

Script e sorgenti del kernel	1589
os16: directory principale	1597
os16: «kernel/devices.h»	1602
os16: «kernel/diag.h»	1606
os16: «kernel/fs.h»	1616
os16: «kernel/ibm_i86.h»	1669
os16: «kernel/k_libc.h»	1682
os16: «kernel/main.h»	1685
os16: «kernel/memory.h»	1690
os16: «kernel/proc.h»	1694
os16: «kernel/tty.h»	1723
Sorgenti della libreria generale	1727
os16: file isolati della directory «lib/»	1735
os16: «lib/dirent.h»	1741
os16: «lib/errno.h»	1744
os16: «lib/fcntl.h»	1747
os16: «lib/grp.h»	1749
os16: «lib/libgen.h»	1749
os16: «lib/pwd.h»	1751
os16: «lib/signal.h»	1752
os16: «lib/stdio.h»	1753
os16: «lib/stdlib.h»	1788
os16: «lib/string.h»	1802
os16: «lib/sys/os16.h»	1811
os16: «lib/sys/stat.h»	1821
os16: «lib/sys/types.h»	1825
os16: «lib/sys/wait.h»	1826
os16: «lib/time.h»	1826
os16: «lib/unistd.h»	1831
os16: «lib/utime.h»	1847
Sorgenti delle applicazioni	1849
os16: directory «applic/»	1850

os16: directory principale	1597
bochs	1597
qemu	1597
makeit	1597
os16: «kernel/devices.h»	1602
kernel/devices/dev_dsk.c	1602
kernel/devices/dev_io.c	1602
kernel/devices/dev_kmem.c	1603
kernel/devices/dev_mem.c	1604
kernel/devices/dev_tty.c	1605
os16: «kernel/diag.h»	1606
kernel/diag/print_fd.c	1607
kernel/diag/print_fd_head.c	1607
kernel/diag/print_fd_list.c	1608
kernel/diag/print_file_head.c	1608
kernel/diag/print_file_list.c	1608
kernel/diag/print_file_num.c	1608
kernel/diag/print_hex_16.c	1609
kernel/diag/print_hex_16_reverse.c	1609
kernel/diag/print_hex_32.c	1609
kernel/diag/print_hex_32_reverse.c	1609
kernel/diag/print_hex_8.c	1609
kernel/diag/print_hex_8_reverse.c	1610
kernel/diag/print_inode.c	1610
kernel/diag/print_inode_head.c	1610
kernel/diag/print_inode_list.c	1611
kernel/diag/print_inode_map.c	1611
kernel/diag/print_inode_zone_list.c	1611
kernel/diag/print_inode_zones.c	1611
kernel/diag/print_inode_zones_head.c	1612
kernel/diag/print_kmem.c	1612
kernel/diag/print_mb_map.c	1612
kernel/diag/print_memory_map.c	1612
kernel/diag/print_proc_head.c	1613
kernel/diag/print_proc_list.c	1613
kernel/diag/print_proc_pid.c	1613
kernel/diag/print_segments.c	1614
kernel/diag/print_superblock.c	1614
kernel/diag/print_time.c	1614
kernel/diag/print_zone_map.c	1614
kernel/diag/reverse_16_bit.c	1615
kernel/diag/reverse_32_bit.c	1615
kernel/diag/reverse_8_bit.c	1615
os16: «kernel/fs.h»	1616
kernel/fs/fd_chmod.c	1618
kernel/fs/fd_chown.c	1619
kernel/fs/fd_close.c	1619
kernel/fs/fd_dup.c	1620
kernel/fs/fd_dup2.c	1620
kernel/fs/fd_fcntl.c	1621
kernel/fs/fd_lseek.c	1622
kernel/fs/fd_open.c	1623
kernel/fs/fd_read.c	1625
kernel/fs/fd_reference.c	1626
kernel/fs/fd_stat.c	1626

kernel/fs/fd_write.c	1627
kernel/fs/file_reference.c	1628
kernel/fs/file_stdio_dev_make.c	1629
kernel/fs/file_table.c	1629
kernel/fs/inode_alloc.c	1629
kernel/fs/inode_check.c	1631
kernel/fs/inode_dir_empty.c	1632
kernel/fs/inode_file_read.c	1632
kernel/fs/inode_file_write.c	1633
kernel/fs/inode_free.c	1635
kernel/fs/inode_fzones_read.c	1635
kernel/fs/inode_fzones_write.c	1636
kernel/fs/inode_get.c	1636
kernel/fs/inode_put.c	1638
kernel/fs/inode_reference.c	1639
kernel/fs/inode_save.c	1640
kernel/fs/inode_stdio_dev_make.c	1641
kernel/fs/inode_table.c	1641
kernel/fs/inode_truncate.c	1641
kernel/fs/inode_zone.c	1643
kernel/fs/path_chdir.c	1647
kernel/fs/path_chmod.c	1647
kernel/fs/path_chown.c	1648
kernel/fs/path_device.c	1649
kernel/fs/path_fix.c	1649
kernel/fs/path_full.c	1650
kernel/fs/path_inode.c	1650
kernel/fs/path_inode_link.c	1653
kernel/fs/path_link.c	1655
kernel/fs/path_mkdir.c	1656
kernel/fs/path_mknod.c	1657
kernel/fs/path_mount.c	1658
kernel/fs/path_stat.c	1659
kernel/fs/path_umount.c	1659
kernel/fs/path_unlink.c	1661
kernel/fs/sb_inode_status.c	1663
kernel/fs/sb_mount.c	1663
kernel/fs/sb_reference.c	1665
kernel/fs/sb_save.c	1665
kernel/fs/sb_table.c	1666
kernel/fs/sb_zone_status.c	1666
kernel/fs/zone_alloc.c	1666
kernel/fs/zone_free.c	1667
kernel/fs/zone_read.c	1668
kernel/fs/zone_write.c	1668
os16: «kernel/ibm_i86.h»	1669
kernel/ibm_i86/_cli.s	1670
kernel/ibm_i86/_in_16.s	1670
kernel/ibm_i86/_in_8.s	1670
kernel/ibm_i86/_int10_00.s	1671
kernel/ibm_i86/_int10_02.s	1671
kernel/ibm_i86/_int10_05.s	1671
kernel/ibm_i86/_int12.s	1672
kernel/ibm_i86/_int13_00.s	1672
kernel/ibm_i86/_int13_02.s	1672
kernel/ibm_i86/_int13_03.s	1673
kernel/ibm_i86/_int16_00.s	1673
kernel/ibm_i86/_int16_01.s	1674

kernel/ibm_i86/_int16_02.s	1674
kernel/ibm_i86/_out_16.s	1675
kernel/ibm_i86/_out_8.s	1675
kernel/ibm_i86/_ram_copy.s	1675
kernel/ibm_i86/_sti.s	1675
kernel/ibm_i86/con_char_read.c	1676
kernel/ibm_i86/con_char_ready.c	1676
kernel/ibm_i86/con_char_wait.c	1676
kernel/ibm_i86/con_init.c	1677
kernel/ibm_i86/con_putc.c	1677
kernel/ibm_i86/con_scroll.c	1678
kernel/ibm_i86/con_select.c	1678
kernel/ibm_i86/dsk_read_bytes.c	1678
kernel/ibm_i86/dsk_read_sectors.c	1679
kernel/ibm_i86/dsk_reset.c	1680
kernel/ibm_i86/dsk_sector_to_chs.c	1680
kernel/ibm_i86/dsk_setup.c	1680
kernel/ibm_i86/dsk_table.c	1680
kernel/ibm_i86/dsk_write_bytes.c	1680
kernel/ibm_i86/dsk_write_sectors.c	1681
kernel/ibm_i86/irq_off.c	1682
kernel/ibm_i86/irq_on.c	1682
os16: «kernel/k_libc.h»	1682
kernel/k_libc/k_clock.c	1683
kernel/k_libc/k_close.c	1683
kernel/k_libc/k_exit.s	1683
kernel/k_libc/k_kill.c	1683
kernel/k_libc/k_open.c	1683
kernel/k_libc/k_perror.c	1684
kernel/k_libc/k_printf.c	1684
kernel/k_libc/k_puts.c	1684
kernel/k_libc/k_read.c	1684
kernel/k_libc/k_stime.c	1685
kernel/k_libc/k_time.c	1685
kernel/k_libc/k_vprintf.c	1685
kernel/k_libc/k_vsprintf.c	1685
os16: «kernel/main.h»	1685
kernel/main/build.h	1685
kernel/main/crt0.s	1686
kernel/main/main.c	1688
kernel/main/menu.c	1689
kernel/main/run.c	1690
os16: «kernel/memory.h»	1690
kernel/memory/address.c	1690
kernel/memory/mb_alloc.c	1691
kernel/memory/mb_alloc_size.c	1691
kernel/memory/mb_free.c	1692
kernel/memory/mb_reference.c	1693
kernel/memory/mb_table.c	1693
kernel/memory/mem_copy.c	1693
kernel/memory/mem_read.c	1694
kernel/memory/mem_write.c	1694
os16: «kernel/proc.h»	1694
kernel/proc/_isr.s	1695
kernel/proc/_ivt_load.s	1698
kernel/proc/proc_available.c	1698
kernel/proc/proc_dump_memory.c	1699

kernel/proc/proc_find.c	1699
kernel/proc/proc_init.c	1700
kernel/proc/proc_reference.c	1701
kernel/proc/proc_sch_signals.c	1701
kernel/proc/proc_sch_terminals.c	1702
kernel/proc/proc_sch_timers.c	1703
kernel/proc/proc_scheduler.c	1703
kernel/proc/proc_sig_chld.c	1705
kernel/proc/proc_sig_cont.c	1705
kernel/proc/proc_sig_core.c	1705
kernel/proc/proc_sig_ignore.c	1706
kernel/proc/proc_sig_off.c	1706
kernel/proc/proc_sig_on.c	1706
kernel/proc/proc_sig_status.c	1706
kernel/proc/proc_sig_stop.c	1707
kernel/proc/proc_sig_term.c	1707
kernel/proc/proc_sys_exec.c	1707
kernel/proc/proc_sys_exit.c	1713
kernel/proc/proc_sys_fork.c	1715
kernel/proc/proc_sys_kill.c	1717
kernel/proc/proc_sys_seteuid.c	1718
kernel/proc/proc_sys_setuid.c	1718
kernel/proc/proc_sys_signal.c	1719
kernel/proc/proc_sys_wait.c	1719
kernel/proc/proc_table.c	1720
kernel/proc/sysroutine.c	1720
os16: «kernel/tty.h»	1723
kernel/tty/tty_console.c	1724
kernel/tty/tty_init.c	1724
kernel/tty/tty_read.c	1725
kernel/tty/tty_reference.c	1725
kernel/tty/tty_table.c	1725
kernel/tty/tty_write.c	1725
address.c	1690
bochs	1597
build.h	1685
con_char_read.c	1676
con_char_ready.c	1676
con_char_wait.c	1676
con_init.c	1677
con_putc.c	1677
con_scroll.c	1678
con_select.c	1678
crt0.s	1686
devices.h	1602
dev_dsk.c	1602
dev_io.c	1602
dev_kmem.c	1603
dev_mem.c	1604
dev_tty.c	1605
diag.h	1606
dsk_read_bytes.c	1678
dsk_read_sectors.c	1679
dsk_reset.c	1680
dsk_sector_to_chs.c	1680
dsk_setup.c	1680
dsk_table.c	1680
dsk_write_bytes.c	1680
dsk_write_sectors.c	1681
fd_chmod.c	1618
fd_chown.c	1619
fd_close.c	1619
fd_dup.c	1620
fd_dup2.c	1620
fd_fcntl.c	1621
fd_lseek.c	1622
fd_open.c	1623
fd_read.c	1625
fd_reference.c	1626
fd_stat.c	1626
fd_write.c	1627
file_reference.c	1628
file_stdio_dev_make.c	1629
file_table.c	1629
fs.h	1616
ibm_i86.h	1669
inode_alloc.c	1629
inode_check.c	1631
inode_dir_empty.c	1632
inode_file_read.c	1632
inode_file_write.c	1633
inode_free.c	1635
inode_fzones_read.c	1635
inode_fzones_write.c	1636
inode_get.c	1636
inode_put.c	1638
inode_reference.c	1639
inode_save.c	1640
inode_stdio_dev_make.c	1641
inode_table.c	1641
inode_truncate.c	1641
inode_zone.c	1643
irq_off.c	1682
irq_on.c	1682
k_clock.c	1683
k_close.c	1683
k_exit.s	1683
k_kill.c	1683
k_libc.h	1682
k_open.c	1683
k_perror.c	1684
k_printf.c	1684
k_puts.c	1684

k_read.c	1684
k_stime.c	1685
k_time.c	1685
k_vprintf.c	1685
k_vsprintf.c	1685
main.c	1688
main.h	1685
makeit	1597
mb_alloc.c	1691
mb_alloc_size.c	1691
mb_free.c	1692
mb_reference.c	1693
mb_table.c	1693
memory.h	1690
mem_copy.c	1693
mem_read.c	1694
mem_write.c	1694
menu.c	1689
path_chdir.c	1647
path_chmod.c	1647
path_chown.c	1648
path_device.c	1649
path_fix.c	1649
path_full.c	1650
path_inode.c	1650
path_inode_link.c	1653
path_link.c	1655
path_mkdir.c	1656
path_mknod.c	1657
path_mount.c	1658
path_stat.c	1659
path_umount.c	1659
path_unlink.c	1661
print_fd.c	1607
print_fd_head.c	1607
print_fd_list.c	1608
print_file_head.c	1608
print_file_list.c	1608
print_file_num.c	1608
print_hex_16.c	1609
print_hex_16_reverse.c	1609
print_hex_32.c	1609
print_hex_32_reverse.c	1609
print_hex_8.c	1609
print_hex_8_reverse.c	1610
print_inode.c	1610
print_inode_head.c	1610
print_inode_list.c	1611
print_inode_map.c	1611
print_inode_zones.c	1611
print_inode_zones_head.c	1612
print_inode_zone_list.c	1611
print_kmem.c	1612
print_mb_map.c	1612
print_memory_map.c	1612
print_proc_head.c	1613
print_proc_list.c	1613
print_proc_pid.c	1613
print_segments.c	1614
print_superblock.c	1614
print_time.c	1614
print_zone_map.c	1614
proc.h	1694
proc_available.c	1698
proc_dump_memory.c	1699
proc_find.c	1699
proc_init.c	1700
proc_reference.c	1701
proc_scheduler.c	1703
proc_sch_signals.c	1701
proc_sch_terminals.c	1702
proc_sch_timers.c	1703
proc_sig_chld.c	1705
proc_sig_cont.c	1705
proc_sig_core.c	1705
proc_sig_ignore.c	1706
proc_sig_off.c	1706
proc_sig_on.c	1706
proc_sig_status.c	1706
proc_sig_stop.c	1707
proc_sig_term.c	1707
proc_sys_exec.c	1707
proc_sys_exit.c	1713
proc_sys_fork.c	1715
proc_sys_kill.c	1717
proc_sys_seteuid.c	1718
proc_sys_setuid.c	1718
proc_sys_signal.c	1719
proc_sys_wait.c	1719
proc_table.c	1720
qemu	1597
reverse_16_bit.c	1615
reverse_32_bit.c	1615
reverse_8_bit.c	1615
run.c	1690
sb_inode_status.c	1663
sb_mount.c	1663
sb_reference.c	1665
sb_save.c	1665
sb_table.c	1666
sb_zone_status.c	1666
sysroutine.c	1720
tty.h	1723
tty_console.c	1724
tty_init.c	1724
tty_read.c	1725
tty_reference.c	1725
tty_table.c	1725
tty_write.c	1725
zone_alloc.c	1666
zone_free.c	1667
zone_read.c	1668
zone_write.c	1668
_cli.s	1670
_int10_00.s	1671
_int10_02.s	1671
_int10_05.s	1671
_int12.s	1672
_int13_00.s	1672
_int13_02.s	1672
_int13_03.s	1673
_int16_00.s	1673
_int16_01.s	1674
_int16_02.s	1674
_in_16.s	1670
_in_8.s	1670
_isr.s	1695
_ivt_load.s	1698
_out_16.s	1675
_out_8.s	1675
_ram_copy.s	1675
_sti.s	1675

os16: directory principale	1597
----------------------------	------

bochs	1597
qemu	1597
makeit	1597

os16: «kernel/devices.h»	1602
--------------------------	------

kernel/devices/dev_dsk.c	1602
kernel/devices/dev_io.c	1602
kernel/devices/dev_kmem.c	1603
kernel/devices/dev_mem.c	1604

kernel/devices/dev_tty.c	1605
os16: «kernel/diag.h»	1606
kernel/diag/print_fd.c	1607
kernel/diag/print_fd_head.c	1607
kernel/diag/print_fd_list.c	1608
kernel/diag/print_file_head.c	1608
kernel/diag/print_file_list.c	1608
kernel/diag/print_file_num.c	1608
kernel/diag/print_hex_16.c	1609
kernel/diag/print_hex_16_reverse.c	1609
kernel/diag/print_hex_32.c	1609
kernel/diag/print_hex_32_reverse.c	1609
kernel/diag/print_hex_8.c	1609
kernel/diag/print_hex_8_reverse.c	1610
kernel/diag/print_inode.c	1610
kernel/diag/print_inode_head.c	1610
kernel/diag/print_inode_list.c	1611
kernel/diag/print_inode_map.c	1611
kernel/diag/print_inode_zone_list.c	1611
kernel/diag/print_inode_zones.c	1611
kernel/diag/print_inode_zones_head.c	1612
kernel/diag/print_kmem.c	1612
kernel/diag/print_mb_map.c	1612
kernel/diag/print_memory_map.c	1612
kernel/diag/print_proc_head.c	1613
kernel/diag/print_proc_list.c	1613
kernel/diag/print_proc_pid.c	1613
kernel/diag/print_segments.c	1614
kernel/diag/print_superblock.c	1614
kernel/diag/print_time.c	1614
kernel/diag/print_zone_map.c	1614
kernel/diag/reverse_16_bit.c	1615
kernel/diag/reverse_32_bit.c	1615
kernel/diag/reverse_8_bit.c	1615
os16: «kernel/fs.h»	1616
kernel/fs/fd_chmod.c	1618
kernel/fs/fd_chown.c	1619
kernel/fs/fd_close.c	1619
kernel/fs/fd_dup.c	1620
kernel/fs/fd_dup2.c	1620
kernel/fs/fd_fcntl.c	1621
kernel/fs/fd_lseek.c	1622
kernel/fs/fd_open.c	1623
kernel/fs/fd_read.c	1625
kernel/fs/fd_reference.c	1626
kernel/fs/fd_stat.c	1626
kernel/fs/fd_write.c	1627
kernel/fs/file_reference.c	1628
kernel/fs/file_stdio_dev_make.c	1629
kernel/fs/file_table.c	1629
kernel/fs/inode_alloc.c	1629
kernel/fs/inode_check.c	1631
kernel/fs/inode_dir_empty.c	1632
kernel/fs/inode_file_read.c	1632
kernel/fs/inode_file_write.c	1633
kernel/fs/inode_free.c	1635
kernel/fs/inode_fzones_read.c	1635

kernel/fs/inode_fzones_write.c	1636
kernel/fs/inode_get.c	1636
kernel/fs/inode_put.c	1638
kernel/fs/inode_reference.c	1639
kernel/fs/inode_save.c	1640
kernel/fs/inode_stdio_dev_make.c	1641
kernel/fs/inode_table.c	1641
kernel/fs/inode_truncate.c	1641
kernel/fs/inode_zone.c	1643
kernel/fs/path_chdir.c	1647
kernel/fs/path_chmod.c	1647
kernel/fs/path_chown.c	1648
kernel/fs/path_device.c	1649
kernel/fs/path_fix.c	1649
kernel/fs/path_full.c	1650
kernel/fs/path_inode.c	1650
kernel/fs/path_inode_link.c	1653
kernel/fs/path_link.c	1655
kernel/fs/path_mkdir.c	1656
kernel/fs/path_mknod.c	1657
kernel/fs/path_mount.c	1658
kernel/fs/path_stat.c	1659
kernel/fs/path_umount.c	1659
kernel/fs/path_unlink.c	1661
kernel/fs/sb_inode_status.c	1663
kernel/fs/sb_mount.c	1663
kernel/fs/sb_reference.c	1665
kernel/fs/sb_save.c	1665
kernel/fs/sb_table.c	1666
kernel/fs/sb_zone_status.c	1666
kernel/fs/zone_alloc.c	1666
kernel/fs/zone_free.c	1667
kernel/fs/zone_read.c	1668
kernel/fs/zone_write.c	1668
os16: «kernel/ibm_i86.h»	1669
kernel/ibm_i86/_cli.s	1670
kernel/ibm_i86/_in_16.s	1670
kernel/ibm_i86/_in_8.s	1670
kernel/ibm_i86/_int10_00.s	1671
kernel/ibm_i86/_int10_02.s	1671
kernel/ibm_i86/_int10_05.s	1671
kernel/ibm_i86/_int12.s	1672
kernel/ibm_i86/_int13_00.s	1672
kernel/ibm_i86/_int13_02.s	1672
kernel/ibm_i86/_int13_03.s	1673
kernel/ibm_i86/_int16_00.s	1673
kernel/ibm_i86/_int16_01.s	1674
kernel/ibm_i86/_int16_02.s	1674
kernel/ibm_i86/_out_16.s	1675
kernel/ibm_i86/_out_8.s	1675
kernel/ibm_i86/_ram_copy.s	1675
kernel/ibm_i86/_sti.s	1675
kernel/ibm_i86/con_char_read.c	1676
kernel/ibm_i86/con_char_ready.c	1676
kernel/ibm_i86/con_char_wait.c	1676
kernel/ibm_i86/con_init.c	1677
kernel/ibm_i86/con_putc.c	1677
kernel/ibm_i86/con_scroll.c	1678

kernel/ibm_i86/con_select.c	1678
kernel/ibm_i86/dsk_read_bytes.c	1678
kernel/ibm_i86/dsk_read_sectors.c	1679
kernel/ibm_i86/dsk_reset.c	1680
kernel/ibm_i86/dsk_sector_to_chs.c	1680
kernel/ibm_i86/dsk_setup.c	1680
kernel/ibm_i86/dsk_table.c	1680
kernel/ibm_i86/dsk_write_bytes.c	1680
kernel/ibm_i86/dsk_write_sectors.c	1681
kernel/ibm_i86/irq_off.c	1682
kernel/ibm_i86/irq_on.c	1682
os16: «kernel/k_libc.h»	1682
kernel/k_libc/k_clock.c	1683
kernel/k_libc/k_close.c	1683
kernel/k_libc/k_exit.s	1683
kernel/k_libc/k_kill.c	1683
kernel/k_libc/k_open.c	1683
kernel/k_libc/k_perror.c	1684
kernel/k_libc/k_printf.c	1684
kernel/k_libc/k_puts.c	1684
kernel/k_libc/k_read.c	1684
kernel/k_libc/k_stime.c	1685
kernel/k_libc/k_time.c	1685
kernel/k_libc/k_vprintf.c	1685
kernel/k_libc/k_vsprintf.c	1685
os16: «kernel/main.h»	1685
kernel/main/build.h	1685
kernel/main/crt0.s	1686
kernel/main/main.c	1688
kernel/main/menu.c	1689
kernel/main/run.c	1690
os16: «kernel/memory.h»	1690
kernel/memory/address.c	1690
kernel/memory/mb_alloc.c	1691
kernel/memory/mb_alloc_size.c	1691
kernel/memory/mb_free.c	1692
kernel/memory/mb_reference.c	1693
kernel/memory/mb_table.c	1693
kernel/memory/mem_copy.c	1693
kernel/memory/mem_read.c	1694
kernel/memory/mem_write.c	1694
os16: «kernel/proc.h»	1694
kernel/proc/_isr.s	1695
kernel/proc/_ivt_load.s	1698
kernel/proc/proc_available.c	1698
kernel/proc/proc_dump_memory.c	1699
kernel/proc/proc_find.c	1699
kernel/proc/proc_init.c	1700
kernel/proc/proc_reference.c	1701
kernel/proc/proc_sch_signals.c	1701
kernel/proc/proc_sch_terminals.c	1702
kernel/proc/proc_sch_timers.c	1703
kernel/proc/proc_scheduler.c	1703
kernel/proc/proc_sig_chld.c	1705
kernel/proc/proc_sig_cont.c	1705
kernel/proc/proc_sig_core.c	1705
kernel/proc/proc_sig_ignore.c	1706

kernel/proc/proc_sig_off.c	1706
kernel/proc/proc_sig_on.c	1706
kernel/proc/proc_sig_status.c	1706
kernel/proc/proc_sig_stop.c	1707
kernel/proc/proc_sig_term.c	1707
kernel/proc/proc_sys_exec.c	1707
kernel/proc/proc_sys_exit.c	1713
kernel/proc/proc_sys_fork.c	1715
kernel/proc/proc_sys_kill.c	1717
kernel/proc/proc_sys_setuid.c	1718
kernel/proc/proc_sys_setuid.c	1718
kernel/proc/proc_sys_signal.c	1719
kernel/proc/proc_sys_wait.c	1719
kernel/proc/proc_table.c	1720
kernel/proc/sysroutine.c	1720
os16: «kernel/tty.h»	1723
kernel/tty/tty_console.c	1724
kernel/tty/tty_init.c	1724
kernel/tty/tty_read.c	1725
kernel/tty/tty_reference.c	1725
kernel/tty/tty_table.c	1725
kernel/tty/tty_write.c	1725

os16: directory principale

bochs

Si veda la sezione [u0.2](#).

```

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

```

qemu

Si veda la sezione [u0.2](#).

```

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

```

makeit

Si veda la sezione [u0.2](#).

```

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

```

```

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

```

1598

```

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

```

1599


```

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

```

```

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

```


os16: «kernel/devices.h»

« Si veda la sezione u0.1.

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

kernel/devices/dev_dsk.c

« Si veda la sezione i159.1.2.

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

kernel/devices/dev_io.c

« Si veda la sezione i159.1.1.

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

1602

```
60033         return (dev_tty (pid, device, rw, offset, buffer, size,
60034 eof));
60035     case DEV_DSK_MAJOR:
60036         return (dev_dsk (pid, device, rw, offset, buffer, size,
60037 eof));
60038     case DEV_KMEM_MAJOR:
60039         return (dev_kmem (pid, device, rw, offset, buffer, size,
60040 eof));
60041     default:
60042         errset (ENODEV);
60043         return (-1);
60044     }
60045 }
```

kernel/devices/dev_kmem.c

« Si veda la sezione i159.1.3.

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

1603

```

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

```

```

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

```

kernel/devices/dev_tty.c

kernel/devices/dev_mem.c

Si veda la sezione [i159.1.4](#).

```

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

```

1604

Si veda la sezione [i159.1.5](#).

```

90001 #include <sys/os16.h>
90002 #include <kernel/devices.h>
90003 #include <sys/types.h>
90004 #include <errno.h>
90005 #include <kernel/memory.h>
90006 #include <kernel/ibm_i86.h>
90007 #include <kernel/proc.h>
90008 #include <string.h>
90009 #include <signal.h>
90010 #include <kernel/k_libc.h>
90011 #include <ctype.h>
90012 #include <kernel/tty.h>
90013 //-----
90014 ssize_t
90015 dev_tty (pid_t pid, dev_t device, int rw, off_t offset, void *buffer,
90016         size_t size, int *eof)
90017 {
90018     uint8_t *buffer08 = (uint8_t *) buffer;
90019     ssize_t n;
90020     proc_t *ps;
90021     //
90022     // Get process. Variable 'ps' will be 'NULL' if the process ID is
90023     // not valid.
90024     //
90025     ps = proc_reference (pid);
90026     //
90027     // Convert 'DEV_TTY' with the controlling terminal for the process.
90028     //
90029     if (device == DEV_TTY)
90030     {
90031         device = ps->device_tty;
90032     }
90033     //

```

1605

```

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

```

os16: «kernel/diag.h»

Si veda la sezione u0.2.

```

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

```

1606

```

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

```

kernel/diag/print_fd.c

Si veda la sezione u0.2.

```

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

```

1607

kernel/diag/print_fd_head.c

Si veda la sezione u0.2.

```

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

```

```

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

```

kernel/diag/print_fd_list.c

Si veda la sezione u0.2.

```

130001 #include <sys/osi16.h>
130002 #include <kernel/diag.h>
130003 #include <kernel/k_libc.h>
130004 //-----
130005 void
130006 print_fd_list (pid_t pid)
130007 {
130008     int     fdn = 0;
130009     fd_t *fd;
130010     fd = fd_reference (pid, &fdn);
130011     print_fd_head ();
130012     for (fdn = 0; fdn < OPEN_MAX; fdn++)
130013     {
130014         if (fd[fdn].file != NULL)
130015         {
130016             k_printf ("%2i ", fdn);
130017             print_fd (fd);
130018         }
130019     }
130020 }

```

kernel/diag/print_file_head.c

Si veda la sezione u0.2.

```

140001 #include <sys/osi16.h>
140002 #include <kernel/diag.h>
140003 #include <kernel/k_libc.h>
140004 //-----
140005 void
140006 print_file_head (void)
140007 {
140008     k_printf ("n. ref flg mode uid size lnks dev ino\n");
140009 }

```

kernel/diag/print_file_list.c

Si veda la sezione u0.2.

```

150001 #include <sys/osi16.h>
150002 #include <kernel/diag.h>
150003 #include <kernel/k_libc.h>
150004 //-----
150005 void
150006 print_file_list (void)
150007 {
150008     int     fno;
150009     file_t *file = file_reference (0);
150010     //
150011     print_file_head ();
150012     //
150013     for (fno = 0; fno < FILE_MAX_SLOTS; fno++)
150014     {
150015         if (file[fno].references > 0)
150016         {
150017             print_file_num (fno);
150018         }
150019     }
150020 }

```

kernel/diag/print_file_num.c

Si veda la sezione u0.2.

```

160001 #include <sys/osi16.h>
160002 #include <kernel/diag.h>
160003 #include <kernel/k_libc.h>
160004 #include <fcntl.h>
160005 //-----
160006 void
160007 print_file_num (int fno)
160008 {
160009     file_t *file = file_reference (fno);
160010
160011     k_printf ("%2i %3i %c/%c %05o %3i %5li %4i %04x %3i",
160012             (unsigned int) fno,
160013             (unsigned int) file->references,
160014             (file->oflags & O_RDONLY ? 'r' : ' '),
160015             (file->oflags & O_WRONLY ? 'w' : ' '),
160016             (unsigned int) file->inode->mode,
160017             (unsigned int) file->inode->uid,
160018             (unsigned long int) file->inode->size,
160019             (unsigned int) file->inode->links,
160020             (unsigned int) file->inode->sb->device,
160021             (unsigned int) file->inode->ino);
160022     k_printf ("\n");
160023 }

```

kernel/diag/print_hex_16.c

Si veda la sezione u0.2.

```

170001 #include <sys/osi16.h>
170002 #include <kernel/diag.h>
170003 #include <kernel/k_libc.h>
170004 #include <inttypes.h>
170005 #include <stdio.h>
170006 //-----
170007 void
170008 print_hex_16 (void *data, size_t elements)
170009 {
170010     uint16_t *element = (uint16_t *) data;
170011     int i;
170012     for (i = 0; i < elements; i++)
170013     {
170014         k_printf ("%04" PRIx16, (uint16_t) element[i]);
170015     }
170016 }

```

kernel/diag/print_hex_16_reverse.c

Si veda la sezione u0.2.

```

180001 #include <sys/osi16.h>
180002 #include <kernel/diag.h>
180003 #include <kernel/k_libc.h>
180004 #include <inttypes.h>
180005 #include <stdio.h>
180006 //-----
180007 void
180008 print_hex_16_reverse (void *data, size_t elements)
180009 {
180010     uint16_t *element = (uint16_t *) data;
180011     int i;
180012     for (i = 0; i < elements; i++)
180013     {
180014         k_printf ("%04" PRIx16, reverse_16_bit (element[i]));
180015     }
180016 }

```

kernel/diag/print_hex_32.c

Si veda la sezione u0.2.

```

190001 #include <sys/osi16.h>
190002 #include <kernel/diag.h>
190003 #include <kernel/k_libc.h>
190004 #include <inttypes.h>
190005 #include <stdio.h>
190006 //-----
190007 void
190008 print_hex_32 (void *data, size_t elements)
190009 {
190010     uint32_t *element = (uint32_t *) data;
190011     int i;
190012     for (i = 0; i < elements; i++)
190013     {
190014         k_printf ("%08" PRIx32, (uint32_t) element[i]);
190015     }
190016 }

```

kernel/diag/print_hex_32_reverse.c

Si veda la sezione u0.2.

```

200001 #include <sys/osi16.h>
200002 #include <kernel/diag.h>
200003 #include <kernel/k_libc.h>
200004 #include <inttypes.h>
200005 #include <stdio.h>
200006 //-----
200007 void
200008 print_hex_32_reverse (void *data, size_t elements)
200009 {
200010     uint32_t *element = (uint32_t *) data;
200011     int i;
200012     for (i = 0; i < elements; i++)
200013     {
200014         k_printf ("%08" PRIx32, reverse_32_bit (element[i]));
200015     }
200016 }

```

kernel/diag/print_hex_8.c

Si veda la sezione u0.2.

```

210001 #include <sys/osi16.h>
210002 #include <kernel/diag.h>
210003 #include <kernel/k_libc.h>
210004 #include <inttypes.h>
210005 #include <stdio.h>
210006 //-----
210007 void
210008 print_hex_8 (void *data, size_t elements)
210009 {
210010     uint8_t *element = (uint8_t *) data;

```

```

230011     int i;
230012     for (i = 0; i < elements; i++)
230013     {
230014         k_printf ("%02" PRIx8, (uint16_t) element[i]);
230015     }
230016 }

```

kernel/diag/print_hex_8_reverse.c

« Si veda la sezione u0.2.

```

230001 #include <sys/os16.h>
230002 #include <kernel/diag.h>
230003 #include <kernel/k_libc.h>
230004 #include <inttypes.h>
230005 #include <stdio.h>
230006 //-----
230007 void
230008 print_hex_8_reverse (void *data, size_t elements)
230009 {
230010     uint8_t *element = (uint8_t *) data;
230011     int i;
230012     for (i = 0; i < elements; i++)
230013     {
230014         k_printf ("%02" PRIx8, reverse_8_bit (element[i]));
230015     }
230016 }

```

kernel/diag/print_inode.c

« Si veda la sezione u0.2.

```

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

```

kernel/diag/print_inode_head.c

« Si veda la sezione u0.2.

```

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

```

```

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

```

kernel/diag/print_inode_list.c

« Si veda la sezione u0.2.

```

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

```

kernel/diag/print_inode_map.c

« Si veda la sezione u0.2.

```

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

```

kernel/diag/print_inode_zone_list.c

« Si veda la sezione u0.2.

```

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

```

kernel/diag/print_inode_zones.c

« Si veda la sezione u0.2.

```

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

```

```

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

```

kernel/diag/print_inode_zones_head.c

« Si veda la sezione u0.2.

```

290001 #include <sys/osi6.h>
290002 #include <kernel/diag.h>
290003 #include <kernel/k_libc.h>
290004 //-----
290005 void
290006 print_inode_zones_head (void)
290007 {
290008     k_printf (" dev ino zn_0 zn_1 zn_2 zn_3 zn_4 zn_5 zn_6 ");
290009     k_printf ("ind1 ind2\n");
290010 }

```

kernel/diag/print_kmem.c

« Si veda la sezione u0.2.

```

300001 #include <sys/osi6.h>
300002 #include <kernel/diag.h>
300003 #include <kernel/k_libc.h>
300004 //-----
300005 extern uint16_t _ksp;
300006 extern uint16_t _etext;
300007 extern uint16_t _edata;
300008 extern uint16_t _end;
300009 //-----
300010 void
300011 print_kmem (void)
300012 {
300013     k_printf ("etext=%04x edata=%04x ebss=%04x ksp=%04x",
300014             (unsigned int) &etext,
300015             (unsigned int) &edata, (unsigned int) &end, _ksp);
300016 }

```

kernel/diag/print_mb_map.c

« Si veda la sezione u0.2.

```

310001 #include <sys/osi6.h>
310002 #include <kernel/diag.h>
310003 #include <kernel/k_libc.h>
310004 //-----
310005 void
310006 print_mb_map (void)
310007 {
310008     uint16_t *mb = mb_reference ();
310009     unsigned int i;
310010     for (i = 0; i < (MEM_MAX_BLOCKS / 16); i++)
310011     {
310012         k_printf ("%04x", mb[i]);
310013     }
310014 }

```

kernel/diag/print_memory_map.c

« Si veda la sezione u0.2.

```

320001 #include <sys/osi6.h>
320002 #include <kernel/diag.h>
320003 #include <kernel/k_libc.h>
320004 //-----
320005 void
320006 print_memory_map (void)
320007 {
320008     uint16_t *mem_block[MEM_BLOCK_SIZE/2];
320009     uint16_t block_rank;
320010     unsigned int b;

```

1612

```

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

```

kernel/diag/print_proc_head.c

« Si veda la sezione u0.2.

```

330001 #include <sys/osi6.h>
330002 #include <kernel/diag.h>
330003 #include <kernel/k_libc.h>
330004 //-----
330005 void
330006 print_proc_head (void)
330007 {
330008     k_printf (
330009         "pp p pg
330010         *id id rp tty uid euid suid usage s iaddr isiz daddr dsiz sp name\n"
330011         );
330012 }

```

kernel/diag/print_proc_list.c

« Si veda la sezione u0.2.

```

340001 #include <sys/osi6.h>
340002 #include <kernel/diag.h>
340003 #include <kernel/k_libc.h>
340004 //-----
340005 void
340006 print_proc_list (void)
340007 {
340008     pid_t pid;
340009     proc_t *ps;
340010     //
340011     print_proc_head ();
340012     //
340013     for (pid = 0; pid < PROCESS_MAX; pid++)
340014     {
340015         ps = proc_reference (pid);
340016         if (ps != NULL && ps->status > 0)
340017         {
340018             print_proc_pid (ps, pid);
340019         }
340020     }
340021 }

```

kernel/diag/print_proc_pid.c

« Si veda la sezione u0.2.

```

350001 #include <sys/osi6.h>
350002 #include <kernel/diag.h>
350003 #include <kernel/k_libc.h>
350004 //-----
350005 void
350006 print_proc_pid (proc_t *ps, pid_t pid)
350007 {
350008     char stat;
350009     switch (ps->status)
350010     {
350011         case PROC_EMPTY : stat = '-'; break;
350012         case PROC_CREATED : stat = 'c'; break;
350013         case PROC_READY : stat = 'r'; break;
350014         case PROC_RUNNING : stat = 'R'; break;
350015         case PROC_SLEEPING : stat = 's'; break;

```

1613

```

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

```

kernel/diag/print_segments.c

« Si veda la sezione u0.2.

```

360001 #include <sys/osal6.h>
360002 #include <kernel/diag.h>
360003 #include <kernel/k_libc.h>
360004 //-----
360005 void
360006 print_segments (void)
360007 {
360008     k_printf ("CS=%04x DS=%04x SS=%04x ES=%04x BP=%04x SP=%04x ",
360009             cs (), ds (), ss (), es (), bp (), sp ());
360010     k_printf ("heap_min=%04x", heap_min ());
360011 }

```

kernel/diag/print_superblock.c

« Si veda la sezione u0.2.

```

370001 #include <sys/osal6.h>
370002 #include <kernel/diag.h>
370003 #include <kernel/k_libc.h>
370004 //-----
370005 void
370006 print_superblock (sb_t *sb)
370007 {
370008     k_printf ("Inodes:           %i\n", sb->inodes);
370009     k_printf ("Blocks:             %i\n", sb->zones);
370010     k_printf ("First data zone:      %i\n", sb->first_data_zone);
370011     k_printf ("Zone size:           %i\n", (1024 << sb->log2_size_zone));
370012     k_printf ("Max file size:       %i\n", sb->max_file_size);
370013     k_printf ("Inode map blocks:    %i\n", sb->map_inode_blocks);
370014     k_printf ("Zone map blocks:     %i\n", sb->map_zone_blocks);
370015 }

```

kernel/diag/print_time.c

« Si veda la sezione u0.2.

```

380001 #include <sys/osal6.h>
380002 #include <kernel/diag.h>
380003 #include <kernel/k_libc.h>
380004 //-----
380005 void
380006 print_time (void)
380007 {
380008     unsigned long int ticks = k_clock ();
380009     unsigned long int seconds = k_time (NULL);
380010     unsigned int h = seconds / 60 / 60;
380011     unsigned int m = seconds / 60 - h * 60;
380012     unsigned int s = seconds - m * 60 - h * 60 * 60;
380013     k_printf ("clock=%08lx, time elapsed=%02u:%02u:%02u",
380014             ticks, h, m, s);
380015 }

```

kernel/diag/print_zone_map.c

« Si veda la sezione u0.2.

```

390001 #include <sys/osal6.h>
390002 #include <kernel/diag.h>
390003 #include <kernel/k_libc.h>
390004 //-----
390005 void
390006 print_zone_map (sb_t *sb, uint16_t *bitmap)
390007 {
390008     size_t size;
390009     unsigned int data_zones = sb->zones - sb->first_data_zone;
390010     if (data_zones % 16)

```

```

390011     {
390012         size = data_zones/16 + 1;
390013     }
390014     else
390015     {
390016         size = data_zones/16;
390017     }
390018     print_hex_l6_reverse (bitmap, size);
390019 }

```

kernel/diag/reverse_16_bit.c

« Si veda la sezione u0.2.

```

400001 #include <sys/osal6.h>
400002 #include <kernel/diag.h>
400003 #include <kernel/k_libc.h>
400004 #include <inttypes.h>
400005 //-----
400006 uint16_t
400007 reverse_16_bit (uint16_t source)
400008 {
400009     uint16_t destination = 0;
400010     uint16_t mask_src;
400011     uint16_t mask_dst;
400012     int i;
400013     for (i = 0; i < 16; i++)
400014     {
400015         mask_src = 0x0001 << i;
400016         mask_dst = 0x8000 >> i;
400017         if (source & mask_src)
400018         {
400019             destination |= mask_dst;
400020         }
400021     }
400022     return (destination);
400023 }

```

kernel/diag/reverse_32_bit.c

« Si veda la sezione u0.2.

```

410001 #include <sys/osal6.h>
410002 #include <kernel/diag.h>
410003 #include <kernel/k_libc.h>
410004 //-----
410005 uint32_t
410006 reverse_32_bit (uint32_t source)
410007 {
410008     uint32_t destination = 0;
410009     uint32_t mask_src;
410010     uint32_t mask_dst;
410011     int i;
410012     for (i = 0; i < 32; i++)
410013     {
410014         mask_src = 0x00000001 << i;
410015         mask_dst = 0x80000000 >> i;
410016         if (source & mask_src)
410017         {
410018             destination |= mask_dst;
410019         }
410020     }
410021     return (destination);
410022 }

```

kernel/diag/reverse_8_bit.c

« Si veda la sezione u0.2.

```

420001 #include <sys/osal6.h>
420002 #include <kernel/diag.h>
420003 #include <kernel/k_libc.h>
420004 #include <inttypes.h>
420005 //-----
420006 uint8_t
420007 reverse_8_bit (uint8_t source)
420008 {
420009     uint8_t destination = 0;
420010     uint8_t mask_src;
420011     uint8_t mask_dst;
420012     int i;
420013     for (i = 0; i < 8; i++)
420014     {
420015         mask_src = 0x01 << i;
420016         mask_dst = 0x80 >> i;
420017         if (source & mask_src)
420018         {
420019             destination |= mask_dst;
420020         }
420021     }
420022     return (destination);
420023 }

```


« Si veda la sezione u0.3.

```

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

```

```

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

```

```

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

```

kernel/fs/fd_chmod.c

Si veda la sezione [i159.3.1](#).

```

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

```

1618

kernel/fs/fd_chown.c

Si veda la sezione [i159.3.2](#).

```

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

```

kernel/fs/fd_close.c

Si veda la sezione [i159.3.3](#).

```

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

```

1619

```

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

```

kernel/fs/fd_dup.c

<<

Si veda la sezione [i159.3.4](#).

```

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

```

kernel/fs/fd_dup2.c

<<

Si veda la sezione [i159.3.4](#).

```

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

```

1620

```

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

```

kernel/fs/fd_fcntl.c

<<

Si veda la sezione [i159.3.6](#).

```

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

```

1621

```

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

```

kernel/fs/fd_lseek.c

Si veda la sezione [i159.3.7](#).

```

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

```

kernel/fs/fd_open.c

Si veda la sezione [i159.3.8](#).

```

510001 #include <kernel/proc.h>
510002 #include <kernel/k_libc.h>
510003 #include <errno.h>
510004 #include <fcntl.h>
510005 //-----
510006 int
510007 fd_open (pid_t pid, const char *path, int oflags, mode_t mode)
510008 {
510009     proc_t *ps;
510010     inode_t *inode;
510011     int *status;
510012     file_t *file;
510013     fd_t *fd;
510014     int fdn;
510015     char full_path[PATH_MAX];
510016     int perm;
510017     tty_t *tty;
510018     mode_t umask;
510019     int errno_save;
510020     //
510021     // Get process.
510022     //
510023     ps = proc_reference (pid);
510024     //
510025     // Correct the mode with the umask. As it is not a directory, to the
510026     // mode are removed execution and sticky permissions.
510027     //
510028     umask = ps->umask | 0111;
510029     mode &= ~umask;
510030     //
510031     // Check open options.
510032     //
510033     if (oflags & O_WRONLY)
510034     {
510035         //
510036         // The file is to be opened for write, or for read/write.
510037         // Try to get inode.
510038         //
510039         inode = path_inode (pid, path);
510040         if (inode == NULL)
510041         {
510042             //
510043             // Cannot get the inode. See if there is the creation
510044             // option.
510045             //
510046             if (oflags & O_CREAT)
510047             {
510048                 //
510049                 // Try to create the missing inode: the file must be a
510050                 // regular one, so add the mode.
510051                 //
510052                 path_full (path, ps->path_cwd, full_path);
510053                 inode = path_inode_link (pid, full_path, NULL,
510054                                         (mode | S_IFREG));
510055                 if (inode == NULL)
510056                 {
510057                     //
510058                     // Sorry: cannot create the inode! Variable 'errno'
510059                     // is already set by 'path_inode_link()'.
510060                     //
510061                     errset (errno);
510062                     return (-1);
510063                 }
510064             }
510065             else
510066             {
510067                 //
510068                 // Cannot open the inode. Variable 'errno'
510069                 // should be already set by 'path_inode()'.
510070                 //
510071                 errset (errno);
510072                 return (-1);
510073             }
510074         }
510075         //
510076         // The inode was read or created: check if it must be
510077         // truncated. It can be truncated only if it is a regular
510078         // file.
510079         //
510080         if (oflags & O_TRUNC && inode->mode & S_IFREG)
510081         {
510082             //
510083             // Truncate inode.
510084             //
510085             status = inode_truncate (inode);
510086             if (status != 0)
510087             {
510088                 //
510089                 // Cannot truncate the inode: release it and return.
510090                 // But this error should never happen, because the
510091                 // function 'inode_truncate()' will not return any
510092                 // other value than zero.
510093                 //
510094                 errno_save = errno;
510095                 inode_put (inode);
510096                 errset (errno_save);
510097                 return (-1);
510098             }
510099         }
510100     }

```

```

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

```

1624

```

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

```

kernel/fs/fd_read.c

Si veda la sezione [i159.3.9](#).

«

```

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

```

1625

```

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

```

kernel/fs/fd_reference.c

Si veda la sezione [i159.3.10](#).

```

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

```

kernel/fs/fd_stat.c

Si veda la sezione [i159.3.50](#).

```

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

```

1626

```

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

```

kernel/fs/fd_write.c

Si veda la sezione [i159.3.12](#).

```

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

```

1627

```

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

```

kernel/fs/file_reference.c

Si veda la sezione [i159.3.13](#).

```

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

```

1628

kernel/fs/file_stdio_dev_make.c

Si veda la sezione [i159.3.14](#).

```

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

```

kernel/fs/file_table.c

Si veda la sezione [i159.3.13](#).

```

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

```

kernel/fs/inode_alloc.c

Si veda la sezione [i159.3.15](#).

```

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

```

1629


```

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

```

1630

```

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

```

kernel/fs/inode_check.c

Si veda la sezione [1159.3.16](#).

«

```

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

```

1631

```

600067     return (0);
600068     }
600069 }

```

kernel/fs/inode_dir_empty.c

Si veda la sezione [i159.3.17](#).

```

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

```

kernel/fs/inode_file_read.c

Si veda la sezione [i159.3.18](#).

```

620001 #include <kernel/fs.h>
620002 #include <errno.h>
620003 #include <kernel/k_libc.h>
620004 //-----
620005 ssize_t
620006 inode_file_read (inode_t *inode, off_t offset,
620007                 void *buffer, size_t count, int *eof)
620008 {
620009     unsigned char *destination = (unsigned char *) buffer;
620010     unsigned char zone_buffer[SB_MAX_ZONE_SIZE];
620011     blkcnt_t      blkcnt_read;
620012     off_t          off_fzone; // File zone offset.
620013     off_t          off_buffer; // Destination buffer offset.
620014     ssize_t        size_read; // Byte transfer counter.
620015     zno_t          fzone;
620016     off_t          off_end;
620017     //
620018     // The inode pointer must be valid, and
620019     // the start byte must be positive.
620020     //
620021     if (inode == NULL || offset < 0)
620022     {
620023         errset (EINVAL); // Invalid argument.
620024         return ((ssize_t) -1);
620025     }
620026     //
620027     // Check if the start address is inside the file size. This is not
620028     // an error, but zero bytes are read and '*eof' is set. Otherwise,
620029     // '*eof' is reset.

```

1632

```

620030     //
620031     if (offset >= inode->size)
620032     {
620033         (eof != NULL)? *eof = 1: 0;
620034         return (0);
620035     }
620036     else
620037     {
620038         (eof != NULL)? *eof = 0: 0;
620039     }
620040     //
620041     // Adjust, if necessary, the size of read, because it cannot be
620042     // larger than the actual file size. The variable 'off_end' is
620043     // used to calculate the position *after* the requested read.
620044     // Remember that the first file position is byte zero; so,
620045     // the byte index inside the file goes from zero to inode->size -1.
620046     //
620047     off_end = offset;
620048     off_end += count;
620049     if (off_end > inode->size)
620050     {
620051         count = (inode->size - offset);
620052     }
620053     //
620054     // Read the first file-zone inside the zone buffer.
620055     //
620056     fzone = offset / inode->sb->blksize;
620057     off_fzone = offset % inode->sb->blksize;
620058     blkcnt_read = inode_fzones_read (inode, fzone, zone_buffer,
620059                                     (blkcnt_t) 1);
620060     if (blkcnt_read <= 0)
620061     {
620062         // Sorry!
620063         //
620064         //
620065         return (0); // Zero bytes read!
620066     }
620067     //
620068     // The first file-zone was read: copy it inside the destination
620069     // buffer and continue reading the other zones needed. Variables
620070     // 'off_buffer' (destination buffer index) and 'size_read' (copy
620071     // byte counter) must be reset here. Variable 'off_fzone' is already
620072     // set with the initial offset inside 'zone_buffer'.
620073     //
620074     off_buffer = 0;
620075     size_read = 0;
620076     //
620077     while (count)
620078     {
620079         //
620080         // Copy the zone buffer into the destination. Variables
620081         // 'off_fzone', 'off_buffer' and 'size_read' must not be
620082         // initialized inside the loop.
620083         //
620084         for (; off_fzone < inode->sb->blksize && count > 0;
620085              off_fzone++, off_buffer++, size_read++,
620086              count--, offset++)
620087         {
620088             destination[off_buffer] = zone_buffer[off_fzone];
620089         }
620090         //
620091         // If not all the bytes are copied, read the next file-zone.
620092         //
620093         if (count)
620094         {
620095             //
620096             // Read another file-zone inside the zone buffer.
620097             // Again, the function 'inode_fzones_read()' might
620098             // return a null pointer, but the variable 'errno' tells if
620099             // it is really an error. For this reason, the variable
620100             // 'errno' must be reset before the read, and checked after
620101             // it.
620102             //
620103             fzone = offset / inode->sb->blksize;
620104             off_fzone = offset % inode->sb->blksize;
620105             blkcnt_read = inode_fzones_read (inode, fzone, zone_buffer,
620106                                             (blkcnt_t) 1);
620107             if (blkcnt_read <= 0)
620108             {
620109                 //
620110                 // Sorry: only 'size_read' bytes read!
620111                 //
620112                 return (size_read);
620113             }
620114         }
620115     }
620116     //
620117     // The requested size was read completely.
620118     //
620119     return (size_read);
620120 }

```

kernel/fs/inode_file_write.c

Si veda la sezione [i159.3.19](#).

```

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

```

1633

```

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

```

1634

kernel/fs/inode_free.c

Si veda la sezione [i159.3.20](#).

```

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

```

kernel/fs/inode_fzones_read.c

Si veda la sezione [i159.3.21](#).

```

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

```

1635

```

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

```

kernel/fs/inode_fzones_write.c

Si veda la sezione [i159.3.21](#).

```

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

```

kernel/fs/inode_get.c

Si veda la sezione [i159.3.23](#).

```

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

```

1636

```

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

```

1637

```

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

```

kernel/fs/inode_put.c

« Si veda la sezione [i159.3.24](#).

```

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

```

1638

```

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

```

kernel/fs/inode_reference.c

Si veda la sezione [i159.3.25](#).

```

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

```

1639

```

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

```

kernel/fs/inode_save.c

Si veda la sezione [i159.3.26](#).

```

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

```

kernel/fs/inode_stdio_dev_make.c

Si veda la sezione [i159.3.27](#).

```

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

```

kernel/fs/inode_table.c

Si veda la sezione [i159.3.25](#).

```

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

```

kernel/fs/inode_truncate.c

Si veda la sezione [i159.3.28](#).

```

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

```

```

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

```

1642

```

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

```

kernel/fs/inode_zone.c

Si veda la sezione [i159.3.29](#).

«

```

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

```

1643


```

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

```

1644

```

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

```

1645

```

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

```

1646

```

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

```

kernel/fs/path_chdir.c

Si veda la sezione [i159.3.30](#).

```

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

```

kernel/fs/path_chmod.c

Si veda la sezione [i159.3.31](#).

```

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

```

1647

```

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

```

```

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

```

kernel/fs/path_device.c

Si veda la sezione [i159.3.33](#).

```

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

```

kernel/fs/path_chown.c

Si veda la sezione [i159.3.32](#).

```

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

```

1648

kernel/fs/path_fix.c

Si veda la sezione [i159.3.34](#).

```

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

```

1649

```

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

```

kernel/fs/path_full.c

<<

Si veda la sezione [i159.3.35](#).

```

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

```

kernel/fs/path_inode.c

<<

Si veda la sezione [i159.3.36](#).

```

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

```

```

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

```

```

810109 //
810110 // The directory is ended: release the inode and return.
810111 //
810112 inode_put (inode);
810113 errset (ENOENT); // No such file or directory.
810114 return (NULL);
810115 }
810116 //
810117 // Calculate how many directory items we have read.
810118 //
810119 dir_size_read = size_read / 16;
810120 //
810121 // Scan the directory to find the current name.
810122 //
810123 for (d = 0; d < dir_size_read; d++)
810124 {
810125 //
810126 // Ensure to have a null terminated string for
810127 // the name found.
810128 //
810129 memcpy (dir_name, dir[d].name, (size_t) NAME_MAX);
810130 dir_name[NAME_MAX] = 0;
810131 //
810132 comp = strcmp (name, dir_name);
810133 if (comp == 0 && dir[d].ino != 0)
810134 {
810135 //
810136 // Found the name and verified that it has a link to
810137 // a inode. Now release the directory inode.
810138 //
810139 inode_put (inode);
810140 //
810141 // Get next inode and break the loop.
810142 //
810143 inode = inode_get (device, dir[d].ino);
810144 break;
810145 }
810146 }
810147 //
810148 // If index 'd' is in a valid range, the name was found.
810149 //
810150 if (d < dir_size_read)
810151 {
810152 //
810153 // The name was found.
810154 //
810155 break;
810156 }
810157 //
810158 //
810159 // If the function is still working, a file or a directory
810160 // was found: see if there is another name after this one
810161 // to look for. If there isn't, just break the loop.
810162 //
810163 if (next == NULL)
810164 {
810165 //
810166 // As no other tokens are to be found, break the loop.
810167 //
810168 break;
810169 }
810170 //
810171 // As there is another name after the current one,
810172 // the current file must be a directory.
810173 //
810174 status = inode_check (inode, S_IFDIR, 1, ps->euid);
810175 if (status != 0)
810176 {
810177 //
810178 // Variable 'errno' is set by 'inode_check()'.
810179 //
810180 errset (errno);
810181 inode_put (inode);
810182 return (NULL);
810183 }
810184 //
810185 // The inode is a directory and the user has the necessary
810186 // permissions: check if it is a mount point and go to the
810187 // new device root directory if necessary.
810188 //
810189 if (inode->sb_attached != NULL)
810190 {
810191 //
810192 // Must find the root directory for the new device, and
810193 // then go to that inode.
810194 //
810195 device = inode->sb_attached->device;
810196 inode_put (inode);
810197 inode = inode_get (device, 1);
810198 status = inode_check (inode, S_IFDIR, 1, ps->euid);
810199 if (status != 0)
810200 {
810201 inode_put (inode);
810202 return (NULL);
810203 }
810204 //
810205 //
810206 // As a directory was found, and another token follows it,
810207 // must continue the token scan.
810208 //
810209 name = next;

```

1652

```

810210 }
810211 //
810212 // Current inode found is the file represented by the requested
810213 // path.
810214 //
810215 return (inode);
810216 }

```

kernel/fs/path_inode_link.c

Si veda la sezione [i159.3.37](#).

«

```

820001 #include <kernel/fs.h>
820002 #include <errno.h>
820003 #include <kernel/proc.h>
820004 #include <libgen.h>
820005 //-----
820006 inode_t *
820007 path_inode_link (pid_t pid, const char *path, inode_t *inode,
820008 mode_t mode)
820009 {
820010 proc_t *ps;
820011 char buffer[SB_MAX_ZONE_SIZE];
820012 off_t start;
820013 int d; // Directory index.
820014 ssize_t size_read;
820015 ssize_t size_written;
820016 directory_t *dir = (directory_t *) buffer;
820017 char path_copy1[PATH_MAX];
820018 char path_copy2[PATH_MAX];
820019 char *path_directory;
820020 char *path_name;
820021 inode_t *inode_directory;
820022 inode_t *inode_new;
820023 dev_t device;
820024 int status;
820025 //
820026 // Check arguments.
820027 //
820028 if (path == NULL || strlen (path) == 0)
820029 {
820030 errset (EINVAL); // Invalid argument:
820031 return (NULL); // the path is mandatory.
820032 }
820033 //
820034 if (inode == NULL && mode == 0)
820035 {
820036 errset (EINVAL); // Invalid argument: if the inode is to
820037 return (NULL); // be created, the mode is mandatory.
820038 }
820039 //
820040 if (inode != NULL)
820041 {
820042 if (mode != 0)
820043 {
820044 errset (EINVAL); // Invalid argument: if the inode is
820045 return (NULL); // already present, the creation mode
820046 } // must not be given.
820047 if (S_ISDIR (inode->mode))
820048 {
820049 errset (EPERM); // Operation not permitted.
820050 return (NULL); // Refuse to link directory.
820051 }
820052 if (inode->links >= LINK_MAX)
820053 {
820054 errset (EMLINK); // Too many links.
820055 return (NULL);
820056 }
820057 }
820058 //
820059 // Get process.
820060 //
820061 ps = proc_reference (pid);
820062 //
820063 // If the destination path already exists, the link cannot be made.
820064 // It does not matter if the inode is known or not.
820065 //
820066 inode_new = path_inode ((uid_t) 0, path);
820067 if (inode_new != NULL)
820068 {
820069 //
820070 // A file already exists with the same name.
820071 //
820072 inode_put (inode_new);
820073 errset (EEXIST); // File exists.
820074 return (NULL);
820075 }
820076 //
820077 // At this point, 'inode_new' is 'NULL'.
820078 // Copy the source path inside the directory path and name arrays.
820079 //
820080 strncpy (path_copy1, path, PATH_MAX);
820081 strncpy (path_copy2, path, PATH_MAX);
820082 //
820083 // Reduce to directory name and find the last name.
820084 //
820085 path_directory = dirname (path_copy1);
820086 path_name = basename (path_copy2);
820087 if (strlen (path_directory) == 0 || strlen (path_name) == 0)
820088 {
820089 errset (EACCES); // Permission denied: maybe the

```

1653

```

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

```

1654

```

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

```

kernel/fs/path_link.c

Si veda la sezione [i159.3.38](#).

```

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

```

1655

```

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

```

kernel/fs/path_mkdir.c

« Si veda la sezione [i159.3.39](#).

```

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

```

1656

```

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

```

kernel/fs/path_mknod.c

« Si veda la sezione [i159.3.40](#).

```

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

```

1657

```

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

```

kernel/fs/path_mount.c

Si veda la sezione [i159.3.41](#).

```

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

```

1658

```

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

```

kernel/fs/path_stat.c

Si veda la sezione [i159.3.50](#).

```

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

```

kernel/fs/path_umount.c

Si veda la sezione [i159.3.41](#).

```

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

```

1659


```

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

```

1660

```

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

```

kernel/fs/path_unlink.c

Si veda la sezione [i159.3.44](#).

```

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

```

1661

```

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

```

1662

kernel/fs/sb_inode_status.c

Si veda la sezione [i159.3.45](#).

```

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

```

kernel/fs/sb_mount.c

Si veda la sezione [i159.3.46](#).

```

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

```

1663

```

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

```

1664

kernel/fs/sb_reference.c

Si veda la sezione [i159.3.47](#).

```

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

```

kernel/fs/sb_save.c

Si veda la sezione [i159.3.48](#).

```

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

```

1665

```

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

```

kernel/fs/sb_table.c

<<

Si veda la sezione [i159.3.47](#).

```

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

```

kernel/fs/sb_zone_status.c

<<

Si veda la sezione [i159.3.45](#).

```

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

```

kernel/fs/zone_alloc.c

<<

Si veda la sezione [i159.3.51](#).

```

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

```

1666

```

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

```

kernel/fs/zone_free.c

>>

Si veda la sezione [i159.3.51](#).

```

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

```

1667

```

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

```

```

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

```

kernel/fs/zone_read.c

Si veda la sezione [i159.3.53](#).

```

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

```

kernel/fs/zone_write.c

Si veda la sezione [i159.3.53](#).

```

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

```

1668

os16: «kernel/ibm_i86.h»

Si veda la sezione [u0.4](#).

```

100001 #ifndef _KERNEL_IBM_I86_H
100002 #define _KERNEL_IBM_I86_H 1
100003
100004 #include <stdint.h>
100005 #include <size_t.h>
100006 #include <kernel/memory.h>
100007 #include <sys/types.h>
100008 //-----
100009 #define IBM_I86_VIDEO_MODE 0x02
100010 #define IBM_I86_VIDEO_PAGES 4
100011
100012 #define IBM_I86_VIDEO_COLUMNS 80
100013 #define IBM_I86_VIDEO_ROWS 25
100014 #define IBM_I86_VIDEO_ADDRESS 0xB8000L, 0xB9000L, 0xBA000L, 0xBB000L
100015 //-----
100016 void _int10_00 (uint16_t video_mode);
100017 void _int10_02 (uint16_t page, uint16_t position);
100018 void _int10_05 (uint16_t page);
100019 uint16_t _int12 (void);
100020 uint16_t _int13_00 (uint16_t drive);
100021 uint16_t _int13_02 (uint16_t drive, uint16_t sectors,
100022                  uint16_t cylinder, uint16_t head,
100023                  uint16_t sector, void *buffer);
100024 uint16_t _int13_03 (uint16_t drive, uint16_t sectors,
100025                  uint16_t cylinder, uint16_t head,
100026                  uint16_t sector, void *buffer);
100027 uint16_t _int16_00 (void);
100028 uint16_t _int16_01 (void);
100029 uint16_t _int16_02 (void);
100030
100031 #define int10_00(video_mode) (_int10_00 ((uint16_t) video_mode))
100032 #define int10_02(page, position) (_int10_02 ((uint16_t) page, \
100033                                             (uint16_t) position))
100034 #define int10_05(page) (_int10_05 ((uint16_t) page))
100035 #define int12() ((unsigned int) _int12 ())
100036
100037 #define int13_00(drive) ((unsigned int) \
100038                        _int13_00 ((uint16_t) drive))
100039 #define int13_02(drive, sectors, cylinder, head, sector, buffer) \
100040 ((unsigned int) \
100041  _int13_02 ((uint16_t) drive, \
100042            (uint16_t) sectors, \
100043            (uint16_t) cylinder, \
100044            (uint16_t) head, \
100045            (uint16_t) sector, \
100046            buffer))
100047 #define int13_03(drive, sectors, cylinder, head, sector, buffer) \
100048 ((unsigned int) \
100049  _int13_03 ((uint16_t) drive, \
100050            (uint16_t) sectors, \
100051            (uint16_t) cylinder, \
100052            (uint16_t) head, \
100053            (uint16_t) sector, \
100054            buffer))
100055 #define int16_00() ((unsigned int) _int16_00 ())
100056 #define int16_01() ((unsigned int) _int16_01 ())
100057 #define int16_02() ((unsigned int) _int16_02 ())
100058 //-----
100059 uint16_t _in_8 (uint16_t port);
100060 uint16_t _in_16 (uint16_t port);
100061 void _out_8 (uint16_t port, uint16_t value);
100062 void _out_16 (uint16_t port, uint16_t value);
100063
100064 #define in_8(port) ((unsigned int) _in_8 ((uint16_t) port))
100065 #define in_16(port) ((unsigned int) _in_16 ((uint16_t) port))
100066 #define out_8(port, value) (_out_8 ((uint16_t) port, \
100067                                   (uint16_t) value))
100068 #define out_16(port, value) (_out_16 ((uint16_t) port, \
100069                                     (uint16_t) value))
100070 //-----
100071 void _cli (void);
100072 void _sti (void);
100073
100074 #define cli() (_cli ())
100075 #define sti() (_sti ())
100076 //-----
100077 void irq_on (unsigned int irq);
100078 void irq_off (unsigned int irq);
100079 //-----
100080 void _ram_copy (segment_t org_seg, offset_t org_off,
100081               segment_t dst_seg, offset_t dst_off,
100082               uint16_t size);
100083

```

1669

```

100084 #define ram_copy(org_seg, org_off, dst_seg, dst_off, size) \
100085     (_ram_copy ((uint16_t) org_seg, \
100086                (uint16_t) org_off, \
100087                (uint16_t) dst_seg, \
100088                (uint16_t) dst_off, \
100089                (uint16_t) size))
100090 //-----
100091 void con_select (int console);
100092 void con_putc (int console, int c);
100093 void con_scroll (int console);
100094 int con_char_wait (void);
100095 int con_char_read (void);
100096 int con_char_ready (void);
100097 void con_init (void);
100098 //-----
100099 #define DSK_MAX 4
100100 #define DSK_SECTOR_SIZE 512 // Fixed!
100101
100102 typedef struct {
100103     unsigned int bios_drive;
100104     unsigned int cylinders;
100105     unsigned int heads;
100106     unsigned int sectors;
100107     unsigned int retry;
100108 } dsk_t;
100109
100110 typedef struct {
100111     unsigned int cylinder;
100112     unsigned int head;
100113     unsigned int sector;
100114 } dsk_chs_t;
100115 //-----
100116 extern dsk_t dsk_table[DSK_MAX];
100117 //-----
100118 void dsk_setup (void);
100119 int dsk_reset (int drive);
100120 void dsk_sector_to_chs (int drive, unsigned int sector,
100121                       dsk_chs_t *chs);
100122 int dsk_read_sectors (int drive, unsigned int start_sector,
100123                      void *buffer, unsigned int n_sectors);
100124 int dsk_write_sectors (int drive, unsigned int start_sector,
100125                       void *buffer, unsigned int n_sectors);
100126 size_t dsk_read_bytes (int drive, off_t offset,
100127                       void *buffer, size_t count);
100128 size_t dsk_write_bytes (int drive, off_t offset,
100129                        void *buffer, size_t count);
100130 //-----
100131
100132 #endif

```

```

103006 ;-----
103007 __in_8:
103008     enter #2, #0 ; 1 local variable.
103009     pushf
103010     cli
103011     pusha
103012     mov dx, 4[bp] ; 1st arg (port number).
103013     in ax, dx
103014     mov ah, #0
103015     mov -2[bp], ax ; Save AX.
103016     popa
103017     popf
103018     mov ax, -2[bp] ; AX is the function return value.
103019     leave
103020     ret

```

kernel/ibm_i86/_int10_00.s

Si veda la sezione u0.4.

```

104001 .global __int10_00
104002 ;-----
104003 .text
104004 ;-----
104005 ; INT 0x10 - video - set video mode
104006 ;
104007 ; AH = 0x00
104008 ; AL = desired video mode:
104009 ; 0x00 = text 40x25 pages 8
104010 ; 0x01 = text 40x25 pages 8
104011 ; 0x02 = text 80x25 pages 4
104012 ; 0x03 = text 80x25 pages 4
104013 ; 0x07 = text 80x25 pages 4?
104014 ;
104015 ; Specify the display mode for the currently active display adapter.
104016 .align 2
104017 __int10_00:
104018     enter #0, #0 ; No local variables.
104019     pushf
104020     cli
104021     pusha
104022     mov ah, #0x00
104023     mov al, 4[bp] ; 1st arg (video mode).
104024     int #0x10
104025     popa
104026     popf
104027     leave
104028     ret

```

kernel/ibm_i86/_cli.s

Si veda la sezione u0.4.

```

101001 .global __cli
101002 ;-----
101003 .text
101004 ;-----
101005 ; Clear interrupt flag.
101006 ;-----
101007 .align 2
101008 __cli:
101009     cli
101010     ret

```

kernel/ibm_i86/_int10_02.s

Si veda la sezione u0.4.

```

105001 .global __int10_02
105002 ;-----
105003 .text
105004 ;-----
105005 ; INT 0x10 - video - set cursor position
105006 ;
105007 ; AH = 0x02
105008 ; BH = page number:
105009 ; 0-7 in modes 0 and 1
105010 ; 0-3 in modes 2 and 3
105011 ; DH = row (0x00 is top)
105012 ; DL = column (0x00 is left)
105013 ;-----
105014 .align 2
105015 __int10_02:
105016     enter #0, #0 ; No local variables.
105017     pushf
105018     cli
105019     pusha
105020     mov ah, #0x02
105021     mov bh, #0x00
105022     mov dx, 4[bp] ; 1st arg (page).
105023     mov dx, 6[bp] ; 2nd arg (pos).
105024     int #0x10
105025     popa
105026     popf
105027     leave
105028     ret

```

kernel/ibm_i86/_in_16.s

Si veda la sezione u0.4.

```

102001 .global __in_16
102002 ;-----
102003 .text
102004 ;-----
102005 ; Port input word.
102006 ;-----
102007 __in_16:
102008     enter #2, #0 ; 1 local variable.
102009     pushf
102010     cli
102011     pusha
102012     mov dx, 4[bp] ; 1st arg (port number).
102013     in ax, dx
102014     mov -2[bp], ax ; Save AX.
102015     popa
102016     popf
102017     mov ax, -2[bp] ; AX is the function return value.
102018     leave
102019     ret

```

kernel/ibm_i86/_int10_05.s

Si veda la sezione u0.4.

```

106001 .global __int10_05
106002 ;-----
106003 .text
106004 ;-----
106005 ; INT 0x10 - video - select active display page
106006 ;
106007 ; AH = 0x05
106008 ; AL = new page number (0x00 is the first)
106009 ;-----
106010 .align 2
106011 __int10_05:
106012     enter #0, #0 ; No local variables.
106013     pushf
106014     cli

```

kernel/ibm_i86/_in_8.s

Si veda la sezione u0.4.

```

103001 .global __in_8
103002 ;-----
103003 .text
103004 ;-----
103005 ; Port input byte.

```

```

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

```

kernel/ibm_i86/_int12.s

<<

Si veda la sezione u0.4.

```

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

```

kernel/ibm_i86/_int13_00.s

<<

Si veda la sezione u0.4.

```

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

```

kernel/ibm_i86/_int13_02.s

<<

Si veda la sezione u0.4.

```

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

```

```

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

```

kernel/ibm_i86/_int13_03.s

<<

Si veda la sezione u0.4.

```

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

```

kernel/ibm_i86/_int16_00.s

<<

Si veda la sezione u0.4.

```

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

```

```

110008 ; AH = BIOS scan code
110009 ; AL = ASCII character
110010 ;-----
110011 .align 2
110012 __int16_00:
110013 ; enter #2, #0 ; 1 local variable.
110014 pushf
110015 cli
110016 pusha
110017 mov ah, #0x00
110018 int #0x16
110019 mov -2[bp], ax ; Save AX.
110020 popa
110021 popf
110022 mov ax, -2[bp] ; AX is the function return value.
110023 leave
110024 ret

```

kernel/ibm_i86/_int16_01.s

« Si veda la sezione u0.4.

```

112001 .global __int16_01
112002 ;-----
112003 .text
112004 ;-----
112005 ; INT 0x16 - keyboard - check for keystroke
112006 ; AH = 0x01
112007 ; Return:
112008 ; ZF set if no keystroke available
112009 ; ZF clear if keystroke available
112010 ; AH = BIOS scan code
112011 ; AL = ASCII character
112012 ;
112013 ; If a keystroke is present, it is not removed from the keyboard buffer.
112014 ;-----
112015 .align 2
112016 __int16_01:
112017 ; enter #2, #0 ; 1 local variable.
112018 pushf
112019 cli
112020 pusha
112021 mov ah, #0x01
112022 int #0x16
112023 jnz __int16_01_ok
112024 mov ax, #0 ; Put zero to AX, if no keystroke is available.
112025 __int16_01_ok:
112026 mov -2[bp], ax ; Save AX.
112027 popa
112028 popf
112029 mov ax, -2[bp] ; AX is the function return value.
112030 leave
112031 ret

```

kernel/ibm_i86/_int16_02.s

« Si veda la sezione u0.4.

```

113001 .global __int16_02
113002 ;-----
113003 .text
113004 ;-----
113005 ; INT 0x16 - keyboard - get shift flags
113006 ; AH = 0x02
113007 ; Return:
113008 ; AL = shift flags
113009 ; AH might be destroyed
113010 ;
113011 ; bit 7 Insert active
113012 ; bit 6 CapsLock active
113013 ; bit 5 NumLock active
113014 ; bit 4 ScrollLock active
113015 ; bit 3 Alt key pressed
113016 ; bit 2 Ctrl key pressed
113017 ; bit 1 left shift key pressed
113018 ; bit 0 right shift key pressed
113019 ;-----
113020 .align 2
113021 __int16_02:
113022 ; enter #2, #0 ; 1 local variable.
113023 pushf
113024 cli
113025 pusha
113026 mov ah, #0x02
113027 int #0x16
113028 mov ah, #0 ; Reset AH.
113029 mov -2[bp], ax ; Save AX.
113030 popa
113031 popf
113032 mov ax, -2[bp] ; AX is the function return value.
113033 leave
113034 ret

```

kernel/ibm_i86/_out_16.s

« Si veda la sezione u0.4.

```

114001 .global __out_16
114002 ;-----
114003 .text
114004 ;-----
114005 ; Port output word.
114006 ;-----
114007 .align 2
114008 __out_16:
114009 ; enter #0, #0 ; No local variables.
114010 pushf
114011 cli
114012 pusha
114013 mov dx, 4[bp] ; 1st arg (port number).
114014 mov ax, 6[bp] ; 2nd arg (value).
114015 out dx, ax
114016 popa
114017 popf
114018 leave
114019 ret

```

kernel/ibm_i86/_out_8.s

« Si veda la sezione u0.4.

```

115001 .global __out_8
115002 ;-----
115003 .text
115004 ;-----
115005 ; Port output byte.
115006 ;-----
115007 .align 2
115008 __out_8:
115009 ; enter #0, #0 ; No local variables.
115010 pushf
115011 cli
115012 pusha
115013 mov dx, 4[bp] ; 1st arg (port number).
115014 mov ax, 6[bp] ; 2nd arg (value).
115015 out dx, al
115016 popa
115017 popf
115018 leave
115019 ret

```

kernel/ibm_i86/_ram_copy.s

« Si veda la sezione u0.4.

```

116001 .global __ram_copy
116002 ;-----
116003 .text
116004 ;-----
116005 ; Copy some bytes between segments.
116006 ;-----
116007 .align 2
116008 __ram_copy:
116009 ; enter #0, #0 ; No local variables.
116010 pushf
116011 cli
116012 pusha
116013 mov ax, 4[bp] ; 1st arg (source segment).
116014 mov si, 6[bp] ; 2nd arg (source offset).
116015 mov bx, 8[bp] ; 3rd arg (destination segment).
116016 mov di, 10[bp] ; 4th arg (destination offset).
116017 mov cx, 12[bp] ; 5th arg (size).
116018 mov dx, ds ; save the data segment.
116019 mov ds, ax ; set DS.
116020 mov es, bx ; set ES.
116021 rep
116022 movsb ; Copy the array of bytes.
116023 mov ds, dx ; Restore the data segment.
116024 mov es, dx ; Restore or fix the extra segment.
116025 popa
116026 popf
116027 leave
116028 ret

```

kernel/ibm_i86/_sti.s

« Si veda la sezione u0.4.

```

117001 .global __sti
117002 ;-----
117003 .text
117004 ;-----
117005 ; Set interrupt flag.
117006 ;-----
117007 .align 2
117008 __sti:
117009 sti
117010 ret

```


<

Si veda la sezione u0.4.

```

180001 #include <kernel/ibm_i86.h>
180002 #include <kernel/k_libc.h>
180003 #include <sys/os16.h>
180004 #include <sys/types.h>
180005 #include <stdint.h>
180006 //-----
180007 int
180008 con_char_read (void)
180009 {
180010     int c;
180011     c = int16_01 ();
180012     //
180013     // Remove special keys that are not used: they have zero in the low
180014     // 8 bits, and something in the upper 8 bits.
180015     //
180016     if ((c & 0xFF00) && !(c & 0x00FF))
180017     {
180018         int16_00 (); // Remove from buffer and return zero:
180019         return (0); // no key.
180020     }
180021     //
180022     // A common key was pressed: filter only che low 8 bits.
180023     //
180024     c = c & 0x00FF;
180025     if (c == 0)
180026     {
180027         return (c); // There is no key.
180028     }
180029     if (c == '\r') // Convert 'CR' to 'LF'.
180030     {
180031         c = '\n';
180032     }
180033     int16_00 (); // Remove the key from buffer and return.
180034     return (c);
180035 }

```

<

Si veda la sezione u0.4.

```

190001 #include <kernel/ibm_i86.h>
190002 #include <kernel/k_libc.h>
190003 #include <sys/os16.h>
190004 #include <sys/types.h>
190005 #include <stdint.h>
190006 //-----
190007 int
190008 con_char_ready (void)
190009 {
190010     int c;
190011     c = int16_01 ();
190012     //
190013     // Remove special keys that are not used: they have zero in the low
190014     // 8 bits, and something in the upper 8 bits.
190015     //
190016     if ((c & 0xFF00) && !(c & 0x00FF))
190017     {
190018         int16_00 (); // Remove from buffer and return zero:
190019         return (0); // no key.
190020     }
190021     //
190022     // A common key was pressed: filter only che low 8 bits.
190023     //
190024     c = c & 0x00FF;
190025     return (c);
190026 }

```

<

Si veda la sezione u0.4.

```

120001 #include <kernel/ibm_i86.h>
120002 #include <kernel/k_libc.h>
120003 #include <sys/os16.h>
120004 #include <sys/types.h>
120005 #include <stdint.h>
120006 //-----
120007 int
120008 con_char_wait (void)
120009 {
120010     int c;
120011     c = int16_00 ();
120012     c = c & 0x00FF;
120013     if (c == '\r')
120014     {
120015         c = '\n';
120016     }
120017     return (c);
120018 }

```

>

Si veda la sezione u0.4.

```

121001 #include <kernel/ibm_i86.h>
121002 #include <kernel/k_libc.h>
121003 #include <sys/os16.h>
121004 #include <sys/types.h>
121005 #include <stdint.h>
121006 //-----
121007 void
121008 con_init (void)
121009 {
121010     int page;
121011     //
121012     int10_00 (IBM_I86_VIDEO_MODE);
121013     int10_05 (0);
121014     //
121015     for (page = 0; page < IBM_I86_VIDEO_PAGES; page++)
121016     {
121017         con_putc (page, '\n');
121018     }
121019 }

```

>

Si veda la sezione u0.4.

```

122001 #include <kernel/ibm_i86.h>
122002 #include <kernel/k_libc.h>
122003 #include <sys/os16.h>
122004 #include <sys/types.h>
122005 #include <stdint.h>
122006 //-----
122007 void
122008 con_putc (int console, int c)
122009 {
122010     static int cursor[IBM_I86_VIDEO_PAGES];
122011     static addr_t address[] = {IBM_I86_VIDEO_ADDRESS};
122012     addr_t address_destination;
122013     size_t size_screen;
122014     size_t size_row;
122015     uint16_t cell;
122016     uint16_t attribute = 0x0700;
122017     int cursor_row;
122018     int cursor_column;
122019     int cursor_combined;
122020
122021     if (console < 0 || console >= IBM_I86_VIDEO_PAGES)
122022     {
122023         //
122024         // No such console.
122025         //
122026         return;
122027     }
122028     //
122029     // Calculate sizes.
122030     //
122031     size_row = IBM_I86_VIDEO_COLUMNS;
122032     size_screen = size_row * IBM_I86_VIDEO_ROWS;
122033     //
122034     // See if it is a special character, or if the cursor position
122035     // requires a scroll up.
122036     //
122037     if (c == '\n')
122038     {
122039         con_scroll (console);
122040         cursor[console] = (size_screen - size_row);
122041     }
122042     else if (c == '\b')
122043     {
122044         cursor[console]--;
122045         if (cursor[console] < 0)
122046         {
122047             cursor[console] = 0;
122048         }
122049     }
122050     else if (cursor[console] == (size_screen - 1))
122051     {
122052         //
122053         // Scroll up.
122054         //
122055         con_scroll (console);
122056         //
122057         cursor[console] -= size_row;
122058     }
122059     //
122060     // If it is not a control character, print it.
122061     //
122062     if (c != '\n' && c != '\b')
122063     {
122064         //
122065         // Write the character.
122066         //
122067         address_destination = address[console];
122068         address_destination += (cursor[console] * 2);
122069         cell = (attribute | (c & 0x00FF));
122070         //
122071         mem_write (address_destination, &cell, sizeof (uint16_t));
122072         //
122073         // and an extra space after it (to be able to show the cursor).

```

```

1230074 //
1230075 cell = (attribute | ' ');
1230076 address_destination += 2;
1230077 mem_write (address_destination, &cell, sizeof (uint16_t));
1230078 //
1230079 //
1230080 //
1230081 cursor[console]++;
1230082 }
1230083 //
1230084 // Update the cursor position on screen.
1230085 //
1230086 cursor_row = cursor[console] / size_row;
1230087 cursor_column = cursor[console] % size_row;
1230088 cursor_combined = (cursor_row << 8) | cursor_column;
1230089 //
1230090 // Set cursor position.
1230091 //
1230092 int10_02 (console, cursor_combined);
1230093 }

```

kernel/ibm_i86/con_scroll.c

Si veda la sezione u0.4.

```

1230091 #include <kernel/ibm_i86.h>
1230092 #include <kernel/k_libc.h>
1230093 #include <sys/os16.h>
1230094 #include <sys/types.h>
1230095 #include <stdint.h>
1230096 //-----
1230097 void
1230098 con_scroll (int console)
1230099 {
1230100     static addr_t address[] = {IBM_I86_VIDEO_ADDRESS};
1230101     addr_t address_source;
1230102     addr_t address_destination;
1230103     size_t size_screen;
1230104     size_t size_row;
1230105     static uint16_t empty_line[IBM_I86_VIDEO_COLUMNS];
1230106     //
1230107     size_row = IBM_I86_VIDEO_COLUMNS;
1230108     size_screen = size_row * IBM_I86_VIDEO_ROWS;
1230109     //
1230110     // Scroll up.
1230111     //
1230112     address_source = address[console];
1230113     address_source += size_row * 2;
1230114     address_destination = address[console];
1230115     //
1230116     mem_copy (address_source, address_destination,
1230117              (size_t) ((size_screen - size_row) * 2));
1230118     //
1230119     address_destination = address[console];
1230120     address_destination += ((size_screen - size_row) * 2);
1230121     //
1230122     mem_write (address_destination, &empty_line,
1230123              (size_t) (size_row * 2));
1230124 }

```

kernel/ibm_i86/con_select.c

Si veda la sezione u0.4.

```

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

```

kernel/ibm_i86/dsk_read_bytes.c

Si veda la sezione u0.4.

```

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

```

```

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

```

kernel/ibm_i86/dsk_read_sectors.c

Si veda la sezione u0.4.

```

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

```

kernel/ibm_i86/dsk_reset.c

<<

Si veda la sezione u0.4.

```

1270001 #include <kernel/ibm_i86.h>
1270002 #include <kernel/k_libc.h>
1270003 #include <sys/os16.h>
1270004 #include <sys/types.h>
1270005 #include <stdint.h>
1270006 //-----
1270007 int
1270008 dsk_reset (int drive)
1270009 {
1270010     unsigned int status;
1270011     status = int13_00 (dsk_table[drive].bios_drive);
1270012     if (status == 0)
1270013     {
1270014         return (0);
1270015     }
1270016     else
1270017     {
1270018         return (-1);
1270019     }
1270020 }

```

kernel/ibm_i86/dsk_sector_to_chs.c

<<

Si veda la sezione u0.4.

```

1280001 #include <kernel/ibm_i86.h>
1280002 #include <kernel/k_libc.h>
1280003 #include <sys/os16.h>
1280004 #include <sys/types.h>
1280005 #include <stdint.h>
1280006 //-----
1280007 void
1280008 dsk_sector_to_chs (int drive, unsigned int sector, dsk_chs_t *chs)
1280009 {
1280010     unsigned int sectors_per_cylinder;
1280011     sectors_per_cylinder = (dsk_table[drive].sectors
1280012         * dsk_table[drive].heads);
1280013     chs->cylinder = sector / sectors_per_cylinder;
1280014     sector = sector % sectors_per_cylinder;
1280015     chs->head = sector / dsk_table[drive].sectors;
1280016     sector = sector % dsk_table[drive].sectors;
1280017     chs->sector = sector + 1;
1280018 }

```

kernel/ibm_i86/dsk_setup.c

<<

Si veda la sezione u0.4.

```

1290001 #include <kernel/ibm_i86.h>
1290002 //-----
1290003 void
1290004 dsk_setup (void)
1290005 {
1290006     dsk_reset (0);
1290007     dsk_table[0].bios_drive = 0x00; // A: 1440 Kibyte floppy disk.
1290008     dsk_table[0].cylinders = 80;
1290009     dsk_table[0].heads = 2;
1290010     dsk_table[0].sectors = 18;
1290011     dsk_table[0].retry = 3;
1290012     dsk_reset (1);
1290013     dsk_table[1].bios_drive = 0x01; // B: 1440 Kibyte floppy disk.
1290014     dsk_table[1].cylinders = 80;
1290015     dsk_table[1].heads = 2;
1290016     dsk_table[1].sectors = 18;
1290017     dsk_table[1].retry = 3;
1290018     dsk_reset (2);
1290019     dsk_table[2].bios_drive = 0x80; // C: like a 2880 Kibyte floppy disk.
1290020     dsk_table[2].cylinders = 80;
1290021     dsk_table[2].heads = 2;
1290022     dsk_table[2].sectors = 36;
1290023     dsk_table[2].retry = 3;
1290024     dsk_reset (3);
1290025     dsk_table[3].bios_drive = 0x81; // D: like a 2880 Kibyte floppy disk.
1290026     dsk_table[3].cylinders = 80;
1290027     dsk_table[3].heads = 2;
1290028     dsk_table[3].sectors = 36;
1290029     dsk_table[3].retry = 3;
1290030 }

```

kernel/ibm_i86/dsk_table.c

<<

Si veda la sezione u0.4.

```

1300001 #include <kernel/ibm_i86.h>
1300002 //-----
1300003 dsk_t dsk_table[DSK_MAX];

```

kernel/ibm_i86/dsk_write_bytes.c

<<

Si veda la sezione u0.4.

```

1310001 #include <kernel/ibm_i86.h>
1310002 #include <kernel/k_libc.h>
1310003 #include <sys/os16.h>

```

1680

```

1310004 #include <sys/types.h>
1310005 #include <stdint.h>
1310006 //-----
1310007 size_t
1310008 dsk_write_bytes (int drive, off_t offset, void *buffer, size_t count)
1310009 {
1310010     unsigned char *data_buffer = (unsigned char *) buffer;
1310011     int status;
1310012     unsigned int sector;
1310013     unsigned char sector_buffer[DSK_SECTOR_SIZE];
1310014     int i;
1310015     int j = 0;
1310016     size_t k = 0;
1310017     size_t m = 0;
1310018
1310019     sector = offset / DSK_SECTOR_SIZE;
1310020     i = offset % DSK_SECTOR_SIZE;
1310021     status = dsk_read_sectors (drive, sector, sector_buffer, 1);
1310022
1310023     if (status != 0)
1310024     {
1310025         return ((size_t) 0);
1310026     }
1310027
1310028     while (count)
1310029     {
1310030         m = k;
1310031         for (; i < DSK_SECTOR_SIZE && count > 0;
1310032             i++, j++, k++, count--, offset++)
1310033         {
1310034             sector_buffer[i] = data_buffer[j];
1310035         }
1310036         status = dsk_write_sectors (drive, sector, sector_buffer, 1);
1310037         if (status != 0)
1310038         {
1310039             return (m);
1310040         }
1310041         if (count)
1310042         {
1310043             sector = offset / DSK_SECTOR_SIZE;
1310044             i = offset % DSK_SECTOR_SIZE;
1310045             status = dsk_read_sectors (drive, sector, sector_buffer, 1);
1310046             if (status != 0)
1310047             {
1310048                 return (m);
1310049             }
1310050         }
1310051     }
1310052     return (k);
1310053 }
1310054
1310055

```

kernel/ibm_i86/dsk_write_sectors.c

<<

Si veda la sezione u0.4.

```

1320001 #include <kernel/ibm_i86.h>
1320002 #include <kernel/k_libc.h>
1320003 #include <sys/os16.h>
1320004 #include <sys/types.h>
1320005 #include <stdint.h>
1320006 //-----
1320007 int
1320008 dsk_write_sectors (int drive, unsigned int start_sector, void *buffer,
1320009     unsigned int n_sectors)
1320010 {
1320011     int status;
1320012     unsigned int retry;
1320013     unsigned int remaining;
1320014     dsk_chs_t chs;
1320015     dsk_sector_to_chs (drive, start_sector, &chs);
1320016     remaining = dsk_table[drive].sectors - chs.sector + 1;
1320017     if (remaining < n_sectors)
1320018     {
1320019         status = dsk_write_sectors (drive, start_sector,
1320020             buffer, remaining);
1320021         if (status == 0)
1320022         {
1320023             status = dsk_write_sectors (drive,
1320024                 start_sector + remaining,
1320025                 buffer, n_sectors - remaining);
1320026         }
1320027         return (status);
1320028     }
1320029     else
1320030     {
1320031         for (retry = 0; retry < dsk_table[drive].retry; retry++)
1320032         {
1320033             status = int13_03 (dsk_table[drive].bios_drive, n_sectors,
1320034                 chs.cylinder, chs.head, chs.sector,
1320035                 buffer);
1320036             status = status & 0x00F0;
1320037             if (status == 0)
1320038             {
1320039                 break;
1320040             }
1320041             else
1320042             {
1320043                 dsk_reset (drive);
1320044             }

```

1681

```

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

```

kernel/ibm_i86/irq_off.c

« Si veda la sezione u0.4.

```

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

```

kernel/ibm_i86/irq_on.c

« Si veda la sezione u0.4.

```

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

```

os16: «kernel/k_libc.h»

« Si veda la sezione u0.5.

```

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

```

```

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

```

kernel/k_libc/k_clock.c

« Si veda la sezione u0.5.

```

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

```

kernel/k_libc/k_close.c

« Si veda la sezione u0.5.

```

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

```

kernel/k_libc/k_exit.s

« Si veda la sezione u0.5.

```

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

```

kernel/k_libc/k_kill.c

« Si veda la sezione u0.5.

```

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

```

kernel/k_libc/k_open.c

« Si veda la sezione u0.5.

```

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

```

kernel/k_libc/k_perror.c

<<

Si veda la sezione u0.5.

```

1440001 #include <kernel/k_libc.h>
1440002 #include <errno.h>
1440003 //-----
1440004 void
1440005 k_perror (const char *s)
1440006 {
1440007     //
1440008     // If errno is zero, there is nothing to show.
1440009     //
1440010     if (errno == 0)
1440011     {
1440012         return;
1440013     }
1440014     //
1440015     // Show the string if there is one.
1440016     //
1440017     if (s != NULL && strlen (s) > 0)
1440018     {
1440019         k_printf ("%s: ", s);
1440020     }
1440021     //
1440022     // Show the translated error.
1440023     //
1440024     if (errfn[0] != 0 && errln != 0)
1440025     {
1440026         k_printf ("%s:%u:%i %s\n",
1440027                 errfn, errln, errno, strerror (errno));
1440028     }
1440029     else
1440030     {
1440031         k_printf ("%i %s\n", errno, strerror (errno));
1440032     }
1440033 }

```

kernel/k_libc/k_printf.c

<<

Si veda la sezione u0.5.

```

1420001 #include <stdarg.h>
1420002 #include <kernel/k_libc.h>
1420003 //-----
1420004 int
1420005 k_printf (const char *restrict format, ...)
1420006 {
1420007     va_list ap;
1420008     va_start (ap, format);
1420009     return k_vprintf (format, ap);
1420010 }

```

kernel/k_libc/k_puts.c

<<

Si veda la sezione u0.5.

```

1430001 #include <sys/os16.h>
1430002 #include <kernel/devices.h>
1430003 #include <kernel/k_libc.h>
1430004 #include <string.h>
1430005 //-----
1430006 int
1430007 k_puts (const char *string)
1430008 {
1430009     dev_io ((pid_t) 0, DEV_TTY, DEV_WRITE, (off_t) 0, string,
1430010           strlen (string), NULL);
1430011     dev_io ((pid_t) 0, DEV_TTY, DEV_WRITE, (off_t) 0, "\n", 1, NULL);
1430012     return 1;
1430013 }

```

kernel/k_libc/k_read.c

<<

Si veda la sezione u0.5.

```

1440001 #include <kernel/k_libc.h>
1440002 #include <kernel/fs.h>
1440003 //-----
1440004 ssize_t
1440005 k_read (int fdn, void *buffer, size_t count)
1440006 {
1440007     int eof;
1440008     ssize_t size;
1440009     //
1440010     eof = 0;
1440011     //
1440012     while (1)
1440013     {
1440014         size += fd_read ((pid_t) 0, fdn, buffer, count, &eof);
1440015         if (size != 0 || eof)
1440016         {
1440017             break;
1440018         }
1440019     }
1440020     return (size);
1440021 }

```

kernel/k_libc/k_stime.c

<<

Si veda la sezione u0.5.

```

1450001 #include <kernel/k_libc.h>
1450002 //-----
1450003 extern time_t _clock_seconds; // uint32_t
1450004 //-----
1450005 int
1450006 k_stime (time_t *timer)
1450007 {
1450008     _clock_seconds = *timer;
1450009     return (0);
1450010 }

```

kernel/k_libc/k_time.c

<<

Si veda la sezione u0.5.

```

1460001 #include <kernel/k_libc.h>
1460002 #include <stddef.h>
1460003 //-----
1460004 extern time_t _clock_seconds; // uint32_t
1460005 //-----
1460006 time_t
1460007 k_time (time_t *timer)
1460008 {
1460009     if (timer != NULL)
1460010     {
1460011         *timer = _clock_seconds;
1460012     }
1460013     return (_clock_seconds);
1460014 }

```

kernel/k_libc/k_vprintf.c

<<

Si veda la sezione u0.5.

```

1470001 #include <sys/os16.h>
1470002 #include <kernel/devices.h>
1470003 #include <stdarg.h>
1470004 #include <kernel/k_libc.h>
1470005 #include <string.h>
1470006 //-----
1470007 int
1470008 k_vprintf (const char *restrict format, va_list arg)
1470009 {
1470010     char string[BUFSIZ];
1470011     int ret;
1470012     string[0] = 0;
1470013     ret = k_vsprintf (string, format, arg);
1470014     dev_io ((pid_t) 0, DEV_CONSOLE, DEV_WRITE, (off_t) 0, string,
1470015           strlen (string), NULL);
1470016     return ret;
1470017 }

```

kernel/k_libc/k_vsprintf.c

<<

Si veda la sezione u0.5.

```

1480001 #include <stdarg.h>
1480002 #include <kernel/k_libc.h>
1480003 #include <stdio.h>
1480004 //-----
1480005 int
1480006 k_vsprintf (char *restrict string, const char *restrict format,
1480007            va_list arg)
1480008 {
1480009     int ret;
1480010     ret = vsprintf (string, BUFSIZ, format, arg);
1480011     return ret;
1480012 }

```

os16: «kernel/main.h»

<<

Si veda la sezione u0.6.

```

1490001 #ifndef _KERNEL_MAIN_H
1490002 #define _KERNEL_MAIN_H 1
1490003
1490004 #include <sys/types.h>
1490005
1490006 void menu (void);
1490007 pid_t run (char *path, char *argv[], char *envp[]);
1490008 int main (int argc, char *argv[], char *envp[]);
1490009
1490010 #endif

```

kernel/main/build.h

<<

Si veda la sezione u0.6.

```

1500001 #define BUILD_DATE "2010.07.26 16:33:58"

```

Si veda la sezione u0.2.

```

150001 .extern _main
150002 .global __mkargv
150003 ;-----
150004 ; Please note that, at the beginning, all the segment registers are
150005 ; the same: CS==DS==ES==SS. But the data segments (DS, ES, SS) are meant
150006 ; to be separated from the code, and they starts at: CS + __segoff.
150007 ; The label "__segoff" is replaced by the linker with a constant value
150008 ; (inside the code) with the segment offset to add to CS: this way
150009 ; it is possible to find where DS and ES should start.
150010 ;-----
150011 ; The following statement says that the code will start at "startup"
150012 ; label.
150013 ;-----
150014 entry startup
150015 ;-----
150016 .text
150017 ;-----
150018 startup:
150019 ;
150020 ; Jump after initial data.
150021 ;
150022 jmp startup_code
150023 ;
150024 filler:
150025 ;
150026 ; After four bytes, from the start, there is the
150027 ; magic number and other data.
150028 ;
150029 .space (0x0004 - (filler - startup))
150030 magic:
150031 ;
150032 ; Add to "/etc/magic" the following line:
150033 ;
150034 ; 4 quad 0x6B65726566731316 os16 kernel
150035 ;
150036 .data4 0x6F731316 ; os16
150037 .data4 0x6B657265 ; kern
150038 ;
150039 segoff: ;
150040 .data2 __segoff ; These values, for a kernel image,
150041 etext: ; are not used.
150042 .data2 __etext ;
150043 edata: ;
150044 .data2 __edata ;
150045 ebas: ;
150046 .data2 __end ;
150047 stack_size: ;
150048 .data2 0x0000 ;
150049 ;
150050 ; At the next label, the work begins.
150051 ;
150052 .align 2
150053 startup_code:
150054 ;
150055 ; Check where we are. If we are at segment 0x1000,
150056 ; then move to 0x3000.
150057 ;
150058 mov cx, cs
150059 xor cx, #0x1000
150060 jcxz move_code_from_0x1000_to_0x3000
150061 ;
150062 ; Check where we are. If we are at segment 0x3000,
150063 ; then move to 0x0050, preserving the IVT and the BDA.
150064 ;
150065 mov cx, cs
150066 xor cx, #0x3000
150067 jcxz move_code_from_0x3000_to_0x0050
150068 ;
150069 ; Check where we are. If we are at segment 0x1050,
150070 ; then jump to the main code.
150071 ;
150072 mov cx, cs
150073 xor cx, #0x1050
150074 jcxz main_code
150075 ;
150076 ; Otherwise, just halt.
150077 ;
150078 hlt
150079 jmp startup_code
150080 ;
150081 move_code_from_0x1000_to_0x3000:
150082 ;
150083 cld ; Clear direction flag.
150084 mov ax, #0x3000 ; Set ES as the destination segment.
150085 mov es, ax ;
150086 mov ax, #0x1000 ; Set DS as the source segment.
150087 mov ds, ax ;
150088 ;
150089 mov cx, #0x8000 ; Move 32768 words = 65536 byte (64 Kibyte).
150090 mov si, #0x0000 ; DS:SI == Source pointer
150091 mov di, #0x0000 ; ES:DI == Destination pointer
150092 rep
150093 movsw ; Copy the array of words
150094 ;
150095 mov ax, #0x4000 ; Set ES as the destination segment.
150096 mov es, ax ;
150097 mov ax, #0x2000 ; Set DS as the source segment.
150098 mov ds, ax ;

```

```

150099 ;
150100 mov cx, #0x8000 ; Move 32768 words = 65536 byte (64 Kibyte).
150101 mov si, #0x0000 ; DS:SI == Source pointer
150102 mov di, #0x0000 ; ES:DI == Destination pointer
150103 rep
150104 movsw ; Copy the array of words
150105 ;
150106 jmp far #0x3000:#0x0000 ; Go to the new kernel copy.
150107 ;
150108 move_code_from_0x3000_to_0x0050:
150109 cld ; Clear direction flag.
150110 ;
150111 ; Text (instructions) is moved at segment 0x1050 (address 0x10500).
150112 ;
150113 mov ax, #0x1050 ; Set ES as the destination segment.
150114 mov es, ax ;
150115 mov ax, #0x3000 ; Set DS as the source segment.
150116 mov ds, ax ;
150117 ;
150118 mov cx, #0x8000 ; Move 32768 words = 65536 byte (64 Kibyte).
150119 mov si, #0x0000 ; DS:SI == Source pointer
150120 mov di, #0x0000 ; ES:DI == Destination pointer
150121 rep
150122 movsw ; Copy the array of words
150123 ;
150124 ; Data is moved at segment 0x0050 (address 0x00500), before the
150125 ; text segment.
150126 ;
150127 mov ax, #0x0050 ; Set ES as the destination segment.
150128 mov es, ax ;
150129 mov ax, #0x3000 ; Calculate where is the data segment:
150130 add ax, __segoff ; it is at 0x3000 + __segoff.
150131 mov ds, ax ; Set DS as the source segment.
150132 ;
150133 mov cx, #0x8000 ; Move 32768 words = 65536 byte (64 Kibyte).
150134 mov si, #0x0000 ; DS:SI == Source pointer
150135 mov di, #0x0000 ; ES:DI == Destination pointer
150136 rep
150137 movsw ; Copy the array of words
150138 ;
150139 jmp far #0x1050:#0x0000 ; Go to the new kernel copy.
150140 ;
150141 ;-----
150142 main_code:
150143 ;
150144 ; Fix data segments!
150145 ;
150146 mov ax, #0x0050
150147 mov ds, ax
150148 mov ss, ax
150149 mov es, ax
150150 ;
150151 ; Fix SP at the kernel stack bottom: the effective stack pointer
150152 ; value should be 0x10000, but only 0x0000 can be written. At the
150153 ; first push SP reaches 0xFFFF.
150154 ;
150155 mov sp, #0x0000
150156 ;
150157 ; Reset flags.
150158 ;
150159 push #0
150160 popf
150161 cli
150162 ;
150163 ; Call C main function, after kernel relocation and segments set up.
150164 ;
150165 push #0 ; This zero means NULL (envp[0] == NULL)
150166 push #0 ; This zero means NULL (argv[0] == NULL)
150167 push #0 ; This other zero means no arguments.
150168 call _main
150169 add sp, #2
150170 add sp, #2
150171 add sp, #2
150172 ;
150173 .align 2
150174 halt:
150175 ;
150176 ; It will never come back from the _main() call, but just for extra
150177 ; security, loop forever.
150178 ;
150179 hlt
150180 jmp halt
150181 ;
150182 ;-----
150183 .align 2
150184 __mkargv: ; Symbol '__mkargv' is used by Bcc inside the function
150185 ret ; 'main()' and must be present for a successful
150186 ; compilation.
150187 ;-----
150188 .align 2
150189 .data
150190 ;
150191 ;-----
150192 .align 2
150193 .bss

```

Si veda la sezione u0.6.

```

152001 #include <kernel/main.h>
152002 #include <errno.h>
152003 #include <fcntl.h>
152004 #include <kernel/main/build.h>
152005 #include <kernel/diag.h>
152006 #include <kernel/fs.h>
152007 #include <kernel/imm_i86.h>
152008 #include <kernel/k_libc.h>
152009 #include <kernel/proc.h>
152010 #include <libgen.h>
152011 #include <stdlib.h>
152012 #include <sys/osl6.h>
152013 #include <sys/stat.h>
152014 #include <sys/types.h>
152015 #include <unistd.h>
152016 //-----
152017 int
152018 main (int argc, char *argv[], char *envp[])
152019 {
152020     unsigned int key;
152021     pid_t pid;
152022     char *exec_argv[2];
152023     int status;
152024     int exit;
152025     //
152026     // Reset video and select the initial console.
152027     //
152028     tty_init ();
152029     //
152030     // Show compilation date and time.
152031     //
152032     k_printf ("osl6 build %s ram %i Kibyte\n", BUILD_DATE, int12 ());
152033     //
152034     // Set up disk management.
152035     //
152036     disk_setup ();
152037     //
152038     // Clear heap for diagnosis.
152039     //
152040     heap_clear ();
152041     //
152042     // Set up process management. Process set up need the file system
152043     // root directory already available.
152044     //
152045     proc_init ();
152046     //
152047     // The kernel will run interactively.
152048     //
152049     menu ();
152050     //
152051     for (exit = 0; exit == 0;)
152052     {
152053         //
152054         // While in kernel code, timer interrupt don't start the
152055         // scheduler. The kernel must leave control to the scheduler
152056         // via a null system call.
152057         //
152058         sys (SYS_0, NULL, 0);
152059         //
152060         // Back to work: read the keyboard from the TTY device.
152061         //
152062         dev_io ((pid_t) 0, DEV_TTY, DEV_READ, 0L, &key, 1, NULL);
152063         //
152064         // Depending on the key, do something.
152065         //
152066         if (key == 0)
152067         {
152068             //
152069             // No key is ready in the buffer keyboard.
152070             //
152071             continue;
152072         }
152073         else
152074         {
152075             //
152076             // Move back the cursor, so that next print will overwrite
152077             // it.
152078             //
152079             k_printf ("\b");
152080         }
152081         //
152082         // A key was pressed: start to check what it was.
152083         //
152084         switch (key)
152085         {
152086             case 'h':
152087                 menu ();
152088                 break;
152089             case 'l':
152090                 k_kill ((pid_t) 1, SIGKILL); // init
152091                 break;
152092             case '2':
152093             case '3':
152094             case '4':
152095             case '5':
152096             case '6':
152097             case '7':
152098             case '8':

```

1688

```

152099         case '9':
152100             k_kill ((pid_t) (key - '0'), SIGTERM); // others
152101             break;
152102         case 'A':
152103         case 'B':
152104         case 'C':
152105         case 'D':
152106         case 'E':
152107         case 'F':
152108             k_kill ((pid_t) (key - 'A' + 10), SIGTERM); // others
152109             break;
152110         case 'a':
152111             run ("/bin/aaa", NULL, NULL);
152112             break;
152113         case 'b':
152114             run ("/bin/bbb", NULL, NULL);
152115             break;
152116         case 'c':
152117             run ("/bin/ccc", NULL, NULL);
152118             break;
152119         case 'f':
152120             print_file_list ();
152121             break;
152122         case 'm':
152123             status = path_mount ((uid_t) 0, "/dev/dsk1", "/usr",
152124                                 MOUNT_DEFAULT);
152125             if (status < 0)
152126             {
152127                 k_perror (NULL);
152128             }
152129             break;
152130         case 'M':
152131             status = path_umount ((uid_t) 0, "/usr");
152132             if (status < 0)
152133             {
152134                 k_perror (NULL);
152135             }
152136             break;
152137         case 'n':
152138             print_inode_list ();
152139             break;
152140         case 'N':
152141             print_inode_zones_list ();
152142             break;
152143         case 'l':
152144             k_kill ((pid_t) 1, SIGCHLD);
152145             break;
152146         case 'p':
152147             k_printf ("\n");
152148             print_proc_list ();
152149             print_segments ();
152150             k_printf (" ");
152151             print_kmem ();
152152             k_printf (" ");
152153             print_time ();
152154             k_printf ("\n");
152155             print_mb_map ();
152156             k_printf ("\n");
152157             break;
152158         case 'x':
152159             exit = 1;
152160             break;
152161         case 'q':
152162             k_printf ("System halted!\n");
152163             return (0);
152164             break;
152165     }
152166     }
152167     //
152168     // Load init.
152169     //
152170     exec_argv[0] = "/bin/init";
152171     exec_argv[1] = NULL;
152172     pid = run ("/bin/init", exec_argv, NULL);
152173     //
152174     // Just sleep.
152175     //
152176     while (1)
152177     {
152178         sys (SYS_0, NULL, 0);
152179     }
152180     //
152181     k_printf ("System halted!\n");
152182     return (0);
152183 }

```

kernel/main/menu.c

Si veda la sezione u0.6.

```

153001 #include <kernel/main.h>
153002 #include <kernel/k_libc.h>
153003 //-----
153004 void
153005 menu (void)
153006 {
153007     k_printf (
153008         "-----\n"
153009         "| [h]   show this menu                |\n"
153010         "| [p]   process status and memory map  |\n"
153011         "| [1]..[9] kill process 1 to 9        |\n"

```

1689

```

1530012 * [A]..[F] kill process 10 to 15 |\n*
1530013 * [l] send SIGCHLD to process 1 |\n*
1530014 * [a]..[c] run programs '/bin/aaa' to '/bin/ccc' in parallel |\n*
1530015 * [f] system file status |\n*
1530016 * [n], [N] list of active inodes |\n*
1530017 * [m], [M] mount/umount '/dev/dsk1' at '/usr/' |\n*
1530018 * [x] exit interaction with kernel and start '/bin/init' |\n*
1530019 * [q] quit kernel |\n*
1530020 *-----|\n*
1530021 };
1530022
1530023 }

```

kernel/main/run.c

Si veda la sezione u0.6.

```

1540001 #include <kernel/main.h>
1540002 #include <kernel/proc.h>
1540003 #include <kernel/k_libc.h>
1540004 #include <unistd.h>
1540005 //-----
1540006 pid_t
1540007 run (char *path, char *argv[], char *envp[])
1540008 {
1540009     pid_t pid;
1540010     //
1540011     pid = fork ();
1540012     if (pid == -1)
1540013     {
1540014         k_perror (NULL);
1540015     }
1540016     else if (pid == 0)
1540017     {
1540018         execve (path, argv, envp);
1540019         k_perror (NULL);
1540020         _exit (0);
1540021     }
1540022     return (pid);
1540023 }
1540024

```

os16: «kernel/memory.h»

Si veda la sezione u0.7.

```

1550001 #ifndef _KERNEL_MEMORY_H
1550002 #define _KERNEL_MEMORY_H 1
1550003
1550004 #include <stdint.h>
1550005 #include <stddef.h>
1550006 #include <sys/types.h>
1550007 //-----
1550008 #define MEM_BLOCK_SIZE 256 // 0x0100
1550009 #define MEM_MAX_BLOCKS 2560 // 655360/256 = 0xA0000/0x0100 = 0xA000
1550010
1550011 extern uint16_t mb_table[MEM_MAX_BLOCKS/16]; // Memory blocks map.
1550012 //-----
1550013 typedef unsigned long int addr_t;
1550014 typedef unsigned int segment_t;
1550015 typedef unsigned int offset_t;
1550016 //-----
1550017 typedef struct {
1550018     addr_t address;
1550019     segment_t segment;
1550020     size_t size;
1550021 } memory_t;
1550022 //-----
1550023 addr_t address (segment_t segment, offset_t offset);
1550024 //-----
1550025 uint16_t *mb_reference (void);
1550026 ssize_t mb_alloc (addr_t address, size_t size);
1550027 void mb_free (addr_t address, size_t size);
1550028 int mb_alloc_size (size_t size, memory_t *allocated);
1550029 //-----
1550030 void mem_copy (addr_t orig, addr_t dest, size_t size);
1550031 size_t mem_read (addr_t start, void *buffer, size_t size);
1550032 size_t mem_write (addr_t start, void *buffer, size_t size);
1550033 //-----
1550034
1550035 #endif

```

kernel/memory/address.c

Si veda la sezione u0.7.

```

1560001 #include <kernel/memory.h>
1560002 //-----
1560003 addr_t
1560004 address (segment_t segment, offset_t offset)
1560005 {
1560006     addr_t a;
1560007     a = segment;
1560008     a += 16;
1560009     a += offset;
1560010     return (a);
1560011 }
1560012

```

```

1560013
1560014
1560015
1560016
1560017
1560018
1560019
1560020

```

kernel/memory/mb_alloc.c

Si veda la sezione u0.7.

```

1570001 #include <kernel/memory.h>
1570002 #include <kernel/ibm_i86.h>
1570003 #include <sys/os16.h>
1570004 #include <kernel/k_libc.h>
1570005 //-----
1570006 static int mb_block_set1 (int block);
1570007 //-----
1570008 ssize_t
1570009 mb_alloc (addr_t address, size_t size)
1570010 {
1570011     unsigned int bstart;
1570012     unsigned int bsize;
1570013     unsigned int bend;
1570014     unsigned int i;
1570015     ssize_t allocated = 0;
1570016     addr_t block_address;
1570017
1570018     if (size == 0)
1570019     {
1570020         //
1570021         // Zero means the maximum size.
1570022         //
1570023         bsize = 0x10000L / MEM_BLOCK_SIZE;
1570024     }
1570025     else
1570026     {
1570027         bsize = size / MEM_BLOCK_SIZE;
1570028     }
1570029
1570030     bstart = address / MEM_BLOCK_SIZE;
1570031
1570032     if (size % MEM_BLOCK_SIZE)
1570033     {
1570034         bend = bstart + bsize;
1570035     }
1570036     else
1570037     {
1570038         bend = bstart + bsize - 1;
1570039     }
1570040
1570041     for (i = bstart; i <= bend; i++)
1570042     {
1570043         if (mb_block_set1 (i))
1570044         {
1570045             allocated += MEM_BLOCK_SIZE;
1570046         }
1570047         else
1570048         {
1570049             block_address = i;
1570050             block_address += MEM_BLOCK_SIZE;
1570051             k_printf ("Kernel alert: mem block %04x, at address ", i);
1570052             k_printf ("%05lx, already allocated!\n", block_address);
1570053             break;
1570054         }
1570055     }
1570056     return (allocated);
1570057 }
1570058 //-----
1570059
1570060 static int
1570061 mb_block_set1 (int block)
1570062 {
1570063     int i = block / 16;
1570064     int j = block % 16;
1570065     uint16_t mask = 0x8000 >> j;
1570066     if (mb_table[i] & mask)
1570067     {
1570068         return (0); // The block is already set to 1 inside the map!
1570069     }
1570070     else
1570071     {
1570072         mb_table[i] = mb_table[i] | mask;
1570073         return (1);
1570074     }
1570075 }

```

kernel/memory/mb_alloc_size.c

Si veda la sezione u0.7.

```

1580001 #include <kernel/memory.h>
1580002 #include <kernel/ibm_i86.h>
1580003 #include <sys/os16.h>
1580004 #include <errno.h>
1580005 //-----
1580006 static int mb_block_status (int block);
1580007 //-----

```



```

158008 int
158009 mb_alloc_size (size_t size, memory_t *allocated)
158010 {
158011     unsigned int bsize;
158012     unsigned int i;
158013     unsigned int j;
158014     unsigned int found = 0;
158015     addr_t alloc_addr;
158016     ssize_t alloc_size;
158017
158018     if (size == 0)
158019     {
158020         //
158021         // Zero means the maximum size.
158022         //
158023         bsize = 0x10000L / MEM_BLOCK_SIZE;
158024     }
158025     else if (size % MEM_BLOCK_SIZE)
158026     {
158027         bsize = size / MEM_BLOCK_SIZE + 1;
158028     }
158029     else
158030     {
158031         bsize = size / MEM_BLOCK_SIZE;
158032     }
158033
158034     for (i = 0; i < (MEM_MAX_BLOCKS - bsize) && !found; i++)
158035     {
158036         for (j = 0; j < bsize; j++)
158037         {
158038             found = mb_block_status (i+j);
158039             if (!found)
158040             {
158041                 i += j;
158042                 break;
158043             }
158044         }
158045     }
158046
158047     if (found && (j == bsize))
158048     {
158049         alloc_addr = i - 1;
158050         alloc_addr += MEM_BLOCK_SIZE;
158051         alloc_size = bsize * MEM_BLOCK_SIZE;
158052         alloc_size = mb_alloc (alloc_addr, (size_t) alloc_size);
158053         if (alloc_size <= 0)
158054         {
158055             errset (ENOMEM);
158056             return (-1);
158057         }
158058         else if (alloc_size < size)
158059         {
158060             mb_free (alloc_addr, (size_t) alloc_size);
158061             errset (ENOMEM);
158062             return (-1);
158063         }
158064         else
158065         {
158066             allocated->address = alloc_addr;
158067             allocated->segment = alloc_addr / 16;
158068             allocated->size = (size_t) alloc_size;
158069         }
158070         return (0);
158071     }
158072     else
158073     {
158074         errset (ENOMEM);
158075         return (-1);
158076     }
158077 }
158078 //-----
158079 static int
158080 mb_block_status (int block)
158081 {
158082     int i = block / 16;
158083     int j = block % 16;
158084     uint16_t mask = 0x8000 >> j;
158085     return ((int) (mb_table[i] & mask));
158086 }

```

```

1590014 //
1590015 // Zero means the maximum size.
1590016 //
1590017 //
1590018 bsize = 0x10000L / MEM_BLOCK_SIZE;
1590019 }
1590020 else
1590021 {
1590022     bsize = size / MEM_BLOCK_SIZE;
1590023 }
1590024
1590025 bstart = address / MEM_BLOCK_SIZE;
1590026
1590027 if (size % MEM_BLOCK_SIZE)
1590028 {
1590029     bend = bstart + bsize;
1590030 }
1590031 else
1590032 {
1590033     bend = bstart + bsize - 1;
1590034 }
1590035
1590036 for (i = bstart; i <= bend; i++)
1590037 {
1590038     if (mb_block_set0 (i))
1590039     {
1590040         ;
1590041     }
1590042     else
1590043     {
1590044         block_address = i;
1590045         block_address += MEM_BLOCK_SIZE;
1590046         k_printf ("Kernel alert: mem block %04x, at address ", i);
1590047         k_printf ("%05lx, already released!\n", block_address);
1590048     }
1590049 }
1590050 //-----
1590051 static int
1590052 mb_block_set0 (int block)
1590053 {
1590054     int i = block / 16;
1590055     int j = block % 16;
1590056     uint16_t mask = 0x8000 >> j;
1590057     if (mb_table[i] & mask)
1590058     {
1590059         mb_table[i] = mb_table[i] & ~mask;
1590060         return (1);
1590061     }
1590062     else
1590063     {
1590064         return (0); // The block is already set to 0 inside the map!
1590065     }
1590066 }

```

kernel/memory/mb_reference.c

Si veda la sezione u0.7.

```

160001 #include <stdint.h>
160002 #include <kernel/memory.h>
160003 //-----
160004 uint16_t *
160005 mb_reference (void)
160006 {
160007     return mb_table;
160008 }
160009

```

kernel/memory/mb_table.c

Si veda la sezione u0.7.

```

161001 #include <kernel/memory.h>
161002 #include <stdint.h>
161003 //-----
161004 uint16_t mb_table[MEM_MAX_BLOCKS/16]; // Memory blocks map.
161005 //-----

```

kernel/memory/mb_free.c

Si veda la sezione u0.7.

```

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

```

kernel/memory/mem_copy.c

Si veda la sezione u0.7.

```

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

```

«

Si veda la sezione u0.7.

```

163001 #include <kernel/memory.h>
163002 #include <kernel/ikm_i86.h>
163003 #include <sys/os16.h>
163004 //-----
163005 size_t
163006 mem_read (addr_t start, void *buffer, size_t size)
163007 {
163008     unsigned int    segment = start / 16;
163009     unsigned int    offset = start % 16;
163010     unsigned long int end;
163011     end = start;
163012     end += size;
163013     if (end > 0x000FFFFFL)
163014     {
163015         size = 0x000FFFFFL - start;
163016     }
163017     ram_copy (segment, offset, seg_d (), (unsigned int) buffer, size);
163018     return (size);
163019 }

```

«

Si veda la sezione u0.7.

```

164001 #include <kernel/memory.h>
164002 #include <kernel/ikm_i86.h>
164003 #include <sys/os16.h>
164004 //-----
164005 size_t
164006 mem_write (addr_t start, void *buffer, size_t size)
164007 {
164008     unsigned int    segment = start / 16;
164009     unsigned int    offset = start % 16;
164010     unsigned long int end;
164011     end = start;
164012     end += size;
164013     if (end > 0x000FFFFFL)
164014     {
164015         size = 0x000FFFFFL - start;
164016     }
164017     ram_copy (seg_d (), (unsigned int) buffer, segment, offset, size);
164018     return (size);
164019 }

```

«

Si veda la sezione u0.8.

```

165001 #ifndef _KERNEL_PROC_H
165002 #define _KERNEL_PROC_H 1
165003
165004 #include <kernel/devices.h>
165005 #include <kernel/memory.h>
165006 #include <kernel/fs.h>
165007 #include <kernel/tty.h>
165008 #include <sys/types.h>
165009 #include <sys/stat.h>
165010 #include <sys/os16.h>
165011 #include <stddef.h>
165012 #include <stdint.h>
165013 #include <time.h>
165014
165015 //-----
165016 #define CLOCK_FREQUENCY_DIVISOR    65535 // [1]
165017 //
165018 // [1]
165019 // Internal clock frequency is (3579545/3) Hz.
165020 // This value is divided by 65535 (0xFFFF) giving 18.2 Hz.
165021 // The divisor value, 65535, is fixed!
165022 //
165023 //-----
165024 #define PROC_EMPTY                0
165025 #define PROC_CREATED              1
165026 #define PROC_READY                2
165027 #define PROC_RUNNING              3
165028 #define PROC_SLEEPING             4
165029 #define PROC_ZOMBIE               5
165030 //-----
165031 #define MAGIC_OS16                0x6F733136L // os16
165032 #define MAGIC_OS16_APPL           0x6170706CL // appl
165033 #define MAGIC_OS16_KERN           0x6B65726EL // kern
165034 //-----
165035 #define PROCESS_MAX               16 // Process slots.
165036
165037 typedef struct {
165038     pid_t        ppid; // Parent PID.
165039     pid_t        pgrp; // Process group ID.
165040     uid_t        uid; // Real user ID
165041     uid_t        euid; // Effective user ID.
165042     uid_t        suid; // Saved user ID.
165043     dev_t        device_tty; // Controlling terminal.
165044     char          path_cwd[PATH_MAX];
165045 // Working directory path.
165046     inode_t      *inode_cwd; // Working directory inode.
165047     int           umask; // File creation mask.
165048     unsigned long int sig_status; // Active signals.

```

```

166049 unsigned long int sig_ignore; // Signals to be ignored.
166050 clock_t          usage; // Clock ticks CPU time usage.
166051 unsigned int     status;
166052 int              wakeup_events; // Wake up for something.
166053 int              wakeup_signal; // Signal waited.
166054 unsigned int     wakeup_timer; // Seconds to wait for.
166055 addr_t           address_i;
166056 segment_t        segment_i;
166057 size_t           size_i;
166058 addr_t           address_d;
166059 segment_t        segment_d;
166060 size_t           size_d;
166061 uint16_t         sp;
166062 int              ret;
166063 char             name[PATH_MAX];
166064 fd_t             fd[FDOPEN_MAX];
166065 } proc_t;
166066
166067 extern proc_t    proc_table[PROCESS_MAX];
166068 //-----
166069 typedef struct {
166070     uint32_t filler0;
166071     uint32_t magic0;
166072     uint32_t magic1;
166073     uint16_t segoff;
166074     uint16_t etext;
166075     uint16_t edata;
166076     uint16_t ebss;
166077     uint16_t ssize;
166078 } header_t;
166079 //-----
166080 void          _ivt_load      (void);
166081 #define ivt_load()          (_ivt_load ())
166082 void          proc_init     (void);
166083 void          proc_scheduler (uint16_t *sp, segment_t *segment);
166084 void          sysroutine    (uint16_t *sp, segment_t *segment,
166085                             uint16_t syscallnr, uint16_t msg_off,
166086                             uint16_t msg_size);
166087 proc_t        *proc_reference (pid_t pid);
166088 //-----
166089 int           proc_sys_exec  (uint16_t *sp, segment_t *segment_d,
166090                             pid_t pid, const char *path,
166091                             unsigned int argc, char *arg_data,
166092                             unsigned int envc, char *env_data);
166093 void          proc_sys_exit  (pid_t pid, int status);
166094 pid_t         proc_sys_fork  (pid_t ppid, uint16_t sp);
166095 int           proc_sys_kill  (pid_t pid_killer, pid_t pid_target,
166096                             int sig);
166097 int           proc_sys_setuid (pid_t pid, uid_t uid);
166098 int           proc_sys_setuid (pid_t pid, uid_t uid);
166099 sighandler_t proc_sys_signal (pid_t pid, int sig,
166100                             sighandler_t handler);
166101 pid_t         proc_sys_wait  (pid_t pid, int *status);
166102 //-----
166103 void          proc_dump_memory (pid_t pid, addr_t address,
166104                                size_t size, char *name);
166105 void          proc_available  (pid_t pid);
166106 pid_t         proc_find       (segment_t segment_d);
166107 void          proc_sch_signals (void);
166108 void          proc_sch_terminals (void);
166109 void          proc_sch_timers (void);
166110 void          proc_sig_chld   (pid_t parent, int sig);
166111 void          proc_sig_cont   (pid_t pid, int sig);
166112 void          proc_sig_core   (pid_t pid, int sig);
166113 int           proc_sig_ignore (pid_t pid, int sig);
166114 void          proc_sig_off    (pid_t pid, int sig);
166115 void          proc_sig_on     (pid_t pid, int sig);
166116 int           proc_sig_status (pid_t pid, int sig);
166117 void          proc_sig_stop   (pid_t pid, int sig);
166118 void          proc_sig_term   (pid_t pid, int sig);
166119
166120 #endif

```

Si veda la sezione i159.8.1.

»

```

166001 .extern _proc_scheduler
166002 .extern _sysroutine
166003 .global __ksp
166004 .global __clock_ticks
166005 .global __clock_seconds
166006 .global isr_1C
166007 .global isr_80
166008 ;-----
166009 ; The kernel code segment starts at 0x10500 (segment 0x1050).
166010 ; The kernel data segments start at 0x00500 (segment 0x0050).
166011 ; To switch to the kernel data segments, DS, ES and SS are set to
166012 ; 0x0050. To identify the kernel context, the DS register is checked:
166013 ; if it is equal to 0x0050, it is the kernel.
166014 ;-----
166015 .data
166016 ;-----
166017 .align 2
166018 proc_ss_0: .word 0x0000
166019 proc_sp_0: .word 0x0000
166020 proc_ss_1: .word 0x0000
166021 proc_sp_1: .word 0x0000
166022 proc_syscallnr: .word 0x0000
166023 proc_msg_offset: .word 0x0000
166024 proc_msg_size: .word 0x0000

```

```

1660025 __ksp: .word 0x0000
1660026 __clock_ticks:
1660027 ticks_lo: .word 0x0000
1660028 ticks_hi: .word 0x0000
1660029 __clock_seconds:
1660030 seconds_lo: .word 0x0000
1660031 seconds_hi: .word 0x0000
1660032 ;-----
1660033 .text
1660034 ;-----
1660035 ; IRQ 0: "timer".
1660036 ; IRQ 0 is associated to INT 8, and after the BIOS work is done,
1660037 ; INT 1C is called. Standard INT 1C has nothing to do, but is
1660038 ; useful to call extra work for the timer. As the original BIOS
1660039 ; interrupts are used, the INT 1C is reprogrammed, keeping intact
1660040 ; the Standard INT 8.
1660041 ;-----
1660042 .align 2
1660043 isr_1c:
1660044 ;-----
1660045 ; Inside the process stack, the CPU already put:
1660046 ;
1660047 ; [omissis]
1660048 ; push flags
1660049 ; push cs
1660050 ; push ip
1660051 ;-----
1660052 ;
1660053 ; Save into process stack:
1660054 ;
1660055 push es ; extra segment
1660056 push ds ; data segment
1660057 push di ; destination index
1660058 push si ; source index
1660059 push bp ; base pointer
1660060 push bx ; BX
1660061 push dx ; DX
1660062 push cx ; CX
1660063 push ax ; AX
1660064 ;
1660065 ; Set the data segments to the kernel data area,
1660066 ; so that the following variables can be accessed.
1660067 ;
1660068 mov ax, #0x0050 ; DS and ES.
1660069 mov ds, ax ;
1660070 mov es, ax ;
1660071 ;
1660072 ; Increment time counters, to keep time.
1660073 ;
1660074 add ticks_lo, #1 ; Clock ticks counter.
1660075 adc ticks_hi, #0 ;
1660076 ;
1660077 mov dx, ticks_hi ;
1660078 mov ax, ticks_lo ; DX := ticks & 18
1660079 mov cx, #18 ;
1660080 div cx ;
1660081 mov ax, #0 ; If the ticks value can be divided by 18,
1660082 cmp ax, dx ; the seconds is incremented by 1.
1660083 jnz L1 ;
1660084 add seconds_lo, #1 ;
1660085 adc seconds_hi, #0 ;
1660086 ;
1660087 L1: ; Save process stack registers into kernel data segment.
1660088 ;
1660089 mov proc_ss_0, ss ; Save process stack segment.
1660090 mov proc_sp_0, sp ; Save process stack pointer.
1660091 ;
1660092 ; Check if it is already in kernel mode.
1660093 ;
1660094 mov dx, proc_ss_0
1660095 mov ax, #0x0050 ; Kernel data area.
1660096 cmp dx, ax
1660097 je L2
1660098 ;
1660099 ; If we are here, a user process was interrupted.
1660100 ; Switch to the kernel stack.
1660101 ;
1660102 mov ax, #0x0050 ; Kernel data area.
1660103 mov ss, ax
1660104 mov sp, __ksp
1660105 ;
1660106 ; Call the scheduler.
1660107 ;
1660108 push #proc_ss_0 ; &proc_ss_0
1660109 push #proc_sp_0 ; &proc_sp_0
1660110 call _proc_scheduler
1660111 add sp, #2
1660112 add sp, #2
1660113 ;
1660114 ; Restore process stack registers from kernel data segment.
1660115 ;
1660116 mov ss, proc_ss_0 ; Restore process stack segment.
1660117 mov sp, proc_sp_0 ; Restore process stack pointer.
1660118 ;
1660119 L2: ; Restore from process stack:
1660120 ;
1660121 pop ax
1660122 pop cx
1660123 pop dx
1660124 pop bx
1660125 pop bp

```

```

1660126 pop si
1660127 pop di
1660128 pop ds
1660129 pop es
1660130 ;
1660131 ; Return from interrupt: will restore CS:IP and FLAGS
1660132 ; from process stack.
1660133 ;
1660134 iret
1660135 ;-----
1660136 ; Syscall.
1660137 ;-----
1660138 .align 2
1660139 isr_80:
1660140 ;-----
1660141 ; Inside the process stack, we already have:
1660142 ; push #message_size
1660143 ; push #message_structure ; the relative address of it
1660144 ; push #syscall_number
1660145 ; push #back_address ; made by a call to _sys function
1660146 ; push flags ; made by int #0x80
1660147 ; push cs ; made by int #0x80
1660148 ; push ip ; made by int #0x80
1660149 ;-----
1660150 ;
1660151 ; Save into process stack:
1660152 ;
1660153 push es ; extra segment
1660154 push ds ; data segment
1660155 push di ; destination index
1660156 push si ; source index
1660157 push bp ; base pointer
1660158 push bx ; BX
1660159 push dx ; DX
1660160 push cx ; CX
1660161 push ax ; AX
1660162 ;
1660163 ; Set the data segments to the kernel data area,
1660164 ; so that the following variables can be accessed.
1660165 ;
1660166 mov ax, #0x0050 ; DS and ES.
1660167 mov ds, ax ;
1660168 mov es, ax ;
1660169 ;
1660170 ; Save process stack registers into kernel data segment.
1660171 ;
1660172 mov proc_ss_1, ss ; Save process stack segment.
1660173 mov proc_sp_1, sp ; Save process stack pointer.
1660174 ;
1660175 ; Save some more data, from the system call.
1660176 ;
1660177 mov bp, sp
1660178 mov ax, +26[bp]
1660179 mov proc_syscallnr, ax
1660180 mov ax, +28[bp]
1660181 mov proc_msg_offset, ax
1660182 mov ax, +30[bp]
1660183 mov proc_msg_size, ax
1660184 ;
1660185 ; Check if it is already the kernel stack.
1660186 ;
1660187 mov dx, ss
1660188 mov ax, #0x0050 ; Kernel data area.
1660189 cmp dx, ax
1660190 jne L3
1660191 ;
1660192 ; It is already the kernel stack, so, the variable "_ksp" is
1660193 ; aligned to current stack pointer. This way, the first syscall
1660194 ; can work without having to set the "_ksp" variable to some
1660195 ; reasonable value.
1660196 ;
1660197 mov __ksp, sp
1660198 ;
1660199 L3: ; Switch to the kernel stack.
1660200 ;
1660201 mov ax, #0x0050 ; Kernel data area.
1660202 mov ss, ax
1660203 mov sp, __ksp
1660204 ;
1660205 ; Call the external hardware interrupt handler.
1660206 ;
1660207 push proc_msg_size
1660208 push proc_msg_offset
1660209 push proc_syscallnr
1660210 push #proc_ss_1 ; &proc_ss_1
1660211 push #proc_sp_1 ; &proc_sp_1
1660212 call _sysroutine
1660213 add sp, #2
1660214 add sp, #2
1660215 add sp, #2
1660216 add sp, #2
1660217 add sp, #2
1660218 ;
1660219 ; Restore process stack registers from kernel data segment.
1660220 ;
1660221 mov ss, proc_ss_1 ; Restore process stack segment.
1660222 mov sp, proc_sp_1 ; Restore process stack pointer.
1660223 ;
1660224 ; Restore from process stack:
1660225 ;
1660226 pop ax

```

```

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

```

kernel/proc/_ivt_load.s

« Si veda la sezione [i159.8.2](#).

```

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

```

kernel/proc/proc_available.c

« Si veda la sezione [i159.8.3](#).

```

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

```

kernel/proc/proc_dump_memory.c

« Si veda la sezione [i159.8.4](#).

```

1690001 #include <kernel/proc.h>
1690002 #include <fcntl.h>
1690003 //-----
1690004 void
1690005 proc_dump_memory (pid_t pid, addr_t address, size_t size, char *name)
1690006 {
1690007     int fdn;
1690008     char buffer[SB_BLOCK_SIZE];
1690009     ssize_t size_written;
1690010     ssize_t size_written_total;
1690011     ssize_t size_read;
1690012     ssize_t size_read_total;
1690013     ssize_t size_total;
1690014     //
1690015     // Dump the code segment to disk.
1690016     //
1690017     fdn = fd_open (pid, name, (O_WRONLY|O_CREAT|O_TRUNC),
1690018                  (mode_t) (S_IFREG|0664));
1690019     if (fdn < 0)
1690020     {
1690021         //
1690022         // There is a problem: just let it go.
1690023         //
1690024         return;
1690025     }
1690026     //
1690027     // Fix size: (size_t) 0 is equivalent to (ssize_t) 0x10000.
1690028     //
1690029     size_total = size;
1690030     if (size_total == 0)
1690031     {
1690032         size_total = 0x10000;
1690033     }
1690034     //
1690035     // Read the memory and write it to disk.
1690036     //
1690037     for (size_read = 0, size_read_total = 0;
1690038         size_read_total < size_total;
1690039         size_read_total += size_read, address += size_read)
1690040     {
1690041         size_read = mem_read (address, buffer, SB_BLOCK_SIZE);
1690042         //
1690043         for (size_written = 0, size_written_total = 0;
1690044             size_written_total < size_read;
1690045             size_written_total += size_written)
1690046         {
1690047             size_written = fd_write (pid, fdn,
1690048                                     &buffer[size_written_total],
1690049                                     (size_t) (size_read - size_written_total));
1690050             //
1690051             if (size_written < 0)
1690052             {
1690053                 fd_close (pid, fdn);
1690054                 return;
1690055             }
1690056         }
1690057     }
1690058     fd_close (pid, fdn);
1690059 }

```

kernel/proc/proc_find.c

« Si veda la sezione [i159.8.5](#).

```

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

```

```

170034     print_mb_map ();
170035     k_printf ("%n");
170036     k_exit (0);
170037     }
170038     return (pid);
170039 }

```

kernel/proc/proc_init.c

« Si veda la sezione [i159.8.6](#).

```

170001 #include <kernel/proc.h>
170002 #include <kernel/k_libc.h>
170003 #include <string.h>
170004 //-----
170005 extern uint16_t _etext;
170006 //-----
170007 void
170008 proc_init (void)
170009 {
170010     uint8_t divisor_lo;
170011     uint8_t divisor_hi;
170012     pid_t pid;
170013     int fdn; // File descriptor index;
170014     addr_t start; // Used for effective memory addresses.
170015     size_t size; // Used for memory allocation.
170016     inode_t *inode;
170017     sb_t *sb;
170018     //
170019     // Clear interrupts (should already be cleared).
170020     //
170021     cli ();
170022     //
170023     // Load Interrupt vector table (IVT).
170024     //
170025     ivt_load ();
170026     //
170027     // Configure the clock: must be the original values, because
170028     // the BIOS depends on it!
170029     //
170030     // Base frequency is 1193181 Hz and it should divided.
170031     // Resulting frequency must be from 18.22 Hz and 1193181 Hz.
170032     // The calculated value (the divisor) must be sent to the
170033     // PIT (programmable interval timer), divided in two pieces.
170034     //
170035     divisor_lo = (CLOCK_FREQUENCY_DIVISOR & 0xFF); // Low byte.
170036     divisor_hi = (CLOCK_FREQUENCY_DIVISOR / 0x100) & 0xFF; // High byte.
170037     out_8 (0x43, 0x36);
170038     out_8 (0x40, divisor_lo); // Lower byte.
170039     out_8 (0x40, divisor_hi); // Higher byte.
170040     //
170041     // Set all memory reference to some invalid data.
170042     //
170043     for (pid = 0; pid < PROCESS_MAX; pid++)
170044     {
170045         proc_available (pid);
170046     }
170047     //
170048     // Mount root file system.
170049     //
170050     inode = NULL;
170051     sb = sb_mount (DEV_DSK0, &inode, MOUNT_DEFAULT);
170052     if (sb == NULL || inode == NULL)
170053     {
170054         k_perror ("Kernel panic: cannot mount root file system*");
170055         k_exit (0);
170056     }
170057     //
170058     // Set up the process table with the kernel.
170059     //
170060     proc_table[0].ppid = 0;
170061     proc_table[0].pgrp = 0;
170062     proc_table[0].uid = 0;
170063     proc_table[0].euid = 0;
170064     proc_table[0].suid = 0;
170065     proc_table[0].device_tty = DEV_UNDEFINED;
170066     proc_table[0].sig_status = 0;
170067     proc_table[0].sig_ignore = 0;
170068     proc_table[0].usage = 0;
170069     proc_table[0].status = PROC_RUNNING;
170070     proc_table[0].wakeup_events = 0;
170071     proc_table[0].wakeup_signal = 0;
170072     proc_table[0].wakeup_timer = 0;
170073     proc_table[0].segment_i = seg_i ();
170074     proc_table[0].address_i = seg_i ();
170075     proc_table[0].address_i *= 16;
170076     proc_table[0].size_i = (size_t) &_etext;
170077     proc_table[0].segment_d = seg_d ();
170078     proc_table[0].address_d = seg_d ();
170079     proc_table[0].address_d *= 16;
170080     proc_table[0].size_d = 0; // Maximum size: 0x10000.
170081     proc_table[0].sp = 0; // To be set at next interrupt.
170082     proc_table[0].ret = 0;
170083     proc_table[0].umask = 0022; // Default umask.
170084     proc_table[0].inode_cwd = inode; // Root fs inode.
170085     strncpy (proc_table[0].path_cwd, "/", PATH_MAX);
170086     strncpy (proc_table[0].name, "osi6 kernel", PATH_MAX);
170087     //
170088     // Ensure to have a terminated string.
170089     //
170090     proc_table[0].name[PATH_MAX-1] = 0;

```

1700

```

170091 //
170092 // Reset file descriptors.
170093 //
170094 for (fdn = 0; fdn < OPEN_MAX; fdn++)
170095 {
170096     proc_table[0].fd[fdn].fl_flags = 0;
170097     proc_table[0].fd[fdn].fd_flags = 0;
170098     proc_table[0].fd[fdn].file = NULL;
170099 }
170100 //
170101 // Allocate memory for the code segment.
170102 //
170103 mb_alloc (proc_table[0].address_i, proc_table[0].size_i);
170104 //
170105 // Allocate memory for the data segment if different.
170106 //
170107 if (seg_d () != seg_i ())
170108 {
170109     mb_alloc (proc_table[0].address_d, proc_table[0].size_d);
170110 }
170111 //
170112 // Allocate memory for the BIOS data area (BDA).
170113 //
170114 mb_alloc (0x00000L, 0x500);
170115 //
170116 // Allocate memory for the extra BIOS at the
170117 // bottom of the 640 Kibyte.
170118 //
170119 start = int12 ();
170120 start += 1024;
170121 size = 0xA0000L - start;
170122 mb_alloc (start, size);
170123 //
170124 // Enable and disable hardware interrupts (IRQ).
170125 //
170126 irq_on (0); // timer.
170127 irq_on (1); // enable keyboard
170128 irq_off (2); //
170129 irq_off (3); //
170130 irq_off (4); //
170131 irq_off (5); //
170132 irq_on (6); // floppy (must be on to let int 13 work!)
170133 irq_off (7); //
170134 //
170135 // Interrupts activation.
170136 //
170137 sti ();
170138 }

```

kernel/proc/proc_reference.c

« Si veda la sezione [i159.8.7](#).

```

172001 #include <kernel/proc.h>
172002 //-----
172003 proc_t *
172004 proc_reference (pid_t pid)
172005 {
172006     if (pid >= 0 && pid < PROCESS_MAX)
172007     {
172008         return (&proc_table[pid]);
172009     }
172010     else
172011     {
172012         return (NULL);
172013     }
172014 }

```

kernel/proc/proc_sch_signals.c

« Si veda la sezione [i159.8.8](#).

```

173001 #include <kernel/proc.h>
173002 //-----
173003 void
173004 proc_sch_signals (void)
173005 {
173006     pid_t pid;
173007     for (pid = 0; pid < PROCESS_MAX; pid++)
173008     {
173009         proc_sig_term (pid, SIGHUP);
173010         proc_sig_term (pid, SIGINT);
173011         proc_sig_core (pid, SIGQUIT);
173012         proc_sig_core (pid, SIGILL);
173013         proc_sig_core (pid, SIGABRT);
173014         proc_sig_core (pid, SIGFPE);
173015         proc_sig_term (pid, SIGKILL);
173016         proc_sig_core (pid, SIGSEGV);
173017         proc_sig_term (pid, SIGPIPE);
173018         proc_sig_term (pid, SIGALRM);
173019         proc_sig_term (pid, SIGTERM);
173020         proc_sig_term (pid, SIGUSR1);
173021         proc_sig_term (pid, SIGUSR2);
173022         proc_sig_chld (pid, SIGCHLD);
173023         proc_sig_cont (pid, SIGCONT);
173024         proc_sig_stop (pid, SIGSTOP);
173025         proc_sig_stop (pid, SIGTSTP);
173026         proc_sig_stop (pid, SIGTTIN);
173027         proc_sig_stop (pid, SIGTTOU);
173028     }

```

1701

```
1740029 ]
```

kernel/proc/proc_sch_terminals.c

Si veda la sezione [i159.8.9.](#)

```

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

```

```

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

```

kernel/proc/proc_sch_timers.c

Si veda la sezione [i159.8.10.](#)

```

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

```

kernel/proc/proc_scheduler.c

Si veda la sezione [i159.8.11.](#)

```

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

```

```

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

```

1704

kernel/proc/proc_sig_chld.c

Si veda la sezione [i159.8.12.](#)

```

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

```

kernel/proc/proc_sig_cont.c

Si veda la sezione [i159.8.13.](#)

```

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

```

kernel/proc/proc_sig_core.c

Si veda la sezione [i159.8.14.](#)

```

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

```

1705

```

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

```

kernel/proc/proc_sig_ignore.c

Si veda la sezione [i159.8.15](#).

```

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

```

kernel/proc/proc_sig_off.c

Si veda la sezione [i159.8.16](#).

```

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

```

kernel/proc/proc_sig_on.c

Si veda la sezione [i159.8.16](#).

```

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

```

kernel/proc/proc_sig_status.c

Si veda la sezione [i159.8.17](#).

```

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

```

kernel/proc/proc_sig_stop.c

Si veda la sezione [i159.8.18](#).

```

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

```

kernel/proc/proc_sig_term.c

Si veda la sezione [i159.8.19](#).

```

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

```

kernel/proc/proc_sys_exec.c

Si veda la sezione [i159.8.20](#).

```

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

```



```

1860046 int          fdn;
1860047 dev_t      device;
1860048 int          eof;
1860049 //
1860050 // Check for limits.
1860051 //
1860052 if (argc > (ARG_MAX/16) || envc > (ARG_MAX/16))
1860053 {
1860054     errset (ENOMEM);
1860055     return (-1);
1860056 }
1860057 //
1860058 // Scan arguments to calculate the full size and the relative
1860059 // pointers. The final size is rounded to 2, for the stack.
1860060 //
1860061 arg = arg_data;
1860062 for (i = 0, j = 0; i < argc; i++)
1860063 {
1860064     argv[i] = (char *) j; // Relative pointer inside
1860065                          // the 'arg_data'.
1860066     size = strlen (arg);
1860067     arg += size + 1;
1860068     j += size + 1;
1860069 }
1860070 arg_data_size = j;
1860071 if (arg_data_size % 2)
1860072 {
1860073     arg_data_size++;
1860074 }
1860075 //
1860076 // Scan environment variables to calculate the full size and the
1860077 // relative pointers. The final size is rounded to 2, for the stack.
1860078 //
1860079 env = env_data;
1860080 for (i = 0, j = 0; i < envc; i++)
1860081 {
1860082     envp[i] = (char *) j; // Relative pointer inside
1860083                          // the 'env_data'.
1860084     size = strlen (env);
1860085     env += size + 1;
1860086     j += size + 1;
1860087 }
1860088 env_data_size = j;
1860089 if (env_data_size % 2)
1860090 {
1860091     env_data_size++;
1860092 }
1860093 //
1860094 // Read the inode related to the executable file name.
1860095 // Function path_inode() includes the inode get procedure.
1860096 //
1860097 inode = path_inode (pid, path);
1860098 if (inode == NULL)
1860099 {
1860100     errset (ENOENT); // No such file or directory.
1860101     return (-1);
1860102 }
1860103 //
1860104 // Check for permissions.
1860105 //
1860106 status = inode_check (inode, S_IFREG, 5, proc_table[pid].euid);
1860107 if (status != 0)
1860108 {
1860109     //
1860110     // File is not of a valid type or permission are not
1860111     // sufficient: release the executable file inode
1860112     // and return with an error.
1860113     //
1860114     inode_put (inode);
1860115     errset (EACCESS); // Permission denied.
1860116     return (-1);
1860117 }
1860118 //
1860119 // Read the header from the executable file.
1860120 //
1860121 size_read = inode_file_read (inode, (off_t) 0, &header,
1860122                             (sizeof header), &eof);
1860123 if (size_read != (sizeof header))
1860124 {
1860125     //
1860126     // The file is shorter than the executable header, so, it isn't
1860127     // an executable: release the file inode and return with an
1860128     // error.
1860129     //
1860130     inode_put (inode);
1860131     errset (ENOEXEC);
1860132     return (-1);
1860133 }
1860134 if (header.magic0 != MAGIC_OS16 || header.magic1 != MAGIC_OS16_APPL)
1860135 {
1860136     //
1860137     // The header does not have the expected magic numbers, so,
1860138     // it isn't a valid executable: release the file inode and
1860139     // return with an error.
1860140     //
1860141     inode_put (inode);
1860142     errset (ENOEXEC);
1860143     return (-1); // This is not a valid executable!
1860144 }
1860145 //
1860146 // Calculate data size.

```

1708

```

1860147 //
1860148 s = header.ebss;
1860149 if (header.ssize == 0)
1860150 {
1860151     s += 0x10000L; // Zero means max size.
1860152 }
1860153 else
1860154 {
1860155     s += header.ssize;
1860156 }
1860157 if (s > 0xFFFF)
1860158 {
1860159     process_size_d = 0x0000; // 0x0000 means the maximum size:
1860160                             // 0x10000.
1860161     new_sp          = 0x0000; // 0x0000 is like 0x10000 and the first
1860162                             // push moves SP to 0xFFFF.
1860163 }
1860164 else
1860165 {
1860166     process_size_d = s;
1860167     new_sp          = process_size_d;
1860168     if (new_sp % 2)
1860169     {
1860170         new_sp--; // The stack pointer should be even.
1860171     }
1860172 }
1860173 //
1860174 // Calculate code size.
1860175 //
1860176 if (header.segoff == 0)
1860177 {
1860178     process_size_i = process_size_d;
1860179 }
1860180 else
1860181 {
1860182     process_size_i = header.segoff + 16;
1860183 }
1860184 //
1860185 // Allocate memory: code and data segments.
1860186 //
1860187 status = mb_alloc_size (process_size_i, &allocated_i);
1860188 if (status < 0)
1860189 {
1860190     //
1860191     // The program instructions (code segment) cannot be loaded
1860192     // into memory: release the executable file inode and return
1860193     // with an error.
1860194     //
1860195     inode_put (inode);
1860196     errset (ENOMEM); // Not enough space.
1860197     return ((pid_t) -1);
1860198 }
1860199 if (header.segoff == 0)
1860200 {
1860201     //
1860202     // Code and data segments are the same: no need
1860203     // to allocate more memory for the data segment.
1860204     //
1860205     allocated_d.address = allocated_i.address;
1860206     allocated_d.segment = allocated_i.segment;
1860207     allocated_d.size    = allocated_i.size;
1860208 }
1860209 else
1860210 {
1860211     //
1860212     // Code and data segments are different: the data
1860213     // segment memory is allocated.
1860214     //
1860215     status = mb_alloc_size (process_size_d, &allocated_d);
1860216     if (status < 0)
1860217     {
1860218         //
1860219         // The separated program data (data segment) cannot be loaded
1860220         // into memory: free the already allocated memory for the
1860221         // program instructions, release the executable file inode
1860222         // and return with an error.
1860223         //
1860224         mb_free (allocated_i.address, allocated_i.size);
1860225         inode_put (inode);
1860226         errset (ENOMEM); // Not enough space.
1860227         return ((pid_t) -1);
1860228     }
1860229 }
1860230 //
1860231 // Load executable in memory.
1860232 //
1860233 if (header.segoff == 0)
1860234 {
1860235     //
1860236     // Code and data share the same segment.
1860237     //
1860238     for (eof = 0, memory_start = allocated_i.address,
1860239          inode_start = 0, size_read = 0;
1860240          inode_start < inode->size && !eof;
1860241          inode_start += size_read)
1860242     {
1860243         memory_start += size_read;
1860244         //
1860245         // Read a block of memory.
1860246         //
1860247         size_read = inode_file_read (inode, inode_start,

```

1709

```

1860248         buffer, MEM_BLOCK_SIZE, &eof);
1860249
1860250     {
1860251         //
1860252         // Free memory and inode.
1860253         //
1860254         mb_free (allocated_i.address, allocated_i.size);
1860255         inode_put (inode);
1860256         errset (EIO);
1860257         return (-1);
1860258     }
1860259     //
1860260     // Copy inside the right position to be executed.
1860261     //
1860262     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE, memory_start, buffer,
1860263             (size_t) size_read, NULL);
1860264 }
1860265 }
1860266 else
1860267 {
1860268     //
1860269     // Code and data with different segments.
1860270     //
1860271     for (eof = 0, memory_start = allocated_i.address,
1860272          inode_start = 0, size_read = 0;
1860273          inode_start < process_size_i && !eof;
1860274          inode_start += size_read)
1860275     {
1860276         memory_start += size_read;
1860277         //
1860278         // Read a block of memory
1860279         //
1860280         size_read = inode_file_read (inode, inode_start,
1860281                                     buffer, MEM_BLOCK_SIZE, &eof);
1860282         if (size_read < 0)
1860283         {
1860284             //
1860285             // Free memory and inode.
1860286             //
1860287             mb_free (allocated_i.address, allocated_i.size);
1860288             mb_free (allocated_d.address, allocated_d.size);
1860289             inode_put (inode);
1860290             errset (EIO);
1860291             return (-1);
1860292         }
1860293         //
1860294         // Copy inside the right position to be executed.
1860295         //
1860296         dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE, memory_start, buffer,
1860297                 (size_t) size_read, NULL);
1860298     }
1860299     for (eof = 0, memory_start = allocated_d.address,
1860300          inode_start = (header.segoff + 16), size_read = 0;
1860301          inode_start < inode_size && !eof;
1860302          inode_start += size_read)
1860303     {
1860304         memory_start += size_read;
1860305         //
1860306         // Read a block of memory
1860307         //
1860308         size_read = inode_file_read (inode, inode_start,
1860309                                     buffer, MEM_BLOCK_SIZE, &eof);
1860310         if (size_read < 0)
1860311         {
1860312             //
1860313             // Free memory and inode.
1860314             //
1860315             mb_free (allocated_i.address, allocated_i.size);
1860316             mb_free (allocated_d.address, allocated_d.size);
1860317             inode_put (inode);
1860318             errset (EIO);
1860319             return (-1);
1860320         }
1860321         dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE, memory_start, buffer,
1860322                 (size_t) size_read, NULL);
1860323     }
1860324 }
1860325 //
1860326 // The executable file was successfully loaded in memory:
1860327 // release the executable file inode.
1860328 //
1860329 inode_put (inode);
1860330 //
1860331 // Put environment data inside the stack.
1860332 //
1860333 new_sp -= env_data_size; //----- environment
1860334 mem_copy (address (seg_d (), (unsigned int) env_data),
1860335           (allocated_d.address + new_sp), env_data_size);
1860336 //
1860337 // Put arguments data inside the stack.
1860338 //
1860339 new_sp -= arg_data_size; //----- arguments
1860340 mem_copy (address (seg_d (), (unsigned int) arg_data),
1860341           (allocated_d.address + new_sp), arg_data_size);
1860342 //
1860343 // Put envp[] inside the stack, updating all the pointers.
1860344 //
1860345 new_sp -= 2; //----- NULL
1860346 stack_element = NULL;
1860347 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860348         (allocated_d.address + new_sp),

```

1710

```

1860349         &stack_element, (sizeof stack_element), NULL);
1860350 //
1860351 p_off = new_sp; //
1860352 p_off += 2; // Calculate memory pointers from
1860353 p_off += arg_data_size; // original relative pointers,
1860354 for (i = 0; i < envc; i++) // inside the environment array
1860355 { // of pointers.
1860356     envp[i] += p_off; //
1860357 } //
1860358 //
1860359 new_sp -= (envc * (sizeof (char *))); //----- *envp[]
1860360 mem_copy (address (seg_d (), (unsigned int) envp),
1860361           (allocated_d.address + new_sp),
1860362           (envc * (sizeof (char *))));
1860363 //
1860364 // Save the envp[] location, needed in the following.
1860365 //
1860366 envp_address = new_sp;
1860367 //
1860368 // Put argv[] inside the stack, updating all the pointers.
1860369 //
1860370 new_sp -= 2; //----- NULL
1860371 stack_element = NULL;
1860372 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860373         (allocated_d.address + new_sp),
1860374         &stack_element, (sizeof stack_element), NULL);
1860375 //
1860376 p_off = new_sp; //
1860377 p_off += 2; // Calculate memory pointers
1860378 p_off += (envc * (sizeof (char *))); // from original relative
1860379 p_off += 2; // pointers, inside the
1860380 for (i = 0; i < argc; i++) // arguments array of
1860381 { // pointers.
1860382     argv[i] += p_off; //
1860383 } //
1860384 //
1860385 new_sp -= (argc * (sizeof (char *))); //----- *argv[]
1860386 mem_copy (address (seg_d (), (unsigned int) argv),
1860387           (allocated_d.address + new_sp),
1860388           (argc * (sizeof (char *))));
1860389 //
1860390 // Save the argv[] location, needed in the following.
1860391 //
1860392 argv_address = new_sp;
1860393 //
1860394 // Put the pointer to the array envp[].
1860395 //
1860396 new_sp -= 2; //----- argc
1860397 stack_element = envp_address;
1860398 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860399         (allocated_d.address + new_sp),
1860400         &stack_element, (sizeof stack_element), NULL);
1860401 //
1860402 // Put the pointer to the array argv[].
1860403 //
1860404 new_sp -= 2; //----- argc
1860405 stack_element = argv_address;
1860406 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860407         (allocated_d.address + new_sp),
1860408         &stack_element, (sizeof stack_element), NULL);
1860409 //
1860410 // Put argc inside the stack.
1860411 //
1860412 new_sp -= 2; //----- argc
1860413 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860414         (allocated_d.address + new_sp),
1860415         &argc, (sizeof argc), NULL);
1860416 //
1860417 // Set the rest of the stack.
1860418 //
1860419 new_sp -= 2; //----- FLAGS
1860420 stack_element = 0x0200;
1860421 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860422         (allocated_d.address + new_sp),
1860423         &stack_element, (sizeof stack_element), NULL);
1860424 new_sp -= 2; //----- CS
1860425 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860426         (allocated_d.address + new_sp),
1860427         &allocated_i.segment, (sizeof allocated_i.segment), NULL);
1860428 new_sp -= 2; //----- IP
1860429 stack_element = 0;
1860430 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860431         (allocated_d.address + new_sp),
1860432         &stack_element, (sizeof stack_element), NULL);
1860433 new_sp -= 2; //----- ES
1860434 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860435         (allocated_d.address + new_sp),
1860436         &allocated_d.segment, (sizeof allocated_d.segment), NULL);
1860437 new_sp -= 2; //----- DS
1860438 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860439         (allocated_d.address + new_sp),
1860440         &allocated_d.segment, (sizeof allocated_d.segment), NULL);
1860441 new_sp -= 2; //----- DI
1860442 stack_element = 0;
1860443 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860444         (allocated_d.address + new_sp),
1860445         &stack_element, (sizeof stack_element), NULL);
1860446 new_sp -= 2; //----- SI
1860447 stack_element = 0;
1860448 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860449         (allocated_d.address + new_sp),

```

1711

```

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

```

1712

```

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

```

kernel/proc/proc_sys_exit.c

Si veda la sezione [1159.8.21](#).

«

```

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

```

1713

```

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

```

1714

kernel/proc/proc_sys_fork.c

Si veda la sezione [i159.8.22](#).

```

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

```

1715

```

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

```

1716

```

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

```

kernel/proc/proc_sys_kill.c

Si veda la sezione [i159.8.23](#).

«

```

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

```

1717

```

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

```

kernel/proc/proc_sys_seteuid.c

« Si veda la sezione [i159.8.24](#).

```

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

```

kernel/proc/proc_sys_setuid.c

« Si veda la sezione [i159.8.25](#).

```

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

```

1718

```

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

```

kernel/proc/proc_sys_signal.c

Si veda la sezione [i159.8.26](#).

«

```

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

```

kernel/proc/proc_sys_wait.c

Si veda la sezione [i159.8.27](#).

«

```

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

```

1719

```

190036     if (child_available)
190037     {
190038         //
190039         // There are child, but all alive.
190040         //
190041         // Go to sleep.
190042         //
190043         proc_table[parent].status      = PROC_SLEEPING;
190044         proc_table[parent].wakeup_events |= WAKEUP_EVENT_SIGNAL;
190045         proc_table[parent].wakeup_signal = SIGCHLD;
190046         return ((pid_t) 0);
190047     }
190048     else
190049     {
190050         //
190051         // There are no child at all.
190052         //
190053         errset (ECHILD);
190054         return ((pid_t) -1);
190055     }
190056 }
190057 }

```

kernel/proc/proc_table.c

Si veda la sezione [i159.8.7](#).

```

194001 #include <kernel/proc.h>
194002 //-----
194003 proc_t  proc_table[PROCESS_MAX];

```

kernel/proc/sysroutine.c

Si veda la sezione [i159.8.28](#).

```

190001 #include <kernel/proc.h>
190002 #include <errno.h>
190003 #include <kernel/k_libc.h>
190004 //-----
190005 static void sysroutine_error_back (int *number, int *line,
190006                                  char *file_name);
190007 //-----
190008 void
190009 sysroutine (uint16_t *sp, segment_t *segment_d, uint16_t syscallnr,
190010            uint16_t msg_off, uint16_t msg_size)
190011 {
190012     pid_t pid      = proc_find (*segment_d);
190013     addr_t msg_addr = address (*segment_d, msg_off);
190014     //
190015     // Inbox.
190016     //
190017     union {
190018         sysmsg_chdir_t   chdir;
190019         sysmsg_chmod_t   chmod;
190020         sysmsg_chown_t   chown;
190021         sysmsg_clock_t   clock;
190022         sysmsg_close_t   close;
190023         sysmsg_dup_t     dup;
190024         sysmsg_dup2_t    dup2;
190025         sysmsg_exec_t    exec;
190026         sysmsg_exit_t    exit;
190027         sysmsg_fchmod_t  fchmod;
190028         sysmsg_fchown_t  fchown;
190029         sysmsg_fcntl_t   fcntl;
190030         sysmsg_fork_t    fork;
190031         sysmsg_fstat_t   fstat;
190032         sysmsg_kill_t    kill;
190033         sysmsg_link_t    link;
190034         sysmsg_lseek_t   lseek;
190035         sysmsg_mkdir_t   mkdir;
190036         sysmsg_mknod_t   mknod;
190037         sysmsg_mount_t   mount;
190038         sysmsg_open_t    open;
190039         sysmsg_read_t    read;
190040         sysmsg_setuid_t  setuid;
190041         sysmsg_setuid_t  setuid;
190042         sysmsg_signal_t  signal;
190043         sysmsg_sleep_t   sleep;
190044         sysmsg_stat_t    stat;
190045         sysmsg_stime_t   stime;
190046         sysmsg_time_t    time;
190047         sysmsg_uarea_t   uarea;
190048         sysmsg_umask_t   umask;
190049         sysmsg_umount_t  umount;
190050         sysmsg_unlink_t  unlink;
190051         sysmsg_wait_t    wait;
190052         sysmsg_write_t   write;
190053         sysmsg_zpchar_t  zpchar;
190054         sysmsg_zpstring_t zpstring;
190055     } msg;
190056     //
190057     // Verify if the system call was emitted from kernel code.
190058     // The kernel can emit only some particular system call:
190059     // SYS_NULL, to let other processes run;
190060     // SYS_FORK, to let fork itself;
190061     // SYS_EXEC, to replace a forked copy of itself.
190062     //
190063     if (pid == 0)
190064     {
190065         //

```

1720

```

190066     // This is the kernel code!
190067     //
190068     if ( syscallnr != SYS_0
190069         && syscallnr != SYS_FORK
190070         && syscallnr != SYS_EXEC)
190071     {
190072         k_printf ("kernel panic: the system call %i ", syscallnr);
190073         k_printf ("was received while running in kernel space!\n");
190074     }
190075 }
190076 //
190077 // Entering a system call, the kernel variable 'errno' must be
190078 // reset, otherwise, a previous kernel code error might be returned
190079 // to the applications.
190080 //
190081 errno = 0;
190082 errln = 0;
190083 errfn[0] = 0;
190084 //
190085 // Get message.
190086 //
190087 dev_io (pid, DEV_MEM, DEV_READ, msg_addr, &msg, msg_size, NULL);
190088 //
190089 // Do the request from the received system call.
190090 //
190091 switch (syscallnr)
190092 {
190093     case SYS_0:
190094         break;
190095     case SYS_CHDIR:
190096         msg.chdir.ret      = path_chdir (pid, msg.chdir.path);
190097         sysroutine_error_back (&msg.chdir.errno, &msg.chdir.errln,
190098                               msg.chdir.errfn);
190099         break;
190100     case SYS_CHMOD:
190101         msg.chmod.ret      = path_chmod (pid, msg.chmod.path,
190102                                         msg.chmod.mode);
190103         sysroutine_error_back (&msg.chmod.errno, &msg.chmod.errln,
190104                               msg.chmod.errfn);
190105         break;
190106     case SYS_CHOWN:
190107         msg.chown.ret      = path_chown (pid, msg.chown.path,
190108                                         msg.chown.uid,
190109                                         msg.chown.gid);
190110         sysroutine_error_back (&msg.chown.errno, &msg.chown.errln,
190111                               msg.chown.errfn);
190112         break;
190113     case SYS_CLOCK:
190114         msg.clock.ret      = k_clock ();
190115         break;
190116     case SYS_CLOSE:
190117         msg.close.ret      = fd_close (pid, msg.close.fdn);
190118         sysroutine_error_back (&msg.close.errno, &msg.close.errln,
190119                               msg.close.errfn);
190120         break;
190121     case SYS_DUP:
190122         msg.dup.ret        = fd_dup (pid, msg.dup.fdn_old, 0);
190123         sysroutine_error_back (&msg.dup.errno, &msg.dup.errln,
190124                               msg.dup.errfn);
190125         break;
190126     case SYS_DUP2:
190127         msg.dup2.ret       = fd_dup2 (pid, msg.dup2.fdn_old,
190128                                       msg.dup2.fdn_new);
190129         sysroutine_error_back (&msg.dup2.errno, &msg.dup2.errln,
190130                               msg.dup2.errfn);
190131         break;
190132     case SYS_EXEC:
190133         msg.exec.ret       = proc_sys_exec (sp, segment_d, pid,
190134                                           msg.exec.path,
190135                                           msg.exec.argc,
190136                                           msg.exec.arg_data,
190137                                           msg.exec.envc,
190138                                           msg.exec.env_data);
190139         msg.exec.uid        = proc_table[pid].uid;
190140         msg.exec.euid       = proc_table[pid].euid;
190141         sysroutine_error_back (&msg.exec.errno, &msg.exec.errln,
190142                               msg.exec.errfn);
190143         break;
190144     case SYS_EXIT:
190145         if (pid == 0)
190146         {
190147             k_printf ("kernel alert: "
190148                      "the kernel cannot exit!\n");
190149         }
190150         else
190151         {
190152             proc_sys_exit (pid, msg.exit.status);
190153         }
190154         break;
190155     case SYS_FCHMOD:
190156         msg.fchmod.ret     = fd_chmod (pid, msg.fchmod.fdn,
190157                                       msg.fchmod.mode);
190158         sysroutine_error_back (&msg.fchmod.errno, &msg.fchmod.errln,
190159                               msg.fchmod.errfn);
190160         break;
190161     case SYS_FCHOWN:
190162         msg.fchown.ret     = fd_chown (pid, msg.fchown.fdn,
190163                                       msg.fchown.uid,
190164                                       msg.fchown.gid);
190165         sysroutine_error_back (&msg.fchown.errno, &msg.fchown.errln,
190166                               msg.fchown.errfn);

```

1721

```

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

```

1722

```

1950268         proc_table[pid].ret      = 0;
1950269         proc_table[pid].wakeup_events = msg.sleep.events;
1950270         proc_table[pid].wakeup_signal = msg.sleep.signal;
1950271         proc_table[pid].wakeup_timer = msg.sleep.seconds;
1950272         break;
1950273     case SYS_STAT:
1950274         msg.stat.ret         = path_stat (pid, msg.stat.path,
1950275                                     &msg.stat.stat);
1950276         sysroutine_error_back (&msg.stat.errno, &msg.stat.errln,
1950277                                 msg.stat.errfn);
1950278         break;
1950279     case SYS_STIME:
1950280         msg.stime.ret        = k_stime (&msg.stime.timer);
1950281         break;
1950282     case SYS_TIME:
1950283         msg.time.ret         = k_time (NULL);
1950284         break;
1950285     case SYS_UAREA:
1950286         msg.uarea.uid        = proc_table[pid].uid;
1950287         msg.uarea.euid       = proc_table[pid].euid;
1950288         msg.uarea.pid        = pid;
1950289         msg.uarea.ppid       = proc_table[pid].ppid;
1950290         msg.uarea.pgrp       = proc_table[pid].pgpr;
1950291         msg.uarea.umask      = proc_table[pid].umask;
1950292         strncpy (msg.uarea.path_cwd,
1950293                 proc_table[pid].path_cwd, PATH_MAX);
1950294         break;
1950295     case SYS_UMASK:
1950296         msg.umask.ret        = proc_table[pid].umask;
1950297         proc_table[pid].umask = (msg.umask.umask & 0077);
1950298         break;
1950299     case SYS_UMOUNT:
1950300         msg.umount.ret       = path_umount (pid,
1950301                                     msg.umount.path_mnt);
1950302         sysroutine_error_back (&msg.umount.errno, &msg.umount.errln,
1950303                                 msg.umount.errfn);
1950304         break;
1950305     case SYS_UNLINK:
1950306         msg.unlink.ret       = path_unlink (pid, msg.unlink.path);
1950307         sysroutine_error_back (&msg.unlink.errno, &msg.unlink.errln,
1950308                                 msg.unlink.errfn);
1950309         break;
1950310     case SYS_WAIT:
1950311         msg.wait.ret         = proc_sys_wait (pid,
1950312                                     &msg.wait.status);
1950313         sysroutine_error_back (&msg.wait.errno, &msg.wait.errln,
1950314                                 msg.wait.errfn);
1950315         break;
1950316     case SYS_WRITE:
1950317         msg.write.ret        = fd_write (pid, msg.write.fdn,
1950318                                     msg.write.buffer,
1950319                                     msg.write.count);
1950320         sysroutine_error_back (&msg.write.errno, &msg.write.errln,
1950321                                 msg.write.errfn);
1950322         break;
1950323     case SYS_ZPCHAR:
1950324         dev_io (pid, DEV_TTY, DEV_WRITE, 0L, &msg.zpchar.c,
1950325                 1, NULL);
1950326         break;
1950327     case SYS_ZPSTRING:
1950328         dev_io (pid, DEV_TTY, DEV_WRITE, 0L,
1950329                 msg.zpstring.string,
1950330                 strlen (msg.zpstring.string), NULL);
1950331         break;
1950332     default:
1950333         k_printf ("kernel alert: unknown system call %i!\n",
1950334                 syscallnr);
1950335         break;
1950336     }
1950337     //
1950338     // Return value with a message back.
1950339     //
1950340     dev_io (pid, DEV_MEM, DEV_WRITE, msg_addr, &msg, msg_size, NULL);
1950341     //
1950342     // Continue with the scheduler.
1950343     //
1950344     proc_scheduler (sp, segment_d);
1950345 }
1950346 //-----
1950347 static void
1950348 sysroutine_error_back (int *number, int *line, char *file_name)
1950349 {
1950350     *number = errno;
1950351     *line   = errln;
1950352     strncpy (file_name, errfn, PATH_MAX);
1950353     file_name[PATH_MAX-1] = 0;
1950354 }

```

os16: «kernel/tty.h»

Si veda la sezione u0.9.

```

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

```

1723

«


```

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

```

kernel/tty/tty_console.c

Si veda la sezione u0.9.

```

1970001 #include <sys/osi16.h>
1970002 #include <kernel/tty.h>
1970003 //-----
1970004 dev_t
1970005 tty_console (dev_t device)
1970006 {
1970007     static dev_t device_active = DEV_CONSOLE0; // First time.
1970008     dev_t device_previous;
1970009     //
1970010     // Check if it required only the current device.
1970011     //
1970012     if (device == 0)
1970013     {
1970014         return (device_active);
1970015     }
1970016     //
1970017     // Fix if the device is not valid.
1970018     //
1970019     if (device > DEV_CONSOLE3 || device < DEV_CONSOLE0)
1970020     {
1970021         device = DEV_CONSOLE0;
1970022     }
1970023     //
1970024     // Update.
1970025     //
1970026     device_previous = device_active;
1970027     device_active = device;
1970028     //
1970029     // Switch.
1970030     //
1970031     con_select (device_active & 0x00FF);
1970032     //
1970033     // Return previous device value.
1970034     //
1970035     return (device_previous);
1970036 }

```

kernel/tty/tty_init.c

Si veda la sezione u0.9.

```

1980001 #include <sys/osi16.h>
1980002 #include <kernel/tty.h>
1980003 //-----
1980004 void
1980005 tty_init (void)
1980006 {
1980007     int page; // console page.
1980008     //
1980009     // Console initialization: console pages correspond to the first
1980010     // terminal items.
1980011     //
1980012     for (page = 0; page < TTYS_CONSOLE; page++)
1980013     {
1980014         tty_table[page].device = DEV_CONSOLE0 + page;
1980015         tty_table[page].pgrp = 0;
1980016         tty_table[page].key = 0;
1980017         tty_table[page].status = TTY_OK;
1980018     }
1980019     //
1980020     // Set video mode.
1980021     //
1980022     con_init ();
1980023     //
1980024     // Select the first console.
1980025     //
1980026     tty_console (DEV_CONSOLE0);
1980027     //
1980028     // Nothing else to configure (only consoles are available).
1980029     //
1980030     return;
1980031 }

```

1724

kernel/tty/tty_read.c

Si veda la sezione u0.9.

```

1990001 #include <sys/osi16.h>
1990002 #include <kernel/tty.h>
1990003 #include <kernel/k_libc.h>
1990004 //-----
1990005 int
1990006 tty_read (dev_t device)
1990007 {
1990008     tty_t *tty;
1990009     int key;
1990010     //
1990011     tty = tty_reference (device);
1990012     if (tty == NULL)
1990013     {
1990014         k_printf ("kernel alert: cannot find terminal device *
1990015                 "0x%04x\n", (int) device);
1990016         //
1990017         return (0);
1990018     }
1990019     //
1990020     // Read key and remove from the source.
1990021     //
1990022     key = tty->key;
1990023     tty->key = 0;
1990024     //
1990025     // Return the key.
1990026     //
1990027     return (key);
1990028 }
1990029 }

```

kernel/tty/tty_reference.c

Si veda la sezione u0.9.

```

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

```

kernel/tty/tty_table.c

Si veda la sezione u0.9.

```

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

```

kernel/tty/tty_write.c

Si veda la sezione u0.9.

```

2020001 #include <sys/osi16.h>
2020002 #include <kernel/tty.h>
2020003 //-----
2020004 void
2020005 tty_write (dev_t device, int c)
2020006 {
2020007     int console;
2020008     //
2020009     if ((device & 0xPF00) == (DEV_CONSOLE_MAJOR << 8))
2020010     {
2020011         console = (device & 0x00FF);
2020012         con_putc (console, c);
2020013     }
2020014 }

```

1725

os16: file isolati della directory «lib/»	1735
lib/NULL.h	1735
lib/SEEK.h	1735
lib/_Bool.h	1735
lib/clock_t.h	1736
lib/const.h	1736
lib/ctype.h	1736
lib/inttypes.h	1736
lib/limits.h	1738
lib/ptrdiff_t.h	1738
lib/restrict.h	1738
lib/size_t.h	1739
lib/stdarg.h	1739
lib/stdbool.h	1739
lib/stddef.h	1739
lib/stdint.h	1739
lib/time_t.h	1740
lib/wchar_t.h	1740
os16: «lib/dirent.h»	1741
lib/dirent/DIR.c	1741
lib/dirent/closedir.c	1741
lib/dirent/opendir.c	1742
lib/dirent/readdir.c	1743
lib/dirent/rewinddir.c	1744
os16: «lib/errno.h»	1744
lib/errno/errno.c	1746
os16: «lib/fcntl.h»	1747
lib/fcntl/creat.c	1748
lib/fcntl/fcntl.c	1748
lib/fcntl/open.c	1748
os16: «lib/grp.h»	1749
lib/grp/getgrgid.c	1749
lib/grp/getgrnam.c	1749
os16: «lib/libgen.h»	1749
lib/libgen/basename.c	1749
lib/libgen/dirname.c	1750
os16: «lib/pwd.h»	1751
lib/pwd/pwent.c	1751
os16: «lib/signal.h»	1752
lib/signal/kill.c	1753
lib/signal/signal.c	1753
os16: «lib/stdio.h»	1753
lib/stdio/FILE.c	1754
lib/stdio/clearerr.c	1755
lib/stdio/fclose.c	1755
lib/stdio/feof.c	1755
lib/stdio/ferror.c	1755
lib/stdio/fflush.c	1755
lib/stdio/fgetc.c	1755
lib/stdio/fgetpos.c	1756
lib/stdio/fgets.c	1756
lib/stdio/fileno.c	1756
lib/stdio/fopen.c	1757

lib/stdio/fprintf.c	1757
lib/stdio/fputc.c	1758
lib/stdio/fputs.c	1758
lib/stdio/fread.c	1758
lib/stdio/freopen.c	1758
lib/stdio/fscanf.c	1759
lib/stdio/fseek.c	1759
lib/stdio/fseeko.c	1759
lib/stdio/fsetpos.c	1760
lib/stdio/ftell.c	1760
lib/stdio/ftello.c	1760
lib/stdio/fwrite.c	1760
lib/stdio/getchar.c	1760
lib/stdio/gets.c	1761
lib/stdio/perror.c	1761
lib/stdio/printf.c	1762
lib/stdio/puts.c	1762
lib/stdio/rewind.c	1762
lib/stdio/scanf.c	1762
lib/stdio/setbuf.c	1762
lib/stdio/setvbuf.c	1762
lib/stdio/snprintf.c	1763
lib/stdio/sprintf.c	1763
lib/stdio/sscanf.c	1763
lib/stdio/vfprintf.c	1763
lib/stdio/vfscanf.c	1764
lib/stdio/vfscanf.c	1764
lib/stdio/vprintf.c	1777
lib/stdio/vscanf.c	1778
lib/stdio/vsnprintf.c	1778
lib/stdio/vsprintf.c	1788
lib/stdio/vsscanf.c	1788
os16: «lib/stdlib.h»	1788
lib/stdlib/_Exit.c	1789
lib/stdlib/abort.c	1789
lib/stdlib/abs.c	1789
lib/stdlib/alloc.c	1790
lib/stdlib/atexit.c	1792
lib/stdlib/atoi.c	1792
lib/stdlib/atol.c	1793
lib/stdlib/div.c	1793
lib/stdlib/environment.c	1793
lib/stdlib/exit.c	1794
lib/stdlib/getenv.c	1794
lib/stdlib/labs.c	1795
lib/stdlib/ldiv.c	1795
lib/stdlib/putenv.c	1795
lib/stdlib/qsort.c	1796
lib/stdlib/rand.c	1798
lib/stdlib/setenv.c	1798
lib/stdlib/strtol.c	1799
lib/stdlib/strtoul.c	1801
lib/stdlib/unsetenv.c	1801
os16: «lib/string.h»	1802
lib/string/memccpy.c	1803
lib/string/memchr.c	1803
lib/string/memcmp.c	1803

lib/string/memcpy.c	1803
lib/string/memmove.c	1803
lib/string/memset.c	1804
lib/string/strcat.c	1804
lib/string/strchr.c	1804
lib/string/strcmp.c	1805
lib/string/strcoll.c	1805
lib/string/strcpy.c	1805
lib/string/strcspn.c	1805
lib/string/strdup.c	1805
lib/string/strerror.c	1806
lib/string/strlen.c	1807
lib/string/strncat.c	1807
lib/string/strncmp.c	1807
lib/string/strncpy.c	1808
lib/string/strpbrk.c	1808
lib/string/strrchr.c	1808
lib/string/strspn.c	1808
lib/string/strstr.c	1809
lib/string/strtok.c	1809
lib/string/strxfrm.c	1810
os16: «lib/sys/os16.h»	1811
lib/sys/os16/_bp.s	1815
lib/sys/os16/_cs.s	1816
lib/sys/os16/_ds.s	1816
lib/sys/os16/_es.s	1816
lib/sys/os16/_seg_d.s	1816
lib/sys/os16/_seg_i.s	1816
lib/sys/os16/_sp.s	1816
lib/sys/os16/_ss.s	1817
lib/sys/os16/heap_clear.c	1817
lib/sys/os16/heap_min.c	1817
lib/sys/os16/input_line.c	1817
lib/sys/os16/mount.c	1818
lib/sys/os16/namep.c	1818
lib/sys/os16/process_info.c	1820
lib/sys/os16/sys.s	1820
lib/sys/os16/umount.c	1820
lib/sys/os16/z_perror.c	1820
lib/sys/os16/z_printf.c	1821
lib/sys/os16/z_putchar.c	1821
lib/sys/os16/z_puts.c	1821
lib/sys/os16/z_vprintf.c	1821
os16: «lib/sys/stat.h»	1821
lib/sys/stat/chmod.c	1823
lib/sys/stat/fchmod.c	1823
lib/sys/stat/fstat.c	1823
lib/sys/stat/mkdir.c	1824
lib/sys/stat/mknod.c	1824
lib/sys/stat/stat.c	1824
lib/sys/stat/umask.c	1825
os16: «lib/sys/types.h»	1825
lib/sys/types/major.c	1825
lib/sys/types/makedev.c	1825
lib/sys/types/minor.c	1826
os16: «lib/sys/wait.h»	1826
lib/sys/wait/wait.c	1826

os16: «lib/time.h»	1826
lib/time/asctime.c	1827
lib/time/clock.c	1828
lib/time/gmtime.c	1828
lib/time/mktime.c	1830
lib/time/stime.c	1831
lib/time/time.c	1831
os16: «lib/unistd.h»	1831
lib/unistd/_exit.c	1832
lib/unistd/access.c	1832
lib/unistd/chdir.c	1833
lib/unistd/chown.c	1833
lib/unistd/close.c	1834
lib/unistd/dup.c	1834
lib/unistd/dup2.c	1834
lib/unistd/envIRON.c	1834
lib/unistd/execl.c	1834
lib/unistd/execle.c	1835
lib/unistd/execlp.c	1835
lib/unistd/execv.c	1836
lib/unistd/execve.c	1836
lib/unistd/execvp.c	1837
lib/unistd/fchdir.c	1837
lib/unistd/fchown.c	1837
lib/unistd/fork.c	1837
lib/unistd/getcwd.c	1838
lib/unistd/geteuid.c	1838
lib/unistd/getopt.c	1839
lib/unistd/getpgrp.c	1841
lib/unistd/getpid.c	1841
lib/unistd/getppid.c	1841
lib/unistd/getuid.c	1842
lib/unistd/isatty.c	1842
lib/unistd/link.c	1842
lib/unistd/lseek.c	1842
lib/unistd/read.c	1843
lib/unistd/rmdir.c	1843
lib/unistd/seteuid.c	1844
lib/unistd/setpgrp.c	1844
lib/unistd/setuid.c	1844
lib/unistd/sleep.c	1845
lib/unistd/ttyname.c	1845
lib/unistd/unlink.c	1846
lib/unistd/write.c	1846
os16: «lib/utime.h»	1847
lib/utime/utime.c	1847
abort.c	1789
abs.c	1789
access.c	1832
alloc.c	1790
asctime.c	1827
atexit.c	1792
atoi.c	1792
atol.c	1793
basename.c	1749
chdir.c	1833
chmod.c	1823
chown.c	1833
clearerr.c	1755
clock.c	1828
clock_t.h	1736
close.c	1834
closedir.c	1741
const.h	1736
creat.c	1748
ctype.h	1736
DIR.c	1741
dirent.h	1741
dirname.c	1750
div.c	1793
dup.c	1834
dup2.c	1834
environ.c	1834
environment.c	1793
errno.c	1746
errno.h	1744
execl.c	1834
execle.c	1835
execlp.c	1835
execv.c	1836
execve.c	1836
execvp.c	1837
exit.c	1794
fchdir.c	1837
fchmod.c	1823
fchown.c	1837
fclose.c	1755
fcntl.c	1748
fcntl.h	1747
feof.c	1755
ferror.c	1755

fflush.c	1755
fgetc.c	1755
fgetpos.c	1756
fgets.c	1756
FILE.c	1754
fileno.c	1756
fopen.c	1757
fork.c	1837
fprintf.c	1757
fputc.c	1758
fputs.c	1758
fread.c	1758
freopen.c	1758
fscanf.c	1759
fseek.c	1759
fseeko.c	1759
fsetpos.c	1760
fstat.c	1823
ftell.c	1760
ftello.c	1760
fwrite.c	1760
getchar.c	1760
getcwd.c	1838
getenv.c	1794
geteuid.c	1838
getgrgid.c	1749
getgrnam.c	1749
getopt.c	1839
getpgrp.c	1841
getpid.c	1841
getppid.c	1841
gets.c	1761
getuid.c	1842
gmtime.c	1828
grp.h	1749
heap_clear.c	1817
heap_min.c	1817
input_line.c	1817
inttypes.h	1736
isatty.c	1842
kill.c	1753
labs.c	1795
ldiv.c	1795
libgen.h	1749
limits.h	1738
link.c	1842
lseek.c	1842
major.c	1825
makedev.c	1825
memcpy.c	1803
memchr.c	1803
memcmp.c	1803
memcpy.c	1803
memmove.c	1803
memset.c	1804
minor.c	1826
mkdir.c	1824
mknod.c	1824
mktime.c	1830
mount.c	1818
namep.c	1818
NULL.h	1735
open.c	1748
opendir.c	1742
os16.h	1811
perror.c	1761
printf.c	1762
process_info.c	1820
ptrdiff_t.h	1738
putenv.c	1795
puts.c	1762
pwd.h	1751
pwent.c	1751
qsort.c	1796
rand.c	1798
read.c	1843
readdir.c	1743
restrict.h	1738
rewind.c	1762
rewinddir.c	1744
rmdir.c	1843
scanf.c	1762
SEEK.h	1735
setbuf.c	1762
setenv.c	1798
seteuid.c	1844
setpgrp.c	1844
setuid.c	1844
setvbuf.c	1762
signal.c	1753
signal.h	1752
size_t.h	1739
sleep.c	1845
snprintf.c	1763
sprintf.c	1763
sscanf.c	1763
stat.c	1824
stat.h	1821
stdarg.h	1739
stdbool.h	1739
stddef.h	1739
stdint.h	1739
stdio.h	1753
stdlib.h	1788
stime.c	1831
strcat.c	1804
strchr.c	1804
strcmp.c	1805
strcoll.c	1805
strcpy.c	1805
strcspn.c	1805
strdup.c	1805
strerror.c	1806
string.h	1802
strlen.c	1807
strncat.c	1807
strncmp.c	1807
strncpy.c	1808
strpbrk.c	1808
strchr.c	1808
strspn.c	1808
strstr.c	1809
strtok.c	1809
strtol.c	1799
strtoul.c	1801
strxfrm.c	1810
sys.s	1820
time.c	1831
time.h	1826
time_t.h	1740
ttyname.c	1845
types.h	1825
umask.c	1825
umount.c	1820
unistd.h	1831
unlink.c	1846
unsetenv.c	1801
utime.c	1847
utime.h	1847
vfprintf.c	1763
vfscanf.c	1764
vfsscanf.c	1764
vprintf.c	1777
vscanf.c	1778
vsprintf.c	1778
vsscanf.c	1778
vsnprintf.c	1778
vsprintf.c	1788
vsscanf.c	1788
wait.c	1826
wait.h	1826
wchar_t.h	1740
write.c	1846
z_perror.c	1820
z_printf.c	1821
z_putchar.c	1821
z_puts.c	1821
z_vprintf.c	1821
_Bool.h	1735
_bp.s	1815
_cs.s	1816
_ds.s	1816
_es.s	1816
_exit.c	1832
_Exit.c	1789
_seg_d.s	1816
_seg_i.s	1816
_sp.s	1816
_ss.s	1817

os16: file isolati della directory «lib/»

lib/NULL.h	1735
lib/SEEK.h	1735
lib/_Bool.h	1735
lib/clock_t.h	1736
lib/const.h	1736
lib/ctype.h	1736
lib/inttypes.h	1736
lib/limits.h	1738
lib/ptrdiff_t.h	1738
lib/restrict.h	1738
lib/size_t.h	1739
lib/stdarg.h	1739
lib/stdbool.h	1739
lib/stddef.h	1739

lib/stdint.h	1739
lib/time_t.h	1740
lib/wchar_t.h	1740
os16: «lib/dirent.h»	1741
lib/dirent/DIR.c	1741
lib/dirent/closedir.c	1741
lib/dirent/opendir.c	1742
lib/dirent/readdir.c	1743
lib/dirent/rewinddir.c	1744
os16: «lib/errno.h»	1744
lib/errno/errno.c	1746
os16: «lib/fcntl.h»	1747
lib/fcntl/creat.c	1748
lib/fcntl/fcntl.c	1748
lib/fcntl/open.c	1748
os16: «lib/grp.h»	1749
lib/grp/getgrgid.c	1749
lib/grp/getgrnam.c	1749
os16: «lib/libgen.h»	1749
lib/libgen/basename.c	1749
lib/libgen/dirname.c	1750
os16: «lib/pwd.h»	1751
lib/pwd/pwent.c	1751
os16: «lib/signal.h»	1752
lib/signal/kill.c	1753
lib/signal/signal.c	1753
os16: «lib/stdio.h»	1753
lib/stdio/FILE.c	1754
lib/stdio/clearerr.c	1755
lib/stdio/fclose.c	1755
lib/stdio/feof.c	1755
lib/stdio/ferror.c	1755
lib/stdio/fflush.c	1755
lib/stdio/fgetc.c	1755
lib/stdio/fgetpos.c	1756
lib/stdio/fgets.c	1756
lib/stdio/fileno.c	1756
lib/stdio/fopen.c	1757
lib/stdio/fprintf.c	1757
lib/stdio/fputc.c	1758
lib/stdio/fputs.c	1758
lib/stdio/fread.c	1758
lib/stdio/freopen.c	1758
lib/stdio/fscanf.c	1759
lib/stdio/fseek.c	1759
lib/stdio/fseeko.c	1759
lib/stdio/fsetpos.c	1760
lib/stdio/ftell.c	1760
lib/stdio/ftello.c	1760
lib/stdio/fwrite.c	1760
lib/stdio/getchar.c	1760
lib/stdio/gets.c	1761
lib/stdio/perror.c	1761
lib/stdio/printf.c	1762
lib/stdio/puts.c	1762

lib/stdio/rewind.c	1762
lib/stdio/scanf.c	1762
lib/stdio/setbuf.c	1762
lib/stdio/setvbuf.c	1762
lib/stdio/snprintf.c	1763
lib/stdio/sprintf.c	1763
lib/stdio/sscanf.c	1763
lib/stdio/vfprintf.c	1763
lib/stdio/vfscanf.c	1764
lib/stdio/vfscanf.c	1764
lib/stdio/vprintf.c	1777
lib/stdio/vscanf.c	1778
lib/stdio/vsnprintf.c	1778
lib/stdio/vsprintf.c	1788
lib/stdio/vsscanf.c	1788
os16: «lib/stdlib.h»	1788
lib/stdlib/_Exit.c	1789
lib/stdlib/abort.c	1789
lib/stdlib/abs.c	1789
lib/stdlib/alloc.c	1790
lib/stdlib/atexit.c	1792
lib/stdlib/atoi.c	1792
lib/stdlib/atol.c	1793
lib/stdlib/div.c	1793
lib/stdlib/environment.c	1793
lib/stdlib/exit.c	1794
lib/stdlib/getenv.c	1794
lib/stdlib/labs.c	1795
lib/stdlib/ldiv.c	1795
lib/stdlib/putenv.c	1795
lib/stdlib/qsort.c	1796
lib/stdlib/rand.c	1798
lib/stdlib/setenv.c	1798
lib/stdlib/strtol.c	1799
lib/stdlib/strtoul.c	1801
lib/stdlib/unsetenv.c	1801
os16: «lib/string.h»	1802
lib/string/memccpy.c	1803
lib/string/memchr.c	1803
lib/string/memcmp.c	1803
lib/string/memcpy.c	1803
lib/string/memmove.c	1803
lib/string/memset.c	1804
lib/string/strcat.c	1804
lib/string/strchr.c	1804
lib/string/stremp.c	1805
lib/string/strcoll.c	1805
lib/string/strcpy.c	1805
lib/string/strcspn.c	1805
lib/string/strdup.c	1805
lib/string/strerror.c	1806
lib/string/strlen.c	1807
lib/string/strncat.c	1807
lib/string/strncmp.c	1807
lib/string/strncpy.c	1808
lib/string/strpbrk.c	1808
lib/string/strrchr.c	1808

lib/string/strspn.c	1808	lib/unistd/execl.c	1834
lib/string/strchr.c	1809	lib/unistd/execlp.c	1835
lib/string/strtok.c	1809	lib/unistd/execl.c	1835
lib/string/strxfrm.c	1810	lib/unistd/execv.c	1836
os16: <lib/sys/os16.h>	1811	lib/unistd/execve.c	1836
lib/sys/os16/_bp.s	1815	lib/unistd/execvp.c	1837
lib/sys/os16/_cs.s	1816	lib/unistd/fchdir.c	1837
lib/sys/os16/_ds.s	1816	lib/unistd/fchown.c	1837
lib/sys/os16/_es.s	1816	lib/unistd/fork.c	1837
lib/sys/os16/_seg_d.s	1816	lib/unistd/getcwd.c	1838
lib/sys/os16/_seg_i.s	1816	lib/unistd/geteuid.c	1838
lib/sys/os16/_sp.s	1816	lib/unistd/getopt.c	1839
lib/sys/os16/_ss.s	1817	lib/unistd/getpgrp.c	1841
lib/sys/os16/heap_clear.c	1817	lib/unistd/getpid.c	1841
lib/sys/os16/heap_min.c	1817	lib/unistd/getppid.c	1841
lib/sys/os16/input_line.c	1817	lib/unistd/getuid.c	1842
lib/sys/os16/mount.c	1818	lib/unistd/isatty.c	1842
lib/sys/os16/namep.c	1818	lib/unistd/link.c	1842
lib/sys/os16/process_info.c	1820	lib/unistd/lseek.c	1842
lib/sys/os16/sys.s	1820	lib/unistd/read.c	1843
lib/sys/os16/umount.c	1820	lib/unistd/rmdir.c	1843
lib/sys/os16/z_perror.c	1820	lib/unistd/seteuid.c	1844
lib/sys/os16/z_printf.c	1821	lib/unistd/setpgrp.c	1844
lib/sys/os16/z_putchar.c	1821	lib/unistd/setuid.c	1844
lib/sys/os16/z_puts.c	1821	lib/unistd/sleep.c	1845
lib/sys/os16/z_vprintf.c	1821	lib/unistd/ttyname.c	1845
os16: <lib/sys/stat.h>	1821	lib/unistd/unlink.c	1846
lib/sys/stat/chmod.c	1823	lib/unistd/write.c	1846
lib/sys/stat/fchmod.c	1823	os16: <lib/utime.h>	1847
lib/sys/stat/fstat.c	1823	lib/utime/utime.c	1847
lib/sys/stat/mkdir.c	1824		
lib/sys/stat/mknod.c	1824	os16: file isolati della directory «lib/»	«
lib/sys/stat/stat.c	1824	lib/NULL.h	«
lib/sys/stat/umask.c	1825	Si veda la sezione u0.2.	
os16: <lib/sys/types.h>	1825	<pre>2303001 #ifndef _NULL_H 2303002 #define _NULL_H 1 2303003 2303004 #define NULL 0 2303005 2303006 #endif</pre>	
lib/sys/types/major.c	1825	lib/SEEK.h	«
lib/sys/types/makedev.c	1825	Si veda la sezione u0.2.	
lib/sys/types/minor.c	1826	<pre>2304001 #ifndef _SEEK_H 2304002 #define _SEEK_H 1 2304003 2304004 //----- 2304005 // These values are used inside <stdio.h> and <unistd.h> 2304006 //----- 2304007 #define SEEK_SET 0 // From the start. 2304008 #define SEEK_CUR 1 // From current position. 2304009 #define SEEK_END 2 // From the end. 2304010 //----- 2304011 2304012 #endif</pre>	
os16: <lib/sys/wait.h>	1826	lib/_Bool.h	«
lib/sys/wait/wait.c	1826	Si veda la sezione u0.2.	
os16: <lib/time.h>	1826	<pre>2305001 #ifndef _BOOL_H 2305002 #define _BOOL_H 1 2305003 2305004 typedef unsigned char _Bool; 2305005 2305006 #endif</pre>	
lib/time/asctime.c	1827		
lib/time/clock.c	1828		
lib/time/gmtime.c	1828		
lib/time/mktime.c	1830		
lib/time/stime.c	1831		
lib/time/time.c	1831		
os16: <lib/unistd.h>	1831		
lib/unistd/_exit.c	1832		
lib/unistd/access.c	1832		
lib/unistd/chdir.c	1833		
lib/unistd/chown.c	1833		
lib/unistd/close.c	1834		
lib/unistd/dup.c	1834		
lib/unistd/dup2.c	1834		
lib/unistd/envIRON.c	1834		

lib/clock_t.h

<

Si veda la sezione u0.2.

```
2060001 #ifndef _CLOCK_T_H
2060002 #define _CLOCK_T_H 1
2060003
2060004 typedef unsigned long int clock_t; // 32 bit unsigned int.
2060005
2060006 #endif
```

lib/const.h

<

Si veda la sezione u0.2.

```
2070001 #ifndef _CONST_H
2070002 #define _CONST_H 1
2070003
2070004 #define const
2070005
2070006 #endif
```

lib/ctype.h

<

Si veda la sezione u0.2.

```
2080001 #ifndef _CTYPE_H
2080002 #define _CTYPE_H 1
2080003 //-----
2080004
2080005 #include <NULL.h>
2080006 //-----
2080007 #define isblank(C) ((int) (C == ' ' || C == '\t'))
2080008 #define isspace(C) ((int) (C == ' ' \
2080009 | C == '\f' \
2080010 | C == '\n' \
2080011 | C == '\r' \
2080012 | C == '\t' \
2080013 | C == '\v'))
2080014
2080015 #define isdigit(C) ((int) (C >= '0' && C <= '9'))
2080016 #define isxdigit(C) ((int) ((C >= '0' && C <= '9' \
2080017 | (C >= 'A' && C <= 'F') \
2080018 | (C >= 'a' && C <= 'f'))))
2080019
2080020 #define isupper(C) ((int) (C >= 'A' && C <= 'Z'))
2080021 #define islower(C) ((int) (C >= 'a' && C <= 'z'))
2080022 #define iscntrl(C) ((int) ((C >= 0x00 && C <= 0x1F) || C == 0x7F))
2080023 #define isgraph(C) ((int) (C >= 0x21 && C <= 0x7E))
2080024 #define isprint(C) ((int) (C >= 0x20 && C <= 0x7E))
2080025 #define isalpha(C) (isupper(C) || islower(C))
2080026 #define isalnum(C) (isalpha(C) || isdigit(C))
2080027 #define ispunct(C) (isgraph(C) && (!isspace(C)) && (!isalnum(C)))
2080028 #define tolower(C) (isupper(C) ? ((C) + 0x20) : (C))
2080029 #define toupper(C) (islower(C) ? ((C) - 0x20) : (C))
2080030 #define toascii(C) (C & 0x7F)
2080031 #define _tolower(C) (isupper(C) ? ((C) + 0x20) : (C))
2080032 #define _toupper(C) (islower(C) ? ((C) - 0x20) : (C))
2080033 //-----
2080034 #endif
```

lib/inttypes.h

<

Si veda la sezione u0.2.

```
2090001 #ifndef _INTTYPES_H
2090002 #define _INTTYPES_H 1
2090003 //-----
2090004
2090005 #include <const.h>
2090006 #include <restrict.h>
2090007 #include <stdint.h>
2090008 #include <wchar_t.h>
2090009 //-----
2090010 typedef struct {
2090011     intmax_t quot;
2090012     intmax_t rem;
2090013 } imaxdiv_t;
2090014 //
2090015 imaxdiv_t imaxdiv (intmax_t numer, intmax_t denom);
2090016 //-----
2090017 // Output typesetting.
2090018 //-----
2090019 #define PRId8 "d"
2090020 #define PRId16 "d"
2090021 #define PRId32 "ld"
2090022 #define PRIdLEAST8 "d"
2090023 #define PRIdLEAST16 "d"
2090024 #define PRIdLEAST32 "ld"
2090025 #define PRIdFAST8 "d"
2090026 #define PRIdFAST16 "d"
2090027 #define PRIdFAST32 "ld"
2090028 #define PRIdMAX "ld"
2090029 #define PRIdPTR "d"
2090030 #define PRIi8 "i"
2090031 #define PRIi16 "i"
2090032 #define PRIi32 "li"
2090033 #define PRIiLEAST8 "i"
2090034 #define PRIiLEAST16 "li"
```

```
2090035 #define PRIiLEAST32 "li"
2090036 #define PRIiFAST8 "i"
2090037 #define PRIiFAST16 "i"
2090038 #define PRIiFAST32 "i"
2090039 #define PRIiMAX "li"
2090040 #define PRIiPTR "i"
2090041 #define PRIo8 "o"
2090042 #define PRIo16 "o"
2090043 #define PRIo32 "lo"
2090044 #define PRIoLEAST8 "o"
2090045 #define PRIoLEAST16 "o"
2090046 #define PRIoLEAST32 "lo"
2090047 #define PRIoFAST8 "o"
2090048 #define PRIoFAST16 "o"
2090049 #define PRIoFAST32 "lo"
2090050 #define PRIoMAX "lo"
2090051 #define PRIoPTR "o"
2090052 #define PRIu8 "u"
2090053 #define PRIu16 "u"
2090054 #define PRIu32 "lu"
2090055 #define PRIuLEAST8 "u"
2090056 #define PRIuLEAST16 "u"
2090057 #define PRIuLEAST32 "lu"
2090058 #define PRIuFAST8 "u"
2090059 #define PRIuFAST16 "u"
2090060 #define PRIuFAST32 "lu"
2090061 #define PRIuMAX "lu"
2090062 #define PRIuPTR "u"
2090063 #define PRIx8 "x"
2090064 #define PRIx16 "x"
2090065 #define PRIx32 "lx"
2090066 #define PRIxLEAST8 "x"
2090067 #define PRIxLEAST16 "x"
2090068 #define PRIxLEAST32 "lx"
2090069 #define PRIxFAST8 "x"
2090070 #define PRIxFAST16 "x"
2090071 #define PRIxFAST32 "lx"
2090072 #define PRIxMAX "lx"
2090073 #define PRIxPTR "x"
2090074 #define PRIX8 "X"
2090075 #define PRIX16 "X"
2090076 #define PRIX32 "LX"
2090077 #define PRIXLEAST8 "X"
2090078 #define PRIXLEAST16 "X"
2090079 #define PRIXLEAST32 "LX"
2090080 #define PRIXFAST8 "X"
2090081 #define PRIXFAST16 "X"
2090082 #define PRIXFAST32 "LX"
2090083 #define PRIXMAX "LX"
2090084 #define PRIXPTR "X"
2090085 //-----
2090086 // Input scan and evaluation.
2090087 //-----
2090088 #define SCNd8 "hhd"
2090089 #define SCNd16 "hd"
2090090 #define SCNd32 "d"
2090091 #define SCNdLEAST8 "hhd"
2090092 #define SCNdLEAST16 "hd"
2090093 #define SCNdLEAST32 "d"
2090094 #define SCNdFAST8 "hhd"
2090095 #define SCNdFAST16 "d"
2090096 #define SCNdFAST32 "d"
2090097 #define SCNdMAX "ld"
2090098 #define SCNdPTR "d"
2090099 #define SCNi8 "hhi"
2090100 #define SCNi16 "hi"
2090101 #define SCNi32 "i"
2090102 #define SCNiLEAST8 "hhi"
2090103 #define SCNiLEAST16 "hi"
2090104 #define SCNiLEAST32 "i"
2090105 #define SCNiFAST8 "hhi"
2090106 #define SCNiFAST16 "i"
2090107 #define SCNiFAST32 "i"
2090108 #define SCNiMAX "li"
2090109 #define SCNiPTR "i"
2090110 #define SCNo8 "hho"
2090111 #define SCNo16 "ho"
2090112 #define SCNo32 "o"
2090113 #define SCNoLEAST8 "hho"
2090114 #define SCNoLEAST16 "ho"
2090115 #define SCNoLEAST32 "o"
2090116 #define SCNoFAST8 "hho"
2090117 #define SCNoFAST16 "o"
2090118 #define SCNoFAST32 "o"
2090119 #define SCNoMAX "lo"
2090120 #define SCNoPTR "o"
2090121 #define SCNu8 "hhu"
2090122 #define SCNu16 "hu"
2090123 #define SCNu32 "u"
2090124 #define SCNuLEAST8 "hhu"
2090125 #define SCNuLEAST16 "hu"
2090126 #define SCNuLEAST32 "u"
2090127 #define SCNuFAST8 "hhu"
2090128 #define SCNuFAST16 "u"
2090129 #define SCNuFAST32 "u"
2090130 #define SCNuMAX "lu"
2090131 #define SCNuPTR "u"
2090132 #define SCNx8 "hhx"
2090133 #define SCNx16 "hx"
2090134 #define SCNx32 "x"
2090135 #define SCNxLEAST8 "hhx"
```

```

2090136 #define SCNxLEAST16 "hx"
2090137 #define SCNxLEAST32 "x"
2090138 #define SCNxFAST8 "hx"
2090139 #define SCNxFAST16 "x"
2090140 #define SCNxFAST32 "x"
2090141 #define SCNxMAX "lx"
2090142 #define SCNxPTR "x"
2090143 //-----
2090144 intmax_t strtoumax (const char *restrict nptr,
2090145 char **restrict endptr, int base);
2090146 uintmax_t strtoumax (const char *restrict nptr,
2090147 char **restrict endptr, int base);
2090148 intmax_t wcstoumax (const wchar_t *restrict nptr,
2090149 wchar_t **restrict endptr, int base);
2090150 uintmax_t wcstoumax (const wchar_t *restrict nptr,
2090151 wchar_t **restrict endptr, int base);
2090152 //-----
2090153
2090154 #endif

```

lib/limits.h

Si veda la sezione u0.2.

```

2100001 #ifndef _LIMITS_H
2100002 #define _LIMITS_H 1
2100003 //-----
2100004 #define CHAR_BIT (8)
2100005 #define SCHAR_MIN (-0x80)
2100006 #define SCHAR_MAX (0x7F)
2100007 #define UCHAR_MAX (0xFF)
2100008 #define CHAR_MIN SCHAR_MIN
2100009 #define CHAR_MAX SCHAR_MAX
2100010 #define MB_LEN_MAX (1)
2100011 #define SHRT_MIN (-0x8000)
2100012 #define SHRT_MAX (0x7FFF)
2100013 #define USHRT_MAX (0xFFFF)
2100014 #define INT_MIN (-0x8000)
2100015 #define INT_MAX (0x7FFF)
2100016 #define UINT_MAX (0xFFFF)
2100017 #define LONG_MIN (-0x80000000L)
2100018 #define LONG_MAX (0x7FFFFFFFL)
2100019 #define ULONG_MAX (0xFFFFFFFFUL)
2100020 //-----
2100021 #define LLONG_MIN (-0x80000000L) // The type 'long long int'
2100022 #define LLONG_MAX (0x7FFFFFFFL) // is not available with
2100023 #define ULLONG_MAX (0xFFFFFFFFUL) // a K&R C compiler.
2100024 //-----
2100025 #define WORD_BIT 16 // POSIX requires at least 32!
2100026 #define LONG_BIT 32
2100027 #define SSIZE_MAX LONG_MAX
2100028 //-----
2100029 #define ARG_MAX 1024 // Arguments+environment max length.
2100030 #define ATEXIT_MAX 32 // Max "at exit" functions.
2100031 #define FILESIZEBITS 32 // File size needs integer size...
2100032 #define LINK_MAX 254 // Max links per file.
2100033 #define NAME_MAX 14 // File name max (Minix 1 fa).
2100034 #define OPEN_MAX 8 // Max open files per process.
2100035 #define PATH_MAX 64 // Path, including final '\0'.
2100036 #define MAX_CANON 1 // Max bytes in canonical tty queue.
2100037 #define MAX_INPUT 1 // Max bytes in tty input queue.
2100038 //-----
2100039 #define CHLD_MAX INT_MAX // Not used.
2100040 #define HOST_NAME_MAX INT_MAX // Not used.
2100041 #define LOGIN_NAME_MAX INT_MAX // Not used.
2100042 #define PAGE_SIZE INT_MAX // Not used.
2100043 #define RE_DUP_MAX INT_MAX // Not used.
2100044 #define STREAM_MAX INT_MAX // Not used.
2100045 #define SYMLINK_MAX INT_MAX // Not used.
2100046 #define TTY_NAME_MAX INT_MAX // Not used.
2100047 #define TZNAME_MAX INT_MAX // Not used.
2100048 #define PIPE_MAX INT_MAX // Not used.
2100049 #define SYMLINK_MAX INT_MAX // Not used.
2100050 //-----
2100051
2100052 #endif

```

lib/ptrdiff_t.h

Si veda la sezione u0.2.

```

2110001 #ifndef _PTRDIFF_T_H
2110002 #define _PTRDIFF_T_H 1
2110003
2110004 typedef int ptrdiff_t;
2110005
2110006 #endif

```

lib/restrict.h

Si veda la sezione u0.2.

```

2120001 #ifndef _RESTRICT_H
2120002 #define _RESTRICT_H 1
2120003
2120004 #define restrict
2120005
2120006 #endif

```

lib/size_t.h

Si veda la sezione u0.2.

```

2130001 #ifndef _SIZE_T_H
2130002 #define _SIZE_T_H 1
2130003 //
2130004 // The type 'size_t' *must* be equal to an 'int'.
2130005 //
2130006 typedef unsigned int size_t;
2130007
2130008 #endif

```

lib/stdarg.h

Si veda la sezione u0.2.

```

2140001 #ifndef _STDARG_H
2140002 #define _STDARG_H 1
2140003 //-----
2140004 typedef unsigned char *va_list;
2140005 //-----
2140006 #define va_start(ap, last) ((void) ((ap) = \
2140007 ((va_list) &(last)) + (sizeof (last))))
2140008
2140009 #define va_end(ap) ((void) ((ap) = 0))
2140010 #define va_copy(dest, src) ((void) ((dest) = (va_list) (src)))
2140011 #define va_arg(ap, type) (((ap) = (ap) + (sizeof (type))), \
2140012 *((type *) ((ap) - (sizeof (type)))))
2140013 //-----
2140014
2140015 #endif

```

lib/stdbool.h

Si veda la sezione u0.2.

```

2150001 #ifndef _STDBOOL_H
2150002 #define _STDBOOL_H 1
2150003 //-----
2150004 typedef unsigned char bool; // [1]
2150005 #define true ((bool) 1)
2150006 #define false ((bool) 0)
2150007 #define __bool_true_false_are_defined 1
2150008 //
2150009 // [1] For some reason, it cannot be defined as a macro expanding to
2150010 // 'bool'. Anyway, it is the same kind of type.
2150011 //
2150012 //-----
2150013
2150014 #endif

```

lib/stddef.h

Si veda la sezione u0.2.

```

2160001 #ifndef _STDDEF_H
2160002 #define _STDDEF_H 1
2160003 //-----
2160004
2160005 #include <ptrdiff_t.h>
2160006 #include <size_t.h>
2160007 #include <wchar_t.h>
2160008 #include <NULL.h>
2160009 //-----
2160010 #define offsetof(type, member) ((size_t) &((type *)0)-member)
2160011 //-----
2160012
2160013 #endif

```

lib/stdint.h

Si veda la sezione u0.2.

```

2170001 #ifndef _STDINT_H
2170002 #define _STDINT_H 1
2170003 //-----
2170004 typedef signed char int8_t;
2170005 typedef short int int16_t;
2170006 typedef long int int32_t;
2170007 typedef unsigned char uint8_t;
2170008 typedef unsigned short int uint16_t;
2170009 typedef unsigned long int uint32_t;
2170010 //
2170011 #define INT8_MIN (-0x80)
2170012 #define INT8_MAX (0x7F)
2170013 #define UINT8_MAX (0xFF)
2170014 #define INT16_MIN (-0x8000)
2170015 #define INT16_MAX (0x7FFF)
2170016 #define UINT16_MAX (0xFFFF)
2170017 #define INT32_MIN (-0x80000000)
2170018 #define INT32_MAX (0x7FFFFFFF)
2170019 #define UINT32_MAX (0xFFFFFFFF)
2170020 //-----
2170021 typedef signed char int_least8_t;
2170022 typedef short int int_least16_t;
2170023 typedef long int int_least32_t;
2170024 typedef unsigned char uint_least8_t;

```



```

2170025 typedef unsigned short int    uint_least16_t;
2170026 typedef unsigned long int    uint_least32_t;
2170027 //
2170028 #define INT_LEAST8_MIN        (-0x80)
2170029 #define INT_LEAST8_MAX        (0x7F)
2170030 #define UINT_LEAST8_MAX      (0xFF)
2170031 #define INT_LEAST16_MIN       (-0x8000)
2170032 #define INT_LEAST16_MAX      (0x7FFF)
2170033 #define UINT_LEAST16_MAX     (0xFFFF)
2170034 #define INT_LEAST32_MIN       (-0x80000000)
2170035 #define INT_LEAST32_MAX      (0x7FFFFFFF)
2170036 #define UINT_LEAST32_MAX     (0xFFFFFFFFU)
2170037 //-----
2170038 #define INT8_C(VAL)           VAL
2170039 #define INT16_C(VAL)          VAL
2170040 #define INT32_C(VAL)          VAL
2170041 #define UINT8_C(VAL)         VAL
2170042 #define UINT16_C(VAL)        VAL
2170043 #define UINT32_C(VAL)        VAL ## U
2170044 //-----
2170045 typedef signed char          int_fast8_t;
2170046 typedef int                  int_fast16_t;
2170047 typedef long int             int_fast32_t;
2170048 typedef unsigned char        uint_fast8_t;
2170049 typedef unsigned int         uint_fast16_t;
2170050 typedef unsigned long int    uint_fast32_t;
2170051 //
2170052 #define INT_FAST8_MIN        (-0x80)
2170053 #define INT_FAST8_MAX        (0x7F)
2170054 #define UINT_FAST8_MAX      (0xFF)
2170055 #define INT_FAST16_MIN       (-0x80000000)
2170056 #define INT_FAST16_MAX      (0x7FFFFFFF)
2170057 #define UINT_FAST16_MAX     (0xFFFFFFFFU)
2170058 #define INT_FAST32_MIN       (-0x80000000)
2170059 #define INT_FAST32_MAX      (0x7FFFFFFF)
2170060 #define UINT_FAST32_MAX     (0xFFFFFFFFU)
2170061 //-----
2170062 typedef int                  intptr_t;
2170063 typedef unsigned int         uintptr_t;
2170064 //
2170065 #define INTPTR_MIN           (-0x80000000)
2170066 #define INTPTR_MAX           (0x7FFFFFFF)
2170067 #define UINTPTR_MAX          (0xFFFFFFFFU)
2170068 //-----
2170069 typedef long int             intmax_t;
2170070 typedef unsigned long int    uintmax_t;
2170071 //
2170072 #define INTMAX_C(VAL)        VAL ## L
2170073 #define UINTMAX_C(VAL)      VAL ## UL
2170074 //
2170075 #define INTMAX_MIN           (-0x80000000L)
2170076 #define INTMAX_MAX           (0x7FFFFFFFL)
2170077 #define UINTMAX_MAX          (0xFFFFFFFFUL)
2170078 //-----
2170079 #define PTRDIFF_MIN          (-0x80000000)
2170080 #define PTRDIFF_MAX          (0x7FFFFFFF)
2170081 //
2170082 #define SIG_ATOMIC_MIN       (-0x80000000)
2170083 #define SIG_ATOMIC_MAX       (0x7FFFFFFF)
2170084 //
2170085 #define SIZE_MAX             (0xFFFFU)
2170086 //
2170087 #define WCHAR_MIN            (0)
2170088 #define WCHAR_MAX            (0xFFU)
2170089 //
2170090 #define WINT_MIN             (-0x80L)
2170091 #define WINT_MAX             (0x7FL)
2170092 //-----
2170093
2170094 #endif

```

lib/time_t.h

Si veda la sezione u0.2.

```

2180001 #ifndef _TIME_T_H
2180002 #define _TIME_T_H        1
2180003
2180004 typedef long int time_t;
2180005
2180006 #endif

```

lib/wchar_t.h

Si veda la sezione u0.2.

```

2190001 #ifndef _WCHAR_T_H
2190002 #define _WCHAR_T_H      1
2190003
2190004 typedef unsigned char wchar_t;
2190005
2190006 #endif

```

os16: «lib/dirent.h»

Si veda la sezione u0.2.

```

2300001 #ifndef _DIRENT_H
2300002 #define _DIRENT_H        1
2300003
2300004 #include <sys/types.h> // ino_t
2300005 #include <limits.h>    // NAME_MAX
2300006 #include <const.h>
2300007
2300008 //-----
2300009 struct dirent {
2300010     ino_t d_ino; // I-node number [1]
2300011     char d_name[NAME_MAX+1]; // NAME_MAX + Null termination
2300012 };
2300013 //
2300014 // [1] The type 'ino_t' must be equal to 'uint16_t', because the
2300015 // directory inside the Minix 1 file system has exactly such
2300016 // size.
2300017 //
2300018 //-----
2300019 #define DOPEN_MAX OPEN_MAX/2 // <limits.h> [1]
2300020 //
2300021 // [1] DOPEN_MAX is not standard, but it is used to define how many
2300022 // directory slot to keep for open directories. As directory streams
2300023 // are opened as file descriptors, the sum of all kind of file open
2300024 // cannot be more than OPEN_MAX.
2300025 //-----
2300026 typedef struct {
2300027     int fdn; // File descriptor number.
2300028     struct dirent dir; // Last directory item read.
2300029 } DIR;
2300030
2300031 extern DIR _directory_stream[]; // Defined inside 'lib/dirent/DIR.c'.
2300032 //-----
2300033 // Function prototypes.
2300034 //-----
2300035 int closedir (DIR *dp);
2300036 DIR *opendir (const char *name);
2300037 struct dirent *readdir (DIR *dp);
2300038 void rewinddir (DIR *dp);
2300039 //-----
2300040
2300041 #endif

```

lib/dirent/DIR.c

Si veda la sezione u0.2.

```

2210001 #include <dirent.h>
2210002 //
2210003 // There must be room for at least 'DOPEN_MAX' elements.
2210004 //
2210005 DIR _directory_stream[DOPEN_MAX];
2210006
2210007 void
2210008 _dirent_directory_stream_setup (void)
2210009 {
2210010     int d;
2210011     //
2210012     for (d = 0; d < DOPEN_MAX; d++)
2210013     {
2210014         _directory_stream[d].fdn = -1;
2210015     }
2210016 }

```

lib/dirent/closedir.c

Si veda la sezione u0.10.

```

2220001 #include <dirent.h>
2220002 #include <fcntl.h>
2220003 #include <const.h>
2220004 #include <sys/types.h>
2220005 #include <sys/stat.h>
2220006 #include <unistd.h>
2220007 #include <errno.h>
2220008 #include <stddef.h>
2220009 //-----
2220010 int
2220011 closedir (DIR *dp)
2220012 {
2220013     //
2220014     // Check for a valid argument
2220015     //
2220016     if (dp == NULL)
2220017     {
2220018         //
2220019         // Not a valid pointer.
2220020         //
2220021         errset (EBADF); // Invalid directory.
2220022         return (-1);
2220023     }
2220024     //
2220025     // Check if it is an open directory stream.
2220026     //
2220027     if (dp->fdn < 0)
2220028     {
2220029         //

```

```

223000 // The stream is closed.
223001 //
223002     errset (EBADF); // Invalid directory.
223003     return (-1);
223004 }
223005 //
223006 // Close the file descriptor. If there is an error,
223007 // the 'errno' variable will be set by 'close()'.
223008 //
223009     return (close (dp->fdn));
223010 }

```

```

223006 // Return the directory pointer.
223007 //
223008     return (dp);
223009 }
223010 }
223011 //
223012 // If we are here, there was no free directory slot available.
223013 //
223014     close (fdn);
223015     errset (EMFILE); // Too many file open.
223016     return (NULL);
223017 }

```

lib/dirent/ opendir.c

Si veda la sezione u0.76.

```

223001 #include <dirent.h>
223002 #include <fcntl.h>
223003 #include <const.h>
223004 #include <sys/types.h>
223005 #include <sys/stat.h>
223006 #include <unistd.h>
223007 #include <errno.h>
223008 #include <stddef.h>
223009 //-----
223010 DIR *
223011 opendir (const char *path)
223012 {
223013     int fdn;
223014     int d;
223015     DIR *dp;
223016     struct stat file_status;
223017     //
223018     // Function 'opendir()' is used only for reading.
223019     //
223020     fdn = open (path, O_RDONLY);
223021     //
223022     // Check the file descriptor returned.
223023     //
223024     if (fdn < 0)
223025     {
223026         //
223027         // The variable 'errno' is already set:
223028         //     EINVAL
223029         //     EMFILE
223030         //     ENFILE
223031         //
223032         errset (errno);
223033         return (NULL);
223034     }
223035     //
223036     // Set the 'FD_CLOEXEC' flag for that file descriptor.
223037     //
223038     if (fcntl (fdn, F_SETFD, FD_CLOEXEC) != 0)
223039     {
223040         //
223041         // The variable 'errno' is already set:
223042         //     EBADF
223043         //
223044         errset (errno);
223045         close (fdn);
223046         return (NULL);
223047     }
223048     //
223049     //
223050     //
223051     if (fstat (fdn, &file_status) != 0)
223052     {
223053         //
223054         // Error should be already set.
223055         //
223056         errset (errno);
223057         close (fdn);
223058         return (NULL);
223059     }
223060     //
223061     // Verify it is a directory
223062     //
223063     if (!S_ISDIR(file_status.st_mode))
223064     {
223065         //
223066         // It is not a directory!
223067         //
223068         close (fdn);
223069         errset (ENOTDIR); // Is not a directory.
223070         return (NULL);
223071     }
223072     //
223073     // A valid file descriptor is available: must find a free
223074     // '_directory_stream[]' slot.
223075     //
223076     for (d = 0; d < DOPEN_MAX; d++)
223077     {
223078         if (_directory_stream[d].fdn < 0)
223079         {
223080             //
223081             // Found a free slot: set it up.
223082             //
223083             dp = &(_directory_stream[d]);
223084             dp->fdn = fdn;
223085             //

```

1742

lib/dirent/ readdir.c

Si veda la sezione u0.86.

```

240001 #include <dirent.h>
240002 #include <fcntl.h>
240003 #include <sys/types.h>
240004 #include <sys/stat.h>
240005 #include <unistd.h>
240006 #include <errno.h>
240007 #include <stddef.h>
240008 //-----
240009 struct dirent *
240010 readdir (DIR *dp)
240011 {
240012     ssize_t size;
240013     //
240014     // Check for a valid argument.
240015     //
240016     if (dp == NULL)
240017     {
240018         //
240019         // Not a valid pointer.
240020         //
240021         errset (EBADF); // Invalid directory.
240022         return (NULL);
240023     }
240024     //
240025     // Check if it is an open directory stream.
240026     //
240027     if (dp->fdn < 0)
240028     {
240029         //
240030         // The stream is closed.
240031         //
240032         errset (EBADF); // Invalid directory.
240033         return (NULL);
240034     }
240035     //
240036     // Read the directory.
240037     //
240038     size = read (dp->fdn, &(dp->dir),
240039                (size_t) 16);
240040     //
240041     // Fix the null termination, if the name is very long.
240042     //
240043     dp->dir_d_name[NAME_MAX] = '\0';
240044     //
240045     // Check what was read.
240046     //
240047     if (size == 0)
240048     {
240049         //
240050         // End of directory, but it is not an error.
240051         //
240052         return (NULL);
240053     }
240054     //
240055     if (size < 0)
240056     {
240057         //
240058         // This is an error. The variable 'errno' is already set.
240059         //
240060         errset (errno);
240061         return (NULL);
240062     }
240063     //
240064     if (dp->dir_d_ino == 0)
240065     {
240066         //
240067         // This is a null directory record.
240068         // Should try to read the next one.
240069         //
240070         return (readdir (dp));
240071     }
240072     //
240073     if (strlen (dp->dir_d_name) == 0)
240074     {
240075         //
240076         // This is a bad directory record: try to read next.
240077         //
240078         return (readdir (dp));
240079     }
240080     //
240081     // A valid directory record should be available now.
240082     //
240083     return (&(dp->dir));
240084 }

```

1743

Si veda la sezione u0.89.

```

2250001 #include <dirent.h>
2250002 #include <fcntl.h>
2250003 #include <const.h>
2250004 #include <sys/types.h>
2250005 #include <sys/stat.h>
2250006 #include <unistd.h>
2250007 #include <errno.h>
2250008 #include <stddef.h>
2250009 #include <stdio.h>
2250010
2250011 //-----
2250012 void
2250013 rewinddir (DIR *dp)
2250014 {
2250015     FILE *fp;
2250016     //
2250017     // Check for a valid argument.
2250018     //
2250019     if (dp == NULL)
2250020     {
2250021         //
2250022         // Nothing to rewind, and no error to set.
2250023         //
2250024         return;
2250025     }
2250026     // Check if it is an open directory stream.
2250027     //
2250028     if (dp->fdn < 0)
2250029     {
2250030         //
2250031         // The stream is closed.
2250032         // Nothing to rewind, and no error to set.
2250033         //
2250034         return;
2250035     }
2250036     //
2250037     //
2250038     //
2250039     fp = &_stream[dp->fdn];
2250040     //
2250041     rewind (fp);
2250042 }

```

os16: «lib/errno.h»

Si veda la sezione u0.18.

```

2260001 #ifndef _ERRNO_H
2260002 #define _ERRNO_H 1
2260003
2260004 #include <limits.h>
2260005 #include <string.h>
2260006 //-----
2260007 // The variable 'errno' is standard, but 'errln' and 'errfn' are added
2260008 // to keep track of the error source. Variable 'errln' is used to save
2260009 // the source file line number; variable 'errfn' is used to save the
2260010 // source file name. To set these variables in a consistent way it is
2260011 // also added a macro-instruction: 'errset'.
2260012 //-----
2260013 extern int errno;
2260014 extern int errln;
2260015 extern char errfn[PATH_MAX];
2260016 #define errset(e) (errln = __LINE__, \
2260017                 strncpy (errfn, __FILE__, PATH_MAX), \
2260018                 errno = e)
2260019 //-----
2260020 // Standard POSIX 'errno' macro variables.
2260021 //-----
2260022 #define E2BIG 1 // Argument list too long.
2260023 #define EACCES 2 // Permission denied.
2260024 #define EADDRINUSE 3 // Address in use.
2260025 #define EADDRNOTAVAIL 4 // Address not available.
2260026 #define EAFNOSUPPORT 5 // Address family not supported.
2260027 #define EAGAIN 6 // Resource unavailable, try again.
2260028 #define EALREADY 7 // Connection already in progress.
2260029 #define EBADF 8 // Bad file descriptor.
2260030 #define EBADMSG 9 // Bad message.
2260031 #define EBUSY 10 // Device or resource busy.
2260032 #define ECANCELED 11 // Operation canceled.
2260033 #define ECHILD 12 // No child processes.
2260034 #define ECONNABORTED 13 // Connection aborted.
2260035 #define ECONNREFUSED 14 // Connection refused.
2260036 #define ECONNRESET 15 // Connection reset.
2260037 #define EDEADLK 16 // Resource deadlock would occur.
2260038 #define EDESTADDRREQ 17 // Destination address required.
2260039 #define EDOM 18 // Mathematics argument out of domain of
2260040 // function.
2260041 #define EDQUOT 19 // Reserved.
2260042 #define EEXIST 20 // File exists.
2260043 #define EFAULT 21 // Bad address.
2260044 #define EFBIG 22 // File too large.
2260045 #define EHOSTUNREACH 23 // Host is unreachable.
2260046 #define EIDRM 24 // Identifier removed.
2260047 #define EILSEQ 25 // Illegal byte sequence.
2260048 #define EINPROGRESS 26 // Operation in progress.
2260049 #define EINTR 27 // Interrupted function.
2260050 #define EINVAL 28 // Invalid argument.

```

```

2260051 #define EIO 29 // I/O error.
2260052 #define EISCONN 30 // Socket is connected.
2260053 #define EISDIR 31 // Is a directory.
2260054 #define ELOOP 32 // Too many levels of symbolic links.
2260055 #define EMFILE 33 // Too many open files.
2260056 #define EMLINK 34 // Too many links.
2260057 #define EMSGSIZE 35 // Message too large.
2260058 #define EMULTIHOP 36 // Reserved.
2260059 #define ENAMETOOLONG 37 // Filename too long.
2260060 #define ENETDOWN 38 // Network is down.
2260061 #define ENETRESET 39 // Connection aborted by network.
2260062 #define ENETUNREACH 40 // Network unreachable.
2260063 #define ENFILE 41 // Too many files open in system.
2260064 #define ENOBUFS 42 // No buffer space available.
2260065 #define ENODATA 43 // No message is available on the stream head
2260066 // read queue.
2260067 #define ENODEV 44 // No such device.
2260068 #define ENOENT 45 // No such file or directory.
2260069 #define ENOEXEC 46 // Executable file format error.
2260070 #define ENOLCK 47 // No locks available.
2260071 #define ENOLINK 48 // Reserved.
2260072 #define ENOMEM 49 // Not enough space.
2260073 #define ENOMSG 50 // No message of the desired type.
2260074 #define ENOPROTOPT 51 // Protocol not available.
2260075 #define ENOSPC 52 // No space left on device.
2260076 #define ENOSR 53 // No stream resources.
2260077 #define ENOSTR 54 // Not a stream.
2260078 #define ENOSYS 55 // Function not supported.
2260079 #define ENOTCONN 56 // The socket is not connected.
2260080 #define ENOTDIR 57 // Not a directory.
2260081 #define ENOTEMPTY 58 // Directory not empty.
2260082 #define ENOTSOCK 59 // Not a socket.
2260083 #define ENOTSUP 60 // Not supported.
2260084 #define ENOTTY 61 // Inappropriate I/O control operation.
2260085 #define ENXIO 62 // No such device or address.
2260086 #define EOPNOTSUPP 63 // Operation not supported on socket.
2260087 #define EOVERFLOW 64 // Value too large to be stored in data type.
2260088 #define EPERM 65 // Operation not permitted.
2260089 #define EPIPE 66 // Broken pipe.
2260090 #define EPROTO 67 // Protocol error.
2260091 #define EPROTONOSUPPORT 68 // Protocol not supported.
2260092 #define EPROTOTYPE 69 // Protocol wrong type for socket.
2260093 #define ERANGE 70 // Result too large.
2260094 #define EROFS 71 // Read-only file system.
2260095 #define ESRPIPE 72 // Invalid seek.
2260096 #define ESRCH 73 // No such process.
2260097 #define ESTALE 74 // Reserved.
2260098 #define ETIME 75 // Stream ioctl() timeout.
2260099 #define ETIMEDOUT 76 // Connection timed out.
2260100 #define ETXTBSY 77 // Text file busy.
2260101 #define EWOULDBLOCK 78 // Operation would block
2260102 // (may be the same as EAGAIN).
2260103 #define EXDEV 79 // Cross-device link.
2260104 //-----
2260105 // Added os16 errors.
2260106 //-----
2260107 #define EUNKNOWN (-1) // Unknown error.
2260108 #define E_FILE_TYPE 80 // File type not compatible.
2260109 #define E_ROOT_INODE_NOT_CACHED 81 // The root directory inode is
2260110 // not cached.
2260111 #define E_CANNOT_READ_SUPERBLOCK 83 // Cannot read super block.
2260112 #define E_MAP_INODE_TOO_BIG 84 // Map inode too big.
2260113 #define E_MAP_ZONE_TOO_BIG 85 // Map zone too big.
2260114 #define E_DATA_ZONE_TOO_BIG 86 // Data zone too big.
2260115 #define E_CANNOT_FIND_ROOT_DEVICE 87 // Cannot find root device.
2260116 #define E_CANNOT_FIND_ROOT_INODE 88 // Cannot find root inode.
2260117 #define E_FILE_TYPE_UNSUPPORTED 89 // File type unsupported.
2260118 #define E_ENV_TOO_BIG 90 // Environment too big.
2260119 #define E_LIMIT 91 // Exceeded implementation
2260120 // limits.
2260121 #define E_NOT_MOUNTED 92 // Not mounted.
2260122 #define E_NOT_IMPLEMENTED 93 // Not implemented.
2260123 //-----
2260124 // Default descriptions for errors.
2260125 //-----
2260126 #define TEXT_E2BIG "Argument list too long."
2260127 #define TEXT_EACCES "Permission denied."
2260128 #define TEXT_EADDRINUSE "Address in use."
2260129 #define TEXT_EADDRNOTAVAIL "Address not available."
2260130 #define TEXT_EAFNOSUPPORT "Address family not supported."
2260131 #define TEXT_EAGAIN "Resource unavailable, * \
2260132 *try again."
2260133 #define TEXT_EALREADY "Connection already in * \
2260134 *progress."
2260135 #define TEXT_EBADF "Bad file descriptor."
2260136 #define TEXT_EBADMSG "Bad message."
2260137 #define TEXT_EBUSY "Device or resource busy."
2260138 #define TEXT_ECANCELED "Operation canceled."
2260139 #define TEXT_ECHILD "No child processes."
2260140 #define TEXT_ECONNABORTED "Connection aborted."
2260141 #define TEXT_ECONNREFUSED "Connection refused."
2260142 #define TEXT_ECONNRESET "Connection reset."
2260143 #define TEXT_EDEADLK "Resource deadlock would occur."
2260144 #define TEXT_EDESTADDRREQ "Destination address required."
2260145 #define TEXT_EDOM "Mathematics argument out of * \
2260146 *domain of function."
2260147 #define TEXT_EDQUOT "Reserved error: EDQUOT"
2260148 #define TEXT_EEXIST "File exists."
2260149 #define TEXT_EFAULT "Bad address."
2260150 #define TEXT_EFBIG "File too large."
2260151 #define TEXT_EHOSTUNREACH "Host is unreachable."

```

```

2260152 #define TEXT_EIDRM "Identifier removed."
2260153 #define TEXT_EILSEQ "Illegal byte sequence."
2260154 #define TEXT_EINPROGRESS "Operation in progress."
2260155 #define TEXT_EINTR "Interrupted function."
2260156 #define TEXT_EINVAL "Invalid argument."
2260157 #define TEXT_EIO "I/O error."
2260158 #define TEXT_EISCONN "Socket is connected."
2260159 #define TEXT_EISDIR "Is a directory."
2260160 #define TEXT_ELOOP "Too many levels of symbolic links."
2260161
2260162 #define TEXT_EMFILE "Too many open files."
2260163 #define TEXT_EMLINK "Too many links."
2260164 #define TEXT_MSGSIZE "Message too large."
2260165 #define TEXT_EMULTIHOP "Reserved error: EMULTIHOP"
2260166 #define TEXT_ENAMETOOLONG "Filename too long."
2260167 #define TEXT_ENETDOWN "Network is down."
2260168 #define TEXT_ENETRESET "Connection aborted by network."
2260169 #define TEXT_ENETUNREACH "Network unreachable."
2260170 #define TEXT_ENFILE "Too many files open in system."
2260171 #define TEXT_ENOBUFS "No buffer space available."
2260172 #define TEXT_ENODATA "No message is available on the stream head read queue."
2260173
2260174 #define TEXT_ENODEV "No such device."
2260175 #define TEXT_ENOENT "No such file or directory."
2260176 #define TEXT_ENOEXEC "Executable file format error."
2260177 #define TEXT_ENOLCK "No locks available."
2260178 #define TEXT_ENOLINK "Reserved error: ENOLINK"
2260179 #define TEXT_ENOMEM "Not enough space."
2260180 #define TEXT_ENOSMSG "No message of the desired type."
2260181
2260182 #define TEXT_ENOPROTOPT "Protocol not available."
2260183 #define TEXT_ENOSPC "No space left on device."
2260184 #define TEXT_ENOSR "No stream resources."
2260185 #define TEXT_ENOSTR "Not a stream."
2260186 #define TEXT_ENOSYS "Function not supported."
2260187 #define TEXT_ENOTCONN "The socket is not connected."
2260188 #define TEXT_ENOTDIR "Not a directory."
2260189 #define TEXT_ENOTEMPTY "Directory not empty."
2260190 #define TEXT_ENOTSOCK "Not a socket."
2260191 #define TEXT_ENOTSUP "Not supported."
2260192 #define TEXT_ENOTTY "Inappropriate I/O control operation."
2260193
2260194 #define TEXT_ENXIO "No such device or address."
2260195 #define TEXT_EOPNOTSUPP "Operation not supported on socket."
2260196
2260197 #define TEXT_EOVERFLOW "Value too large to be stored in data type."
2260198
2260199 #define TEXT_EPERM "Operation not permitted."
2260200 #define TEXT_EPIPE "Broken pipe."
2260201 #define TEXT_EPROTO "Protocol error."
2260202 #define TEXT_EPROTONOSUPPORT "Protocol not supported."
2260203 #define TEXT_EPROTOTYPE "Protocol wrong type for socket."
2260204
2260205 #define TEXT_ERANGE "Result too large."
2260206 #define TEXT_EROFS "Read-only file system."
2260207 #define TEXT_ESPIPE "Invalid seek."
2260208 #define TEXT_ESRCH "No such process."
2260209 #define TEXT_ESTALE "Reserved error: ESTALE"
2260210 #define TEXT_ETIME "Stream ioctl() timeout."
2260211 #define TEXT_ETIMEDOUT "Connection timed out."
2260212 #define TEXT_ETXTBSY "Text file busy."
2260213 #define TEXT_EWOULDBLOCK "Operation would block."
2260214 #define TEXT_EXDEV "Cross-device link."
2260215
2260216 #define TEXT_EUNKNOWN "Unknown error."
2260217 #define TEXT_E_FILE_TYPE "File type not compatible."
2260218 #define TEXT_E_ROOT_INODE_NOT_CACHED "The root directory inode is not cached."
2260219
2260220 #define TEXT_E_CANNOT_READ_SUPERBLOCK "Cannot read super block."
2260221 #define TEXT_E_MAP_INODE_TOO_BIG "Map inode too big."
2260222 #define TEXT_E_MAP_ZONE_TOO_BIG "Map zone too big."
2260223 #define TEXT_E_DATA_ZONE_TOO_BIG "Data zone too big."
2260224 #define TEXT_E_CANNOT_FIND_ROOT_DEVICE "Cannot find root device."
2260225 #define TEXT_E_CANNOT_FIND_ROOT_INODE "Cannot find root inode."
2260226 #define TEXT_E_FILE_TYPE_UNSUPPORTED "File type unsupported."
2260227 #define TEXT_E_ENV_TOO_BIG "Environment too big."
2260228 #define TEXT_E_LLIMIT "Exceeded implementation limits."
2260229
2260230 #define TEXT_E_NOT_MOUNTED "Not mounted."
2260231 #define TEXT_E_NOT_IMPLEMENTED "Not implemented."
2260232
2260233
2260234 // The function 'error()' is not standard and is used to return a
2260235 // pointer to a string containing the default description of the
2260236 // error contained inside 'errno'.
2260237
2260238 char *error (void); // Not standard!
2260239
2260240 #endif

```

lib/errno/errno.c

Si veda la sezione u0.18.

```

2270001 //-----
2270002 // This file does not include the 'errno.h' header, because here 'errno'
2270003 // should not be declared as an extern variable!
2270004 //-----
2270005
2270006 #include <limits.h>
2270007 //-----

```

1746

```

2270008 // The variable 'errno' is standard, but 'errln' and 'errfn' are added
2270009 // to keep track of the error source. Variable 'errln' is used to save
2270010 // the source file line number; variable 'errfn' is used to save the
2270011 // source file name. To set these variables in a consistent way it is
2270012 // also added a macro-instruction: 'errset'.
2270013 //-----
2270014 int errno;
2270015 int errln;
2270016 char errfn[PATH_MAX];
2270017 //-----

```

os16: «lib/fcntl.h»

Si veda la sezione u0.2.

```

2280001 #ifndef _FCNTL_H
2280002 #define _FCNTL_H 1
2280003
2280004 #include <const.h>
2280005 #include <sys/types.h> // mode_t
2280006 // off_t
2280007 // pid_t
2280008
2280009 // Values for the second parameter of function 'fcntl()'.
2280010 //-----
2280011 #define F_DUPFD 0 // Duplicate file descriptor.
2280012 #define F_GETFD 1 // Get file descriptor flags.
2280013 #define F_SETFD 2 // Set file descriptor flags.
2280014 #define F_GETFL 3 // Get file status flags.
2280015 #define F_SETFL 4 // Set file status flags.
2280016 #define F_GETLK 5 // Get record locking information.
2280017 #define F_SETLK 6 // Set record locking information.
2280018 #define F_SETLKW 7 // Set record locking information;
2280019 // wait if blocked.
2280020 #define F_GETOWN 8 // Set owner of socket.
2280021 #define F_SETOWN 9 // Get owner of socket.
2280022 //-----
2280023 // Flags to be set with:
2280024 // fcntl (fd, F_SETFD, ...);
2280025 //-----
2280026 #define FD_CLOEXEC 1 // Close the file descriptor upon
2280027 // execution of an exec() family
2280028 // function.
2280029 //-----
2280030 // Values for type 'l_type', used for record locking with 'fcntl()'.
2280031 //-----
2280032 #define F_RDLCK 0 // Read lock.
2280033 #define F_WRLCK 1 // Write lock.
2280034 #define F_UNLCK 2 // Remove lock.
2280035 //-----
2280036 // Flags for file creation, in place of 'oflag' parameter for function
2280037 // 'open()'.
2280038 //-----
2280039 #define O_CREAT 000010 // Create file if it does not exist.
2280040 #define O_EXCL 000020 // Exclusive use flag.
2280041 #define O_NOCTTY 000040 // Do not assign a controlling terminal.
2280042 #define O_TRUNC 000100 // Truncation flag.
2280043 //-----
2280044 // Flags for the file status, used with 'open()' and 'fcntl()'.
2280045 //-----
2280046 #define O_APPEND 000200 // Write append.
2280047 #define O_DSYNC 000400 // Synchronized write operations.
2280048 #define O_NONBLOCK 001000 // Non-blocking mode.
2280049 #define O_RSYNC 002000 // Synchronized read operations.
2280050 #define O_SYNC 004000 // Synchronized read and write.
2280051 //-----
2280052 // File access mask selection.
2280053 //-----
2280054 #define O_ACCMODE 000003 // Mask to select the last three bits,
2280055 // used to specify the main access
2280056 // modes: read, write and both.
2280057 //-----
2280058 // Main access modes.
2280059 //-----
2280060 #define O_RDONLY 000001 // Read.
2280061 #define O_WRONLY 000002 // Write.
2280062 #define O_RDWR (O_RDONLY | O_WRONLY) // Both read and write.
2280063 //-----
2280064 // Structure 'lock', used to file lock for POSIX standard. It is not
2280065 // used inside os16.
2280066 //-----
2280067 struct flock {
2280068     short int l_type; // Type of lock: F_RDLCK, F_WRLCK, or F_UNLCK.
2280069     short int l_whence; // Start reference point.
2280070     off_t l_start; // Offset, from 'l_whence', for the area start.
2280071     off_t l_len; // Locked area size. Zero means up to the end of
2280072 // the file.
2280073     pid_t l_pid; // The process id blocking the area.
2280074 };
2280075 //-----
2280076 // Function prototypes.
2280077 //-----
2280078 int creat (const char *path, mode_t mode);
2280079 int fcntl (int fdn, int cmd, ...);
2280080 int open (const char *path, int oflags, ...);
2280081 //-----
2280082
2280083 #endif

```

1747

lib/fcntl/creat.c

«

Si veda la sezione [u0.11](#).

```
230001 #include <fcntl.h>
230002 #include <sys/types.h>
230003 #include <const.h>
230004 //-----
230005 int
230006 creat (const char *path, mode_t mode)
230007 {
230008     return (open (path, O_WRONLY|O_CREAT|O_TRUNC, mode));
230009 }
```

lib/fcntl/fcntl.c

«

Si veda la sezione [u0.13](#).

```
230001 #include <fcntl.h>
230002 #include <stdarg.h>
230003 #include <stddef.h>
230004 #include <string.h>
230005 #include <errno.h>
230006 #include <sys/osl6.h>
230007 #include <const.h>
230008 #include <limits.h>
230009 //-----
230010 int
230011 fcntl (int fdn, int cmd, ...)
230012 {
230013     va_list ap;
230014     sysmsg_fcntl_t msg;
230015     va_start (ap, cmd);
230016     //
230017     // Well known arguments.
230018     //
230019     msg.fdn = fdn;
230020     msg.cmd = cmd;
230021     //
230022     // Select other arguments.
230023     //
230024     switch (cmd)
230025     {
230026     case F_DUPFD:
230027     case F_SETFD:
230028     case F_SETFL:
230029         msg.arg = va_arg (ap, int);
230030         break;
230031     case F_GETFD:
230032     case F_GETFL:
230033         break;
230034     case F_GETOWN:
230035     case F_SETOWN:
230036     case F_GETLK:
230037     case F_SETLK:
230038     case F_SETLKW:
230039         errset (E_NOT_IMPLEMENTED); // Not implemented.
230040         return (-1);
230041     default:
230042         errset (EINVAL); // Not implemented.
230043         return (NULL);
230044     }
230045     //
230046     // Do the system call.
230047     //
230048     sys (SYS_FCNTL, &msg, (sizeof msg));
230049     errno = msg.errno;
230050     errln = msg.errln;
230051     strncpy (errfn, msg.errfn, PATH_MAX);
230052     return (msg.ret);
230053 }
```

lib/fcntl/open.c

«

Si veda la sezione [u0.28](#).

```
230001 #include <fcntl.h>
230002 #include <stdarg.h>
230003 #include <stddef.h>
230004 #include <string.h>
230005 #include <errno.h>
230006 #include <sys/osl6.h>
230007 #include <const.h>
230008 #include <limits.h>
230009 //-----
230010 int
230011 open (const char *path, int oflags, ...)
230012 {
230013     va_list ap;
230014     sysmsg_open_t msg;
230015     va_start (ap, oflags);
230016     if (path == NULL || strlen (path) == 0)
230017     {
230018         errset (EINVAL); // Invalid argument.
230019         return (-1);
230020     }
230021     strncpy (msg.path, path, PATH_MAX);
230022     msg.flags = oflags;
230023     msg.mode = va_arg (ap, mode_t);
230024     sys (SYS_OPEN, &msg, (sizeof msg));
```

1748

```
230025     errno = msg.errno;
230026     errln = msg.errln;
230027     strncpy (errfn, msg.errfn, PATH_MAX);
230028     return (msg.ret);
230029 }
```

osl6: «lib/grp.h»

Si veda la sezione [u0.2](#).

«

```
230001 //-----
230002 // osl6 does not have a group management!
230003 //-----
230004
230005 #ifndef _GRP_H
230006 #define _GRP_H 1
230007
230008 #include <const.h>
230009 #include <restrict.h>
230010 #include <sys/types.h> // gid_t
230011
230012 //-----
230013 struct group {
230014     char *gr_name;
230015     gid_t gr_gid;
230016     char **gr_mem;
230017 };
230018 //-----
230019 struct group *getgrgid (gid_t gid);
230020 struct group *getgrnam (const char *name);
230021 //-----
230022 #endif
230023
```

lib/grp/getgrgid.c

«

Si veda la sezione [u0.2](#).

```
230001 #include <grp.h>
230002 #include <NULL.h>
230003 //-----
230004 struct group *
230005 getgrgid (gid_t gid)
230006 {
230007     static char *name = "none";
230008     static struct group grp;
230009     //
230010     // osl6 does not have a group management, so the answer is always
230011     // the same.
230012     //
230013     grp.gr_name = name;
230014     grp.gr_gid = (gid_t) -1;
230015     grp.gr_mem = NULL;
230016     //
230017     return (&grp);
230018 }
```

lib/grp/getgrnam.c

«

Si veda la sezione [u0.2](#).

```
234001 #include <grp.h>
234002 //-----
234003 struct group *
234004 getgrnam (const char *name)
234005 {
234006     return (getgrgid ((gid_t) 0));
234007 }
```

osl6: «lib/libgen.h»

Si veda la sezione [u0.2](#).

«

```
235001 #ifndef _LIBGEN_H
235002 #define _LIBGEN_H 1
235003
235004 //-----
235005 char *basename (char *path);
235006 char *dirname (char *path);
235007 //-----
235008 #endif
235009
```

lib/libgen/basename.c

«

Si veda la sezione [u0.7](#).

```
236001 #include <libgen.h>
236002 #include <limits.h>
236003 #include <stddef.h>
236004 #include <string.h>
236005 //-----
236006 char *
236007 basename (char *path)
236008 {
236009     static char *point = "."; // When 'path' is NULL.
```

1749

```

236010     char *p;           // Pointer inside 'path'.
236011     int i;            // Scan index inside 'path'.
236012     //
236013     // Empty path.
236014     //
236015     if (path == NULL || strlen (path) == 0)
236016     {
236017         return (point);
236018     }
236019     //
236020     // Remove all final '/' if it exists, excluded the first character:
236021     // 'i' is kept greater than zero.
236022     //
236023     for (i = (strlen (path) - 1); i > 0 && path[i] == '/'; i--)
236024     {
236025         path[i] = 0;
236026     }
236027     //
236028     // After removal of extra final '/', if there is only one '/', this
236029     // is to be returned.
236030     //
236031     if (strncmp (path, "/", PATH_MAX) == 0)
236032     {
236033         return (path);
236034     }
236035     //
236036     // If there are no '/'.
236037     //
236038     if (strchr (path, '/') == NULL)
236039     {
236040         return (path);
236041     }
236042     //
236043     // Find the last '/' and calculate a pointer to the base name.
236044     //
236045     p = strrchr (path, (unsigned int) '/');
236046     p++;
236047     //
236048     // Return the pointer to the base name.
236049     //
236050     return (p);
236051 }

```

lib/libgen/dirname.c

Si veda la sezione u0.7.

```

237001 #include <libgen.h>
237002 #include <limits.h>
237003 #include <stddef.h>
237004 #include <string.h>
237005 //-----
237006 char *
237007 dirname (char *path)
237008 {
237009     static char *point = ".*";           // When 'path' is NULL.
237010     char *p;                             // Pointer inside 'path'.
237011     int i;                                // Scan index inside 'path'.
237012     //
237013     // Empty path.
237014     //
237015     if (path == NULL || strlen (path) == 0)
237016     {
237017         return (point);
237018     }
237019     //
237020     // Simple cases.
237021     //
237022     if (strncmp (path, "/", PATH_MAX) == 0 ||
237023         strncmp (path, ".", PATH_MAX) == 0 ||
237024         strncmp (path, "..", PATH_MAX) == 0)
237025     {
237026         return (path);
237027     }
237028     //
237029     // Remove all final '/' if it exists, excluded the first character:
237030     // 'i' is kept greater than zero.
237031     //
237032     for (i = (strlen (path) - 1); i > 0 && path[i] == '/'; i--)
237033     {
237034         path[i] = 0;
237035     }
237036     //
237037     // After removal of extra final '/', if there is only one '/', this
237038     // is to be returned.
237039     //
237040     if (strncmp (path, "/", PATH_MAX) == 0)
237041     {
237042         return (path);
237043     }
237044     //
237045     // If there are no '/'.
237046     //
237047     if (strchr (path, '/') == NULL)
237048     {
237049         return (point);
237050     }
237051     //
237052     // If there is only a '/' a the beginning.
237053     //
237054     if (path[0] == '/'

```

1750

```

237055     strchr (&path[1], (unsigned int) '/') == NULL)
237056     {
237057         path[1] = 0;
237058         return (path);
237059     }
237060     //
237061     // Replace the last '/' with zero.
237062     //
237063     p = strrchr (path, (unsigned int) '/');
237064     *p = 0;
237065     //
237066     // Now remove extra duplicated final '/', except the very first
237067     // character: 'i' is kept greater than zero.
237068     //
237069     for (i = (strlen (path) - 1); i > 0 && path[i] == '/'; i--)
237070     {
237071         path[i] = 0;
237072     }
237073     //
237074     // Now 'path' appears as a reduced string: the original path string
237075     // is modified.
237076     //
237077     return (path);
237078 }

```

os16: «lib/pwd.h»

Si veda la sezione u0.2.

```

238001 #ifndef _PWD_H
238002 #define _PWD_H           1
238003
238004 #include <const.h>
238005 #include <restrict.h>
238006 #include <sys/types.h>           // gid_t, uid_t
238007 //-----
238008 struct passwd {
238009     char *pw_name;
238010     char *pw_passwd;
238011     uid_t pw_uid;
238012     gid_t pw_gid;
238013     char *pw_gecos;
238014     char *pw_dir;
238015     char *pw_shell;
238016 };
238017 //-----
238018 struct passwd *getpwent (void);
238019 void setpwent (void);
238020 void endpwent (void);
238021 struct passwd *getpwnam (const char *name);
238022 struct passwd *getpwuid (uid_t uid);
238023 //-----
238024 #endif

```

lib/pwd/pwent.c

Si veda la sezione u0.53.

```

239001 #include <pwd.h>
239002 #include <stdio.h>
239003 #include <string.h>
239004 #include <stdlib.h>
239005 //-----
239006 static char buffer[BUFSIZ];
239007 static struct passwd pw;
239008 static FILE *fp = NULL;
239009 //-----
239010 struct passwd *
239011 getpwent (void)
239012 {
239013     {
239014         void *pstatus;
239015         char *char_uid;
239016         char *char_gid;
239017         //
239018         if (fp == NULL)
239019         {
239020             fp = fopen ("/etc/passwd", "r");
239021             if (fp == NULL)
239022             {
239023                 return NULL;
239024             }
239025         }
239026         //
239027         pstatus = fgets (buffer, BUFSIZ, fp);
239028         if (pstatus == NULL)
239029         {
239030             return (NULL);
239031         }
239032         //
239033         pw.pw_name = strtok (buffer, ":");
239034         pw.pw_passwd = strtok (NULL, ":");
239035         char_uid = strtok (NULL, ":");
239036         char_gid = strtok (NULL, ":");
239037         pw.pw_gecos = strtok (NULL, ":");
239038         pw.pw_dir = strtok (NULL, ":");
239039         pw.pw_shell = strtok (NULL, ":");
239040         pw.pw_uid = (uid_t) atoi (char_uid);

```

1751

```

230041     pw.pw_gid  = (gid_t) atoi (char_gid);
230042     //
230043     return (&pw);
230044 }
230045 //-----
230046 void
230047 endpwent (void)
230048 {
230049     int status;
230050     //
230051     if (fp != NULL)
230052     {
230053         fclose (fp);
230054         if (status != NULL)
230055         {
230056             fp = NULL;
230057         }
230058     }
230059 }
230060 //-----
230061 void
230062 setpwent (void)
230063 {
230064     if (fp != NULL)
230065     {
230066         rewind (fp);
230067     }
230068 }
230069 //-----
230070 struct passwd *
230071 getpwnam (const char *name)
230072 {
230073     struct passwd *pw;
230074     //
230075     setpwent ();
230076     //
230077     for (;;)
230078     {
230079         pw = getpwent ();
230080         if (pw == NULL)
230081         {
230082             return (NULL);
230083         }
230084         if (strcmp (pw->pw_name, name) == 0)
230085         {
230086             return (pw);
230087         }
230088     }
230089 }
230090 //-----
230091 struct passwd *
230092 getpwuid (uid_t uid)
230093 {
230094     struct passwd *pw;
230095     //
230096     setpwent ();
230097     //
230098     for (;;)
230099     {
230100         pw = getpwent ();
230101         if (pw == NULL)
230102         {
230103             return (NULL);
230104         }
230105         if (pw->pw_uid == uid)
230106         {
230107             return (pw);
230108         }
230109     }
230110 }

```

os16: «lib/signal.h»

« Si veda la sezione u0.2.

```

240001 #ifndef _SIGNAL_H
240002 #define _SIGNAL_H    1
240003
240004 #include <sys/types.h>
240005 //-----
240006 #define SIGHUP      1
240007 #define SIGINT      2
240008 #define SIGQUIT     3
240009 #define SIGILL      4
240010 #define SIGABRT     6
240011 #define SIGFPE      8
240012 #define SIGKILL     9
240013 #define SIGSEGV    11
240014 #define SIGPIPE    13
240015 #define SIGALRM    14
240016 #define SIGTERM    15
240017 #define SIGSTOP    17
240018 #define SIGTSTP    18
240019 #define SIGCONT    19
240020 #define SIGCHLD    20
240021 #define SIGTTIN    21
240022 #define SIGTTOU    22
240023 #define SIGUSR1    30
240024 #define SIGUSR2    31
240025 //-----
240026 typedef int sig_atomic_t;

```

1752

```

240027 typedef void (*sighandler_t) (int); // The type 'sighandler_t' is a
240028 // pointer to a function for the
240029 // signal handling, with a parameter
240030 // of type 'int', returning 'void'.
240031 //
240032 // Special undeclarable functions.
240033 //
240034 #define SIG_ERR ((sighandler_t) -1) // It transform an integer number
240035 #define SIG_DFL ((sighandler_t) 0) // into a 'sighandler_t' type,
240036 #define SIG_IGN ((sighandler_t) 1) // that is, a pointer to a function
240037 // that does not exists really.
240038 //-----
240039 sighandler_t signal (int sig, sighandler_t handler);
240040 int kill (pid_t pid, int sig);
240041 int raise (int sig);
240042 //-----
240043
240044 #endif

```

lib/signal/kill.c

« Si veda la sezione u0.22.

```

241001 #include <sys/os16.h>
241002 #include <sys/types.h>
241003 #include <signal.h>
241004 #include <errno.h>
241005 #include <string.h>
241006 //-----
241007 int
241008 kill (pid_t pid, int sig)
241009 {
241010     sysmsg_kill_t msg;
241011     if (pid < -1) // Currently unsupported.
241012     {
241013         errset (ESRCH);
241014         return (-1);
241015     }
241016     msg.pid = pid;
241017     msg.signal = sig;
241018     msg.ret = 0;
241019     msg.errno = 0;
241020     sys (SYS_KILL, &msg, (sizeof msg));
241021     errno = msg.errno;
241022     errln = msg.errln;
241023     strncpy (errfn, msg.errfn, PATH_MAX);
241024     return (msg.ret);
241025 }

```

lib/signal/signal.c

« Si veda la sezione u0.34.

```

242001 #include <sys/os16.h>
242002 #include <sys/types.h>
242003 #include <signal.h>
242004 #include <errno.h>
242005 #include <string.h>
242006 //-----
242007 sighandler_t
242008 signal (int sig, sighandler_t handler)
242009 {
242010     sysmsg_signal_t msg;
242011
242012     msg.signal = sig;
242013     msg.handler = handler;
242014     msg.ret = SIG_DFL;
242015     msg.errno = 0;
242016     sys (SYS_SIGNAL, &msg, (sizeof msg));
242017     errno = msg.errno;
242018     errln = msg.errln;
242019     strncpy (errfn, msg.errfn, PATH_MAX);
242020     return (msg.ret);
242021 }

```

os16: «lib/stdio.h»

« Si veda la sezione u0.103.

```

243001 #ifndef _STDIO_H
243002 #define _STDIO_H    1
243003
243004 #include <const.h>
243005 #include <restrict.h>
243006 #include <stdarg.h>
243007 #include <stdint.h>
243008 #include <limits.h>
243009 #include <NULL.h>
243010 #include <size_t.h>
243011 #include <sys/types.h>
243012 #include <SEEK.h> // SEEK_CUR, SEEK_SET, SEEK_END
243013 //-----
243014 #define BUFSIZ      2048 // Like the file system max zone
243015 // size.
243016 #define _IOFBF      0 // Input-output fully buffered.
243017 #define _IOLBF      1 // Input-output line buffered.
243018 #define _IONBF      2 // Input-output with no buffering.
243019

```

1753

```

2430020 #define L_tmpnam FILENAME_MAX // <limits.h>
2430021
2430022 #define FOPEN_MAX OPEN_MAX // <limits.h>
2430023 #define FILENAME_MAX NAME_MAX // <limits.h>
2430024 #define TMP_MAX 0x7FFF
2430025
2430026 #define EOF (-1) // Must be a negative value.
2430027 //-----
2430028 typedef off_t fpos_t; // 'off_t' defined in <sys/types.h>.
2430029
2430030 typedef struct {
2430031     int fdn; // File descriptor number.
2430032     char error; // Error indicator.
2430033     char eof; // End of file indicator.
2430034 } FILE;
2430035
2430036 extern FILE _stream[]; // Defined inside 'lib/stdio/FILE.c'.
2430037
2430038 #define stdin (&_stream[0])
2430039 #define stdout (&_stream[1])
2430040 #define stderr (&_stream[2])
2430041 //-----
2430042 void clearerr (FILE *fp);
2430043 int fclose (FILE *fp);
2430044 int feof (FILE *fp);
2430045 int ferror (FILE *fp);
2430046 int fflush (FILE *fp);
2430047 int fgetc (FILE *fp);
2430048 int fgetpos (FILE *restrict fp, fpos_t *restrict pos);
2430049 char *fgets (char *restrict string, int n, FILE *restrict fp);
2430050 int fileno (FILE *fp);
2430051 FILE *fopen (const char *path, const char *mode);
2430052 int fprintf (FILE *fp, const char *format, ...);
2430053 int fputc (int c, FILE *fp);
2430054 int fputs (const char *restrict string, FILE *restrict fp);
2430055 size_t fread (void *restrict buffer, size_t size, size_t nmemb,
2430056 FILE *restrict fp);
2430057 FILE *freopen (const char *restrict path,
2430058 const char *restrict mode,
2430059 FILE *restrict fp);
2430060 int fscanf (FILE *restrict fp, const char *restrict format,
2430061 ...);
2430062 int fseek (FILE *fp, long int offset, int whence);
2430063 int fsetpos (FILE *fp, const fpos_t *pos);
2430064 long int ftell (FILE *fp);
2430065 off_t ftello (FILE *fp);
2430066 size_t fwrite (const void *restrict buffer, size_t size,
2430067 size_t nmemb, FILE *restrict fp);
2430068 #define getc(p) (fgetc (p))
2430069 int getchar (void);
2430070 char *gets (char *string);
2430071 void perror (const char *string);
2430072 int printf (const char *restrict format, ...);
2430073 #define putc(c, p) (fputc ((c), (p)))
2430074 #define putchar(c) (fputc ((c), (stdout)))
2430075 int puts (const char *string);
2430076 void rewind (FILE *fp);
2430077 int scanf (const char *restrict format, ...);
2430078 void setbuf (FILE *restrict fp, char *restrict buffer);
2430079 int setvbuf (FILE *restrict fp, char *restrict buffer,
2430080 int buf_mode, size_t size);
2430081 int snprintf (char *restrict string, size_t size,
2430082 const char *restrict format, ...);
2430083 int sprintf (char *restrict string, const char *restrict format,
2430084 ...);
2430085 int sscanf (char *restrict string, const char *restrict format,
2430086 ...);
2430087 int vfprintf (FILE *fp, const char *restrict format, va_list arg);
2430088 int vfscanf (FILE *restrict fp, const char *restrict format,
2430089 va_list arg);
2430090 int vprintf (const char *restrict format, va_list arg);
2430091 int vscanf (const char *restrict format, va_list ap);
2430092 int vsprintf (char *restrict string, size_t size,
2430093 const char *restrict format, va_list arg);
2430094 int vsprintf (char *restrict string, const char *restrict format,
2430095 va_list arg);
2430096 int vsscanf (const char *string, const char *format,
2430097 va_list ap);
2430098
2430099 #endif

```

lib/stdio/FILE.c

Si veda la sezione [u0.2](#).

```

2440001 #include <stdio.h>
2440002 //
2440003 // There must be room for at least 'FOPEN_MAX' elements.
2440004 //
2440005 FILE _stream[FOPEN_MAX];
2440006 //-----
2440007 void
2440008 _stdio_stream_setup (void)
2440009 {
2440010     _stream[0].fdn = 0;
2440011     _stream[0].error = 0;
2440012     _stream[0].eof = 0;
2440013
2440014     _stream[1].fdn = 1;
2440015     _stream[1].error = 0;
2440016     _stream[1].eof = 0;

```

1754

```

2440017     _stream[2].fdn = 2;
2440018     _stream[2].error = 0;
2440019     _stream[2].eof = 0;
2440020 }
2440021

```

lib/stdio/clearerr.c

Si veda la sezione [u0.9](#).

```

2450001 #include <stdio.h>
2450002 //-----
2450003 void
2450004 clearerr (FILE *fp)
2450005 {
2450006     if (fp != NULL)
2450007     {
2450008         fp->error = 0;
2450009         fp->eof = 0;
2450010     }
2450011 }

```

lib/stdio/fclose.c

Si veda la sezione [u0.27](#).

```

2460001 #include <stdio.h>
2460002 #include <unistd.h>
2460003 //-----
2460004 int
2460005 fclose (FILE *fp)
2460006 {
2460007     return (close (fp->fdn));
2460008 }

```

lib/stdio/feof.c

Si veda la sezione [u0.28](#).

```

2470001 #include <stdio.h>
2470002 //-----
2470003 int
2470004 feof (FILE *fp)
2470005 {
2470006     if (fp != NULL)
2470007     {
2470008         return (fp->eof);
2470009     }
2470010     return (0);
2470011 }

```

lib/stdio/ferror.c

Si veda la sezione [u0.29](#).

```

2480001 #include <stdio.h>
2480002 //-----
2480003 int
2480004 ferror (FILE *fp)
2480005 {
2480006     if (fp != NULL)
2480007     {
2480008         return (fp->error);
2480009     }
2480010     return (0);
2480011 }

```

lib/stdio/fflush.c

Si veda la sezione [u0.30](#).

```

2490001 #include <stdio.h>
2490002 //-----
2490003 int
2490004 fflush (FILE *fp)
2490005 {
2490006     //
2490007     // The os16 library does not have any buffered data.
2490008     //
2490009     return (0);
2490010 }

```

lib/stdio/fgetc.c

Si veda la sezione [u0.31](#).

```

2500001 #include <stdio.h>
2500002 #include <sys/types.h>
2500003 #include <unistd.h>
2500004 //-----
2500005 int
2500006 fgetc (FILE *fp)
2500007 {
2500008     ssize_t size_read;
2500009     int c; // Character read.

```

1755


```

250010 //
250011 for (c = 0;;)
250012 {
250013     size_read = read (fp->fdn, &c, (size_t) 1);
250014     //
250015     if (size_read <= 0)
250016     {
250017         //
250018         // It is the end of file (zero) otherwise there is a
250019         // problem (a negative value): return 'EOF'.
250020         //
250021         return (EOF);
250022     }
250023     //
250024     // Valid read: end of scan.
250025     //
250026     return (c);
250027 }
250028 }

```

lib/stdio/fgetpos.c

« Si veda la sezione u0.32.

```

250001 #include <stdio.h>
250002 //-----
250003 int
250004 fgetpos (FILE *restrict fp, fpos_t *restrict pos)
250005 {
250006     long int position;
250007     //
250008     if (fp != NULL)
250009     {
250010         position = ftell (fp);
250011         if (position >= 0)
250012         {
250013             *pos = position;
250014             return (0);
250015         }
250016     }
250017     return (-1);
250018 }

```

lib/stdio/fgets.c

« Si veda la sezione u0.33.

```

250001 #include <stdio.h>
250002 #include <sys/types.h>
250003 #include <unistd.h>
250004 #include <stddef.h>
250005 //-----
250006 char *
250007 fgets (char *restrict string, int n, FILE *restrict fp)
250008 {
250009     ssize_t size_read;
250010     int b; // Index inside the string buffer.
250011     //
250012     for (b = 0; b < (n-1); b++, string[b] = 0)
250013     {
250014         size_read = read (fp->fdn, &string[b], (size_t) 1);
250015         //
250016         if (size_read <= 0)
250017         {
250018             //
250019             // It is the end of file (zero) otherwise there is a
250020             // problem (a negative value).
250021             //
250022             string[b] = 0;
250023             break;
250024         }
250025         //
250026         if (string[b] == '\n')
250027         {
250028             b++;
250029             string[b] = 0;
250030             break;
250031         }
250032     }
250033     //
250034     // If 'b' is zero, nothing was read and 'NULL' is returned.
250035     //
250036     if (b == 0)
250037     {
250038         return (NULL);
250039     }
250040     else
250041     {
250042         return (string);
250043     }
250044 }

```

lib/stdio/fileno.c

« Si veda la sezione u0.34.

```

250001 #include <stdio.h>
250002 #include <errno.h>

```

```

250003 //-----
250004 int
250005 fileno (FILE *fp)
250006 {
250007     if (fp != NULL)
250008     {
250009         return (fp->fdn);
250010     }
250011     errset (EBADF); // Bad file descriptor.
250012     return (-1);
250013 }

```

lib/stdio/fopen.c

« Si veda la sezione u0.35.

```

250001 #include <fcntl.h>
250002 #include <stdarg.h>
250003 #include <stddef.h>
250004 #include <string.h>
250005 #include <errno.h>
250006 #include <sys/osi6.h>
250007 #include <const.h>
250008 #include <limits.h>
250009 #include <stdio.h>
250010 //-----
250011 FILE *
250012 fopen (const char *path, const char *mode)
250013 {
250014     int fdn;
250015     //
250016     if (strcmp (mode, "r") ||
250017         strcmp (mode, "rb"))
250018     {
250019         fdn = open (path, O_RDONLY);
250020     }
250021     else if (strcmp (mode, "r+") ||
250022             strcmp (mode, "r+b") ||
250023             strcmp (mode, "rb+"))
250024     {
250025         fdn = open (path, O_RDWR);
250026     }
250027     else if (strcmp (mode, "w") ||
250028             strcmp (mode, "wb"))
250029     {
250030         fdn = open (path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
250031     }
250032     else if (strcmp (mode, "w+") ||
250033             strcmp (mode, "w+b") ||
250034             strcmp (mode, "wb+"))
250035     {
250036         fdn = open (path, O_RDWR|O_CREAT|O_TRUNC, 0666);
250037     }
250038     else if (strcmp (mode, "a") ||
250039             strcmp (mode, "ab"))
250040     {
250041         fdn = open (path, O_WRONLY|O_APPEND|O_CREAT|O_TRUNC, 0666);
250042     }
250043     else if (strcmp (mode, "a+") ||
250044             strcmp (mode, "a+b") ||
250045             strcmp (mode, "ab+"))
250046     {
250047         fdn = open (path, O_RDWR|O_APPEND|O_CREAT|O_TRUNC, 0666);
250048     }
250049     else
250050     {
250051         errset (EINVAL); // Invalid argument.
250052         return (NULL);
250053     }
250054     //
250055     // Check the file descriptor returned.
250056     //
250057     if (fdn < 0)
250058     {
250059         //
250060         // The variable 'errno' is already set.
250061         //
250062         errset (errno);
250063         return (NULL);
250064     }
250065     //
250066     // A valid file descriptor is available: convert it into a file
250067     // stream. Please note that the file descriptor number must be
250068     // saved inside the corresponding '_stream[]' array, because the
250069     // file pointer do not have knowledge of the relative position
250070     // inside the array.
250071     //
250072     _stream[fdn].fdn = fdn; // Saved the file descriptor number.
250073     //
250074     return (&_stream[fdn]); // Returned the file stream pointer.
250075 }

```

lib/stdio/fprintf.c

« Si veda la sezione u0.78.

```

250001 #include <stdio.h>
250002 //-----

```

```

258004 int
258005 fprintf (FILE *fp, char *restrict format, ...)
258006 {
258007     va_list ap;
258008     va_start (ap, format);
258009     return (vfprintf (fp, format, ap));
258010 }

```

lib/stdio/fputc.c

<<

Si veda la sezione [u0.37](#).

```

256001 #include <stdio.h>
256002 #include <sys/types.h>
256003 #include <sys/osl6.h>
256004 #include <string.h>
256005 #include <unistd.h>
256006 //-----
256007 int
256008 fputc (int c, FILE *fp)
256009 {
256010     ssize_t size_written;
256011     char character = (char) c;
256012     size_written = write (fp->fdn, &character, (size_t) 1);
256013     if (size_written < 0)
256014     {
256015         fp->eof = 1;
256016         return (EOF);
256017     }
256018     return (c);
256019 }

```

lib/stdio/fputs.c

<<

Si veda la sezione [u0.38](#).

```

257001 #include <stdio.h>
257002 #include <string.h>
257003 //-----
257004 int
257005 fputs (const char *restrict string, FILE *restrict fp)
257006 {
257007     int i; // Index inside the string to be printed.
257008     int status;
257009
257010     for (i = 0; i < strlen (string); i++)
257011     {
257012         status = fputc (string[i], fp);
257013         if (status == EOF)
257014         {
257015             fp->eof = 1;
257016             return (EOF);
257017         }
257018     }
257019     return (0);
257020 }

```

lib/stdio/fread.c

<<

Si veda la sezione [u0.39](#).

```

258001 #include <unistd.h>
258002 #include <stdio.h>
258003 //-----
258004 size_t
258005 fread (void *restrict buffer, size_t size, size_t nmemb,
258006        FILE *restrict fp)
258007 {
258008     ssize_t size_read;
258009     size_read = read (fp->fdn, buffer, (size_t) (size * nmemb));
258010     if (size_read == 0)
258011     {
258012         fp->eof = 1;
258013         return ((size_t) 0);
258014     }
258015     else if (size_read < 0)
258016     {
258017         fp->error = 1;
258018         return ((size_t) 0);
258019     }
258020     else
258021     {
258022         return ((size_t) (size_read / size));
258023     }
258024 }

```

lib/stdio/freopen.c

<<

Si veda la sezione [u0.35](#).

```

259001 #include <fcntl.h>
259002 #include <stdarg.h>
259003 #include <stddef.h>
259004 #include <string.h>
259005 #include <errno.h>
259006 #include <sys/osl6.h>
259007 #include <const.h>

```

```

259008 #include <limits.h>
259009 #include <stdio.h>
259010
259011 //-----
259012 FILE *
259013 freopen (const char *restrict path, const char *restrict mode,
259014         FILE *restrict fp)
259015 {
259016     int status;
259017     FILE *fp_new;
259018     //
259019     if (fp == NULL)
259020     {
259021         return (NULL);
259022     }
259023     //
259024     status = fclose (fp);
259025     if (status != 0)
259026     {
259027         fp->error = 1;
259028         return (NULL);
259029     }
259030     //
259031     fp_new = fopen (path, mode);
259032     //
259033     if (fp_new == NULL)
259034     {
259035         return (NULL);
259036     }
259037     //
259038     if (fp_new != fp)
259039     {
259040         fclose (fp_new);
259041         return (NULL);
259042     }
259043     //
259044     return (fp_new);
259045 }

```

lib/stdio/fscanf.c

<<

Si veda la sezione [u0.90](#).

```

260001 #include <stdio.h>
260002 //-----
260003 int
260004 fscanf (FILE *restrict fp, const char *restrict format, ...)
260005 {
260006     va_list ap;
260007     va_start (ap, format);
260008     return vscanf (fp, format, ap);
260009 }

```

lib/stdio/fseek.c

<<

Si veda la sezione [u0.43](#).

```

261001 #include <stdio.h>
261002 #include <unistd.h>
261003 //-----
261004 int
261005 fseek (FILE *fp, long int offset, int whence)
261006 {
261007     off_t off_new;
261008     off_new = lseek (fp->fdn, (off_t) offset, whence);
261009     if (off_new < 0)
261010     {
261011         fp->error = 1;
261012         return (-1);
261013     }
261014     else
261015     {
261016         fp->eof = 0;
261017         return (0);
261018     }
261019 }

```

lib/stdio/fseeko.c

<<

Si veda la sezione [u0.43](#).

```

262001 #include <stdio.h>
262002 #include <unistd.h>
262003 //-----
262004 int
262005 fseeko (FILE *fp, off_t offset, int whence)
262006 {
262007     off_t off_new;
262008     off_new = lseek (fp->fdn, offset, whence);
262009     if (off_new < 0)
262010     {
262011         fp->error = 1;
262012         return (-1);
262013     }
262014     else
262015     {
262016         return (0);
262017     }
262018 }

```

lib/stdio/fsetpos.c

<<

Si veda la sezione [u0.32](#).

```
2630001 #include <stdio.h>
2630002 //-----
2630003 int
2630004 fsetpos (FILE *restrict fp, fpos_t *restrict pos)
2630005 {
2630006     long int position;
2630007     //
2630008     if (fp != NULL)
2630009     {
2630010         position = fseek (fp, (long int) *pos, SEEK_SET);
2630011         if (position >= 0)
2630012         {
2630013             *pos = position;
2630014             return (0);
2630015         }
2630016     }
2630017     return (-1);
2630018 }
```

```
2670022         return (EOF);
2670023     }
2670024     //
2670025     // Valid read.
2670026     //
2670027     if (size_read == 0)
2670028     {
2670029         //
2670030         // If no character is ready inside the keyboard buffer, just
2670031         // retry.
2670032         //
2670033         continue;
2670034     }
2670035     //
2670036     // End of scan.
2670037     //
2670038     return (c);
2670039 }
2670040 }
```

lib/stdio/ftell.c

<<

Si veda la sezione [u0.46](#).

```
2640001 #include <stdio.h>
2640002 #include <unistd.h>
2640003 //-----
2640004 long int
2640005 ftell (FILE *fp)
2640006 {
2640007     return ((long int) lseek (fp->fdn, (off_t) 0, SEEK_CUR));
2640008 }
```

lib/stdio/gets.c

<<

Si veda la sezione [u0.33](#).

```
2680001 #include <stdio.h>
2680002 #include <sys/types.h>
2680003 #include <unistd.h>
2680004 #include <stddef.h>
2680005 //-----
2680006 char *
2680007 gets (char *string)
2680008 {
2680009     ssize_t size_read;
2680010     int b; // Index inside the string buffer.
2680011     //
2680012     for (b = 0; b++, string[b] = 0)
2680013     {
2680014         size_read = read (STDIN_FILENO, &string[b], (size_t) 1);
2680015         //
2680016         if (size_read <= 0)
2680017         {
2680018             //
2680019             // It is the end of file (zero) otherwise there is a
2680020             // problem (a negative value).
2680021             //
2680022             _stream[STDIN_FILENO].eof = 1;
2680023             string[b] = 0;
2680024             break;
2680025         }
2680026         //
2680027         if (string[b] == '\n')
2680028         {
2680029             b++;
2680030             string[b] = 0;
2680031             break;
2680032         }
2680033     }
2680034     //
2680035     // If 'b' is zero, nothing was read and 'NULL' is returned.
2680036     //
2680037     if (b == 0)
2680038     {
2680039         return (NULL);
2680040     }
2680041     else
2680042     {
2680043         return (string);
2680044     }
2680045 }
```

lib/stdio/ftello.c

<<

Si veda la sezione [u0.46](#).

```
2650001 #include <stdio.h>
2650002 #include <unistd.h>
2650003 //-----
2650004 off_t
2650005 ftello (FILE *fp)
2650006 {
2650007     return (lseek (fp->fdn, (off_t) 0, SEEK_CUR));
2650008 }
```

lib/stdio/fwrite.c

<<

Si veda la sezione [u0.48](#).

```
2660001 #include <unistd.h>
2660002 #include <stdio.h>
2660003 //-----
2660004 size_t
2660005 fwrite (const void *restrict buffer, size_t size, size_t nmemb,
2660006         FILE *restrict fp)
2660007 {
2660008     ssize_t size_written;
2660009     size_written = write (fp->fdn, buffer, (size_t) (size * nmemb));
2660010     if (size_written < 0)
2660011     {
2660012         fp->error = 1;
2660013         return ((size_t) 0);
2660014     }
2660015     else
2660016     {
2660017         return ((size_t) (size_written / size));
2660018     }
2660019 }
```

lib/stdio/perror.c

<<

Si veda la sezione [u0.77](#).

```
2690001 #include <stdio.h>
2690002 #include <errno.h>
2690003 #include <stddef.h>
2690004 #include <string.h>
2690005 //-----
2690006 void
2690007 perror (const char *string)
2690008 {
2690009     //
2690010     // If errno is zero, there is nothing to show.
2690011     //
2690012     if (errno == 0)
2690013     {
2690014         return;
2690015     }
2690016     //
2690017     // Show the string if there is one.
2690018     //
2690019     if (string != NULL && strlen (string) > 0)
2690020     {
2690021         printf ("%s: ", string);
2690022     }
2690023     //
2690024     // Show the translated error.
2690025     //
2690026     if (errfn[0] != 0 && errln != 0)
```

lib/stdio/getchar.c

<<

Si veda la sezione [u0.31](#).

```
2670001 #include <stdio.h>
2670002 #include <sys/types.h>
2670003 #include <unistd.h>
2670004 //-----
2670005 int
2670006 getchar (void)
2670007 {
2670008     ssize_t size_read;
2670009     int c; // Character read.
2670010     //
2670011     for (c = 0;;)
2670012     {
2670013         size_read = read (STDIN_FILENO, &c, (size_t) 1);
2670014         //
2670015         if (size_read <= 0)
2670016         {
2670017             //
2670018             // It is the end of file (zero) otherwise there is a
2670019             // problem (a negative value): return 'EOF'.
2670020             //
2670021             _stream[STDIN_FILENO].eof = 1;
```

```

2690027     {
2690028         printf ("%s:%u:%i] %s\n",
2690029                errfn, errln, errno, strerror (errno));
2690030     }
2690031     else
2690032     {
2690033         printf ("%i] %s\n", errno, strerror (errno));
2690034     }
2690035 }

```

lib/stdio/printf.c

« Si veda la sezione [u0.78](#).

```

2700001 #include <stdio.h>
2700002 //-----
2700003 int
2700004 printf (char *restrict format, ...)
2700005 {
2700006     va_list ap;
2700007     va_start (ap, format);
2700008     return (vprintf (format, ap));
2700009 }

```

lib/stdio/puts.c

« Si veda la sezione [u0.38](#).

```

2710001 #include <stdio.h>
2710002 //-----
2710003 int
2710004 puts (const char *string)
2710005 {
2710006     int status;
2710007     status = printf ("%s\n", string);
2710008     if (status < 0)
2710009     {
2710010         return (EOF);
2710011     }
2710012     else
2710013     {
2710014         return (status);
2710015     }
2710016 }

```

lib/stdio/rewind.c

« Si veda la sezione [u0.88](#).

```

2720001 #include <stdio.h>
2720002 //-----
2720003 void
2720004 rewind (FILE *fp)
2720005 {
2720006     (void) fseek (fp, 0L, SEEK_SET);
2720007     fp->error = 0;
2720008 }

```

lib/stdio/scanf.c

« Si veda la sezione [u0.90](#).

```

2730001 #include <stdio.h>
2730002 //-----
2730003 int
2730004 scanf (const char *restrict format, ...)
2730005 {
2730006     va_list ap;
2730007     va_start (ap, format);
2730008     return vfscanf (stdin, format, ap);
2730009 }

```

lib/stdio/setbuf.c

« Si veda la sezione [u0.93](#).

```

2740001 #include <stdio.h>
2740002 //-----
2740003 void
2740004 setbuf (FILE *restrict fp, char *restrict buffer)
2740005 {
2740006     //
2740007     // The os16 library does not have any buffered data.
2740008     //
2740009     return;
2740010 }

```

lib/stdio/setvbuf.c

« Si veda la sezione [u0.93](#).

```

2750001 #include <stdio.h>
2750002 //-----
2750003 int
2750004 setvbuf (FILE *restrict fp, char *restrict buffer, int buf_mode,

```

```

2760005     size_t size)
2760006     {
2760007         //
2760008         // The os16 library does not have any buffered data.
2760009         //
2760010         return (0);
2760011     }

```

lib/stdio/snprintf.c

« Si veda la sezione [u0.78](#).

```

2760001 #include <stdio.h>
2760002 #include <stdarg.h>
2760003 //-----
2760004 int
2760005 snprintf (char *restrict string, size_t size,
2760006           const char *restrict format, ...)
2760007     {
2760008         va_list ap;
2760009         va_start (ap, format);
2760010         return vsnprintf (string, size, format, ap);
2760011     }

```

lib/stdio/sprintf.c

« Si veda la sezione [u0.78](#).

```

2770001 #include <stdio.h>
2770002 #include <stdarg.h>
2770003 //-----
2770004 int
2770005 sprintf (char *restrict string, const char *restrict format,
2770006         ...)
2770007     {
2770008         va_list ap;
2770009         va_start (ap, format);
2770010         return vsnprintf (string, (size_t) BUFSIZ, format, ap);
2770011     }

```

lib/stdio/sscanf.c

« Si veda la sezione [u0.90](#).

```

2780001 #include <stdio.h>
2780002 //-----
2780003 int
2780004 sscanf (char *restrict string, const char *restrict format, ...)
2780005     {
2780006         va_list ap;
2780007         va_start (ap, format);
2780008         return vsscanf (string, format, ap);
2780009     }

```

lib/stdio/vfprintf.c

« Si veda la sezione [u0.128](#).

```

2790001 #include <stdio.h>
2790002 #include <sys/types.h>
2790003 #include <sys/os16.h>
2790004 #include <string.h>
2790005 #include <unistd.h>
2790006 //-----
2790007 int
2790008 vfprintf (FILE *fp, char *restrict format, va_list arg)
2790009     {
2790010         ssize_t size_written;
2790011         size_t size;
2790012         size_t size_total;
2790013         int status;
2790014         char string[BUFSIZ];
2790015         char *buffer = string;
2790016         //
2790017         buffer[0] = 0;
2790018         status = vsprintf (buffer, format, arg);
2790019         //
2790020         size = strlen (buffer);
2790021         if (size >= BUFSIZ)
2790022         {
2790023             size = BUFSIZ;
2790024         }
2790025         //
2790026         for (size_total = 0, size_written = 0;
2790027              size_total < size;
2790028              size_total += size_written, buffer += size_written)
2790029         {
2790030             size_written = write (fp->fdn, buffer, size - size_total);
2790031             if (size_written < 0)
2790032             {
2790033                 return (size_total);
2790034             }
2790035         }
2790036         return (size);
2790037     }

```

<

Si veda la sezione u0.129.

```

280001 #include <stdio.h>
280002
280003 //-----
280004 int vfscanf (FILE *restrict fp, const char *string,
280005             const char *restrict format, va_list ap);
280006 //-----
280007 int
280008 vfscanf (FILE *restrict fp, const char *restrict format, va_list ap)
280009 {
280010     return (vfscanf (fp, NULL, format, ap));
280011 }
280012 //-----

```

<

Si veda la sezione u0.129.

```

280001 #include <stdint.h>
280002 #include <stdbool.h>
280003 #include <stdlib.h>
280004 #include <string.h>
280005 #include <stdio.h>
280006 #include <stdarg.h>
280007 #include <ctype.h>
280008 #include <errno.h>
280009 #include <stddef.h>
280010 //-----
280011 //
280012 // This function is not standard and is able to do the work of both
280013 // 'vfscanf()' and 'vscanf()'.
280014 //
280015 //-----
280016 #define WIDTH_MAX      64
280017 //-----
280018 static intmax_t strtointmax (const char *restrict string,
280019                             char **restrict endptr, int base,
280020                             size_t max_width);
280021 static int      ass_or_eof (int consumed, int assigned);
280022 //-----
280023 int
280024 vfscanf (FILE *restrict fp, const char *string,
280025          const char *restrict format, va_list ap)
280026 {
280027     int      f = 0;           // Format index.
280028     char     buffer[BUFSIZ];
280029     const char *input = string; // Default.
280030     const char *start = input; // Default.
280031     char     *next = NULL;
280032     int      scanned = 0;
280033     //
280034     bool     stream = 0;
280035     bool     flag_star = 0;
280036     bool     specifier = 0;
280037     bool     specifier_flags = 0;
280038     bool     specifier_width = 0;
280039     bool     specifier_type = 0;
280040     bool     inverted = 0;
280041     //
280042     char     *ptr_char;
280043     signed char *ptr_schar;
280044     unsigned char *ptr_uchar;
280045     short int *ptr_sshort;
280046     unsigned short int *ptr_ushort;
280047     int *ptr_sint;
280048     unsigned int *ptr_uint;
280049     long int *ptr_slong;
280050     unsigned long int *ptr_ulong;
280051     intmax_t *ptr_simax;
280052     uintmax_t *ptr_uimax;
280053     size_t *ptr_size;
280054     ptrdiff_t *ptr_ptrdiff;
280055     void **ptr_void;
280056     //
280057     size_t width;
280058     char width_string[WIDTH_MAX+1];
280059     int w; // Index inside width string.
280060     int assigned = 0; // Assignment counter.
280061     int consumed = 0; // Consumed counter.
280062     //
280063     intmax_t value_i;
280064     uintmax_t value_u;
280065     //
280066     const char *end_format;
280067     const char *end_input;
280068     int count; // Generic counter.
280069     int index; // Generic index.
280070     bool ascii[128];
280071     //
280072     void *pstatus;
280073     //
280074     // Initialize some data.
280075     //
280076     width_string[0] = '\0';
280077     end_format = format + (strlen (format));
280078     //
280079     // Check arguments and find where input comes.
280080     //

```

```

280081     if (fp == NULL && (string == NULL || string[0] == 0))
280082     {
280083         errset (EINVAL); // Invalid argument.
280084         return (EOF);
280085     }
280086     //
280087     if (fp != NULL && string != NULL && string[0] != 0)
280088     {
280089         errset (EINVAL); // Invalid argument.
280090         return (EOF);
280091     }
280092     //
280093     if (fp != NULL)
280094     {
280095         stream = 1;
280096     }
280097     //
280098     //
280099     //
280100     for (;;)
280101     {
280102         if (stream)
280103         {
280104             pstatus = fgets (buffer, BUFSIZ, fp);
280105             //
280106             if (pstatus == NULL)
280107             {
280108                 return (ass_or_eof (consumed, assigned));
280109             }
280110             //
280111             input = buffer;
280112             start = input;
280113             next = NULL;
280114         }
280115         //
280116         // Calculate end input.
280117         //
280118         end_input = input + (strlen (input));
280119         //
280120         // Scan format and input strings. Index 'f' is not reset.
280121         //
280122         while (&format[f] < end_format && input < end_input)
280123         {
280124             if (!specifier)
280125             {
280126                 //----- The context is not inside a specifier.
280127                 if (isspace (format[f]))
280128                 {
280129                     //----- Space.
280130                     while (isspace (*input))
280131                     {
280132                         input++;
280133                     }
280134                     //
280135                     // Verify that the input string is not finished.
280136                     //
280137                     if (input[0] == 0)
280138                     {
280139                         //
280140                         // As the input string is finished, the format
280141                         // string index is not advanced, because there
280142                         // might be more spaces on the next line (if
280143                         // there is a next line, of course).
280144                         //
280145                         continue;
280146                     }
280147                     else
280148                     {
280149                         f++;
280150                         continue;
280151                     }
280152                 }
280153                 if (format[f] != '%')
280154                 {
280155                     //----- Ordinary character.
280156                     if (format[f] == *input)
280157                     {
280158                         input++;
280159                         f++;
280160                         continue;
280161                     }
280162                     else
280163                     {
280164                         return (ass_or_eof (consumed, assigned));
280165                     }
280166                 }
280167                 if (format[f] == '%' && format[f+1] == '%')
280168                 {
280169                     //----- Matching a literal '%'.
280170                     f++;
280171                     if (format[f] == *input)
280172                     {
280173                         input++;
280174                         f++;
280175                         continue;
280176                     }
280177                     else
280178                     {
280179                         return (ass_or_eof (consumed, assigned));
280180                     }
280181                 }

```

```

280182         if (format[f] == '%')
280183         {
280184             //----- Percent of a specifier.
280185             f++;
280186             specifier      = 1;
280187             specifier_flags = 1;
280188             continue;
280189         }
280190     }
280191     //
280192     if (specifier && specifier_flags)
280193     {
280194         //----- The context is inside specifier flags.
280195         if (format[f] == '+')
280196         {
280197             //----- Assignment suppression star.
280198             flag_star = 1;
280199             f++;
280200         }
280201         else
280202         {
280203             //----- End of flags and begin of specifier length.
280204             specifier_flags = 0;
280205             specifier_width = 1;
280206         }
280207     }
280208     //
280209     if (specifier && specifier_width)
280210     {
280211         //----- The context is inside a specifier width.
280212         for (w = 0;
280213              format[f] >= '0'
280214              && format[f] <= '9'
280215              && w < WIDTH_MAX;
280216              w++)
280217         {
280218             width_string[w] = format[f];
280219             f++;
280220         }
280221         width_string[w] = '\0';
280222         width = atoi (width_string);
280223         if (width > WIDTH_MAX)
280224         {
280225             width = WIDTH_MAX;
280226         }
280227         //
280228         // A zero width means an unspecified limit for the field
280229         // length.
280230         //
280231         //----- End of spec. width and begin of spec. type.
280232         specifier_width = 0;
280233         specifier_type  = 1;
280234     }
280235     //
280236     if (specifier && specifier_type)
280237     {
280238         //
280239         // Specifiers with length modifier.
280240         //
280241         if (format[f] == 'h' && format[f+1] == 'h')
280242         {
280243             //----- char.
280244             if (format[f+2] == 'd')
280245             {
280246                 //----- signed char, base 10.
280247                 value_i = strtointmax (input, &next, 10, width);
280248                 if (input == next)
280249                 {
280250                     return (ass_or_eof (consumed, assigned));
280251                 }
280252                 consumed++;
280253                 if (!flag_star)
280254                 {
280255                     ptr_schar = va_arg (ap, signed char *);
280256                     *ptr_schar = value_i;
280257                     assigned++;
280258                 }
280259                 f += 3;
280260                 input = next;
280261             }
280262             else if (format[f+2] == 'i')
280263             {
280264                 //----- signed char, base unknown.
280265                 value_i = strtointmax (input, &next, 0, width);
280266                 if (input == next)
280267                 {
280268                     return (ass_or_eof (consumed, assigned));
280269                 }
280270                 consumed++;
280271                 if (!flag_star)
280272                 {
280273                     ptr_schar = va_arg (ap, signed char *);
280274                     *ptr_schar = value_i;
280275                     assigned++;
280276                 }
280277                 f += 3;
280278                 input = next;
280279             }
280280             else if (format[f+2] == 'o')
280281             {
280282                 //----- signed char, base 8.

```

```

280283         value_i = strtointmax (input, &next, 8, width);
280284         if (input == next)
280285         {
280286             return (ass_or_eof (consumed, assigned));
280287         }
280288         consumed++;
280289         if (!flag_star)
280290         {
280291             ptr_schar = va_arg (ap, signed char *);
280292             *ptr_schar = value_i;
280293             assigned++;
280294         }
280295         f += 3;
280296         input = next;
280297     }
280298     else if (format[f+2] == 'u')
280299     {
280300         //----- unsigned char, base 10.
280301         value_u = strtointmax (input, &next, 10, width);
280302         if (input == next)
280303         {
280304             return (ass_or_eof (consumed, assigned));
280305         }
280306         consumed++;
280307         if (!flag_star)
280308         {
280309             ptr_uchar = va_arg (ap, unsigned char *);
280310             *ptr_uchar = value_u;
280311             assigned++;
280312         }
280313         f += 3;
280314         input = next;
280315     }
280316     else if (format[f+2] == 'x' || format[f+2] == 'X')
280317     {
280318         //----- signed char, base 16.
280319         value_i = strtointmax (input, &next, 16, width);
280320         if (input == next)
280321         {
280322             return (ass_or_eof (consumed, assigned));
280323         }
280324         consumed++;
280325         if (!flag_star)
280326         {
280327             ptr_schar = va_arg (ap, signed char *);
280328             *ptr_schar = value_i;
280329             assigned++;
280330         }
280331         f += 3;
280332         input = next;
280333     }
280334     else if (format[f+2] == 'n')
280335     {
280336         //----- signed char, string index counter.
280337         ptr_schar = va_arg (ap, signed char *);
280338         *ptr_schar = (signed char)
280339                     (input - start + scanned);
280340         f += 3;
280341     }
280342     else
280343     {
280344         //----- unsupported or unknown specifier.
280345         f += 2;
280346     }
280347 }
280348 else if (format[f] == 'h')
280349 {
280350     //----- short.
280351     if (format[f+1] == 'd')
280352     {
280353         //----- signed short, base 10.
280354         value_i = strtointmax (input, &next, 10, width);
280355         if (input == next)
280356         {
280357             return (ass_or_eof (consumed, assigned));
280358         }
280359         consumed++;
280360         if (!flag_star)
280361         {
280362             ptr_sshort = va_arg (ap, signed short *);
280363             *ptr_sshort = value_i;
280364             assigned++;
280365         }
280366         f += 2;
280367         input = next;
280368     }
280369     else if (format[f+1] == 'i')
280370     {
280371         //----- signed short, base unknown.
280372         value_i = strtointmax (input, &next, 0, width);
280373         if (input == next)
280374         {
280375             return (ass_or_eof (consumed, assigned));
280376         }
280377         consumed++;
280378         if (!flag_star)
280379         {
280380             ptr_sshort = va_arg (ap, signed short *);
280381             *ptr_sshort = value_i;
280382             assigned++;
280383         }

```

```

2810384         f += 2;
2810385         input = next;
2810386     }
2810387     else if (format[f+1] == 'o')
2810388     {
2810389         //----- signed short, base 8.
2810390         value_i = strtointmax (input, &next, 8, width);
2810391         if (input == next)
2810392         {
2810393             return (ass_or_eof (consumed, assigned));
2810394         }
2810395         consumed++;
2810396         if (!flag_star)
2810397         {
2810398             ptr_sshort = va_arg (ap, signed short *);
2810399             *ptr_sshort = value_i;
2810400             assigned++;
2810401         }
2810402         f += 2;
2810403         input = next;
2810404     }
2810405     else if (format[f+1] == 'u')
2810406     {
2810407         //----- unsigned short, base 10.
2810408         value_u = strtointmax (input, &next, 10, width);
2810409         if (input == next)
2810410         {
2810411             return (ass_or_eof (consumed, assigned));
2810412         }
2810413         consumed++;
2810414         if (!flag_star)
2810415         {
2810416             ptr_ushort = va_arg (ap, unsigned short *);
2810417             *ptr_ushort = value_u;
2810418             assigned++;
2810419         }
2810420         f += 2;
2810421         input = next;
2810422     }
2810423     else if (format[f+1] == 'x' || format[f+2] == 'X')
2810424     {
2810425         //----- signed short, base 16.
2810426         value_i = strtointmax (input, &next, 16, width);
2810427         if (input == next)
2810428         {
2810429             return (ass_or_eof (consumed, assigned));
2810430         }
2810431         consumed++;
2810432         if (!flag_star)
2810433         {
2810434             ptr_sshort = va_arg (ap, signed short *);
2810435             *ptr_sshort = value_i;
2810436             assigned++;
2810437         }
2810438         f += 2;
2810439         input = next;
2810440     }
2810441     else if (format[f+1] == 'n')
2810442     {
2810443         //----- signed char, string index counter.
2810444         ptr_sshort = va_arg (ap, signed short *);
2810445         *ptr_sshort = (signed short)
2810446             (input - start + scanned);
2810447         f += 2;
2810448     }
2810449     else
2810450     {
2810451         //----- unsupported or unknown specifier.
2810452         f += 1;
2810453     }
2810454 }
2810455 //----- There is no 'long long int'.
2810456 else if (format[f] == 'l')
2810457 {
2810458     //----- long int.
2810459     if (format[f+1] == 'd')
2810460     {
2810461         //----- signed long, base 10.
2810462         value_i = strtointmax (input, &next, 10, width);
2810463         if (input == next)
2810464         {
2810465             return (ass_or_eof (consumed, assigned));
2810466         }
2810467         consumed++;
2810468         if (!flag_star)
2810469         {
2810470             ptr_slong = va_arg (ap, signed long *);
2810471             *ptr_slong = value_i;
2810472             assigned++;
2810473         }
2810474         f += 2;
2810475         input = next;
2810476     }
2810477     else if (format[f+1] == 'i')
2810478     {
2810479         //----- signed long, base unknown.
2810480         value_i = strtointmax (input, &next, 0, width);
2810481         if (input == next)
2810482         {
2810483             return (ass_or_eof (consumed, assigned));
2810484         }

```

```

2810485         consumed++;
2810486         if (!flag_star)
2810487         {
2810488             ptr_slong = va_arg (ap, signed long *);
2810489             *ptr_slong = value_i;
2810490             assigned++;
2810491         }
2810492         f += 2;
2810493         input = next;
2810494     }
2810495     else if (format[f+1] == 'o')
2810496     {
2810497         //----- signed long, base 8.
2810498         value_i = strtointmax (input, &next, 8, width);
2810499         if (input == next)
2810500         {
2810501             return (ass_or_eof (consumed, assigned));
2810502         }
2810503         consumed++;
2810504         if (!flag_star)
2810505         {
2810506             ptr_slong = va_arg (ap, signed long *);
2810507             *ptr_slong = value_i;
2810508             assigned++;
2810509         }
2810510         f += 2;
2810511         input = next;
2810512     }
2810513     else if (format[f+1] == 'u')
2810514     {
2810515         //----- unsigned long, base 10.
2810516         value_u = strtointmax (input, &next, 10, width);
2810517         if (input == next)
2810518         {
2810519             return (ass_or_eof (consumed, assigned));
2810520         }
2810521         consumed++;
2810522         if (!flag_star)
2810523         {
2810524             ptr_ulong = va_arg (ap, unsigned long *);
2810525             *ptr_ulong = value_u;
2810526             assigned++;
2810527         }
2810528         f += 2;
2810529         input = next;
2810530     }
2810531     else if (format[f+1] == 'x' || format[f+2] == 'X')
2810532     {
2810533         //----- signed long, base 16.
2810534         value_i = strtointmax (input, &next, 16, width);
2810535         if (input == next)
2810536         {
2810537             return (ass_or_eof (consumed, assigned));
2810538         }
2810539         consumed++;
2810540         if (!flag_star)
2810541         {
2810542             ptr_slong = va_arg (ap, signed long *);
2810543             *ptr_slong = value_i;
2810544             assigned++;
2810545         }
2810546         f += 2;
2810547         input = next;
2810548     }
2810549     else if (format[f+1] == 'n')
2810550     {
2810551         //----- signed char, string index counter.
2810552         ptr_slong = va_arg (ap, signed long *);
2810553         *ptr_slong = (signed long)
2810554             (input - start + scanned);
2810555         f += 2;
2810556     }
2810557     else
2810558     {
2810559         //----- unsupported or unknown specifier.
2810560         f += 1;
2810561     }
2810562 }
2810563 else if (format[f] == 'j')
2810564 {
2810565     //----- intmax_t.
2810566     if (format[f+1] == 'd')
2810567     {
2810568         //----- intmax_t, base 10.
2810569         value_i = strtointmax (input, &next, 10, width);
2810570         if (input == next)
2810571         {
2810572             return (ass_or_eof (consumed, assigned));
2810573         }
2810574         consumed++;
2810575         if (!flag_star)
2810576         {
2810577             ptr_simax = va_arg (ap, intmax_t *);
2810578             *ptr_simax = value_i;
2810579             assigned++;
2810580         }
2810581         f += 2;
2810582         input = next;
2810583     }
2810584     else if (format[f+1] == 'i')
2810585     {

```

```

2810586 //----- intmax_t, base unknown.
2810587 value_i = strtointmax (input, &next, 0, width);
2810588 if (input == next)
2810589 {
2810590     return (ass_or_eof (consumed, assigned));
2810591 }
2810592 consumed++;
2810593 if (!flag_star)
2810594 {
2810595     ptr_simax = va_arg (ap, intmax_t *);
2810596     *ptr_simax = value_i;
2810597     assigned++;
2810598 }
2810599 f += 2;
2810600 input = next;
2810601 }
2810602 else if (format[f+1] == 'o')
2810603 {
2810604     //----- intmax_t, base 8.
2810605     value_i = strtointmax (input, &next, 8, width);
2810606     if (input == next)
2810607     {
2810608         return (ass_or_eof (consumed, assigned));
2810609     }
2810610     consumed++;
2810611     if (!flag_star)
2810612     {
2810613         ptr_simax = va_arg (ap, intmax_t *);
2810614         *ptr_simax = value_i;
2810615         assigned++;
2810616     }
2810617     f += 2;
2810618     input = next;
2810619 }
2810620 else if (format[f+1] == 'u')
2810621 {
2810622     //----- uintmax_t, base 10.
2810623     value_u = strtointmax (input, &next, 10, width);
2810624     if (input == next)
2810625     {
2810626         return (ass_or_eof (consumed, assigned));
2810627     }
2810628     consumed++;
2810629     if (!flag_star)
2810630     {
2810631         ptr_uimax = va_arg (ap, uintmax_t *);
2810632         *ptr_uimax = value_u;
2810633         assigned++;
2810634     }
2810635     f += 2;
2810636     input = next;
2810637 }
2810638 else if (format[f+1] == 'x' || format[f+2] == 'X')
2810639 {
2810640     //----- intmax_t, base 16.
2810641     value_i = strtointmax (input, &next, 16, width);
2810642     if (input == next)
2810643     {
2810644         return (ass_or_eof (consumed, assigned));
2810645     }
2810646     consumed++;
2810647     if (!flag_star)
2810648     {
2810649         ptr_simax = va_arg (ap, intmax_t *);
2810650         *ptr_simax = value_i;
2810651         assigned++;
2810652     }
2810653     f += 2;
2810654     input = next;
2810655 }
2810656 else if (format[f+1] == 'n')
2810657 {
2810658     //----- signed char, string index counter.
2810659     ptr_simax = va_arg (ap, intmax_t *);
2810660     *ptr_simax = (intmax_t)
2810661         (input - start + scanned);
2810662     f += 2;
2810663 }
2810664 else
2810665 {
2810666     //----- unsupported or unknown specifier.
2810667     f += 1;
2810668 }
2810669 }
2810670 else if (format[f] == 'z')
2810671 {
2810672     //----- size_t.
2810673     if (format[f+1] == 'd')
2810674     {
2810675         //----- size_t, base 10.
2810676         value_i = strtointmax (input, &next, 10, width);
2810677         if (input == next)
2810678         {
2810679             return (ass_or_eof (consumed, assigned));
2810680         }
2810681         consumed++;
2810682         if (!flag_star)
2810683         {
2810684             ptr_size = va_arg (ap, size_t *);
2810685             *ptr_size = value_i;
2810686             assigned++;

```

1770

```

2810687     }
2810688     f += 2;
2810689     input = next;
2810690 }
2810691 else if (format[f+1] == 'i')
2810692 {
2810693     //----- size_t, base unknown.
2810694     value_i = strtointmax (input, &next, 0, width);
2810695     if (input == next)
2810696     {
2810697         return (ass_or_eof (consumed, assigned));
2810698     }
2810699     consumed++;
2810700     if (!flag_star)
2810701     {
2810702         ptr_size = va_arg (ap, size_t *);
2810703         *ptr_size = value_i;
2810704         assigned++;
2810705     }
2810706     f += 2;
2810707     input = next;
2810708 }
2810709 else if (format[f+1] == 'o')
2810710 {
2810711     //----- size_t, base 8.
2810712     value_i = strtointmax (input, &next, 8, width);
2810713     if (input == next)
2810714     {
2810715         return (ass_or_eof (consumed, assigned));
2810716     }
2810717     consumed++;
2810718     if (!flag_star)
2810719     {
2810720         ptr_size = va_arg (ap, size_t *);
2810721         *ptr_size = value_i;
2810722         assigned++;
2810723     }
2810724     f += 2;
2810725     input = next;
2810726 }
2810727 else if (format[f+1] == 'u')
2810728 {
2810729     //----- size_t, base 10.
2810730     value_u = strtointmax (input, &next, 10, width);
2810731     if (input == next)
2810732     {
2810733         return (ass_or_eof (consumed, assigned));
2810734     }
2810735     consumed++;
2810736     if (!flag_star)
2810737     {
2810738         ptr_size = va_arg (ap, size_t *);
2810739         *ptr_size = value_u;
2810740         assigned++;
2810741     }
2810742     f += 2;
2810743     input = next;
2810744 }
2810745 else if (format[f+1] == 'x' || format[f+2] == 'X')
2810746 {
2810747     //----- size_t, base 16.
2810748     value_i = strtointmax (input, &next, 16, width);
2810749     if (input == next)
2810750     {
2810751         return (ass_or_eof (consumed, assigned));
2810752     }
2810753     consumed++;
2810754     if (!flag_star)
2810755     {
2810756         ptr_size = va_arg (ap, size_t *);
2810757         *ptr_size = value_i;
2810758         assigned++;
2810759     }
2810760     f += 2;
2810761     input = next;
2810762 }
2810763 else if (format[f+1] == 'n')
2810764 {
2810765     //----- signed char, string index counter.
2810766     ptr_size = va_arg (ap, size_t *);
2810767     *ptr_size = (size_t) (input - start + scanned);
2810768     f += 2;
2810769 }
2810770 else
2810771 {
2810772     //----- unsupported or unknown specifier.
2810773     f += 1;
2810774 }
2810775 }
2810776 else if (format[f] == 't')
2810777 {
2810778     //----- ptrdiff_t.
2810779     if (format[f+1] == 'd')
2810780     {
2810781         //----- ptrdiff_t, base 10.
2810782         value_i = strtointmax (input, &next, 10, width);
2810783         if (input == next)
2810784         {
2810785             return (ass_or_eof (consumed, assigned));
2810786         }
2810787         consumed++;

```

1771


```

2810788     if (!flag_star)
2810789     {
2810790         ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810791         *ptr_ptrdiff = value_i;
2810792         assigned++;
2810793     }
2810794     f += 2;
2810795     input = next;
2810796 }
2810797 else if (format[f+1] == 'i')
2810798 {
2810799     //----- ptrdiff_t, base unknown.
2810800     value_i = strtointmax (input, &next, 0, width);
2810801     if (input == next)
2810802     {
2810803         return (ass_or_eof (consumed, assigned));
2810804     }
2810805     consumed++;
2810806     if (!flag_star)
2810807     {
2810808         ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810809         *ptr_ptrdiff = value_i;
2810810         assigned++;
2810811     }
2810812     f += 2;
2810813     input = next;
2810814 }
2810815 else if (format[f+1] == 'o')
2810816 {
2810817     //----- ptrdiff_t, base 8.
2810818     value_i = strtointmax (input, &next, 8, width);
2810819     if (input == next)
2810820     {
2810821         return (ass_or_eof (consumed, assigned));
2810822     }
2810823     consumed++;
2810824     if (!flag_star)
2810825     {
2810826         ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810827         *ptr_ptrdiff = value_i;
2810828         assigned++;
2810829     }
2810830     f += 2;
2810831     input = next;
2810832 }
2810833 else if (format[f+1] == 'u')
2810834 {
2810835     //----- ptrdiff_t, base 10.
2810836     value_u = strtointmax (input, &next, 10, width);
2810837     if (input == next)
2810838     {
2810839         return (ass_or_eof (consumed, assigned));
2810840     }
2810841     consumed++;
2810842     if (!flag_star)
2810843     {
2810844         ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810845         *ptr_ptrdiff = value_u;
2810846         assigned++;
2810847     }
2810848     f += 2;
2810849     input = next;
2810850 }
2810851 else if (format[f+1] == 'x' || format[f+2] == 'X')
2810852 {
2810853     //----- ptrdiff_t, base 16.
2810854     value_i = strtointmax (input, &next, 16, width);
2810855     if (input == next)
2810856     {
2810857         return (ass_or_eof (consumed, assigned));
2810858     }
2810859     consumed++;
2810860     if (!flag_star)
2810861     {
2810862         ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810863         *ptr_ptrdiff = value_i;
2810864         assigned++;
2810865     }
2810866     f += 2;
2810867     input = next;
2810868 }
2810869 else if (format[f+1] == 'n')
2810870 {
2810871     //----- signed char, string index counter.
2810872     ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810873     *ptr_ptrdiff = (ptrdiff_t)
2810874         (input - start + scanned);
2810875     f += 2;
2810876 }
2810877 else
2810878 {
2810879     //----- unsupported or unknown specifier.
2810880     f += 1;
2810881 }
2810882 }
2810883 //
2810884 // Specifiers with no length modifier.
2810885 //
2810886 if (format[f] == 'd')
2810887 {
2810888     //----- signed short, base 10.

```

```

2810889     value_i = strtointmax (input, &next, 10, width);
2810890     if (input == next)
2810891     {
2810892         return (ass_or_eof (consumed, assigned));
2810893     }
2810894     consumed++;
2810895     if (!flag_star)
2810896     {
2810897         ptr_sshort = va_arg (ap, signed short *);
2810898         *ptr_sshort = value_i;
2810899         assigned++;
2810900     }
2810901     f += 1;
2810902     input = next;
2810903 }
2810904 else if (format[f] == 'i')
2810905 {
2810906     //----- signed int, base unknown.
2810907     value_i = strtointmax (input, &next, 0, width);
2810908     if (input == next)
2810909     {
2810910         return (ass_or_eof (consumed, assigned));
2810911     }
2810912     consumed++;
2810913     if (!flag_star)
2810914     {
2810915         ptr_sint = va_arg (ap, signed int *);
2810916         *ptr_sint = value_i;
2810917         assigned++;
2810918     }
2810919     f += 1;
2810920     input = next;
2810921 }
2810922 else if (format[f] == 'o')
2810923 {
2810924     //----- signed int, base 8.
2810925     value_i = strtointmax (input, &next, 8, width);
2810926     if (input == next)
2810927     {
2810928         return (ass_or_eof (consumed, assigned));
2810929     }
2810930     consumed++;
2810931     if (!flag_star)
2810932     {
2810933         ptr_sint = va_arg (ap, signed int *);
2810934         *ptr_sint = value_i;
2810935         assigned++;
2810936     }
2810937     f += 1;
2810938     input = next;
2810939 }
2810940 else if (format[f] == 'u')
2810941 {
2810942     //----- unsigned short, base 10.
2810943     value_u = strtointmax (input, &next, 10, width);
2810944     if (input == next)
2810945     {
2810946         return (ass_or_eof (consumed, assigned));
2810947     }
2810948     consumed++;
2810949     if (!flag_star)
2810950     {
2810951         ptr_uint = va_arg (ap, unsigned int *);
2810952         *ptr_uint = value_u;
2810953         assigned++;
2810954     }
2810955     f += 1;
2810956     input = next;
2810957 }
2810958 else if (format[f] == 'x' || format[f] == 'X')
2810959 {
2810960     //----- signed short, base 16.
2810961     value_i = strtointmax (input, &next, 16, width);
2810962     if (input == next)
2810963     {
2810964         return (ass_or_eof (consumed, assigned));
2810965     }
2810966     consumed++;
2810967     if (!flag_star)
2810968     {
2810969         ptr_sint = va_arg (ap, signed int *);
2810970         *ptr_sint = value_i;
2810971         assigned++;
2810972     }
2810973     f += 1;
2810974     input = next;
2810975 }
2810976 else if (format[f] == 'c')
2810977 {
2810978     //----- char[].
2810979     if (width == 0) width = 1;
2810980     //
2810981     if (!flag_star) ptr_char = va_arg (ap, char *);
2810982     //
2810983     for (count = 0;
2810984         width > 0 && *input != 0;
2810985         width--, ptr_char++, input++)
2810986     {
2810987         if (!flag_star) *ptr_char = *input;
2810988         //
2810989         count++;

```

```

281090     }
281091     //
281092     if (count)          consumed++;
281093     if (count && !flag_star) assigned++;
281094     //
281095     f += 1;
281096   }
281097   else if (format[f] == 's')
281098   {
281099     //----- string.
281100     if (!flag_star) ptr_char = va_arg (ap, char *);
281101     //
281102     for (count = 0;
281103          lisspace (*input) && *input != 0;
281104          ptr_char++, input++)
281105     {
281106       if (!flag_star) *ptr_char = *input;
281107       //
281108       count++;
281109     }
281110     if (!flag_star) *ptr_char = 0;
281111     //
281112     if (count)          consumed++;
281113     if (count && !flag_star) assigned++;
281114     //
281115     f += 1;
281116   }
281117   else if (format[f] == '[')
281118   {
281119     //
281120     f++;
281121     //
281122     if (format[f] == '^')
281123     {
281124       inverted = 1;
281125       f++;
281126     }
281127     else
281128     {
281129       inverted = 0;
281130     }
281131     //
281132     // Reset ascii array.
281133     //
281134     for (index = 0; index < 128; index++)
281135     {
281136       ascii[index] = inverted;
281137     }
281138     //
281139     //
281140     //
281141     for (count = 0; &format[f] < end_format; count++)
281142     {
281143       if (format[f] == '[' && count > 0)
281144       {
281145         break;
281146       }
281147       //
281148       // Check for an interval.
281149       //
281150       if (format[f+1] == '-'
281151           && format[f+2] != '['
281152           && format[f+2] != 0)
281153       {
281154         //
281155         // Interval.
281156         //
281157         for (index = format[f];
281158              index <= format[f+2];
281159              index++)
281160         {
281161           ascii[index] = !inverted;
281162         }
281163         f += 3;
281164         continue;
281165       }
281166       //
281167       // Single character.
281168       //
281169       index = format[f];
281170       ascii[index] = !inverted;
281171       f++;
281172     }
281173     //
281174     // Is the scan correctly finished?.
281175     //
281176     if (format[f] != ']')
281177     {
281178       return (ass_or_eof (consumed, assigned));
281179     }
281180     //
281181     // The ascii table is populated.
281182     //
281183     if (width == 0) width = SIZE_MAX;
281184     //
281185     // Scan the input string.
281186     //
281187     if (!flag_star) ptr_char = va_arg (ap, char *);
281188     //
281189     for (count = 0;
281190          width > 0 && *input != 0;

```

```

281091     width--, ptr_char++, input++)
281092   {
281093     index = *input;
281094     if (ascii[index])
281095     {
281096       if (!flag_star) *ptr_char = *input;
281097       count++;
281098     }
281099     else
281100     {
281101       break;
281102     }
281103   }
281104   //
281105   if (count)          consumed++;
281106   if (count && !flag_star) assigned++;
281107   //
281108   f += 1;
281109   }
281110   else if (format[f] == 'p')
281111   {
281112     //----- void *.
281113     value_i = strtointmax (input, &next, 16, width);
281114     if (input == next)
281115     {
281116       return (ass_or_eof (consumed, assigned));
281117     }
281118     consumed++;
281119     if (!flag_star)
281120     {
281121       ptr_void = va_arg (ap, void **);
281122       *ptr_void = (void *) ((int) value_i);
281123       assigned++;
281124     }
281125     f += 1;
281126     input = next;
281127   }
281128   else if (format[f] == 'n')
281129   {
281130     //----- signed char, string index counter.
281131     ptr_sint = va_arg (ap, signed int *);
281132     *ptr_sint = (signed char) (input - start + scanned);
281133     f += 1;
281134   }
281135   else
281136   {
281137     //----- unsupported or unknown specifier.
281138     ;
281139   }
281140   //-----
281141   // End of specifier.
281142   //-----
281143   width_string[0] = '\0';
281144   specifier       = 0;
281145   specifier_flags = 0;
281146   specifier_width = 0;
281147   specifier_type  = 0;
281148   flag_star      = 0;
281149   }
281150   //
281151   // The format or the input string is terminated.
281152   //
281153   if (&format[f] < end_format && stream)
281154   {
281155     //
281156     // Only the input string is finished, and the input comes
281157     // from a stream, so another read will be done.
281158     //
281159     scanned += (int) (input - start);
281160     continue;
281161   }
281162   //
281163   // The format string is terminated.
281164   //
281165   return (ass_or_eof (consumed, assigned));
281166   }
281167   }
281168   //-----
281169   static intmax_t
281170   strtointmax (const char *restrict string, char **endptr,
281171               int base, size_t max_width)
281172   {
281173     int i;
281174     int d;          // Digits counter.
281175     int sign = +1;
281176     intmax_t number;
281177     intmax_t previous;
281178     int digit;
281179     //
281180     bool flag_prefix_oct = 0;
281181     bool flag_prefix_exa = 0;
281182     bool flag_prefix_dec = 0;
281183     //
281184     // If the 'max_width' value is zero, fix it to the maximum
281185     // that it can represent.
281186     //
281187     if (max_width == 0)

```

```

281192     {
281193         max_width = SIZE_MAX;
281194     }
281195     //
281196     // Eat initial spaces, but if there are spaces, there is an
281197     // error inside the calling function!
281198     //
281199     for (i = 0; isspace (string[i]); i++)
281200     {
281201         fprintf (stderr, "libc error: file \"%s\", line %i\n",
281202                 _FILE_, _LINE_);
281203     }
281204     //
281205     // Check sign. The 'max_width' counts also the sign, if there is
281206     // one.
281207     //
281208     if (string[i] == '+')
281209     {
281210         sign = +1;
281211         i++;
281212         max_width--;
281213     }
281214     else if (string[i] == '-')
281215     {
281216         sign = -1;
281217         i++;
281218         max_width--;
281219     }
281220     //
281221     // Check for prefix.
281222     //
281223     if (string[i] == '0')
281224     {
281225         if (string[i+1] == 'x' || string[i+1] == 'X')
281226         {
281227             flag_prefix_hex = 1;
281228         }
281229         if (isdigit (string[i+1]))
281230         {
281231             flag_prefix_oct = 1;
281232         }
281233     }
281234     //
281235     if (string[i] > '0' && string[i] <= '9')
281236     {
281237         flag_prefix_dec = 1;
281238     }
281239     //
281240     // Check compatibility with requested base.
281241     //
281242     if (flag_prefix_hex)
281243     {
281244         if (base == 0)
281245         {
281246             base = 16;
281247         }
281248         else if (base == 16)
281249         {
281250             ; // Ok.
281251         }
281252         else
281253         {
281254             //
281255             // Incompatible sequence: only the initial zero is reported.
281256             //
281257             *endptr = &string[i+1];
281258             return ((intmax_t) 0);
281259         }
281260         //
281261         // Move on, after the '0x' prefix.
281262         //
281263         i += 2;
281264     }
281265     //
281266     if (flag_prefix_oct)
281267     {
281268         if (base == 0)
281269         {
281270             base = 8;
281271         }
281272         //
281273         // Move on, after the '0' prefix.
281274         //
281275         i += 1;
281276     }
281277     //
281278     if (flag_prefix_dec)
281279     {
281280         if (base == 0)
281281         {
281282             base = 10;
281283         }
281284     }
281285     //
281286     // Scan the string.
281287     //
281288     for (d = 0, number = 0; d < max_width && string[i] != 0; i++, d++)
281289     {
281290         if (string[i] >= '0' && string[i] <= '9')

```

1776

```

281293         digit = string[i] - '0';
281294     }
281295     else if (string[i] >= 'A' && string[i] <= 'F')
281296     {
281297         digit = string[i] - 'A' + 10;
281298     }
281299     else if (string[i] >= 'a' && string[i] <= 'f')
281300     {
281301         digit = string[i] - 'a' + 10;
281302     }
281303     else
281304     {
281305         digit = 999;
281306     }
281307     //
281308     // Give a sign to the digit.
281309     //
281310     digit *= sign;
281311     //
281312     // Compare with the base.
281313     //
281314     if (base > (digit * sign))
281315     {
281316         //
281317         // Check if the current digit can be safely computed.
281318         //
281319         previous = number;
281320         number *= base;
281321         number += digit;
281322         if (number / base != previous)
281323         {
281324             //
281325             // Out of range.
281326             //
281327             *endptr = &string[i+1];
281328             errset (ERANGE); // Result too large.
281329             if (sign > 0)
281330             {
281331                 return (INTMAX_MAX);
281332             }
281333             else
281334             {
281335                 return (INTMAX_MIN);
281336             }
281337         }
281338     }
281339     else
281340     {
281341         *endptr = &string[i];
281342         return (number);
281343     }
281344 }
281345 //
281346 // The string is finished or the max digits length is reached.
281347 //
281348 *endptr = &string[i];
281349 //
281350 return (number);
281351 }
281352 //-----
281353 static int
281354 ass_or_eof (int consumed, int assigned)
281355 {
281356     if (consumed == 0)
281357     {
281358         return (EOF);
281359     }
281360     else
281361     {
281362         return (assigned);
281363     }
281364 }
281365 //-----

```

lib/stdio/vprintf.c

Si veda la sezione u0.128.

```

282001 #include <stdio.h>
282002 #include <sys/types.h>
282003 #include <sys/os16.h>
282004 #include <string.h>
282005 #include <unistd.h>
282006 //-----
282007 int
282008 vprintf (char *restrict format, va_list arg)
282009 {
282010     ssize_t   size_written;
282011     size_t    size;
282012     size_t    size_total;
282013     int       status;
282014     char      string[BUFSIZ];
282015     char      *buffer = string;
282016
282017     buffer[0] = 0;
282018     status = vsprintf (buffer, format, arg);
282019
282020     size = strlen (buffer);
282021     if (size >= BUFSIZ)
282022     {
282023         size = BUFSIZ;

```

1777

```

2830024     }
2830025
2830026     for (size_total = 0, size_written = 0;
2830027         size_total < size;
2830028         size_total += size_written, buffer += size_written)
2830029     {
2830030         //
2830031         // Write to the standard output: file descriptor n. 1.
2830032         //
2830033         size_written = write (STDOUT_FILENO, buffer, size - size_total);
2830034         if (size_written < 0)
2830035             {
2830036                 return (size_total);
2830037             }
2830038     }
2830039     return (size);
2830040 }

```

lib/stdio/vscanf.c

<

Si veda la sezione [u0.129](#).

```

2830001 #include <stdio.h>
2830002 //-----
2830003 int
2830004 vscanf (const char *restrict format, va_list ap)
2830005 {
2830006     return (vscanf (stdin, format, ap));
2830007 }
2830008 //-----

```

lib/stdio/vsnprintf.c

<

Si veda la sezione [u0.128](#).

```

2840001 #include <stdint.h>
2840002 #include <stdbool.h>
2840003 #include <stdlib.h>
2840004 #include <string.h>
2840005 #include <stdio.h>
2840006 //-----
2840007 static size_t uimaxtoa (uintmax_t integer, char *buffer, int base,
2840008                        int uppercase, size_t size);
2840009 static size_t imaxtoa (intmax_t integer, char *buffer, int base,
2840010                       int uppercase, size_t size);
2840011 static size_t simaxtoa (intmax_t integer, char *buffer, int base,
2840012                        int uppercase, size_t size);
2840013 static size_t uimaxtoa_fill (uintmax_t integer, char *buffer, int base,
2840014                              int uppercase, int width, int filler,
2840015                              int max);
2840016 static size_t imaxtoa_fill (intmax_t integer, char *buffer, int base,
2840017                             int uppercase, int width, int filler,
2840018                             int max);
2840019 static size_t simaxtoa_fill (intmax_t integer, char *buffer, int base,
2840020                              int uppercase, int width, int filler,
2840021                              int max);
2840022 static size_t strtostri_fill (char *string, char *buffer, int width,
2840023                               int filler, int max);
2840024 //-----
2840025 int
2840026 vsnprintf (char *restrict string, size_t size,
2840027            const char *restrict format, va_list ap)
2840028 {
2840029     //
2840030     // We produce at most 'size-1' characters, + '\0'.
2840031     // 'size' is used also as the max size for internal
2840032     // strings, but only if it is not too big.
2840033     //
2840034     int f = 0;
2840035     int s = 0;
2840036     int remain = size - 1;
2840037     //
2840038     bool specifier = 0;
2840039     bool specifier_flags = 0;
2840040     bool specifier_width = 0;
2840041     bool specifier_precision = 0;
2840042     bool specifier_type = 0;
2840043     //
2840044     bool flag_plus = 0;
2840045     bool flag_minus = 0;
2840046     bool flag_space = 0;
2840047     bool flag_alternate = 0;
2840048     bool flag_zero = 0;
2840049     //
2840050     int alignment;
2840051     int filler;
2840052     //
2840053     intmax_t value_i;
2840054     uintmax_t value_ui;
2840055     char *value_cp;
2840056     //
2840057     size_t width;
2840058     size_t precision;
2840059     #define str_size BUFSIZ/2
2840060     char width_string[str_size];
2840061     char precision_string[str_size];
2840062     int w;
2840063     int p;
2840064     //
2840065     width_string[0] = '\0';

```

```

2840066     precision_string[0] = '\0';
2840067     //
2840068     while (format[f] != 0 && s < (size - 1))
2840069     {
2840070         if (!specifier)
2840071             {
2840072                 //----- The context is not inside a specifier.
2840073                 if (format[f] != '%$')
2840074                     {
2840075                         string[s] = format[f];
2840076                         s++;
2840077                         remain--;
2840078                         f++;
2840079                         continue;
2840080                     }
2840081                 if (format[f] == '%' && format[f+1] == '%')
2840082                     {
2840083                         string[s] = '%';
2840084                         f++;
2840085                         f++;
2840086                         s++;
2840087                         remain--;
2840088                         continue;
2840089                     }
2840090                 if (format[f] == '$')
2840091                     {
2840092                         f++;
2840093                         specifier = 1;
2840094                         specifier_flags = 1;
2840095                         continue;
2840096                     }
2840097             }
2840098         //
2840099         if (specifier && specifier_flags)
2840100             {
2840101                 //----- The context is inside specifier flags.
2840102                 if (format[f] == '+')
2840103                     {
2840104                         flag_plus = 1;
2840105                         f++;
2840106                         continue;
2840107                     }
2840108                 else if (format[f] == '-')
2840109                     {
2840110                         flag_minus = 1;
2840111                         f++;
2840112                         continue;
2840113                     }
2840114                 else if (format[f] == ' ')
2840115                     {
2840116                         flag_space = 1;
2840117                         f++;
2840118                         continue;
2840119                     }
2840120                 else if (format[f] == '#')
2840121                     {
2840122                         flag_alternate = 1;
2840123                         f++;
2840124                         continue;
2840125                     }
2840126                 else if (format[f] == '0')
2840127                     {
2840128                         flag_zero = 1;
2840129                         f++;
2840130                         continue;
2840131                     }
2840132                 else
2840133                     {
2840134                         specifier_flags = 0;
2840135                         specifier_width = 1;
2840136                     }
2840137             }
2840138         //
2840139         if (specifier && specifier_width)
2840140             {
2840141                 //----- The context is inside specifier width.
2840142                 for (w = 0; format[f] >= '0' && format[f] <= '9'
2840143                     && w < str_size; w++)
2840144                     {
2840145                         width_string[w] = format[f];
2840146                         f++;
2840147                     }
2840148                 width_string[w] = '\0';
2840149
2840150                 specifier_width = 0;
2840151
2840152                 if (format[f] == '.')
2840153                     {
2840154                         specifier_precision = 1;
2840155                         f++;
2840156                     }
2840157                 else
2840158                     {
2840159                         specifier_precision = 0;
2840160                         specifier_type = 1;
2840161                     }
2840162             }
2840163         //
2840164         if (specifier && specifier_precision)
2840165             {
2840166                 //----- The context is inside specifier precision.

```

```

2840167     for (p = 0; format[f] >= '0' && format[f] <= '9'
2840168         && p < str_size; p++)
2840169     {
2840170         precision_string[p] = format[f];
2840171         p++;
2840172     }
2840173     precision_string[p] = '\0';
2840174
2840175     specifier_precision = 0;
2840176     specifier_type      = 1;
2840177 }
2840178 //
2840179 if (specifier && specifier_type)
2840180 {
2840181     //----- The context is inside specifier type.
2840182     width      = atoi (width_string);
2840183     precision  = atoi (precision_string);
2840184     filler     = ' ';
2840185     if (flag_zero) filler = '0';
2840186     if (flag_space) filler = ' ';
2840187     alignment  = width;
2840188     if (flag_minus)
2840189     {
2840190         alignment = -alignment;
2840191         filler = ' '; // The filler character cannot
2840192                     // be zero, so it is black.
2840193     }
2840194     //
2840195     if (format[f] == 'h' && format[f+1] == 'h')
2840196     {
2840197         if (format[f+2] == 'd' || format[f+2] == 'i')
2840198         {
2840199             //----- signed char, base 10.
2840200             value_i = va_arg (ap, int);
2840201             if (flag_plus)
2840202             {
2840203                 s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840204                                     alignment, filler, remain);
2840205             }
2840206             else
2840207             {
2840208                 s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840209                                     alignment, filler, remain);
2840210             }
2840211             f += 3;
2840212         }
2840213         else if (format[f+2] == 'u')
2840214         {
2840215             //----- unsigned char, base 10.
2840216             value_ui = va_arg (ap, unsigned int);
2840217             s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840218                                 alignment, filler, remain);
2840219             f += 3;
2840220         }
2840221         else if (format[f+2] == 'o')
2840222         {
2840223             //----- unsigned char, base 8.
2840224             value_ui = va_arg (ap, unsigned int);
2840225             s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840226                                 alignment, filler, remain);
2840227             f += 3;
2840228         }
2840229         else if (format[f+2] == 'x')
2840230         {
2840231             //----- unsigned char, base 16.
2840232             value_ui = va_arg (ap, unsigned int);
2840233             s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840234                                 alignment, filler, remain);
2840235             f += 3;
2840236         }
2840237         else if (format[f+2] == 'X')
2840238         {
2840239             //----- unsigned char, base 16.
2840240             value_ui = va_arg (ap, unsigned int);
2840241             s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840242                                 alignment, filler, remain);
2840243             f += 3;
2840244         }
2840245         else
2840246         {
2840247             //----- unsupported or unknown specifier.
2840248             f += 2;
2840249         }
2840250     }
2840251     else if (format[f] == 'h')
2840252     {
2840253         if (format[f+1] == 'd' || format[f+1] == 'i')
2840254         {
2840255             //----- short int, base 10.
2840256             value_i = va_arg (ap, int);
2840257             if (flag_plus)
2840258             {
2840259                 s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840260                                     alignment, filler, remain);
2840261             }
2840262             else
2840263             {
2840264                 s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840265                                     alignment, filler, remain);
2840266             }
2840267             f += 2;

```

```

2840268     }
2840269     else if (format[f+1] == 'u')
2840270     {
2840271         //----- unsigned short int, base 10.
2840272         value_ui = va_arg (ap, unsigned int);
2840273         s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840274                             alignment, filler, remain);
2840275         f += 2;
2840276     }
2840277     else if (format[f+1] == 'o')
2840278     {
2840279         //----- unsigned short int, base 8.
2840280         value_ui = va_arg (ap, unsigned int);
2840281         s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840282                             alignment, filler, remain);
2840283         f += 2;
2840284     }
2840285     else if (format[f+1] == 'x')
2840286     {
2840287         //----- unsigned short int, base 16.
2840288         value_ui = va_arg (ap, unsigned int);
2840289         s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840290                             alignment, filler, remain);
2840291         f += 2;
2840292     }
2840293     else if (format[f+1] == 'X')
2840294     {
2840295         //----- unsigned short int, base 16.
2840296         value_ui = va_arg (ap, unsigned int);
2840297         s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840298                             alignment, filler, remain);
2840299         f += 2;
2840300     }
2840301     else
2840302     {
2840303         //----- unsupported or unknown specifier.
2840304         f += 1;
2840305     }
2840306 }
2840307 //-----
2840308 // There is no 'long long int'.
2840309 //-----
2840310
2840311     else if (format[f] == 'l')
2840312     {
2840313         if (format[f+1] == 'd' || format[f+1] == 'i')
2840314         {
2840315             //----- long int base 10.
2840316             value_i = va_arg (ap, long int);
2840317             if (flag_plus)
2840318             {
2840319                 s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840320                                     alignment, filler, remain);
2840321             }
2840322             else
2840323             {
2840324                 s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840325                                     alignment, filler, remain);
2840326             }
2840327             f += 2;
2840328         }
2840329         else if (format[f+1] == 'u')
2840330         {
2840331             //----- Unsigned long int base 10.
2840332             value_ui = va_arg (ap, unsigned long int);
2840333             s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840334                                 alignment, filler, remain);
2840335             f += 2;
2840336         }
2840337         else if (format[f+1] == 'o')
2840338         {
2840339             //----- Unsigned long int base 8.
2840340             value_ui = va_arg (ap, unsigned long int);
2840341             s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840342                                 alignment, filler, remain);
2840343             f += 2;
2840344         }
2840345         else if (format[f+1] == 'x')
2840346         {
2840347             //----- Unsigned long int base 16.
2840348             value_ui = va_arg (ap, unsigned long int);
2840349             s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840350                                 alignment, filler, remain);
2840351             f += 2;
2840352         }
2840353         else if (format[f+1] == 'X')
2840354         {
2840355             //----- Unsigned long int base 16.
2840356             value_ui = va_arg (ap, unsigned long int);
2840357             s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840358                                 alignment, filler, remain);
2840359             f += 2;
2840360         }
2840361         else
2840362         {
2840363             //----- unsupported or unknown specifier.
2840364             f += 1;
2840365         }
2840366     }
2840367 }
2840368     else if (format[f] == 'j')

```

```

2840369 {
2840370     if (format[f+1] == 'd' || format[f+1] == 'i')
2840371     {
2840372         //----- intmax_t base 10.
2840373         value_ui = va_arg (ap, intmax_t);
2840374         if (flag_plus)
2840375         {
2840376             s += simaxtoa_fill (value_ui, &string[s], 10, 0,
2840377                               alignment, filler, remain);
2840378         }
2840379         else
2840380         {
2840381             s += imaxtoa_fill (value_ui, &string[s], 10, 0,
2840382                               alignment, filler, remain);
2840383         }
2840384         f += 2;
2840385     }
2840386     else if (format[f+1] == 'u')
2840387     {
2840388         //----- uintmax_t base 10.
2840389         value_ui = va_arg (ap, uintmax_t);
2840390         s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840391                             alignment, filler, remain);
2840392         f += 2;
2840393     }
2840394     else if (format[f+1] == 'o')
2840395     {
2840396         //----- uintmax_t base 8.
2840397         value_ui = va_arg (ap, uintmax_t);
2840398         s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840399                             alignment, filler, remain);
2840400         f += 2;
2840401     }
2840402     else if (format[f+1] == 'x')
2840403     {
2840404         //----- uintmax_t base 16.
2840405         value_ui = va_arg (ap, uintmax_t);
2840406         s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840407                             alignment, filler, remain);
2840408         f += 2;
2840409     }
2840410     else if (format[f+1] == 'X')
2840411     {
2840412         //----- uintmax_t base 16.
2840413         value_ui = va_arg (ap, uintmax_t);
2840414         s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840415                             alignment, filler, remain);
2840416         f += 2;
2840417     }
2840418     else
2840419     {
2840420         //----- unsupported or unknown specifier.
2840421         f += 1;
2840422     }
2840423 }
2840424 else if (format[f] == 'z')
2840425 {
2840426     if (format[f+1] == 'd'
2840427         || format[f+1] == 'i'
2840428         || format[f+1] == 'i')
2840429     {
2840430         //----- size_t base 10.
2840431         value_ui = va_arg (ap, unsigned long int);
2840432         s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840433                             alignment, filler, remain);
2840434         f += 2;
2840435     }
2840436     else if (format[f+1] == 'o')
2840437     {
2840438         //----- size_t base 8.
2840439         value_ui = va_arg (ap, unsigned long int);
2840440         s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840441                             alignment, filler, remain);
2840442         f += 2;
2840443     }
2840444     else if (format[f+1] == 'x')
2840445     {
2840446         //----- size_t base 16.
2840447         value_ui = va_arg (ap, unsigned long int);
2840448         s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840449                             alignment, filler, remain);
2840450         f += 2;
2840451     }
2840452     else if (format[f+1] == 'X')
2840453     {
2840454         //----- size_t base 16.
2840455         value_ui = va_arg (ap, unsigned long int);
2840456         s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840457                             alignment, filler, remain);
2840458         f += 2;
2840459     }
2840460     else
2840461     {
2840462         //----- unsupported or unknown specifier.
2840463         f += 1;
2840464     }
2840465 }
2840466 else if (format[f] == 't')
2840467 {
2840468     if (format[f+1] == 'd' || format[f+1] == 'i')
2840469     {

```

```

2840470 //----- ptrdiff_t base 10.
2840471 value_i = va_arg (ap, long int);
2840472 if (flag_plus)
2840473 {
2840474     s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840475                         alignment, filler, remain);
2840476 }
2840477 else
2840478 {
2840479     s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840480                       alignment, filler, remain);
2840481 }
2840482 f += 2;
2840483 }
2840484 else if (format[f+1] == 'u')
2840485 {
2840486     //----- ptrdiff_t base 10, without sign.
2840487     value_ui = va_arg (ap, unsigned long int);
2840488     s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840489                         alignment, filler, remain);
2840490     f += 2;
2840491 }
2840492 else if (format[f+1] == 'o')
2840493 {
2840494     //----- ptrdiff_t base 8, without sign.
2840495     value_ui = va_arg (ap, unsigned long int);
2840496     s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840497                         alignment, filler, remain);
2840498     f += 2;
2840499 }
2840500 else if (format[f+1] == 'x')
2840501 {
2840502     //----- ptrdiff_t base 16, without sign.
2840503     value_ui = va_arg (ap, unsigned long int);
2840504     s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840505                         alignment, filler, remain);
2840506     f += 2;
2840507 }
2840508 else if (format[f+1] == 'X')
2840509 {
2840510     //----- ptrdiff_t base 16, without sign.
2840511     value_ui = va_arg (ap, unsigned long int);
2840512     s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840513                         alignment, filler, remain);
2840514     f += 2;
2840515 }
2840516 else
2840517 {
2840518     //----- unsupported or unknown specifier.
2840519     f += 1;
2840520 }
2840521 }
2840522 if (format[f] == 'd' || format[f] == 'i')
2840523 {
2840524     //----- int base 10.
2840525     value_i = va_arg (ap, int);
2840526     if (flag_plus)
2840527     {
2840528         s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840529                             alignment, filler, remain);
2840530     }
2840531     else
2840532     {
2840533         s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840534                             alignment, filler, remain);
2840535     }
2840536     f += 1;
2840537 }
2840538 else if (format[f] == 'u')
2840539 {
2840540     //----- unsigned int base 10.
2840541     value_ui = va_arg (ap, unsigned int);
2840542     s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840543                         alignment, filler, remain);
2840544     f += 1;
2840545 }
2840546 else if (format[f] == 'o')
2840547 {
2840548     //----- unsigned int base 8.
2840549     value_ui = va_arg (ap, unsigned int);
2840550     s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840551                         alignment, filler, remain);
2840552     f += 1;
2840553 }
2840554 else if (format[f] == 'x')
2840555 {
2840556     //----- unsigned int base 16.
2840557     value_ui = va_arg (ap, unsigned int);
2840558     s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840559                         alignment, filler, remain);
2840560     f += 1;
2840561 }
2840562 else if (format[f] == 'X')
2840563 {
2840564     //----- unsigned int base 16.
2840565     value_ui = va_arg (ap, unsigned int);
2840566     s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840567                         alignment, filler, remain);
2840568     f += 1;
2840569 }
2840570 else if (format[f] == 'c')

```

```

284071     {
284072         //----- unsigned char.
284073         value_ui = va_arg (ap, unsigned int);
284074         string[s] = (char) value_ui;
284075         s += 1;
284076         f += 1;
284077     }
284078     else if (format[f] == 's')
284079     {
284080         //----- string.
284081         value_cp = va_arg (ap, char *);
284082         filler = ' ';
284083
284084         s += strtost_r_fill (value_cp, &string[s], alignment,
284085                             filler, remain);
284086         f += 1;
284087     }
284088     else
284089     {
284090         //----- unsupported or unknown specifier.
284091         ;
284092     }
284093     //-----
284094     // End of specifier.
284095     //-----
284096     width_string[0] = '\0';
284097     precision_string[0] = '\0';
284098
284099     specifier           = 0;
284100     specifier_flags     = 0;
284101     specifier_width     = 0;
284102     specifier_precision = 0;
284103     specifier_type      = 0;
284104
284105     flag_plus          = 0;
284106     flag_minus        = 0;
284107     flag_space        = 0;
284108     flag_alterate     = 0;
284109     flag_zero         = 0;
284110 }
284111 }
284112 string[s] = '\0';
284113 return s;
284114 }
284115 //-----
284116 // Static functions.
284117 //-----
284118 static size_t
284119 uimaxtoa (uintmax_t integer, char *buffer, int base, int uppercase,
284120           size_t size)
284121 {
284122     //-----
284123     // Convert a maximum rank integer into a string.
284124     //-----
284125     uintmax_t integer_copy = integer;
284126     size_t digits;
284127     int b;
284128     unsigned char remainder;
284129
284130     for (digits = 0; integer_copy > 0; digits++)
284131     {
284132         integer_copy = integer_copy / base;
284133     }
284134
284135     if (buffer == NULL && integer == 0) return 1;
284136     if (buffer == NULL && integer > 0) return digits;
284137
284138     if (integer == 0)
284139     {
284140         buffer[0] = '0';
284141         buffer[1] = '\0';
284142         return 1;
284143     }
284144     //
284145     // Fix the maximum number of digits.
284146     //
284147     if (size > 0 && digits > size) digits = size;
284148     //
284149     *(buffer + digits) = '\0'; // End of string.
284150
284151     for (b = digits - 1; integer != 0 && b >= 0; b--)
284152     {
284153         remainder = integer % base;
284154         integer = integer / base;
284155
284156         if (remainder <= 9)
284157         {
284158             *(buffer + b) = remainder + '0';
284159         }
284160         else
284161         {
284162             if (uppercase)
284163             {
284164                 *(buffer + b) = remainder - 10 + 'A';
284165             }
284166             else
284167             {
284168                 *(buffer + b) = remainder - 10 + 'a';
284169             }
284170         }
284171     }

```

1784

```

284072     }
284073     return digits;
284074 }
284075 //-----
284076 static size_t
284077 imaxtoa (intmax_t integer, char *buffer, int base, int uppercase,
284078          size_t size)
284079 {
284080     //-----
284081     // Convert a maximum rank integer with sign into a string.
284082     //-----
284083     if (integer >= 0)
284084     {
284085         return uimaxtoa (integer, buffer, base, uppercase, size);
284086     }
284087     //
284088     // At this point, there is a negative number, less than zero.
284089     //
284090     if (buffer == NULL)
284091     {
284092         return uimaxtoa (-integer, NULL, base, uppercase, size) + 1;
284093     }
284094     *buffer = '-'; // The minus sign is needed at the beginning.
284095     if (size == 1)
284096     {
284097         *(buffer + 1) = '\0';
284098         return 1;
284099     }
284100     else
284101     {
284102         return uimaxtoa (-integer, buffer+1, base, uppercase, size-1)
284103             + 1;
284104     }
284105 }
284106 //-----
284107 static size_t
284108 simaxtoa (intmax_t integer, char *buffer, int base, int uppercase,
284109           size_t size)
284110 {
284111     //-----
284112     // Convert a maximum rank integer with sign into a string, placing
284113     // the sign also if it is positive.
284114     //-----
284115     if (buffer == NULL && integer >= 0)
284116     {
284117         return uimaxtoa (integer, NULL, base, uppercase, size) + 1;
284118     }
284119     if (buffer == NULL && integer < 0)
284120     {
284121         return uimaxtoa (-integer, NULL, base, uppercase, size) + 1;
284122     }
284123     //
284124     // At this point, 'buffer' is different from NULL.
284125     //
284126     if (integer >= 0)
284127     {
284128         *buffer = '+';
284129     }
284130     else
284131     {
284132         *buffer = '-';
284133     }
284134     if (size == 1)
284135     {
284136         *(buffer + 1) = '\0';
284137         return 1;
284138     }
284139     if (integer >= 0)
284140     {
284141         return uimaxtoa (integer, buffer+1, base, uppercase, size-1)
284142             + 1;
284143     }
284144     else
284145     {
284146         return uimaxtoa (-integer, buffer+1, base, uppercase, size-1)
284147             + 1;
284148     }
284149 }
284150 //-----
284151 static size_t
284152 uimaxtoa_fill (uintmax_t integer, char *buffer, int base,
284153               int uppercase, int width, int filler, int max)
284154 {
284155     //-----
284156     // Convert a maximum rank integer without sign into a string,
284157     // taking care of the alignment.
284158     //-----
284159     size_t size_i;
284160     size_t size_f;
284161
284162     if (max < 0) return 0; // <max> deve essere un valore positivo.
284163
284164     size_i = uimaxtoa (integer, NULL, base, uppercase, 0);

```

1785

```

2840773 if (width > 0 && max > 0 && width > max) width = max;
2840774 if (width < 0 && -max < 0 && width < -max) width = -max;
2840775
2840776 if (size_i > abs (width))
2840777 {
2840778     return uimaxtoa (integer, buffer, base, uppercase, abs (width));
2840779 }
2840780
2840781 if (width == 0 && max > 0)
2840782 {
2840783     return uimaxtoa (integer, buffer, base, uppercase, max);
2840784 }
2840785
2840786 if (width == 0)
2840787 {
2840788     return uimaxtoa (integer, buffer, base, uppercase, abs (width));
2840789 }
2840790 //
2840791 // size_i <= abs (width).
2840792 //
2840793 size_f = abs (width) - size_i;
2840794
2840795 if (width < 0)
2840796 {
2840797     // Left alignment.
2840798     uimaxtoa (integer, buffer, base, uppercase, 0);
2840799     memset (buffer + size_i, filler, size_f);
2840800 }
2840801 else
2840802 {
2840803     // Right alignment.
2840804     memset (buffer, filler, size_f);
2840805     uimaxtoa (integer, buffer + size_f, base, uppercase, 0);
2840806 }
2840807 *(buffer + abs (width)) = '\0';
2840808
2840809 return abs (width);
2840810 }
2840811 //-----
2840812 static size_t
2840813 imaxtoa_fill (intmax_t integer, char *buffer, int base,
2840814              int uppercase, int width, int filler, int max)
2840815 {
2840816     //-----
2840817     // Convert a maximum rank integer with sign into a string,
2840818     // taking care of the alignment.
2840819     //-----
2840820
2840821     size_t size_i;
2840822     size_t size_f;
2840823
2840824     if (max < 0) return 0; // 'max' must be a positive value.
2840825
2840826     size_i = imaxtoa (integer, NULL, base, uppercase, 0);
2840827
2840828     if (width > 0 && max > 0 && width > max) width = max;
2840829     if (width < 0 && -max < 0 && width < -max) width = -max;
2840830
2840831     if (size_i > abs (width))
2840832     {
2840833         return imaxtoa (integer, buffer, base, uppercase, abs (width));
2840834     }
2840835
2840836     if (width == 0 && max > 0)
2840837     {
2840838         return imaxtoa (integer, buffer, base, uppercase, max);
2840839     }
2840840
2840841     if (width == 0)
2840842     {
2840843         return imaxtoa (integer, buffer, base, uppercase, abs (width));
2840844     }
2840845
2840846     // size_i <= abs (width).
2840847
2840848     size_f = abs (width) - size_i;
2840849
2840850     if (width < 0)
2840851     {
2840852         // Left alignment.
2840853         imaxtoa (integer, buffer, base, uppercase, 0);
2840854         memset (buffer + size_i, filler, size_f);
2840855     }
2840856     else
2840857     {
2840858         // Right alignment.
2840859         memset (buffer, filler, size_f);
2840860         imaxtoa (integer, buffer + size_f, base, uppercase, 0);
2840861     }
2840862     *(buffer + abs (width)) = '\0';
2840863
2840864     return abs (width);
2840865 }
2840866 //-----
2840867 static size_t
2840868 simaxtoa_fill (intmax_t integer, char *buffer, int base,
2840869              int uppercase, int width, int filler, int max)
2840870 {
2840871     //-----
2840872     // Convert a maximum rank integer with sign into a string,
2840873     // placing the sign also if it is positive and taking care of the

```

1786

```

2840874 // alignment.
2840875 //-----
2840876
2840877 size_t size_i;
2840878 size_t size_f;
2840879
2840880 if (max < 0) return 0; // 'max' must be a positive value.
2840881
2840882 size_i = simaxtoa (integer, NULL, base, uppercase, 0);
2840883
2840884 if (width > 0 && max > 0 && width > max) width = max;
2840885 if (width < 0 && -max < 0 && width < -max) width = -max;
2840886
2840887 if (size_i > abs (width))
2840888 {
2840889     return simaxtoa (integer, buffer, base, uppercase, abs (width));
2840890 }
2840891
2840892 if (width == 0 && max > 0)
2840893 {
2840894     return simaxtoa (integer, buffer, base, uppercase, max);
2840895 }
2840896
2840897 if (width == 0)
2840898 {
2840899     return simaxtoa (integer, buffer, base, uppercase, abs (width));
2840900 }
2840901 //
2840902 // size_i <= abs (width).
2840903 //
2840904 size_f = abs (width) - size_i;
2840905
2840906 if (width < 0)
2840907 {
2840908     // Left alignment.
2840909     simaxtoa (integer, buffer, base, uppercase, 0);
2840910     memset (buffer + size_i, filler, size_f);
2840911 }
2840912 else
2840913 {
2840914     // Right alignment.
2840915     memset (buffer, filler, size_f);
2840916     simaxtoa (integer, buffer + size_f, base, uppercase, 0);
2840917 }
2840918 *(buffer + abs (width)) = '\0';
2840919
2840920 return abs (width);
2840921 }
2840922 //-----
2840923 static size_t
2840924 strtosr_fill (char *string, char *buffer, int width, int filler,
2840925              int max)
2840926 {
2840927     //-----
2840928     // Transfer a string with care for the alignment.
2840929     //-----
2840930
2840931     size_t size_s;
2840932     size_t size_f;
2840933
2840934     if (max < 0) return 0; // 'max' must be a positive value.
2840935
2840936     size_s = strlen (string);
2840937
2840938     if (width > 0 && max > 0 && width > max) width = max;
2840939     if (width < 0 && -max < 0 && width < -max) width = -max;
2840940
2840941     if (width != 0 && size_s > abs (width))
2840942     {
2840943         memcpy (buffer, string, abs (width));
2840944         buffer[width] = '\0';
2840945         return width;
2840946     }
2840947
2840948     if (width == 0 && max > 0 && size_s > max)
2840949     {
2840950         memcpy (buffer, string, max);
2840951         buffer[max] = '\0';
2840952         return max;
2840953     }
2840954
2840955     if (width == 0 && max > 0 && size_s < max)
2840956     {
2840957         memcpy (buffer, string, size_s);
2840958         buffer[size_s] = '\0';
2840959         return size_s;
2840960     }
2840961     //
2840962     // width =! 0
2840963     // size_s <= abs (width)
2840964     //
2840965     size_f = abs (width) - size_s;
2840966
2840967     if (width < 0)
2840968     {
2840969         // Right alignment.
2840970         memset (buffer, filler, size_f);
2840971         strncpy (buffer+size_f, string, size_s);
2840972     }
2840973     else
2840974     {

```

1787


```

284975 // Left alignment.
284976 strncpy (buffer, string, size_s);
284977 memset (buffer+size_s, filler, size_f);
284978 }
284979 *(buffer + abs (width)) = '\0';
284980
284981 return abs (width);
284982 }
284983
284984

```

lib/stdio/vsprintf.c

<<

Si veda la sezione u0.128.

```

285001 #include <stdio.h>
285002 #include <sys/osl6.h>
285003 //-----
285004 int
285005 vsprintf (char *string, char *restrict format, va_list arg)
285006 {
285007     return (vsnprintf (string, BUFSIZ, format, arg));
285008 }

```

lib/stdio/vsscanf.c

<<

Si veda la sezione u0.129.

```

286001 #include <stdio.h>
286002
286003 //-----
286004 int vfscanf (FILE *restrict fp, const char *string,
286005             const char *restrict format, va_list ap);
286006 //-----
286007 int
286008 vsscanf (const char *string, const char *restrict format, va_list ap)
286009 {
286010     return (vfscanf (NULL, string, format, ap));
286011 }
286012 //-----

```

osl6: «lib/stdlib.h»

<<

Si veda la sezione u0.2.

```

287001 #ifndef _STDLIB_H
287002 #define _STDLIB_H 1
287003 //-----
287004
287005 #include <size_t.h>
287006 #include <wchar_t.h>
287007 #include <NULL.h>
287008 #include <limits.h>
287009 #include <const.h>
287010 #include <restrict.h>
287011 //-----
287012 typedef struct {
287013     int quot;
287014     int rem;
287015 } div_t;
287016 //-----
287017 typedef struct {
287018     long int quot;
287019     long int rem;
287020 } ldiv_t;
287021 //-----
287022 typedef void (*atexit_t) (void); // Non standard. [1]
287023 //
287024 // [1] The type 'atexit_t' is a pointer to a function for the "at exit"
287025 // procedure, with no parameters and returning void. With the
287026 // declaration of type 'atexit_t', the function prototype of
287027 // 'atexit()' is easier to declare and to understand. Original
287028 // declaration is:
287029 //
287030 // int atexit (void (*function) (void));
287031 //
287032 //-----
287033 #define EXIT_FAILURE 1
287034 #define EXIT_SUCCESS 0
287035 #define RAND_MAX INT_MAX
287036 #define MB_CUR_MAX ((size_t) MB_LEN_MAX)
287037 //-----
287038 void _Exit (int status);
287039 void abort (void);
287040 int abs (int j);
287041 int atexit (atexit_t function);
287042 int atoi (const char *string);
287043 long int atol (const char *string);
287044 //void *bsearch (const void *key, const void *base,
287045 // size_t nmemb, size_t size,
287046 // int (*compar) (const void *, const void *));
287047 #define calloc(b, s) (malloc ((b) * (s)))
287048 div_t div (int numer, int denom);
287049 void exit (int status);
287050 void free (void *ptr);
287051 char *getenv (const char *name);
287052 long int labs (long int j);
287053 ldiv_t ldiv (long int numer, long int denom);

```

1788

```

287054 void *malloc (size_t size);
287055 int putenv (const char *string);
287056 void qsort (void *base, size_t nmemb, size_t size,
287057            int (*compare) (const void *,
287058                            const void *));
287059 int rand (void);
287060 void *realloc (void *ptr, size_t size);
287061 int setenv (const char *name, const char *value,
287062            int overwrite);
287063 void srand (unsigned int seed);
287064 long int strtol (const char *restrict string,
287065                char **restrict endptr, int base);
287066 unsigned long int strtoul (const char *restrict string,
287067                            char ** restrict endptr, int base);
287068 //int system (const char *string);
287069 int unsetenv (const char *name);
287070
287071 #endif

```

lib/stdlib/_Exit.c

Si veda la sezione u0.2.

<<

```

288001 #include <stdlib.h>
288002 #include <sys/osl6.h>
288003 //-----
288004 void
288005 _Exit (int status)
288006 {
288007     sysmsg_exit_t msg;
288008     //
288009     // Only the low eight bit are returned.
288010     //
288011     msg.status = (status & 0xFF);
288012     //
288013     //
288014     //
288015     sys (SYS_EXIT, &msg, (sizeof msg));
288016     //
288017     // Should not return from system call, but if it does, loop
288018     // forever:
288019     //
288020     while (1);
288021 }

```

lib/stdlib/abort.c

<<

Si veda la sezione u0.2.

```

289001 #include <stdlib.h>
289002 #include <sys/types.h>
289003 #include <signal.h>
289004 #include <unistd.h>
289005 //-----
289006 void
289007 abort (void)
289008 {
289009     pid_t pid;
289010     sighandler_t sig_previous;
289011     //
289012     // Set 'SIGABRT' to a default action.
289013     //
289014     sig_previous = signal (SIGABRT, SIG_DFL);
289015     //
289016     // If the previous action was something different than symbolic
289017     // ones, configure again the previous action.
289018     //
289019     if (sig_previous != SIG_DFL &&
289020         sig_previous != SIG_IGN &&
289021         sig_previous != SIG_ERR)
289022     {
289023         signal (SIGABRT, sig_previous);
289024     }
289025     //
289026     // Get current process ID and sent the signal.
289027     //
289028     pid = getpid ();
289029     kill (pid, SIGABRT);
289030     //
289031     // Second chance
289032     //
289033     for (;;)
289034     {
289035         signal (SIGABRT, SIG_DFL);
289036         pid = getpid ();
289037         kill (pid, SIGABRT);
289038     }
289039 }

```

lib/stdlib/abs.c

<<

Si veda la sezione u0.3.

```

290001 #include <stdlib.h>
290002 //-----
290003 int
290004 abs (int j)
290005 {

```

1789

```

290006     if (j < 0)
290007     {
290008         return -j;
290009     }
290010     else
290011     {
290012         return j;
290013     }
290014 }

```

lib/stdlib/alloc.c

« Si veda la sezione u0.66.

```

290001 #include <stdlib.h>
290002 #include <string.h>
290003 #include <errno.h>
290004 #include <limits.h>
290005 #include <stdio.h>
290006 //-----
290007 #define MEMORY_BLOCK_SIZE    1024
290008 //-----
290009 static char  _alloc_memory[LONG_BIT][MEMORY_BLOCK_SIZE]; // [1]
290010 static size_t _alloc_size[LONG_BIT]; // [2]
290011 static long int _alloc_map; // [3]
290012 //
290013 // [1] Memory to be allocated.
290014 // [2] Sizes allocated.
290015 // [3] Memory block map. The memory map is made of a single integer and
290016 //     the rightmost bit is the first memory block.
290017 //-----
290018 void *
290019 malloc (size_t size)
290020 {
290021     size_t size_free; // Size free found that might be allocated.
290022     int m; // Index inside '_alloc_memory[...]' table.
290023     int s; // Start index for a free memory area.
290024     long int mask; // Mask to compare with '_alloc_map'.
290025     long int alloc; // New allocation map.
290026     //
290027     // Check for arguments.
290028     //
290029     if (size == 0)
290030     {
290031         return (NULL);
290032     }
290033     //
290034     for (s = 0, m = 0; m < LONG_BIT; m++)
290035     {
290036         mask = 1;
290037         mask <<= m;
290038         //
290039         if (_alloc_map & mask)
290040         {
290041             //
290042             // The memory block is not free.
290043             //
290044             s = m + 1;
290045             size_free = 0;
290046             alloc = 0;
290047         }
290048         else
290049         {
290050             alloc |= mask;
290051             size_free += MEMORY_BLOCK_SIZE;
290052         }
290053         if (size_free >= size)
290054         {
290055             //
290056             // Space found: update '_alloc_size[]' table, the map inside
290057             // '_alloc_map' and return the memory address.
290058             //
290059             _alloc_size[s] = size_free;
290060             _alloc_map |= alloc;
290061             return ((void *) &_alloc_memory[s][0]);
290062         }
290063     }
290064     //
290065     // No space left.
290066     //
290067     errset (ENOMEM); // Not enough space.
290068     //
290069     return (NULL);
290070 }
290071 //-----
290072 void
290073 free (void *address)
290074 {
290075     size_t size_free; // Size to make free.
290076     int m; // Index inside '_alloc_memory[...]' table.
290077     int s; // Start index.
290078     long int mask; // Mask to compare with '_alloc_map'.
290079     long int alloc; // New allocation map.
290080     //
290081     // Check argument.
290082     //
290083     if (address == NULL)
290084     {
290085         return;
290086     }
290087     //

```

1790

```

290088 // Find the original allocated address inside '_alloc_memory[...]'
290089 // table.
290090 //
290091 for (m = 0; m < LONG_BIT; m++)
290092 {
290093     if (address == (void *) &_alloc_memory[m][0])
290094     {
290095         //
290096         // This is the right memory block.
290097         //
290098         if (_alloc_size[m] == 0)
290099         {
290100             //
290101             // The block found is not allocated.
290102             //
290103             return;
290104         }
290105         else
290106         {
290107             //
290108             // Build the map of the memory to set free.
290109             //
290110             size_free = _alloc_size[m];
290111             for (alloc = 0, s = m;
290112                  size_free > 0 && s < LONG_BIT;
290113                  size_free -= MEMORY_BLOCK_SIZE, s++)
290114             {
290115                 mask = 1;
290116                 mask <<= s;
290117                 alloc |= mask;
290118             }
290119             //
290120             // Compare the map of memory to be freed with the
290121             // reality allocated one, then free the memory.
290122             //
290123             if ((_alloc_map & alloc) == alloc)
290124             {
290125                 _alloc_map &= ~alloc;
290126                 _alloc_size[m] = 0;
290127                 return;
290128             }
290129             //
290130             // The real map does not report the same amount of
290131             // allocated memory, so nothing is freed.
290132             //
290133             return;
290134         }
290135     }
290136 }
290137 //
290138 // Address not allocated.
290139 //
290140 return;
290141 }
290142 //-----
290143 void *
290144 realloc (void *address, size_t size)
290145 {
290146     char *address_new;
290147     char *address_old = (char *) address;
290148     size_t size_old = 0;
290149     size_t size_new = size;
290150     int m; // Index inside the memory table;
290151     //
290152     // Check arguments.
290153     //
290154     if (size == 0) return (NULL);
290155     if (address == NULL) return (malloc (size));
290156     //
290157     // Locate original allocation.
290158     //
290159     for (m = 0; m < LONG_BIT; m++)
290160     {
290161         if (address_old == (char *) &_alloc_memory[m][0])
290162         {
290163             size_old = _alloc_size[m];
290164             break;
290165         }
290166     }
290167     //
290168     // Check if a valid size was found.
290169     //
290170     if (size_old == 0)
290171     {
290172         //
290173         // Address not found or size not valid.
290174         //
290175         return (NULL);
290176     }
290177     //
290178     // Allocate the new memory.
290179     //
290180     address_new = malloc (size);
290181     //
290182     // Check allocation. If there is an error, the variable 'errno'
290183     // is already updated by 'malloc()'.
290184     //
290185     if (address_new == NULL)
290186     {
290187         return (NULL);
290188     }

```

1791

```

290189 //
290190 // Copy old memory.
290191 //
290192 for (; size_old > 0 && size_new > 0;
290193     size_old--, size_new--, address_new++, address_old++)
290194     {
290195     *address_new = *address_old;
290196     }
290197 //
290198 // Free old memory.
290199 //
290200 free (address);
290201 //
290202 // Return the new address.
290203 //
290204 return (address_new);
290205 }
290206 //-----

```

lib/stdlib/atexit.c

Si veda la sezione u0.4.

```

290001 #include <stdlib.h>
290002 #include <stdio.h>
290003 //-----
290004 atexit_t _atexit_table[ATEXIT_MAX];
290005 //-----
290006 void
290007 _atexit_setup (void)
290008 {
290009     int a;
290010     //
290011     for (a = 0; a < ATEXIT_MAX; a++)
290012     {
290013     _atexit_table[a] = NULL;
290014     }
290015 }
290016 //-----
290017 int
290018 atexit (atexit_t function)
290019 {
290020     int a;
290021     //
290022     if (function == NULL)
290023     {
290024     return (-1);
290025     }
290026     //
290027     for (a = 0; a < ATEXIT_MAX; a++)
290028     {
290029     if (_atexit_table[a] == NULL)
290030     {
290031     _atexit_table[a] = function;
290032     return (0);
290033     }
290034     }
290035     //
290036     return (-1);
290037 }

```

lib/stdlib/atol.c

Si veda la sezione u0.5.

```

290001 #include <stdlib.h>
290002 #include <ctype.h>
290003 //-----
290004 int
290005 atoi (const char *string)
290006 {
290007     int i;
290008     int sign = +1;
290009     int number;
290010     //
290011     for (i = 0; isspace (string[i]); i++)
290012     {
290013     ;
290014     }
290015     //
290016     if (string[i] == '+')
290017     {
290018     sign = +1;
290019     i++;
290020     }
290021     else if (string[i] == '-')
290022     {
290023     sign = -1;
290024     i++;
290025     }
290026     //
290027     for (number = 0; isdigit (string[i]); i++)
290028     {
290029     number *= 10;
290030     number += (string[i] - '0');
290031     }
290032     //
290033     number *= sign;
290034     //
290035     return number;

```

```

290036 }

```

lib/stdlib/atol.c

Si veda la sezione u0.5.

```

290001 #include <stdlib.h>
290002 #include <ctype.h>
290003 //-----
290004 long int
290005 atol (const char *string)
290006 {
290007     int i;
290008     int sign = +1;
290009     long int number;
290010     //
290011     for (i = 0; isspace (string[i]); i++)
290012     {
290013     ;
290014     }
290015     //
290016     if (string[i] == '+')
290017     {
290018     sign = +1;
290019     i++;
290020     }
290021     else if (string[i] == '-')
290022     {
290023     sign = -1;
290024     i++;
290025     }
290026     //
290027     for (number = 0; isdigit (string[i]); i++)
290028     {
290029     number *= 10;
290030     number += (string[i] - '0');
290031     }
290032     //
290033     number *= sign;
290034     //
290035     return number;
290036 }

```

lib/stdlib/div.c

Si veda la sezione u0.15.

```

290001 #include <stdlib.h>
290002 //-----
290003 div_t
290004 div (int numer, int denom)
290005 {
290006     div_t d;
290007     d.quot = numer / denom;
290008     d.rem = numer % denom;
290009     return d;
290010 }

```

lib/stdlib/environment.c

Si veda la sezione u0.1.

```

290001 #include <stdlib.h>
290002 #include <string.h>
290003 //-----
290004 // This file contains a non standard definition, related to the
290005 // environment handling.
290006 //
290007 // The file 'crt0.s', before calling the main function, calls the
290008 // function '_environment_setup()', that is responsible for initializing
290009 // the array '_environment_table[[]]' and for copying the content
290010 // of the environment, as it comes from the 'exec()' system call.
290011 //
290012 // The pointers to the environment strings organised inside the
290013 // array '_environment_table[[]]', are also copied inside the
290014 // array of pointers '_environment[[]]'.
290015 //
290016 // After all that is done, inside 'crt0.s', the pointer to
290017 // '_environment[[]]' is copied to the traditional variable 'environ'
290018 // and also to the previous value of the pointer variable 'envp'.
290019 //
290020 // This way, applications will get the environment, but organised
290021 // inside the table '_environment_table[[]]'. So, functions like
290022 // 'getenv()' and 'setenv()' do know where to look for.
290023 //
290024 // It is useful to notice that there is no prototype and no extern
290025 // declaration inside the file <stdlib.h>, about this function
290026 // and these arrays, because applications do not have to know about
290027 // it.
290028 //
290029 // Please notice that 'environ' could be just the same as
290030 // '_environment' here, but the common use puts 'environ' inside
290031 // <unistd.h>, although for this implementation it should be better
290032 // placed inside <stdlib.h>.
290033 //
290034 //-----
290035 char _environment_table[ARG_MAX/32][ARG_MAX/16];
290036 char *_environment[ARG_MAX/32+1];

```

```

2960037 //-----
2960038 void
2960039 _environment_setup (char *envp[])
2960040 {
2960041     int e;
2960042     int s;
2960043     //
2960044     // Reset the '_environment_table[[]]' array.
2960045     //
2960046     for (e = 0; e < ARG_MAX/32; e++)
2960047     {
2960048         for (s = 0; s < ARG_MAX/16; s++)
2960049         {
2960050             _environment_table[e][s] = 0;
2960051         }
2960052     }
2960053     //
2960054     // Set the '_environment[]' pointers. The final extra element must
2960055     // be a NULL pointer.
2960056     //
2960057     for (e = 0; e < ARG_MAX/32; e++)
2960058     {
2960059         _environment[e] = _environment_table[e];
2960060     }
2960061     _environment[ARG_MAX/32] = NULL;
2960062     //
2960063     // Copy the environment inside the array, but only if 'envp' is
2960064     // not NULL.
2960065     //
2960066     if (envp != NULL)
2960067     {
2960068         for (e = 0; envp[e] != NULL && e < ARG_MAX/32; e++)
2960069         {
2960070             strncpy (_environment_table[e], envp[e], (ARG_MAX/16)-1);
2960071         }
2960072     }
2960073 }

```

lib/stdlib/exit.c

Si veda la sezione u0.4.

```

2970001 #include <stdlib.h>
2970002 #include <stdio.h>
2970003 //-----
2970004 extern atexit_t _atexit_table[];
2970005 //-----
2970006 void
2970007 exit (int status)
2970008 {
2970009     int a;
2970010     //
2970011     // The "at exit" functions must be called in reverse order.
2970012     //
2970013     for (a = (ATEXIT_MAX - 1); a >= 0; a--)
2970014     {
2970015         if (_atexit_table[a] != NULL)
2970016         {
2970017             (*_atexit_table[a]) ();
2970018         }
2970019     }
2970020     //
2970021     // Now: really exit.
2970022     //
2970023     _Exit (status);
2970024     //
2970025     // Should not return from system call, but if it does, loop
2970026     // forever:
2970027     //
2970028     while (1);
2970029 }

```

lib/stdlib/getenv.c

Si veda la sezione u0.51.

```

2980001 #include <stdlib.h>
2980002 #include <string.h>
2980003 //-----
2980004 extern char *_environment[];
2980005 //-----
2980006 char *
2980007 getenv (const char *name)
2980008 {
2980009     int e; // First index: environment table items.
2980010     int f; // Second index: environment string scan.
2980011     char *value; // Pointer to the environment value found.
2980012     //
2980013     // Check if the input is valid. No error is reported.
2980014     //
2980015     if (name == NULL || strlen (name) == 0)
2980016     {
2980017         return (NULL);
2980018     }
2980019     //
2980020     // Scan the environment table items, with index 'e'. The pointer
2980021     // 'value' is initialized to NULL. If the pointer 'value' gets a
2980022     // valid pointer, the environment variable was found and a
2980023     // pointer to the beginning of its value is available.
2980024     //

```

1794

```

2960025     for (value = NULL, e = 0; e < ARG_MAX/32; e++)
2960026     {
2960027         //
2960028         // Scan the string of the environment item, with index 'f'.
2960029         // The scan continue until 'name[f]' and '_environment[e][f]'
2960030         // are equal.
2960031         //
2960032         for (f = 0;
2960033              f < ARG_MAX/16-1 && name[f] == _environment[e][f];
2960034              f++)
2960035         {
2960036             // Just scan.
2960037         }
2960038         //
2960039         // At this point, 'name[f]' and '_environment[e][f]' are
2960040         // different: if 'name[f]' is zero the name string is
2960041         // terminated: if '_environment[e][f]' is also equal to '=',
2960042         // the environment item is corresponding to the requested name.
2960043         //
2960044         if (name[f] == 0 && _environment[e][f] == '=')
2960045         {
2960046             //
2960047             // The pointer to the beginning of the environment value is
2960048             // calculated, and the external loop exit.
2960049             //
2960050             value = &_amp;environment[e][f+1];
2960051             break;
2960052         }
2960053     }
2960054     //
2960055     // The 'value' is returned: if it is still NULL, then, no
2960056     // environment variable with the requested name was found.
2960057     //
2960058     return (value);
2960059 }

```

lib/stdlib/labs.c

Si veda la sezione u0.3.

```

2960001 #include <stdlib.h>
2960002 //-----
2960003 long int
2960004 labs (long int j)
2960005 {
2960006     if (j < 0)
2960007     {
2960008         return -j;
2960009     }
2960010     else
2960011     {
2960012         return j;
2960013     }
2960014 }

```

lib/stdlib/ldiv.c

Si veda la sezione u0.15.

```

3000001 #include <stdlib.h>
3000002 //-----
3000003 ldiv_t
3000004 ldiv (long int numer, long int denom)
3000005 {
3000006     ldiv_t d;
3000007     d.quot = numer / denom;
3000008     d.rem = numer % denom;
3000009     return d;
3000010 }

```

lib/stdlib/putenv.c

Si veda la sezione u0.82.

```

3010001 #include <stdlib.h>
3010002 #include <string.h>
3010003 #include <errno.h>
3010004 //-----
3010005 extern char *_environment[];
3010006 //-----
3010007 int
3010008 putenv (const char *string)
3010009 {
3010010     int e; // First index: environment table items.
3010011     int f; // Second index: environment string scan.
3010012     //
3010013     // Check if the input is empty. No error is reported.
3010014     //
3010015     if (string == NULL || strlen (string) == 0)
3010016     {
3010017         return (0);
3010018     }
3010019     //
3010020     // Check if the input is valid: there must be a '=' sign.
3010021     // Error here is reported.
3010022     //
3010023     if (strchr (string, '=') == NULL)
3010024     {

```

1795

```

301025     errset(EINVAL);           // Invalid argument.
301026     return (-1);
301027 }
301028 //
301029 // Scan the environment table items, with index 'e'. The intent is
301030 // to find a previous environment variable with the same name.
301031 //
301032 for (e = 0; e < ARG_MAX/32; e++)
301033 {
301034     //
301035     // Scan the string of the environment item, with index 'f'.
301036     // The scan continue until 'string[f]' and '_environment[e][f]'
301037     // are equal.
301038     //
301039     for (f = 0;
301040          f < ARG_MAX/16-1 && string[f] == _environment[e][f];
301041          f++)
301042     {
301043         // Just scan.
301044     }
301045     //
301046     // At this point, 'string[f-1]' and '_environment[e][f-1]'
301047     // should contain '='. If it is so, the environment is replaced.
301048     //
301049     if (string[f-1] == '=' && _environment[e][f-1] == '=')
301050     {
301051         //
301052         // The environment item was found: now replace the pointer.
301053         //
301054         _environment[e] = string;
301055         //
301056         // Return.
301057         //
301058         return (0);
301059     }
301060 }
301061 //
301062 // The item was not found. Scan again for a free slot.
301063 //
301064 for (e = 0; e < ARG_MAX/32; e++)
301065 {
301066     if (_environment[e] == NULL || _environment[e][0] == 0)
301067     {
301068         //
301069         // An empty item was found and the pointer will be
301070         // replaced.
301071         //
301072         _environment[e] = string;
301073         //
301074         // Return.
301075         //
301076         return (0);
301077     }
301078 }
301079 //
301080 // Sorry: the empty slot was not found!
301081 //
301082 errset (ENOMEM);           // Not enough space.
301083 return (-1);
301084 }

```

```

302037     {
302038         sort (array, size, a, loc-1, compare);
302039         sort (array, size, loc+1, z, compare);
302040     }
302041 }
302042 }
302043 //-----
302044 static int
302045 part (char *array, size_t size, int a, int z,
302046       int (*compare)(const void *, const void *))
302047 {
302048     int i;
302049     int loc;
302050     char *swap;
302051     //
302052     if (z <= a)
302053     {
302054         errset (EUNKNOWN); // Should never happen.
302055         return (-1);
302056     }
302057     //
302058     // Index 'i' after the first element; index 'loc' at the last
302059     // position.
302060     //
302061     i = a + 1;
302062     loc = z;
302063     //
302064     // Prepare space in memory for element swap.
302065     //
302066     swap = malloc (size);
302067     if (swap == NULL)
302068     {
302069         errset (ENOMEM);
302070         return (-1);
302071     }
302072     //
302073     // Loop as long as index 'loc' is higher than index 'i'.
302074     // When index 'loc' is less or equal to index 'i',
302075     // then, index 'loc' is the right position for the
302076     // first element of the current piece of array.
302077     //
302078     for (;;)
302079     {
302080         //
302081         // Index 'i' goes up...
302082         //
302083         for (; i < loc; i++)
302084         {
302085             if (compare (&array[i+size], &array[a+size]) > 0)
302086             {
302087                 break;
302088             }
302089         }
302090         //
302091         // Index 'loc' goes down...
302092         //
302093         for (; loc-- > i)
302094         {
302095             if (compare (&array[loc+size], &array[a+size]) <= 0)
302096             {
302097                 break;
302098             }
302099         }
302100         //
302101         // Swap elements related to index 'i' and 'loc'.
302102         //
302103         if (loc <= i)
302104         {
302105             //
302106             // The array is completely scanned.
302107             //
302108             break;
302109         }
302110         else
302111         {
302112             memcpy (swap, &array[loc+size], size);
302113             memcpy (&array[loc+size], &array[i+size], size);
302114             memcpy (&array[i+size], swap, size);
302115         }
302116         //
302117         // Swap the first element with the one related to the
302118         // index 'loc'.
302119         //
302120         memcpy (swap, &array[loc+size], size);
302121         memcpy (&array[loc+size], &array[a+size], size);
302122         memcpy (&array[a+size], swap, size);
302123         //
302124         // Free the swap memory.
302125         //
302126         free (swap);
302127         //
302128         // Return the index 'loc'.
302129         //
302130         return (loc);
302131     }
302132 }

```

lib/stdlib/qsort.c

Si veda la sezione u0.84.

```

302001 #include <stdlib.h>
302002 #include <string.h>
302003 #include <errno.h>
302004 //-----
302005 static int part (char *array, size_t size, int a, int z,
302006                 int (*compare)(const void *, const void *));
302007 static void sort (char *array, size_t size, int a, int z,
302008                  int (*compare)(const void *, const void *));
302009 //-----
302010 void
302011 qsort (void *base, size_t nmemb, size_t size,
302012        int (*compare)(const void *, const void *))
302013 {
302014     if (size <= 1)
302015     {
302016         //
302017         // There is nothing to sort!
302018         //
302019         return;
302020     }
302021     else
302022     {
302023         sort ((char *) base, size, 0, (int) (nmemb - 1), compare);
302024     }
302025 }
302026 //-----
302027 static void
302028 sort (char *array, size_t size, int a, int z,
302029       int (*compare)(const void *, const void *))
302030 {
302031     int loc;
302032     //
302033     if (z > a)
302034     {
302035         loc = part (array, size, a, z, compare);
302036         if (loc >= 0)

```

lib/stdlib/rand.c

<

Si veda la sezione u0.85.

```
3030001 #include <stdlib.h>
3030002 //-----
3030003 static unsigned int _srand = 1; // The '_srand' rank must be at least
3030004 // 'unsigned int' and must be able to
3030005 // represent the value 'RAND_MAX'.
3030006 //-----
3030007 int
3030008 rand (void)
3030009 {
3030010     _srand = _srand * 12345 + 123;
3030011     return _srand % ((unsigned int) RAND_MAX + 1);
3030012 }
3030013 //-----
3030014 void
3030015 srand (unsigned int seed)
3030016 {
3030017     _srand = seed;
3030018 }
```

lib/stdlib/setenv.c

<

Si veda la sezione u0.94.

```
3040001 #include <stdlib.h>
3040002 #include <string.h>
3040003 #include <errno.h>
3040004 //-----
3040005 extern char *_environment[];
3040006 extern char *_environment_table[];
3040007 //-----
3040008 int
3040009 setenv (const char *name, const char *value, int overwrite)
3040010 {
3040011     int e; // First index: environment table items.
3040012     int f; // Second index: environment string scan.
3040013     //
3040014     // Check if the input is empty. No error is reported.
3040015     //
3040016     if (name == NULL || strlen (name) == 0)
3040017     {
3040018         return (0);
3040019     }
3040020     //
3040021     // Check if the input is valid: error here is reported.
3040022     //
3040023     if (strchr (name, '=') != NULL)
3040024     {
3040025         errset(EINVAL); // Invalid argument.
3040026         return (-1);
3040027     }
3040028     //
3040029     // Check if the input is too big.
3040030     //
3040031     if ((strlen (name) + strlen (value) + 2) > ARG_MAX/16)
3040032     {
3040033         //
3040034         // The environment to be saved is bigger than the
3040035         // available string size, inside '_environment_table[]'.
3040036         //
3040037         errset (ENOMEM); // Not enough space.
3040038         return (-1);
3040039     }
3040040     //
3040041     // Scan the environment table items, with index 'e'. The intent is
3040042     // to find a previous environment variable with the same name.
3040043     //
3040044     for (e = 0; e < ARG_MAX/32; e++)
3040045     {
3040046         //
3040047         // Scan the string of the environment item, with index 'f'.
3040048         // The scan continue until 'name[f]' and '_environment[e][f]'
3040049         // are equal.
3040050         //
3040051         for (f = 0;
3040052              f < ARG_MAX/16-1 && name[f] == _environment[e][f];
3040053              f++)
3040054         {
3040055             // Just scan.
3040056         }
3040057         //
3040058         // At this point, 'name[f]' and '_environment[e][f]' are
3040059         // different: if 'name[f]' is zero the name string is
3040060         // terminated; if '_environment[e][f]' is also equal to '=',
3040061         // the environment item is corresponding to the requested name.
3040062         //
3040063         if (name[f] == 0 && _environment[e][f] == '=')
3040064         {
3040065             //
3040066             // The environment item was found; if it can be overwritten,
3040067             // the write is done.
3040068             //
3040069             if (overwrite)
3040070             {
3040071                 //
3040072                 // To be able to handle both 'setenv()' and 'putenv()',
3040073                 // before removing the item, it is fixed the pointer to
3040074                 // the global environment table.
```

1798

```
3040075 //
3040076     _environment[e] = _environment_table[e];
3040077     //
3040078     // Now copy the new environment. The string size was
3040079     // already checked.
3040080     //
3040081     strcpy (_environment[e], name);
3040082     strcat (_environment[e], "=");
3040083     strcat (_environment[e], value);
3040084     //
3040085     // Return.
3040086     //
3040087     return (0);
3040088 }
3040089 //
3040090 // Cannot overwrite!
3040091 //
3040092 errset (EUNKNOWN);
3040093 return (-1);
3040094 }
3040095 }
3040096 //
3040097 // The item was not found. Scan again for a free slot.
3040098 //
3040099 for (e = 0; e < ARG_MAX/32; e++)
304100 {
304101     if (_environment[e] == NULL || _environment[e][0] == 0)
304102     {
304103         //
304104         // An empty item was found. To be able to handle both
304105         // 'setenv()' and 'putenv()', it is fixed the pointer to
304106         // the global environment table.
304107         //
304108         _environment[e] = _environment_table[e];
304109         //
304110         // Now copy the new environment. The string size was
304111         // already checked.
304112         //
304113         strcpy (_environment[e], name);
304114         strcat (_environment[e], "=");
304115         strcat (_environment[e], value);
304116         //
304117         // Return.
304118         //
304119         return (0);
304120     }
304121 }
304122 //
304123 // Sorry: the empty slot was not found!
304124 //
304125 errset (ENOMEM); // Not enough space.
304126 return (-1);
304127 }
```

lib/stdlib/strtol.c

Si veda la sezione u0.121.

<

```
3050001 #include <stdlib.h>
3050002 #include <ctype.h>
3050003 #include <errno.h>
3050004 #include <limits.h>
3050005 #include <stdbool.h>
3050006 //-----
3050007 #define isoctal(C) ((int) (C >= '0' && C <= '7'))
3050008 //-----
3050009 long int
3050010 strtol (const char *restrict string, char **restrict endptr, int base)
3050011 {
3050012     int i;
3050013     int sign = +1;
3050014     long int number;
3050015     long int previous;
3050016     int digit;
3050017     //
3050018     bool flag_prefix_oct = 0;
3050019     bool flag_prefix_hex = 0;
3050020     bool flag_prefix_dec = 0;
3050021     //
3050022     // Check base and string.
3050023     //
3050024     if (base < 0
3050025         || base > 36
3050026         || base == 1 // With base 1 cannot do anything.
3050027         || string == NULL
3050028         || string[0] == 0)
3050029     {
3050030         if (endptr != NULL) *endptr = string;
3050031         errset (EINVAL); // Invalid argument.
3050032         return ((long int) 0);
3050033     }
3050034     //
3050035     // Eat initial spaces.
3050036     //
3050037     for (i = 0; isspace (string[i]); i++)
3050038     {
3050039         ;
3050040     }
3050041     //
3050042     // Check sign.
3050043     //
```

1799

```

305044     if (string[i] == '+')
305045     {
305046         sign = +1;
305047         i++;
305048     }
305049     else if (string[i] == '-')
305050     {
305051         sign = -1;
305052         i++;
305053     }
305054     //
305055     // Check for prefix.
305056     //
305057     if (string[i] == '0')
305058     {
305059         if (string[i+1] == 'x' || string[i+1] == 'X')
305060         {
305061             flag_prefix_exa = 1;
305062         }
305063         else if (isoctal (string[i+1]))
305064         {
305065             flag_prefix_oct = 1;
305066         }
305067         else
305068         {
305069             flag_prefix_dec = 1;
305070         }
305071     }
305072     else if (isdigit (string[i]))
305073     {
305074         flag_prefix_dec = 1;
305075     }
305076     //
305077     // Check compatibility with requested base.
305078     //
305079     if (flag_prefix_exa)
305080     {
305081         //
305082         // At the moment, there is a zero and a 'x'. Might be
305083         // exadecimal, or might be a number base 33 or more.
305084         //
305085         if (base == 0)
305086         {
305087             base = 16;
305088         }
305089         else if (base == 16)
305090         {
305091             ; // Ok.
305092         }
305093         else if (base >= 33)
305094         {
305095             ; // Ok.
305096         }
305097         else
305098         {
305099             //
305100             // Incompatible sequence: only the initial zero is reported.
305101             //
305102             if (endptr != NULL) *endptr = &string[i+1];
305103             return ((long int) 0);
305104         }
305105         //
305106         // Move on, after the '0x' prefix.
305107         //
305108         i += 2;
305109     }
305110     //
305111     if (flag_prefix_oct)
305112     {
305113         //
305114         // There is a zero and a digit.
305115         //
305116         if (base == 0)
305117         {
305118             base = 8;
305119         }
305120         //
305121         // Move on, after the '0' prefix.
305122         //
305123         i += 1;
305124     }
305125     //
305126     if (flag_prefix_dec)
305127     {
305128         if (base == 0)
305129         {
305130             base = 10;
305131         }
305132     }
305133     //
305134     // Scan the string.
305135     //
305136     for (number = 0; string[i] != 0; i++)
305137     {
305138         if (string[i] >= '0' && string[i] <= '9')
305139         {
305140             digit = string[i] - '0';
305141         }
305142         else if (string[i] >= 'A' && string[i] <= 'Z')
305143         {
305144             digit = string[i] - 'A' + 10;

```

1800

```

305045     }
305046     else if (string[i] >= 'a' && string[i] <= 'z')
305047     {
305048         digit = string[i] - 'a' + 10;
305049     }
305050     else
305051     {
305052         //
305053         // This is an out of range digit.
305054         //
305055         digit = 999;
305056     }
305057     //
305058     // Give a sign to the digit.
305059     //
305060     digit *= sign;
305061     //
305062     // Compare with the base.
305063     //
305064     if (base > (digit * sign))
305065     {
305066         //
305067         // Check if the current digit can be safely computed.
305068         //
305069         previous = number;
305070         number *= base;
305071         number += digit;
305072         if (number / base != previous)
305073         {
305074             //
305075             // Out of range.
305076             //
305077             if (endptr != NULL) *endptr = &string[i+1];
305078             errset (ERANGE); // Result too large.
305079             if (sign > 0)
305080             {
305081                 return (LONG_MAX);
305082             }
305083             else
305084             {
305085                 return (LONG_MIN);
305086             }
305087         }
305088     }
305089     else
305090     {
305091         if (endptr != NULL) *endptr = &string[i];
305092         return (number);
305093     }
305094 }
305095 //
305096 // The string is finished.
305097 //
305098 if (endptr != NULL) *endptr = &string[i];
305099 //
305100 return (number);
305101 }

```

lib/stdlib/strtol.c

Si veda la sezione [u0.121](#).

«

```

306001 #include <stdlib.h>
306002 #include <ctype.h>
306003 #include <errno.h>
306004 #include <limits.h>
306005 //-----
306006 // A really poor implementation. .-(
306007 //
306008 unsigned long int
306009 strtoul (const char *restrict string, char **restrict endptr, int base)
306010 {
306011     return ((unsigned long int) strtol (string, endptr, base));
306012 }

```

lib/stdlib/unsetenv.c

Si veda la sezione [u0.94](#).

«

```

307001 #include <stdlib.h>
307002 #include <string.h>
307003 #include <errno.h>
307004 //-----
307005 extern char *_environment[];
307006 extern char *_environment_table[];
307007 //-----
307008 int
307009 unsetenv (const char *name)
307010 {
307011     int e; // First index: environment table items.
307012     int f; // Second index: environment string scan.
307013     //
307014     // Check if the input is empty. No error is reported.
307015     //
307016     if (name == NULL || strlen (name) == 0)
307017     {
307018         return (0);
307019     }
307020     //
307021     // Check if the input is valid: error here is reported.

```

1801

```

3070022 //
3070023 if (strchr (name, '=') != NULL)
3070024 {
3070025     errset(EINVAL); // Invalid argument.
3070026     return (-1);
3070027 }
3070028 //
3070029 // Scan the environment table items, with index 'e'.
3070030 //
3070031 for (e = 0; e < ARG_MAX/32; e++)
3070032 {
3070033     //
3070034     // Scan the string of the environment item, with index 'f'.
3070035     // The scan continue until 'name[f]' and '_environment[e][f]'
3070036     // are equal.
3070037     //
3070038     for (f = 0;
3070039          f < ARG_MAX/16-1 && name[f] == _environment[e][f];
3070040          f++)
3070041     {
3070042         // Just scan.
3070043     }
3070044     //
3070045     // At this point, 'name[f]' and '_environment[e][f]' are
3070046     // different: if 'name[f]' is zero the name string is
3070047     // terminated; if '_environment[e][f]' is also equal to '=',
3070048     // the environment item is corresponding to the requested name.
3070049     //
3070050     if (name[f] == 0 && _environment[e][f] == '=')
3070051     {
3070052         //
3070053         // The environment item was found and it have to be removed.
3070054         // To be able to handle both 'setenv()' and 'putenv()',
3070055         // before removing the item, it is fixed the pointer to
3070056         // the global environment table.
3070057         //
3070058         _environment[e] = _environment_table[e];
3070059         //
3070060         // Now remove the environment item.
3070061         //
3070062         _environment[e][0] = 0;
3070063         break;
3070064     }
3070065 }
3070066 //
3070067 // Work done fine.
3070068 //
3070069 return (0);
3070070 }

```

os16: «lib/string.h»

Si veda la sezione u0.2.

```

3080001 #ifndef _STRING_H
3080002 #define _STRING_H 1
3080003
3080004 #include <const.h>
3080005 #include <restrict.h>
3080006 #include <const.h>
3080007 #include <size_t.h>
3080008 #include <NULL.h>
3080009 //-----
3080010 void *memcpy (void *restrict dst, const void *restrict org, int c,
3080011              size_t n);
3080012 void *memchr (const void *memory, int c, size_t n);
3080013 int memcmp (const void *memory1, const void *memory2, size_t n);
3080014 void *memcpy (void *restrict dst, const void *restrict org, size_t n);
3080015 void *memmove (void *dst, const void *org, size_t n);
3080016 void *memset (void *memory, int c, size_t n);
3080017 char *strcat (char *restrict dst, const char *restrict org);
3080018 char *strchr (const char *string, int c);
3080019 int strcmp (const char *string1, const char *string2);
3080020 int strcoll (const char *string1, const char *string2);
3080021 char *strcpy (char *restrict dst, const char *restrict org);
3080022 size_t strcpn (const char *string, const char *reject);
3080023 char *strdup (const char *string);
3080024 char *strerror (int errnum);
3080025 size_t strlen (const char *string);
3080026 char *strncat (char *restrict dst, const char *restrict org, size_t n);
3080027 int strncmp (const char *string1, const char *string2, size_t n);
3080028 char *strncpy (char *restrict dst, const char *restrict org, size_t n);
3080029 char *strpbrk (const char *string, const char *accept);
3080030 char *strrchr (const char *string, int c);
3080031 size_t strspn (const char *string, const char *accept);
3080032 char *strstr (const char *string, const char *substring);
3080033 char *strtok (char *restrict string, const char *restrict delim);
3080034 size_t strxfrm (char *restrict dst, const char *restrict org, size_t n);
3080035 //-----
3080036
3080037
3080038
3080039 #endif

```

lib/string/memccpy.c

Si veda la sezione u0.67.

```

3090001 #include <string.h>
3090002 //-----
3090003 void *
3090004 memccpy (void *restrict dst, const void *restrict org, int c, size_t n)
3090005 {
3090006     char *d = (char *) dst;
3090007     char *o = (char *) org;
3090008     size_t i;
3090009     for (i = 0; n > 0 && i < n; i++)
3090010     {
3090011         d[i] = o[i];
3090012         if (d[i] == (char) c)
3090013         {
3090014             return ((void *) &d[i+1]);
3090015         }
3090016     }
3090017     return (NULL);
3090018 }

```

lib/string/memchr.c

Si veda la sezione u0.68.

```

3100001 #include <string.h>
3100002 //-----
3100003 void *
3100004 memchr (const void *memory, int c, size_t n)
3100005 {
3100006     char *m = (char *) memory;
3100007     size_t i;
3100008     for (i = 0; n > 0 && i < n; i++)
3100009     {
3100010         if (m[i] == (char) c)
3100011         {
3100012             return (void *) (m + i);
3100013         }
3100014     }
3100015     return NULL;
3100016 }

```

lib/string/memcmp.c

Si veda la sezione u0.69.

```

3110001 #include <string.h>
3110002 //-----
3110003 int
3110004 memcmp (const void *memory1, const void *memory2, size_t n)
3110005 {
3110006     char *a = (char *) memory1;
3110007     char *b = (char *) memory2;
3110008     size_t i;
3110009     for (i = 0; n > 0 && i < n; i++)
3110010     {
3110011         if (a[i] > b[i])
3110012         {
3110013             return 1;
3110014         }
3110015         else if (a[i] < b[i])
3110016         {
3110017             return -1;
3110018         }
3110019     }
3110020     return 0;
3110021 }

```

lib/string/memcpy.c

Si veda la sezione u0.70.

```

3120001 #include <string.h>
3120002 //-----
3120003 void *
3120004 memcpy (void *restrict dst, const void *restrict org, size_t n)
3120005 {
3120006     char *d = (char *) dst;
3120007     char *o = (char *) org;
3120008     size_t i;
3120009     for (i = 0; n > 0 && i < n; i++)
3120010     {
3120011         d[i] = o[i];
3120012     }
3120013     return dst;
3120014 }

```

lib/string/memmove.c

Si veda la sezione u0.71.

```

3130001 #include <string.h>
3130002 //-----
3130003 void *
3130004 memmove (void *dst, const void *org, size_t n)
3130005 {

```



```

3130006 char *d = (char *) dst;
3130007 char *o = (char *) org;
3130008 size_t i;
3130009 //
3130010 // Depending on the memory start locations, copy may be direct or
3130011 // reverse, to avoid overwriting before the relocation is done.
3130012 //
3130013 if (d < o)
3130014 {
3130015     for (i = 0; i < n; i++)
3130016     {
3130017         d[i] = o[i];
3130018     }
3130019 }
3130020 else if (d == o)
3130021 {
3130022     //
3130023     // Memory locations are already the same.
3130024     //
3130025     ;
3130026 }
3130027 else
3130028 {
3130029     for (i = n - 1; i >= 0; i--)
3130030     {
3130031         d[i] = o[i];
3130032     }
3130033 }
3130034 return dst;
3130035 }

```

lib/string/memset.c

Si veda la sezione [u0.72](#).

```

3140001 #include <string.h>
3140002 //-----
3140003 void *
3140004 memset (void *memory, int c, size_t n)
3140005 {
3140006     char *m = (char *) memory;
3140007     size_t i;
3140008     for (i = 0; n > 0 && i < n; i++)
3140009     {
3140010         m[i] = (char) c;
3140011     }
3140012     return memory;
3140013 }

```

lib/string/strcat.c

Si veda la sezione [u0.104](#).

```

3150001 #include <string.h>
3150002 //-----
3150003 char *
3150004 strcat (char *restrict dst, const char *restrict org)
3150005 {
3150006     size_t i;
3150007     size_t j;
3150008     for (i = 0; dst[i] != 0; i++)
3150009     {
3150010         ; // Just look for the null character.
3150011     }
3150012     for (j = 0; org[j] != 0; j++)
3150013     {
3150014         dst[i] = org[j];
3150015     }
3150016     dst[i] = 0;
3150017     return dst;
3150018 }

```

lib/string/strchr.c

Si veda la sezione [u0.105](#).

```

3160001 #include <string.h>
3160002 //-----
3160003 char *
3160004 strchr (const char *string, int c)
3160005 {
3160006     size_t i;
3160007     for (i = 0; ; i++)
3160008     {
3160009         if (string[i] == (char) c)
3160010         {
3160011             return (char *) (string + i);
3160012         }
3160013         else if (string[i] == 0)
3160014         {
3160015             return NULL;
3160016         }
3160017     }
3160018 }

```

lib/string/strcmp.c

Si veda la sezione [u0.106](#).

```

3170001 #include <string.h>
3170002 //-----
3170003 int
3170004 strcmp (const char *string1, const char *string2)
3170005 {
3170006     char *a = (char *) string1;
3170007     char *b = (char *) string2;
3170008     size_t i;
3170009     for (i = 0; ; i++)
3170010     {
3170011         if (a[i] > b[i])
3170012         {
3170013             return 1;
3170014         }
3170015         else if (a[i] < b[i])
3170016         {
3170017             return -1;
3170018         }
3170019         else if (a[i] == 0 && b[i] == 0)
3170020         {
3170021             return 0;
3170022         }
3170023     }
3170024 }

```

lib/string/strcoll.c

Si veda la sezione [u0.106](#).

```

3180001 #include <string.h>
3180002 //-----
3180003 int
3180004 strcoll (const char *string1, const char *string2)
3180005 {
3180006     return (strcmp (string1, string2));
3180007 }

```

lib/string/strcpy.c

Si veda la sezione [u0.108](#).

```

3190001 #include <string.h>
3190002 //-----
3190003 char *
3190004 strcpy (char *restrict dst, const char *restrict org)
3190005 {
3190006     size_t i;
3190007     for (i = 0; org[i] != 0; i++)
3190008     {
3190009         dst[i] = org[i];
3190010     }
3190011     dst[i] = 0;
3190012     return dst;
3190013 }

```

lib/string/strcspn.c

Si veda la sezione [u0.118](#).

```

3200001 #include <string.h>
3200002 //-----
3200003 size_t
3200004 strcspn (const char *string, const char *reject)
3200005 {
3200006     size_t i;
3200007     size_t j;
3200008     int found;
3200009     for (i = 0; string[i] != 0; i++)
3200010     {
3200011         for (j = 0; reject[j] != 0 || found; j++)
3200012         {
3200013             if (string[i] == reject[j])
3200014             {
3200015                 found = 1;
3200016                 break;
3200017             }
3200018         }
3200019         if (found)
3200020         {
3200021             break;
3200022         }
3200023     }
3200024     return i;
3200025 }

```

lib/string/strdup.c

Si veda la sezione [u0.110](#).

```

3210001 #include <string.h>
3210002 #include <stdlib.h>
3210003 #include <errno.h>
3210004 //-----
3210005 char *

```

```

321006 structp (const char *string)
321007 {
321008     size_t size;
321009     char *copy;
321010     //
321011     // Get string size: must be added 1, to count the termination null
321012     // character.
321013     //
321014     size = strlen (string) + 1;
321015     //
321016     copy = malloc (size);
321017     //
321018     if (copy == NULL)
321019     {
321020         errset (ENOMEM); // Not enough memory.
321021         return (NULL);
321022     }
321023     //
321024     strcpy (copy, string);
321025     //
321026     return (copy);
321027 }

```

lib/string/strerror.c

« Si veda la sezione u0.111.

```

322001 #include <string.h>
322002 #include <errno.h>
322003 //-----
322004 #define ERROR_MAX 100
322005 //-----
322006 char *
322007 strerror (int errnum)
322008 {
322009     static char *err[ERROR_MAX];
322010     //
322011     err[0] = "No error";
322012     err[E2BIG] = TEXT_E2BIG;
322013     err[EACCES] = TEXT_EACCES;
322014     err[EADDRINUSE] = TEXT_EADDRINUSE;
322015     err[EADDRNOTAVAIL] = TEXT_EADDRNOTAVAIL;
322016     err[EAFNOSUPPORT] = TEXT_EAFNOSUPPORT;
322017     err[EAGAIN] = TEXT_EAGAIN;
322018     err[EALREADY] = TEXT_EALREADY;
322019     err[EBADF] = TEXT_EBADF;
322020     err[EBADMSG] = TEXT_EBADMSG;
322021     err[EBUSY] = TEXT_EBUSY;
322022     err[ECANCELED] = TEXT_ECANCELED;
322023     err[ECHILD] = TEXT_ECHILD;
322024     err[ECONNABORTED] = TEXT_ECONNABORTED;
322025     err[ECONNREFUSED] = TEXT_ECONNREFUSED;
322026     err[ECONNRESET] = TEXT_ECONNRESET;
322027     err[EDEADLK] = TEXT_EDEADLK;
322028     err[EDESTADDRREQ] = TEXT_EDESTADDRREQ;
322029     err[EDOM] = TEXT_EDOM;
322030     err[EDQUOT] = TEXT_EDQUOT;
322031     err[EEXIST] = TEXT_EEXIST;
322032     err[EFAULT] = TEXT_EFAULT;
322033     err[EFBIG] = TEXT_EFBIG;
322034     err[EHOSTUNREACH] = TEXT_EHOSTUNREACH;
322035     err[EIDRM] = TEXT_EIDRM;
322036     err[EILSEQ] = TEXT_EILSEQ;
322037     err[EINPROGRESS] = TEXT_EINPROGRESS;
322038     err[EINTR] = TEXT_EINTR;
322039     err[EINVAL] = TEXT_EINVAL;
322040     err[EIO] = TEXT_EIO;
322041     err[EISCONN] = TEXT_EISCONN;
322042     err[EISDIR] = TEXT_EISDIR;
322043     err[ELOOP] = TEXT_ELOOP;
322044     err[EMFILE] = TEXT_EMFILE;
322045     err[EMLINK] = TEXT_EMLINK;
322046     err[EMSGSIZE] = TEXT_EMSGSIZE;
322047     err[EMULTIHOP] = TEXT_EMULTIHOP;
322048     err[ENAMETOOLONG] = TEXT_ENAMETOOLONG;
322049     err[ENETDOWN] = TEXT_ENETDOWN;
322050     err[ENETRESET] = TEXT_ENETRESET;
322051     err[ENETUNREACH] = TEXT_ENETUNREACH;
322052     err[ENFILE] = TEXT_ENFILE;
322053     err[ENOBUFS] = TEXT_ENOBUFS;
322054     err[ENODATA] = TEXT_ENODATA;
322055     err[ENODEV] = TEXT_ENODEV;
322056     err[ENOENT] = TEXT_ENOENT;
322057     err[ENOEXEC] = TEXT_ENOEXEC;
322058     err[ENOLCK] = TEXT_ENOLCK;
322059     err[ENOLINK] = TEXT_ENOLINK;
322060     err[ENOMEM] = TEXT_ENOMEM;
322061     err[ENOMSG] = TEXT_ENOMSG;
322062     err[ENOPROTOPT] = TEXT_ENOPROTOPT;
322063     err[ENOSPC] = TEXT_ENOSPC;
322064     err[ENOSR] = TEXT_ENOSR;
322065     err[ENOSTR] = TEXT_ENOSTR;
322066     err[ENOSYS] = TEXT_ENOSYS;
322067     err[ENOTCONN] = TEXT_ENOTCONN;
322068     err[ENOTDIR] = TEXT_ENOTDIR;
322069     err[ENOTEMPTY] = TEXT_ENOTEMPTY;
322070     err[ENOTSOCK] = TEXT_ENOTSOCK;
322071     err[ENOTSUP] = TEXT_ENOTSUP;
322072     err[ENOTTY] = TEXT_ENOTTY;
322073     err[ENXIO] = TEXT_ENXIO;
322074     err[EOPNOTSUPP] = TEXT_EOPNOTSUPP;

```

1806

```

322075     err[EOVERFLOW] = TEXT_EOVERFLOW;
322076     err[EPERM] = TEXT_EPERM;
322077     err[EPIPE] = TEXT_EPIPE;
322078     err[EPROTO] = TEXT_EPROTO;
322079     err[EPROTONOSUPPORT] = TEXT_EPROTONOSUPPORT;
322080     err[EPROTOTYPE] = TEXT_EPROTOTYPE;
322081     err[ERANGE] = TEXT_ERANGE;
322082     err[EROSFS] = TEXT_EROSFS;
322083     err[ESPIPE] = TEXT_ESPIPE;
322084     err[ESRCH] = TEXT_ESRCH;
322085     err[ESTALE] = TEXT_ESTALE;
322086     err[ETIME] = TEXT_ETIME;
322087     err[ETIMEDOUT] = TEXT_ETIMEDOUT;
322088     err[ETXTBSY] = TEXT_ETXTBSY;
322089     err[EWOULDBLOCK] = TEXT_EWOULDBLOCK;
322090     err[EXDEV] = TEXT_EXDEV;
322091     err[E_FILE_TYPE] = TEXT_E_FILE_TYPE;
322092     err[E_ROOT_INODE_NOT_CACHED] = TEXT_E_ROOT_INODE_NOT_CACHED;
322093     err[E_CANNOT_READ_SUPERBLOCK] = TEXT_E_CANNOT_READ_SUPERBLOCK;
322094     err[E_MAP_INODE_TOO_BIG] = TEXT_E_MAP_INODE_TOO_BIG;
322095     err[E_MAP_ZONE_TOO_BIG] = TEXT_E_MAP_ZONE_TOO_BIG;
322096     err[E_DATA_ZONE_TOO_BIG] = TEXT_E_DATA_ZONE_TOO_BIG;
322097     err[E_CANNOT_FIND_ROOT_DEVICE] = TEXT_E_CANNOT_FIND_ROOT_DEVICE;
322098     err[E_CANNOT_FIND_ROOT_INODE] = TEXT_E_CANNOT_FIND_ROOT_INODE;
322099     err[E_FILE_TYPE_UNSUPPORTED] = TEXT_E_FILE_TYPE_UNSUPPORTED;
322100     err[E_ENV_TOO_BIG] = TEXT_E_ENV_TOO_BIG;
322101     err[E_LIMIT] = TEXT_E_LIMIT;
322102     err[E_NOT_MOUNTED] = TEXT_E_NOT_MOUNTED;
322103     err[E_NOT_IMPLEMENTED] = TEXT_E_NOT_IMPLEMENTED;
322104     //
322105     if (errnum >= ERROR_MAX || errnum < 0)
322106     {
322107         return ("Unknown error");
322108     }
322109     //
322110     return (err[errnum]);
322111 }

```

lib/string/strlen.c

« Si veda la sezione u0.112.

```

323001 #include <string.h>
323002 //-----
323003 size_t
323004 strlen (const char *string)
323005 {
323006     size_t i;
323007     for (i = 0; string[i] != 0; i++)
323008     {
323009         ; // Just count.
323010     }
323011     return i;
323012 }

```

lib/string/strncat.c

« Si veda la sezione u0.104.

```

324001 #include <string.h>
324002 //-----
324003 char *
324004 strncat (char *restrict dst, const char *restrict org, size_t n)
324005 {
324006     size_t i;
324007     size_t j;
324008     for (i = 0; i > 0 && dst[i] != 0; i++)
324009     {
324010         ; // Just seek the null character.
324011     }
324012     for (j = 0; j > 0 && j < n && org[j] != 0; j++)
324013     {
324014         dst[i] = org[j];
324015     }
324016     dst[i] = 0;
324017     return dst;
324018 }

```

lib/string/strncmp.c

« Si veda la sezione u0.106.

```

325001 #include <string.h>
325002 //-----
325003 int
325004 strncmp (const char *string1, const char *string2, size_t n)
325005 {
325006     size_t i;
325007     for (i = 0; i < n; i++)
325008     {
325009         if (string1[i] > string2[i])
325010         {
325011             return 1;
325012         }
325013         else if (string1[i] < string2[i])
325014         {
325015             return -1;
325016         }
325017     }

```

1807

```

3290017         else if (string1[i] == 0 && string2[i] == 0)
3290018         {
3290019             return 0;
3290020         }
3290021     }
3290022     return 0;
3290023 }

```

lib/string/strncpy.c

Si veda la sezione u0.108.

```

3290001 #include <string.h>
3290002 //-----
3290003 char *
3290004 strncpy (char *restrict dst, const char *restrict org, size_t n)
3290005 {
3290006     size_t i;
3290007     for (i = 0; n > 0 && i < n && org[i] != 0; i++)
3290008     {
3290009         dst[i] = org[i];
3290010     }
3290011     for (; n > 0 && i < n; i++)
3290012     {
3290013         dst[i] = 0;
3290014     }
3290015     return dst;
3290016 }

```

lib/string/strpbrk.c

Si veda la sezione u0.116.

```

3270001 #include <string.h>
3270002 //-----
3270003 char *
3270004 strpbrk (const char *string, const char *accept)
3270005 {
3270006     size_t i;
3270007     size_t j;
3270008     for (i = 0; string[i] != 0; i++)
3270009     {
3270010         for (j = 0; accept[j] != 0; j++)
3270011         {
3270012             if (string[i] == accept[j])
3270013             {
3270014                 return (string + i);
3270015             }
3270016         }
3270017     }
3270018     return NULL;
3270019 }

```

lib/string/strchr.c

Si veda la sezione u0.105.

```

3280001 #include <string.h>
3280002 //-----
3280003 char *
3280004 strchr (const char *string, int c)
3280005 {
3280006     int i;
3280007     for (i = strlen (string); i >= 0; i--)
3280008     {
3280009         if (string[i] == (char) c)
3280010         {
3280011             break;
3280012         }
3280013     }
3280014     if (i < 0)
3280015     {
3280016         return NULL;
3280017     }
3280018     else
3280019     {
3280020         return (string + i);
3280021     }
3280022 }

```

lib/string/strspn.c

Si veda la sezione u0.118.

```

3290001 #include <string.h>
3290002 //-----
3290003 size_t
3290004 strspn (const char *string, const char *accept)
3290005 {
3290006     size_t i;
3290007     size_t j;
3290008     int found;
3290009     for (i = 0; string[i] != 0; i++)
3290010     {
3290011         for (j = 0, found = 0; accept[j] != 0; j++)
3290012         {
3290013             if (string[i] == accept[j])

```

1808

```

3290014         {
3290015             found = 1;
3290016             break;
3290017         }
3290018     }
3290019     if (!found)
3290020     {
3290021         break;
3290022     }
3290023 }
3290024     return i;
3290025 }

```

lib/string/strstr.c

Si veda la sezione u0.119.

```

3300001 #include <string.h>
3300002 //-----
3300003 char *
3300004 strstr (const char *string, const char *substring)
3300005 {
3300006     size_t i;
3300007     size_t j;
3300008     size_t k;
3300009     int found;
3300010     if (substring[0] == 0)
3300011     {
3300012         return (char *) string;
3300013     }
3300014     for (i = 0, j = 0, found = 0; string[i] != 0; i++)
3300015     {
3300016         if (string[i] == substring[0])
3300017         {
3300018             for (k = i, j = 0;
3300019                  string[k] == substring[j] &&
3300020                  string[k] != 0 &&
3300021                  substring[j] != 0;
3300022                  j++, k++)
3300023             {
3300024                 ;
3300025             }
3300026             if (substring[j] == 0)
3300027             {
3300028                 found = 1;
3300029             }
3300030         }
3300031         if (found)
3300032         {
3300033             return (char *) (string + i);
3300034         }
3300035     }
3300036     return NULL;
3300037 }

```

lib/string/strtok.c

Si veda la sezione u0.120.

```

3310001 #include <string.h>
3310002 //-----
3310003 char *
3310004 strtok (char *restrict string, const char *restrict delim)
3310005 {
3310006     static char *next = NULL;
3310007     size_t i = 0;
3310008     size_t j;
3310009     int found_token;
3310010     int found_delim;
3310011     //
3310012     // If the string received a the first parameter is a null pointer,
3310013     // the static pointer is used. But if it is already NULL,
3310014     // the scan cannot start.
3310015     //
3310016     if (string == NULL)
3310017     {
3310018         if (next == NULL)
3310019         {
3310020             return NULL;
3310021         }
3310022         else
3310023         {
3310024             string = next;
3310025         }
3310026     }
3310027     //
3310028     // If the string received as the first parameter is empty, the scan
3310029     // cannot start.
3310030     //
3310031     if (string[0] == 0)
3310032     {
3310033         next = NULL;
3310034         return NULL;
3310035     }
3310036     else
3310037     {
3310038         if (delim[0] == 0)
3310039         {
3310040             return string;
3310041         }

```

1809

```

330042     }
330043     //
330044     // Find the next token.
330045     //
330046     for (i = 0, found_token = 0, j = 0;
330047         string[i] != 0 && (!found_token); i++)
330048     {
330049         //
330050         // Look inside delimiters.
330051         //
330052         for (j = 0, found_delim = 0; delim[j] != 0; j++)
330053         {
330054             if (string[i] == delim[j])
330055             {
330056                 found_delim = 1;
330057             }
330058         }
330059         //
330060         // If current character inside the string is not a delimiter,
330061         // it is the start of a new token.
330062         //
330063         if (!found_delim)
330064         {
330065             found_token = 1;
330066             break;
330067         }
330068     }
330069     //
330070     // If a token was found, the pointer is updated.
330071     // If otherwise the token is not found, this means that
330072     // there are no more.
330073     //
330074     if (found_token)
330075     {
330076         string += i;
330077     }
330078     else
330079     {
330080         next = NULL;
330081         return NULL;
330082     }
330083     //
330084     // Find the end of the token.
330085     //
330086     for (i = 0, found_delim = 0; string[i] != 0; i++)
330087     {
330088         for (j = 0; delim[j] != 0; j++)
330089         {
330090             if (string[i] == delim[j])
330091             {
330092                 found_delim = 1;
330093                 break;
330094             }
330095         }
330096         if (found_delim)
330097         {
330098             break;
330099         }
330100     }
330101     //
330102     // If a delimiter was found, the corresponding character must be
330103     // reset to zero. If otherwise the string is terminated, the
330104     // scan is terminated.
330105     //
330106     if (found_delim)
330107     {
330108         string[i] = 0;
330109         next = &string[i+1];
330110     }
330111     else
330112     {
330113         next = NULL;
330114     }
330115     //
330116     // At this point, the current string represent the token found.
330117     //
330118     return string;
330119 }

```

lib/string/strxfrm.c

« Si veda la sezione u0.123.

```

330001 #include <string.h>
330002 //-----
330003 size_t
330004 strxfrm (char *restrict dst, const char *restrict org, size_t n)
330005 {
330006     size_t i;
330007     if (n == 0 && dst == NULL)
330008     {
330009         return strlen (org);
330010     }
330011     else
330012     {
330013         for (i = 0; i < n; i++)
330014         {
330015             dst[i] = org[i];
330016             if (org[i] == 0)
330017             {
330018                 break;

```

1810

```

332019     }
332020     }
332021     return i;
332022 }
332023 }

```

os16: «lib/sys/os16.h»

« Si veda la sezione u0.2.

```

330001 #ifndef _SYS_OS16_H
330002 #define _SYS_OS16_H 1
330003 //-----
330004 // This file contains all the declarations that don't have a better
330005 // place inside standard headers files. Even declarations related to
330006 // device numbers and system calls is contained here.
330007 //-----
330008 // Please remember that system calls should never be used (called)
330009 // inside the kernel code, because system calls cannot be nested for
330010 // the os16 simple architecture!
330011 // If a particular function is necessary inside the kernel, that usually
330012 // is made by a system call, an appropriate k_...() function must be
330013 // made, to avoid the problem.
330014 //-----
330015
330016 #include <sys/types.h>
330017 #include <sys/stat.h>
330018 #include <stdint.h>
330019 #include <signal.h>
330020 #include <limits.h>
330021 #include <stdio.h>
330022 #include <stdint.h>
330023 #include <stddef.h>
330024 #include <const.h>
330025 #include <restrict.h>
330026 #include <stdarg.h>
330027 //-----
330028 // Device numbers.
330029 //-----
330030 #define DEV_UNDEFINED_MAJOR 0x00
330031 #define DEV_UNDEFINED 0x0000
330032 #define DEV_MEM_MAJOR 0x01
330033 #define DEV_MEM 0x0101
330034 #define DEV_NULL 0x0102
330035 #define DEV_PORT 0x0103
330036 #define DEV_ZERO 0x0104
330037 #define DEV_TTY_MAJOR 0x02
330038 #define DEV_TTY 0x0200
330039 #define DEV_DSK_MAJOR 0x03
330040 #define DEV_DSK0 0x0300
330041 #define DEV_DSK1 0x0301
330042 #define DEV_DSK2 0x0302
330043 #define DEV_DSK3 0x0303
330044 #define DEV_KMEM_MAJOR 0x04
330045 #define DEV_KMEM_PS 0x0401
330046 #define DEV_KMEM_MMP 0x0402
330047 #define DEV_KMEM_SB 0x0403
330048 #define DEV_KMEM_INODE 0x0404
330049 #define DEV_KMEM_FILE 0x0405
330050 #define DEV_CONSOLE_MAJOR 0x05
330051 #define DEV_CONSOLE 0x05FF
330052 #define DEV_CONSOLE0 0x0500
330053 #define DEV_CONSOLE1 0x0501
330054 #define DEV_CONSOLE2 0x0502
330055 #define DEV_CONSOLE3 0x0503
330056 #define DEV_CONSOLE4 0x0504
330057 //-----
330058 // Current segments.
330059 //-----
330060 uint16_t _seg_i (void);
330061 uint16_t _seg_d (void);
330062 uint16_t _cs (void);
330063 uint16_t _ds (void);
330064 uint16_t _ss (void);
330065 uint16_t _es (void);
330066 uint16_t _sp (void);
330067 uint16_t _bp (void);
330068 #define seg_i() ((unsigned int) _seg_i ())
330069 #define seg_d() ((unsigned int) _seg_d ())
330070 #define cs() ((unsigned int) _cs ())
330071 #define ds() ((unsigned int) _ds ())
330072 #define ss() ((unsigned int) _ss ())
330073 #define es() ((unsigned int) _es ())
330074 #define sp() ((unsigned int) _sp ())
330075 #define bp() ((unsigned int) _bp ())
330076 //-----
330077 #define min(a, b) (a < b ? a : b)
330078 #define max(a, b) (a > b ? a : b)
330079 //-----
330080 #define INPUT_LINE_HIDDEN 0
330081 #define INPUT_LINE_ECHO 1
330082 #define INPUT_LINE_STARS 2
330083 //-----
330084 #define MOUNT_DEFAULT 0 // Default mount options.
330085 #define MOUNT_RO 1 // Read only mount option.
330086 //-----
330087 #define SYS_0 0 // Nothing to do.
330088 #define SYS_CHDIR 1
330089 #define SYS_CHMOD 2
330090 #define SYS_CLOCK 3
330091 #define SYS_CLOSE 4

```

1811

```

330092 #define SYS_EXEC 5
330093 #define SYS_EXIT 6 // [1] see below.
330094 #define SYS_FCHMOD 7
330095 #define SYS_FORK 8
330096 #define SYS_FSTAT 9
330097 #define SYS_KILL 10
330098 #define SYS_LSEEK 11
330099 #define SYS_MKDIR 12
330100 #define SYS_MKNOD 13
330101 #define SYS_MOUNT 14
330102 #define SYS_OPEN 15
330103 #define SYS_PGRP 16
330104 #define SYS_READ 17
330105 #define SYS_SETEUID 18
330106 #define SYS_SETUID 19
330107 #define SYS_SIGNAL 20
330108 #define SYS_SLEEP 21
330109 #define SYS_STAT 22
330110 #define SYS_TIME 23
330111 #define SYS_UMASK 24
330112 #define SYS_UMOUNT 25
330113 #define SYS_WAIT 26
330114 #define SYS_WRITE 27
330115 #define SYS_2PCHAR 28
330116 #define SYS_2PSTR 29 // [2] see below.
330117 #define SYS_2PSTRING 30 // [2]
330118 #define SYS_CHOWN 31
330119 #define SYS_DUP 33
330120 #define SYS_DUP2 34
330121 #define SYS_LINK 35
330122 #define SYS_UNLINK 36
330123 #define SYS_FCNTL 37
330124 #define SYS_STIME 38
330125 #define SYS_FCHOWN 39
330126 //
330127 // [1] The files 'crt0...' need to know the value used for the
330128 // exit system call. If this value is modified, all the file
330129 // 'crt0...' have also to be modified the same way.
330130 //
330131 // [2] These system calls were developed at the beginning, when no
330132 // standard I/O was available. They are to be considered as a
330133 // last resort for debugging purposes.
330134 //
330135 //-----
330136 typedef struct {
330137     char path[PATH_MAX];
330138     int ret;
330139     int errno;
330140     int errln;
330141     char errfn[PATH_MAX];
330142 } sysmsg_chdir_t;
330143 //-----
330144 typedef struct {
330145     char path[PATH_MAX];
330146     mode_t mode;
330147     int ret;
330148     int errno;
330149     int errln;
330150     char errfn[PATH_MAX];
330151 } sysmsg_chmod_t;
330152 //-----
330153 typedef struct {
330154     char path[PATH_MAX];
330155     uid_t uid;
330156     uid_t gid;
330157     int ret;
330158     int errno;
330159     int errln;
330160     char errfn[PATH_MAX];
330161 } sysmsg_chown_t;
330162 //-----
330163 typedef struct {
330164     clock_t ret;
330165 } sysmsg_clock_t;
330166 //-----
330167 typedef struct {
330168     int fdn;
330169     int ret;
330170     int errno;
330171     int errln;
330172     char errfn[PATH_MAX];
330173 } sysmsg_close_t;
330174 //-----
330175 typedef struct {
330176     int fdn_old;
330177     int ret;
330178     int errno;
330179     int errln;
330180     char errfn[PATH_MAX];
330181 } sysmsg_dup_t;
330182 //-----
330183 typedef struct {
330184     int fdn_old;
330185     int fdn_new;
330186     int ret;
330187     int errno;
330188     int errln;
330189     char errfn[PATH_MAX];
330190 } sysmsg_dup2_t;
330191 //-----
330192 typedef struct {

```

```

330193     char path[PATH_MAX];
330194     int argc;
330195     int envc;
330196     char arg_data[ARG_MAX/2];
330197     char env_data[ARG_MAX/2];
330198     uid_t uid;
330199     uid_t euid;
330200     int ret;
330201     int errno;
330202     int errln;
330203     char errfn[PATH_MAX];
330204 } sysmsg_exec_t;
330205 //-----
330206 typedef struct {
330207     int status;
330208 } sysmsg_exit_t;
330209 //-----
330210 typedef struct {
330211     int fdn;
330212     mode_t mode;
330213     int ret;
330214     int errno;
330215     int errln;
330216     char errfn[PATH_MAX];
330217 } sysmsg_fchmod_t;
330218 //-----
330219 typedef struct {
330220     int fdn;
330221     uid_t uid;
330222     uid_t gid;
330223     int ret;
330224     int errno;
330225     int errln;
330226     char errfn[PATH_MAX];
330227 } sysmsg_fchown_t;
330228 //-----
330229 typedef struct {
330230     int fdn;
330231     int cmd;
330232     int arg;
330233     int ret;
330234     int errno;
330235     int errln;
330236     char errfn[PATH_MAX];
330237 } sysmsg_fcntl_t;
330238 //-----
330239 typedef struct {
330240     pid_t ret;
330241     int errno;
330242     int errln;
330243     char errfn[PATH_MAX];
330244 } sysmsg_fork_t;
330245 //-----
330246 typedef struct {
330247     int fdn;
330248     struct stat stat;
330249     int ret;
330250     int errno;
330251     int errln;
330252     char errfn[PATH_MAX];
330253 } sysmsg_fstat_t;
330254 //-----
330255 typedef struct {
330256     pid_t pid;
330257     int signal;
330258     int ret;
330259     int errno;
330260     int errln;
330261     char errfn[PATH_MAX];
330262 } sysmsg_kill_t;
330263 //-----
330264 typedef struct {
330265     char path_old[PATH_MAX];
330266     char path_new[PATH_MAX];
330267     int ret;
330268     int errno;
330269     int errln;
330270     char errfn[PATH_MAX];
330271 } sysmsg_link_t;
330272 //-----
330273 typedef struct {
330274     int fdn;
330275     off_t offset;
330276     int whence;
330277     int ret;
330278     int errno;
330279     int errln;
330280     char errfn[PATH_MAX];
330281 } sysmsg_lseek_t;
330282 //-----
330283 typedef struct {
330284     char path[PATH_MAX];
330285     mode_t mode;
330286     int ret;
330287     int errno;
330288     int errln;
330289     char errfn[PATH_MAX];
330290 } sysmsg_mkdir_t;
330291 //-----
330292 typedef struct {
330293     char path[PATH_MAX];

```

```

330024 mode_t mode;
330025 dev_t device;
330026 int ret;
330027 int errno;
330028 int errln;
330029 char errfn[PATH_MAX];
330030 } sysmsg_mknod_t;
330031 //-----
330032 typedef struct {
330033     char path_dev[PATH_MAX];
330034     char path_mnt[PATH_MAX];
330035     int options;
330036     int ret;
330037     int errno;
330038     int errln;
330039     char errfn[PATH_MAX];
330040 } sysmsg_mount_t;
330041 //-----
330042 typedef struct {
330043     char path[PATH_MAX];
330044     int flags;
330045     mode_t mode;
330046     int ret;
330047     int errno;
330048     int errln;
330049     char errfn[PATH_MAX];
330050 } sysmsg_open_t;
330051 //-----
330052 typedef struct {
330053     int fdn;
330054     char buffer[BUFSIZ];
330055     size_t count;
330056     int eof;
330057     ssize_t ret;
330058     int errno;
330059     int errln;
330060     char errfn[PATH_MAX];
330061 } sysmsg_read_t;
330062 //-----
330063 typedef struct {
330064     uid_t euid;
330065     int ret;
330066     int errno;
330067     int errln;
330068     char errfn[PATH_MAX];
330069 } sysmsg_seteuid_t;
330070 //-----
330071 typedef struct {
330072     uid_t uid;
330073     uid_t euid;
330074     uid_t suid;
330075     int ret;
330076     int errno;
330077     int errln;
330078     char errfn[PATH_MAX];
330079 } sysmsg_setuid_t;
330080 //-----
330081 typedef struct {
330082     sighandler_t handler;
330083     int signal;
330084     sighandler_t ret;
330085     int errno;
330086     int errln;
330087     char errfn[PATH_MAX];
330088 } sysmsg_signal_t;
330089 //-----
330090 #define WAKEUP_EVENT_SIGNAL 1 // 1, 2, 4, 8, 16, ...
330091 #define WAKEUP_EVENT_TIMER 2 // so that can be 'OR' combined.
330092 #define WAKEUP_EVENT_TTY 4 //
330093 typedef struct {
330094     int events;
330095     int signal;
330096     unsigned int seconds;
330097     time_t ret;
330098 } sysmsg_sleep_t;
330099 //-----
330100 typedef struct {
330101     char path[PATH_MAX];
330102     struct stat stat;
330103     int ret;
330104     int errno;
330105     int errln;
330106     char errfn[PATH_MAX];
330107 } sysmsg_stat_t;
330108 //-----
330109 typedef struct {
330110     time_t ret;
330111 } sysmsg_time_t;
330112 //-----
330113 typedef struct {
330114     time_t timer;
330115     int ret;
330116 } sysmsg_stime_t;
330117 //-----
330118 typedef struct {
330119     uid_t uid; // Read user ID.
330120     uid_t euid; // Effective user ID.
330121     uid_t suid; // Saved user ID.
330122     pid_t pid; // Process ID.
330123     pid_t ppid; // Parent PID.
330124     pid_t pgrp; // Process group.

```

1814

```

330095 mode_t umask; // Access permission mask.
330096 char path_cwd[PATH_MAX];
330097 } sysmsg_uarea_t;
330098 //-----
330099 typedef struct {
330100     mode_t umask;
330101     mode_t ret;
330102 } sysmsg_umask_t;
330103 //-----
330104 typedef struct {
330105     char path_mnt[PATH_MAX];
330106     int ret;
330107     int errno;
330108     int errln;
330109     char errfn[PATH_MAX];
330110 } sysmsg_umount_t;
330111 //-----
330112 typedef struct {
330113     char path[PATH_MAX];
330114     int ret;
330115     int errno;
330116     int errln;
330117     char errfn[PATH_MAX];
330118 } sysmsg_unlink_t;
330119 //-----
330120 typedef struct {
330121     int status;
330122     pid_t ret;
330123     int errno;
330124     int errln;
330125     char errfn[PATH_MAX];
330126 } sysmsg_wait_t;
330127 //-----
330128 typedef struct {
330129     int fdn;
330130     char buffer[BUFSIZ];
330131     size_t count;
330132     ssize_t ret;
330133     int errno;
330134     int errln;
330135     char errfn[PATH_MAX];
330136 } sysmsg_write_t;
330137 //-----
330138 typedef struct {
330139     char c;
330140 } sysmsg_zpchar_t;
330141 //-----
330142 typedef struct {
330143     char string[BUFSIZ];
330144 } sysmsg_zpstring_t;
330145 //-----
330146 void heap_clear (void);
330147 int heap_min (void);
330148 void input_line (char *line, char *prompt, size_t size, int type);
330149 int mount (const char *path_dev, const char *path_mnt,
330150           int options);
330151 int namep (const char *name, char *path, size_t size);
330152 void process_info (void);
330153 void sys (int syscallnr, void *message, size_t size);
330154 int umount (const char *path_mnt);
330155 void z_perror (const char *string);
330156 int z_printf (const char *restrict format, ...);
330157 int z_putchar (int c);
330158 int z_puts (const char *string);
330159 int z_vprintf (const char *restrict format, va_list arg);
330160 //int z_vsprintf (char *restrict string, const char *restrict format,
330161 //               va_list arg);
330162 //-----
330163 #endif
330164 #endif

```

lib/sys/os16/_bp.s

Si veda la sezione u0.12.

«

```

334001 .global __bp
334002 .text
334003 ;-----
334004 ; Read the base pointer, as it is before this call.
334005 ;-----
334006 .align 2
334007 __bp:
334008     enter #2, #0 // 1 local variable.
334009     pushf
334010     cli
334011     pusha
334012     mov ax, [bp] // The previous BP value is saved at *BP.
334013     mov -2[bp], ax // Save the calculated old SP value.
334014     popa
334015     popf
334016     mov ax, -2[bp] // AX is the function return value.
334017     leave
334018     ret

```

1815

<<

Si veda la sezione u0.12.

```

3350001 .global __cs
3350002 .text
3350003 ;-----
3350004 ; Read the code segment value.
3350005 ;-----
3350006 .align 2
3350007 __cs:
3350008     mov ax, cs
3350009     ret

```

<<

Si veda la sezione u0.12.

```

3360001 .global __ds
3360002 .text
3360003 ;-----
3360004 ; Read the data segment value.
3360005 ;-----
3360006 .align 2
3360007 __ds:
3360008     mov ax, ds
3360009     ret

```

<<

Si veda la sezione u0.12.

```

3370001 .global __es
3370002 .text
3370003 ;-----
3370004 ; Read the extra segment value.
3370005 ;-----
3370006 .align 2
3370007 __es:
3370008     mov ax, es
3370009     ret

```

<<

Si veda la sezione u0.91.

```

3380001 .global __seg_d
3380002 .text
3380003 ;-----
3380004 ; Read the data segment value.
3380005 ;-----
3380006 .align 2
3380007 __seg_d:
3380008     mov ax, ds
3380009     ret

```

<<

Si veda la sezione u0.91.

```

3390001 .global __seg_i
3390002 .text
3390003 ;-----
3390004 ; Read the instruction segment value.
3390005 ;-----
3390006 .align 2
3390007 __seg_i:
3390008     mov ax, cs
3390009     ret

```

<<

Si veda la sezione u0.12.

```

3400001 .global __sp
3400002 .text
3400003 ;-----
3400004 ; Read the stack pointer, as it is before this call.
3400005 ;-----
3400006 .align 2
3400007 __sp:
3400008     enter #2, #0 ; 1 local variable.
3400009     pushf
3400010     cli
3400011     pusha
3400012     mov ax, bp ; The previous SP is equal to BP + 2 + 2.
3400013     add ax, #4 ;
3400014     mov -2[bp], ax ; Save the calculated old SP value.
3400015     popa
3400016     popf
3400017     mov ax, -2[bp] ; AX is the function return value.
3400018     leave
3400019     ret

```

>>

Si veda la sezione u0.12.

```

3410001 .global __ss
3410002 .text
3410003 ;-----
3410004 ; Read the stack segment value.
3410005 ;-----
3410006 .align 2
3410007 __ss:
3410008     mov ax, ss
3410009     ret

```

>>

Si veda la sezione u0.57.

```

3420001 #include <sys/os16.h>
3420002 //-----
3420003 extern uint16_t _end;
3420004 //-----
3420005 void heap_clear (void)
3420006 {
3420007     uint16_t *a = &_end;
3420008     uint16_t *z = (void *) (sp () - 2);
3420009     for (; a < z; a++)
3420010     {
3420011         *a = 0xFFFF;
3420012     }
3420013 }

```

>>

Si veda la sezione u0.57.

```

3430001 #include <sys/os16.h>
3430002 //-----
3430003 extern uint16_t _end;
3430004 //-----
3430005 int heap_min (void)
3430006 {
3430007     uint16_t *a = &_end;
3430008     uint16_t *z = (void *) (sp () - 2);
3430009     int count;
3430010     for (count = 0; a < z && *a == 0xFFFF; a++, count++);
3430011     return (count * 2);
3430012 }

```

>>

Si veda la sezione u0.60.

```

3440001 #include <sys/os16.h>
3440002 #include <string.h>
3440003 #include <stdio.h>
3440004 //-----
3440005 void
3440006 input_line (char *line, char *prompt, size_t size, int type)
3440007 {
3440008     int i; // Index inside the 'line[]' array.
3440009     int c; // Character received from keyboard.
3440010
3440011     if (prompt != NULL || strlen (prompt) > 0)
3440012     {
3440013         printf ("%s ", prompt);
3440014     }
3440015     //
3440016     // Loop for character input. Please note that the loop
3440017     // will exit only through 'break', where the input line
3440018     // will also be correctly terminated with '\0'.
3440019     //
3440020     for (i = 0; i++)
3440021     {
3440022         c = getchar ();
3440023         //
3440024         // Control codes.
3440025         //
3440026         if (c == EOF)
3440027         {
3440028             line[i] = 0;
3440029             break;
3440030         }
3440031         else if (c == 4) // [Ctrl D]
3440032         {
3440033             line[i] = 0;
3440034             break;
3440035         }
3440036         else if (c == 10) // [Enter]
3440037         {
3440038             line[i] = 0;
3440039             break;
3440040         }
3440041         else if (c == 8) // [Backspace]
3440042         {
3440043             if (i == 0)
3440044             {
3440045                 //
3440046                 // It is already the lowest position, so the video

```

```

3440047 // cursor is moved forward again, so that the prompt
3440048 // is not overwritten.
3440049 // The index is set to -1, so that on the next loop,
3440050 // it will be again zero.
3440051 //
3440052 printf (" ");
3440053 i = -1;
3440054 }
3440055 else
3440056 {
3440057     i -= 2;
3440058 }
3440059 continue;
3440060 }
3440061 //
3440062 // If 'i' is equal 'size - 1', it is not allowed to continue
3440063 // typing.
3440064 //
3440065 if (i == (size - 1))
3440066 {
3440067     //
3440068     // Ignore typing, move back the cursor, delete the character
3440069     // typed and move back again.
3440070     //
3440071     printf ("\b \b");
3440072     i--;
3440073     continue;
3440074 }
3440075 //
3440076 // Typing is allowed.
3440077 //
3440078 line[i] = c;
3440079 //
3440080 // Verify if it should be hidden.
3440081 //
3440082 if (type == INPUT_LINE_HIDDEN)
3440083 {
3440084     printf ("\b "); // Space: at least you see something.
3440085 }
3440086 else if (type == INPUT_LINE_STARS)
3440087 {
3440088     printf ("\b*");
3440089 }
3440090 }
3440091 }

```

lib/sys/os16/mount.c

Si veda la sezione u0.27.

```

3450001 #include <sys/types.h>
3450002 #include <errno.h>
3450003 #include <sys/os16.h>
3450004 #include <stddef.h>
3450005 #include <string.h>
3450006 #include <const.h>
3450007 //-----
3450008 int
3450009 mount (const char *path_dev, const char *path_mnt, int options)
3450010 {
3450011     sysmsg_mount_t msg;
3450012     //
3450013     strncpy (msg.path_dev, path_dev, PATH_MAX);
3450014     strncpy (msg.path_mnt, path_mnt, PATH_MAX);
3450015     msg.options = options;
3450016     msg.ret = 0;
3450017     msg.errno = 0;
3450018     //
3450019     sys (SYS_MOUNT, &msg, (sizeof msg));
3450020     //
3450021     errno = msg.errno;
3450022     errln = msg.errln;
3450023     strncpy (errfn, msg.errfn, PATH_MAX);
3450024     return (msg.ret);
3450025 }

```

lib/sys/os16/namep.c

Si veda la sezione u0.74.

```

3460001 #include <sys/os16.h>
3460002 #include <stdlib.h>
3460003 #include <errno.h>
3460004 #include <unistd.h>
3460005 //-----
3460006 int
3460007 namep (const char *name, char *path, size_t size)
3460008 {
3460009     char command[PATH_MAX];
3460010     char *env_path;
3460011     int p; // Index used inside the path environment.
3460012     int c; // Index used inside the command string.
3460013     int status;
3460014     //
3460015     // Check for valid input.
3460016     //
3460017     if (name == NULL || name[0] == 0 || path == NULL || name == path)
3460018     {
3460019         errset (EINVAL); // Invalid argument.
3460020         return (-1);

```

1818

```

}
3460021 //
3460022 // Check if the original command contains at least a '/'. Otherwise
3460023 // a scan for the environment variable 'PATH' must be done.
3460024 //
3460025 if (strchr (name, '/') == NULL)
3460026 {
3460027     //
3460028     // Ok: no '/' there. Get the environment variable 'PATH'.
3460029     //
3460030     env_path = getenv ("PATH");
3460031     if (env_path == NULL)
3460032     {
3460033         //
3460034         // There is no 'PATH' environment value.
3460035         //
3460036         errset (ENOENT); // No such file or directory.
3460037         return (-1);
3460038     }
3460039     //
3460040     // Scan paths and try to find a file with that name.
3460041     //
3460042     for (p = 0; env_path[p] != 0;)
3460043     {
3460044         for (c = 0;
3460045              c < (PATH_MAX - strlen(name) - 2) &&
3460046              env_path[p] != 0 &&
3460047              env_path[p] != ':' &&
3460048              c++, p++)
3460049         {
3460050             command[c] = env_path[p];
3460051         }
3460052         //
3460053         // If the loop is ended because the command array does not
3460054         // have enough room for the full path, then must return an
3460055         // error.
3460056         //
3460057         if (env_path[p] != ':' && env_path[p] != 0)
3460058         {
3460059             errset (ENAMETOOLONG); // Filename too long.
3460060             return (-1);
3460061         }
3460062         //
3460063         // The command array has enough space. At index 'c' must
3460064         // place a zero, to terminate current string.
3460065         //
3460066         command[c] = 0;
3460067         //
3460068         // Add the rest of the path.
3460069         //
3460070         strcat (command, "/");
3460071         strcat (command, name);
3460072         //
3460073         // Verify to have something with that full path name.
3460074         //
3460075         status = access (command, F_OK);
3460076         if (status == 0)
3460077         {
3460078             //
3460079             // Verify to have enough room inside the destination
3460080             // path.
3460081             //
3460082             if (strlen (command) >= size)
3460083             {
3460084                 //
3460085                 // Sorry: too big. There must be room also for
3460086                 // the string termination null character.
3460087                 //
3460088                 errset (ENAMETOOLONG); // Filename too long.
3460089                 return (-1);
3460090             }
3460091             //
3460092             // Copy the path and return.
3460093             //
3460094             strncpy (path, command, size);
3460095             return (0);
3460096         }
3460097         //
3460098         // That path was not good: try again. But before returning
3460099         // to the external loop, must verify if 'p' is to be
3460100         // incremented, after a ':', because the external loop
3460101         // does not touch the index 'p',
3460102         //
3460103         if (env_path[p] == ':')
3460104         {
3460105             p++;
3460106         }
3460107     }
3460108     //
3460109     // At this point, there is no match with the paths.
3460110     //
3460111     errset (ENOENT); // No such file or directory.
3460112     return (-1);
3460113 }
3460114 //
3460115 // At this point, a path was given and the environment variable
3460116 // 'PATH' was not scanned. Just copy the same path. But must verify
3460117 // that the receiving path has enough room for it.
3460118 //
3460119 if (strlen (name) >= size)
3460120 {
3460121

```

1819


```

3460122 //
3460123 // Sorry: too big.
3460124 //
3460125 errset (ENAMETOOLONG); // Filename too long.
3460126 return (-1);
3460127 }
3460128 //
3460129 // Ok: copy and return.
3460130 //
3460131 strncpy (path, name, size);
3460132 return (0);
3460133 }

```

lib/sys/os16/process_info.c

<<

Si veda la sezione u0.79.

```

3470001 #include <sys/os16.h>
3470002 #include <stdio.h>
3470003
3470004 extern uint16_t _edata;
3470005 extern uint16_t _end;
3470006 //-----
3470007 void
3470008 process_info (void)
3470009 {
3470010     printf ("cs=%04x ds=%04x ss=%04x es=%04x bp=%04x sp=%04x ",
3470011           cs (), ds (), ss (), es (), bp (), sp ());
3470012     printf ("edata=%04x ebss=%04x heap=%04x\n",
3470013           (int) &_edata, (int) &_end, heap_min ());
3470014 }

```

lib/sys/os16/sys.s

<<

Si veda la sezione u0.37.

```

3480001 .global _sys
3480002 .text
3480003 ;-----
3480004 ; Call a system call.
3480005 ;
3480006 ; Please remember that system calls should never be used (called) inside
3480007 ; the kernel code, because system calls cannot be nested for the os16
3480008 ; simple architecture!
3480009 ; If a particular function is necessary inside the kernel, that usually
3480010 ; is made by a system call, an appropriate k_...() function must be
3480011 ; made, to avoid the problem.
3480012 ;
3480013 ;-----
3480014 .align 2
3480015 _sys:
3480016     int    #0x80
3480017     ret

```

lib/sys/os16/umount.c

<<

Si veda la sezione u0.27.

```

3490001 #include <sys/types.h>
3490002 #include <errno.h>
3490003 #include <sys/os16.h>
3490004 #include <stddef.h>
3490005 #include <string.h>
3490006 //-----
3490007 int
3490008 umount (const char *path_mnt)
3490009 {
3490010     sysmsg_umount_t msg;
3490011     //
3490012     strncpy (msg.path_mnt, path_mnt, PATH_MAX);
3490013     msg.ret = 0;
3490014     msg.errno = 0;
3490015     //
3490016     sys (SYS_UMOUNT, &msg, (sizeof msg));
3490017     //
3490018     errno = msg.errno;
3490019     errln = msg.errln;
3490020     strncpy (errfn, msg.errfn, PATH_MAX);
3490021     return (msg.ret);
3490022 }

```

lib/sys/os16/z_perror.c

<<

Si veda la sezione u0.45.

```

3500001 #include <sys/os16.h>
3500002 #include <errno.h>
3500003 #include <stddef.h>
3500004 #include <string.h>
3500005 //-----
3500006 void
3500007 z_perror (const char *string)
3500008 {
3500009     //
3500010     // If errno is zero, there is nothing to show.
3500011     //
3500012     if (errno == 0)

```

```

3500013 {
3500014     return;
3500015 }
3500016 //
3500017 // Show the string if there is one.
3500018 //
3500019 if (string != NULL && strlen (string) > 0)
3500020 {
3500021     z_printf ("%s ", string);
3500022 }
3500023 //
3500024 // Show the translated error.
3500025 //
3500026 if (errfn[0] != 0 && errln != 0)
3500027 {
3500028     z_printf ("[%s:%u:%i] %s\n",
3500029             errfn, errln, errno, strerror (errno));
3500030 }
3500031 else
3500032 {
3500033     z_printf ("[%i] %s\n", errno, strerror (errno));
3500034 }
3500035 }

```

lib/sys/os16/z_printf.c

<<

Si veda la sezione u0.45.

```

3510001 #include <sys/os16.h>
3510002 //-----
3510003 int
3510004 z_printf (char *format, ...)
3510005 {
3510006     va_list ap;
3510007     va_start (ap, format);
3510008     return z_vprintf (format, ap);
3510009 }

```

lib/sys/os16/z_putchar.c

<<

Si veda la sezione u0.45.

```

3520001 #include <sys/os16.h>
3520002 //-----
3520003 int
3520004 z_putchar (int c)
3520005 {
3520006     sysmsg_zpchar_t msg;
3520007     msg.c = c;
3520008     sys (SYS_ZPCHAR, &msg, (sizeof msg));
3520009     return (c);
3520010 }

```

lib/sys/os16/z_puts.c

<<

Si veda la sezione u0.45.

```

3530001 #include <sys/os16.h>
3530002 //-----
3530003 int
3530004 z_puts (char *string)
3530005 {
3530006     unsigned int i;
3530007     for (i = 0; string[i] != 0; string++)
3530008     {
3530009         z_putchar ((int) string[i]);
3530010     }
3530011     z_putchar ((int) '\n');
3530012     return (1);
3530013 }

```

lib/sys/os16/z_vprintf.c

<<

Si veda la sezione u0.45.

```

3540001 #include <sys/os16.h>
3540002 //-----
3540003 int
3540004 z_vprintf (char *format, va_list arg)
3540005 {
3540006     int                ret;
3540007     sysmsg_vpstring_t msg;
3540008     msg.string[0] = 0;
3540009     ret = vsprintf (msg.string, format, arg);
3540010     sys (SYS_ZPSTRING, &msg, (sizeof msg));
3540011     return ret;
3540012 }

```

os16: «lib/sys/stat.h»

<<

Si veda la sezione u0.2.

```

3550001 #ifndef _SYS_STAT_H
3550002 #define _SYS_STAT_H    1
3550003
3550004 #include <restrict.h>

```

```

3550005 #include <const.h>
3550006 #include <sys/types.h> // dev_t
3550007 // off_t
3550008 // blkcnt_t
3550009 // blksize_t
3550010 // ino_t
3550011 // mode_t
3550012 // nlink_t
3550013 // uid_t
3550014 // gid_t
3550015 // time_t
3550016 //-----
3550017 // File type.
3550018 //-----
3550019 #define S_IFMT 0170000 // File type mask.
3550020 //
3550021 #define S_IFBLK 0060000 // Block device file.
3550022 #define S_IFCHR 0020000 // Character device file.
3550023 #define S_IFIFO 0010000 // Pipe (FIFO) file.
3550024 #define S_IFREG 0100000 // Regular file.
3550025 #define S_IFDIR 0040000 // Directory.
3550026 #define S_IFLNK 0120000 // Symbolic link.
3550027 #define S_IFSOCK 0140000 // Unix domain socket.
3550028 //-----
3550029 // Owner user access permissions.
3550030 //-----
3550031 #define S_IRWXU 0000700 // Owner user access permissions mask.
3550032 //
3550033 #define S_IRUSR 0000400 // Owner user read access permission.
3550034 #define S_IWUSR 0000200 // Owner user write access permission.
3550035 #define S_IXUSR 0000100 // Owner user execution or cross perm.
3550036 //-----
3550037 // Group owner access permissions.
3550038 //-----
3550039 #define S_IRWXG 0000070 // Owner group access permissions mask.
3550040 //
3550041 #define S_IRGRP 0000040 // Owner group read access permission.
3550042 #define S_IWGRP 0000020 // Owner group write access permission.
3550043 #define S_IXGRP 0000010 // Owner group execution or cross perm.
3550044 //-----
3550045 // Other users access permissions.
3550046 //-----
3550047 #define S_IRWXO 0000007 // Other users access permissions mask.
3550048 //
3550049 #define S_IROTH 0000004 // Other users read access permission.
3550050 #define S_IWOTH 0000002 // Other users write access permissions.
3550051 #define S_IXOTH 0000001 // Other users execution or cross perm.
3550052 //-----
3550053 // S-bit: in this case there is no mask to select all of them.
3550054 //-----
3550055 #define S_ISUID 0004000 // S-UID.
3550056 #define S_ISGID 0002000 // S-GID.
3550057 #define S_ISVTX 0001000 // Sticky.
3550058 //-----
3550059 // Macro-instructions to verify the type of file.
3550060 //-----
3550061 #define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK) // Block device.
3550062 #define S_ISCHR(m) (((m) & S_IFMT) == S_IFCHR) // Character device.
3550063 #define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO) // FIFO.
3550064 #define S_ISREG(m) (((m) & S_IFMT) == S_IFREG) // Regular file.
3550065 #define S_ISDIR(m) (((m) & S_IFMT) == S_IFDIR) // Directory.
3550066 #define S_ISLNK(m) (((m) & S_IFMT) == S_IFLNK) // Symbolic link.
3550067 #define S_ISSOCK(m) (((m) & S_IFMT) == S_IFSOCK) // Socket.
3550068 //-----
3550069 // Structure 'stat'.
3550070 //-----
3550071 struct stat {
3550072     dev_t    st_dev; // Device containing the file.
3550073     ino_t    st_ino; // File serial number (inode number).
3550074     mode_t   st_mode; // File type and permissions.
3550075     nlink_t  st_nlink; // Links to the file.
3550076     uid_t    st_uid; // Owner user id.
3550077     gid_t    st_gid; // Owner group id.
3550078     dev_t    st_rdev; // Device number if it is a device file.
3550079     off_t    st_size; // File size.
3550080     time_t   st_atime; // Last access time.
3550081     time_t   st_mtime; // Last modification time.
3550082     time_t   st_ctime; // Last inode modification.
3550083     blksize_t st_blksize; // Block size for I/O operations.
3550084     blkcnt_t st_blocks; // File size / block size.
3550085 };
3550086 //-----
3550087 // Function prototypes.
3550088 //-----
3550089 int chmod (const char *path, mode_t mode);
3550090 int fchmod (int fdn, mode_t mode);
3550091 int fstat (int fdn, struct stat *buffer);
3550092 int lstat (const char *restrict path, struct stat *restrict buffer);
3550093 int mkdir (const char *path, mode_t mode);
3550094 int mkfifo (const char *path, mode_t mode);
3550095 int mknod (const char *path, mode_t mode, dev_t dev);
3550096 int stat (const char *restrict path, struct stat *restrict buffer);
3550097 mode_t umask (mode_t mask);
3550098
3550099 #endif // _SYS_STAT_H

```

lib/sys/stat/chmod.c

Si veda la sezione u0.4.

```

3560001 #include <sys/stat.h>
3560002 #include <string.h>
3560003 #include <const.h>
3560004 #include <sys/osl6.h>
3560005 #include <errno.h>
3560006 #include <limits.h>
3560007 //-----
3560008 int
3560009 chmod (const char *path, mode_t mode)
3560010 {
3560011     sysmsg_chmod_t msg;
3560012 //
3560013     strncpy (msg.path, path, PATH_MAX);
3560014     msg.mode = mode;
3560015 //
3560016     sys (SYS_CHMOD, &msg, (sizeof msg));
3560017 //
3560018     errno = msg.errno;
3560019     errln = msg.errln;
3560020     strncpy (errfn, msg.errfn, PATH_MAX);
3560021     return (msg.ret);
3560022 }

```

lib/sys/stat/fchmod.c

Si veda la sezione u0.4.

```

3570001 #include <sys/stat.h>
3570002 #include <string.h>
3570003 #include <const.h>
3570004 #include <sys/osl6.h>
3570005 #include <errno.h>
3570006 #include <limits.h>
3570007 //-----
3570008 int
3570009 fchmod (int fdn, mode_t mode)
3570010 {
3570011     sysmsg_fchmod_t msg;
3570012 //
3570013     msg.fdn = fdn;
3570014     msg.mode = mode;
3570015 //
3570016     sys (SYS_FCHMOD, &msg, (sizeof msg));
3570017 //
3570018     errno = msg.errno;
3570019     errln = msg.errln;
3570020     strncpy (errfn, msg.errfn, PATH_MAX);
3570021     return (msg.ret);
3570022 }

```

lib/sys/stat/fstat.c

Si veda la sezione u0.36.

```

3580001 #include <unistd.h>
3580002 #include <errno.h>
3580003 #include <sys/osl6.h>
3580004 #include <string.h>
3580005 //-----
3580006 int
3580007 fstat (int fdn, struct stat *buffer)
3580008 {
3580009     sysmsg_fstat_t msg;
3580010 //
3580011     msg.fdn = fdn;
3580012     msg.stat.st_dev = buffer->st_dev;
3580013     msg.stat.st_ino = buffer->st_ino;
3580014     msg.stat.st_mode = buffer->st_mode;
3580015     msg.stat.st_nlink = buffer->st_nlink;
3580016     msg.stat.st_uid = buffer->st_uid;
3580017     msg.stat.st_gid = buffer->st_gid;
3580018     msg.stat.st_rdev = buffer->st_rdev;
3580019     msg.stat.st_size = buffer->st_size;
3580020     msg.stat.st_atime = buffer->st_atime;
3580021     msg.stat.st_mtime = buffer->st_mtime;
3580022     msg.stat.st_ctime = buffer->st_ctime;
3580023     msg.stat.st_blksize = buffer->st_blksize;
3580024     msg.stat.st_blocks = buffer->st_blocks;
3580025 //
3580026     sys (SYS_FSTAT, &msg, (sizeof msg));
3580027 //
3580028     buffer->st_dev = msg.stat.st_dev;
3580029     buffer->st_ino = msg.stat.st_ino;
3580030     buffer->st_mode = msg.stat.st_mode;
3580031     buffer->st_nlink = msg.stat.st_nlink;
3580032     buffer->st_uid = msg.stat.st_uid;
3580033     buffer->st_gid = msg.stat.st_gid;
3580034     buffer->st_rdev = msg.stat.st_rdev;
3580035     buffer->st_size = msg.stat.st_size;
3580036     buffer->st_atime = msg.stat.st_atime;
3580037     buffer->st_mtime = msg.stat.st_mtime;
3580038     buffer->st_ctime = msg.stat.st_ctime;
3580039     buffer->st_blksize = msg.stat.st_blksize;
3580040     buffer->st_blocks = msg.stat.st_blocks;
3580041 //
3580042     errno = msg.errno;

```

```

3580043     errln = msg.errln;
3580044     strncpy (errfn, msg.errfn, PATH_MAX);
3580045     return (msg.ret);
3580046 }

```

lib/sys/stat/mkdir.c

« Si veda la sezione u0.25.

```

3590001 #include <sys/stat.h>
3590002 #include <string.h>
3590003 #include <const.h>
3590004 #include <sys/os16.h>
3590005 #include <errno.h>
3590006 #include <limits.h>
3590007 //-----
3590008 int
3590009 mkdir (const char *path, mode_t mode)
3590010 {
3590011     sysmsg_mkdir_t msg;
3590012     //
3590013     strncpy (msg.path, path, PATH_MAX);
3590014     msg.mode = mode;
3590015     //
3590016     sys (SYS_MKDIR, &msg, (sizeof msg));
3590017     //
3590018     errno = msg.errno;
3590019     errln = msg.errln;
3590020     strncpy (errfn, msg.errfn, PATH_MAX);
3590021     return (msg.ret);
3590022 }

```

```

3610037     buffer->st_atime = msg.stat.st_atime;
3610038     buffer->st_mtime = msg.stat.st_mtime;
3610039     buffer->st_ctime = msg.stat.st_ctime;
3610040     buffer->st_blksize = msg.stat.st_blksize;
3610041     buffer->st_blocks = msg.stat.st_blocks;
3610042     //
3610043     errno = msg.errno;
3610044     errln = msg.errln;
3610045     strncpy (errfn, msg.errfn, PATH_MAX);
3610046     return (msg.ret);
3610047 }

```

lib/sys/stat/umask.c

« Si veda la sezione u0.40.

```

3620001 #include <sys/stat.h>
3620002 #include <string.h>
3620003 #include <const.h>
3620004 #include <sys/os16.h>
3620005 #include <errno.h>
3620006 #include <limits.h>
3620007 //-----
3620008 mode_t
3620009 umask (mode_t mask)
3620010 {
3620011     sysmsg_umask_t msg;
3620012     msg.umask = mask;
3620013     sys (SYS_UMASK, &msg, (sizeof msg));
3620014     return (msg.ret);
3620015 }

```

lib/sys/stat/mknod.c

« Si veda la sezione u0.26.

```

3600001 #include <unistd.h>
3600002 #include <errno.h>
3600003 #include <sys/os16.h>
3600004 #include <string.h>
3600005 //-----
3600006 int
3600007 mknod (const char *path, mode_t mode, dev_t device)
3600008 {
3600009     sysmsg_mknod_t msg;
3600010     //
3600011     strncpy (msg.path, path, PATH_MAX);
3600012     msg.mode = mode;
3600013     msg.device = device;
3600014     //
3600015     sys (SYS_MKNOD, &msg, (sizeof msg));
3600016     //
3600017     errno = msg.errno;
3600018     errln = msg.errln;
3600019     strncpy (errfn, msg.errfn, PATH_MAX);
3600020     return (msg.ret);
3600021 }

```

os16: «lib/sys/types.h»

« Si veda la sezione u0.2.

```

3600001 #ifndef _SYS_TYPES_H
3600002 #define _SYS_TYPES_H 1
3600003 //-----
3600004
3600005 #include <clock_t.h>
3600006 #include <time_t.h>
3600007 #include <size_t.h>
3600008 #include <stdint.h>
3600009 //-----
3600010 typedef long int blkcnt_t;
3600011 typedef long int blksize_t;
3600012 typedef uint16_t dev_t; // Traditional device size.
3600013 typedef unsigned int id_t;
3600014 typedef unsigned int gid_t;
3600015 typedef unsigned int uid_t;
3600016 typedef uint16_t ino_t; // Minix 1 file system inode size.
3600017 typedef uint16_t mode_t; // Minix 1 file system mode size.
3600018 typedef unsigned int nlink_t;
3600019 typedef long int off_t;
3600020 typedef int pid_t;
3600021 typedef unsigned int pthread_t;
3600022 typedef long int ssize_t;
3600023 //-----
3600024 // Common extentions.
3600025 //
3600026 dev_t makedev (int major, int minor);
3600027 int major (dev_t device);
3600028 int minor (dev_t device);
3600029 //-----
3600030
3600031 #endif

```

lib/sys/stat/stat.c

« Si veda la sezione u0.36.

```

3610001 #include <unistd.h>
3610002 #include <errno.h>
3610003 #include <sys/os16.h>
3610004 #include <string.h>
3610005 //-----
3610006 int
3610007 stat (const char *path, struct stat *buffer)
3610008 {
3610009     sysmsg_stat_t msg;
3610010     //
3610011     strncpy (msg.path, path, PATH_MAX);
3610012     //
3610013     msg.stat.st_dev = buffer->st_dev;
3610014     msg.stat.st_ino = buffer->st_ino;
3610015     msg.stat.st_mode = buffer->st_mode;
3610016     msg.stat.st_nlink = buffer->st_nlink;
3610017     msg.stat.st_uid = buffer->st_uid;
3610018     msg.stat.st_gid = buffer->st_gid;
3610019     msg.stat.st_rdev = buffer->st_rdev;
3610020     msg.stat.st_size = buffer->st_size;
3610021     msg.stat.st_atime = buffer->st_atime;
3610022     msg.stat.st_mtime = buffer->st_mtime;
3610023     msg.stat.st_ctime = buffer->st_ctime;
3610024     msg.stat.st_blksize = buffer->st_blksize;
3610025     msg.stat.st_blocks = buffer->st_blocks;
3610026     //
3610027     sys (SYS_STAT, &msg, (sizeof msg));
3610028     //
3610029     buffer->st_dev = msg.stat.st_dev;
3610030     buffer->st_ino = msg.stat.st_ino;
3610031     buffer->st_mode = msg.stat.st_mode;
3610032     buffer->st_nlink = msg.stat.st_nlink;
3610033     buffer->st_uid = msg.stat.st_uid;
3610034     buffer->st_gid = msg.stat.st_gid;
3610035     buffer->st_rdev = msg.stat.st_rdev;
3610036     buffer->st_size = msg.stat.st_size;

```

lib/sys/types/major.c

« Si veda la sezione u0.65.

```

3640001 #include <sys/types.h>
3640002 //-----
3640003 int
3640004 major (dev_t device)
3640005 {
3640006     return ((int) (device / 256));
3640007 }

```

lib/sys/types/makedev.c

« Si veda la sezione u0.65.

```

3650001 #include <sys/types.h>
3650002 //-----
3650003 dev_t
3650004 makedev (int major, int minor)
3650005 {
3650006     return ((dev_t) (major * 256 + minor));
3650007 }

```

lib/sys/types/minor.c

«

Si veda la sezione u0.65.

```
3660001 #include <sys/types.h>
3660002 //-----
3660003 int
3660004 minor (dev_t device)
3660005 {
3660006     return ((dev_t) (device & 0x00FF));
3660007 }
```

os16: «lib/sys/wait.h»

«

Si veda la sezione u0.2.

```
3670001 #ifndef _SYS_WAIT_H
3670002 #define _SYS_WAIT_H    1
3670003
3670004 #include <sys/types.h>
3670005
3670006 //-----
3670007 pid_t wait (int *status);
3670008 //-----
3670009
3670010 #endif
```

lib/sys/wait/wait.c

«

Si veda la sezione u0.43.

```
3680001 #include <sys/types.h>
3680002 #include <errno.h>
3680003 #include <sys/os16.h>
3680004 #include <stddef.h>
3680005 #include <string.h>
3680006 //-----
3680007 pid_t
3680008 wait (int *status)
3680009 {
3680010     sysmsg_wait_t msg;
3680011     msg.ret = 0;
3680012     msg.errno = 0;
3680013     msg.status = 0;
3680014     while (msg.ret == 0)
3680015     {
3680016         //
3680017         // Loop as long as there are children, an none is dead.
3680018         //
3680019         sys (SYS_WAIT, &msg, (sizeof msg));
3680020     }
3680021     errno = msg.errno;
3680022     errln = msg.errln;
3680023     strncpy (errfn, msg.errfn, PATH_MAX);
3680024     //
3680025     if (status != NULL)
3680026     {
3680027         //
3680028         // Only the low eight bits are returned.
3680029         //
3680030         *status = (msg.status & 0x00FF);
3680031     }
3680032     return (msg.ret);
3680033 }
```

os16: «lib/time.h»

«

Si veda la sezione u0.2.

```
3690001 #ifndef _TIME_H
3690002 #define _TIME_H    1
3690003 //-----
3690004
3690005 #include <const.h>
3690006 #include <restrict.h>
3690007 #include <size_t.h>
3690008 #include <time_t.h>
3690009 #include <clock_t.h>
3690010 #include <NULL.h>
3690011 #include <stdint.h>
3690012 //-----
3690013 #define CLOCKS_PER_SEC 18 // Should be 18.22 Hz, but it is a 'int'.
3690014 //-----
3690015 struct tm {int tm_sec; int tm_min; int tm_hour;
3690016           int tm_mday; int tm_mon; int tm_year;
3690017           int tm_wday; int tm_yday; int tm_isdst};
3690018 //-----
3690019 clock_t clock (void);
3690020 time_t time (time_t *timer);
3690021 int stime (time_t *timer);
3690022 double difftime (time_t time1, time_t time0);
3690023 time_t mktime (struct tm *timeptr);
3690024 struct tm *gmtime (const time_t *timer);
3690025 struct tm *localtime (const time_t *timer);
3690026 char *asctime (const struct tm *timeptr);
3690027 char *ctime (const time_t *timer);
3690028 size_t strftime (char * restrict s, size_t maxsize,
3690029                const char * restrict format,
3690030                const struct tm * restrict timeptr);
```

1826

```
3690031 //-----
3690032 #define difftime(t1,t0) ((double)((t1)-(t0)))
3690033 #define ctime(t) (asctime (localtime (t)))
3690034 #define localtime(t) (gmtime (t))
3690035 //-----
3690036
3690037 #endif
```

lib/time/asctime.c

Si veda la sezione u0.13.

«

```
3700001 #include <time.h>
3700002 #include <string.h>
3700003 #include <stdio.h>
3700004
3700005 //-----
3700006 char *
3700007 asctime (const struct tm *timeptr)
3700008 {
3700009     static char time_string[25]; // 'Sun Jan 30 24:00:00 2111'
3700010     //
3700011     // Check argument.
3700012     //
3700013     if (timeptr == NULL)
3700014     {
3700015         return (NULL);
3700016     }
3700017     //
3700018     // Set week day.
3700019     //
3700020     switch (timeptr->tm_wday)
3700021     {
3700022     case 0:
3700023         strcpy (&time_string[0], "Sun");
3700024         break;
3700025     case 1:
3700026         strcpy (&time_string[0], "Mon");
3700027         break;
3700028     case 2:
3700029         strcpy (&time_string[0], "Tue");
3700030         break;
3700031     case 3:
3700032         strcpy (&time_string[0], "Wed");
3700033         break;
3700034     case 4:
3700035         strcpy (&time_string[0], "Thu");
3700036         break;
3700037     case 5:
3700038         strcpy (&time_string[0], "Fri");
3700039         break;
3700040     case 6:
3700041         strcpy (&time_string[0], "Sat");
3700042         break;
3700043     default:
3700044         strcpy (&time_string[0], "Err");
3700045     }
3700046     //
3700047     // Set month.
3700048     //
3700049     switch (timeptr->tm_mon)
3700050     {
3700051     case 1:
3700052         strcpy (&time_string[3], " Jan");
3700053         break;
3700054     case 2:
3700055         strcpy (&time_string[3], " Feb");
3700056         break;
3700057     case 3:
3700058         strcpy (&time_string[3], " Mar");
3700059         break;
3700060     case 4:
3700061         strcpy (&time_string[3], " Apr");
3700062         break;
3700063     case 5:
3700064         strcpy (&time_string[3], " May");
3700065         break;
3700066     case 6:
3700067         strcpy (&time_string[3], " Jun");
3700068         break;
3700069     case 7:
3700070         strcpy (&time_string[3], " Jul");
3700071         break;
3700072     case 8:
3700073         strcpy (&time_string[3], " Aug");
3700074         break;
3700075     case 9:
3700076         strcpy (&time_string[3], " Sep");
3700077         break;
3700078     case 10:
3700079         strcpy (&time_string[3], " Oct");
3700080         break;
3700081     case 11:
3700082         strcpy (&time_string[3], " Nov");
3700083         break;
3700084     case 12:
3700085         strcpy (&time_string[3], " Dec");
3700086         break;
3700087     default:
3700088         strcpy (&time_string[3], " Err");
3700089     }
3700090 }
```

1827

```

370090 //
370091 // Set day of month, hour, minute, second and year.
370092 //
370093 sprintf (&time_string[7], " %2i %2i:%2i:%2i %4i",
370094         timeptr->tm_mday, timeptr->tm_hour, timeptr->tm_min,
370095         timeptr->tm_sec, timeptr->tm_year);
370096 //
370097 //
370098 //
370099 return (&time_string[0]);
370100 }

```

lib/time/clock.c

Si veda la sezione u0.6.

```

371001 #include <time.h>
371002 #include <sys/os16.h>
371003 //-----
371004 clock_t
371005 clock (void)
371006 {
371007     sysmsg_clock_t msg;
371008     msg.ret = 0;
371009     sys (SYS_CLOCK, &msg, (sizeof msg));
371010     return (msg.ret);
371011 }
371012

```

lib/time/gmtime.c

Si veda la sezione u0.13.

```

372001 #include <time.h>
372002 //-----
372003 static int leap_year (int year);
372004 //-----
372005 struct tm *
372006 gmtime (const time_t *timer)
372007 {
372008     static struct tm tms;
372009     int loop;
372010     unsigned int remainder;
372011     unsigned int days;
372012     //
372013     // Check argument.
372014     //
372015     if (timer == NULL)
372016     {
372017         return (NULL);
372018     }
372019     //
372020     // Days since epoch. There are 86400 seconds per day.
372021     // At the moment, the field 'tm_yday' will contain
372022     // all days since epoch.
372023     //
372024     days = *timer / 86400L;
372025     remainder = *timer % 86400L;
372026     //
372027     // Minutes, after full days.
372028     //
372029     tms.tm_min = remainder / 60U;
372030     //
372031     // Seconds, after full minutes.
372032     //
372033     tms.tm_sec = remainder % 60U;
372034     //
372035     // Hours, after full days.
372036     //
372037     tms.tm_hour = tms.tm_min / 60;
372038     //
372039     // Minutes, after full hours.
372040     //
372041     tms.tm_min = tms.tm_min % 60;
372042     //
372043     // Find the week day. Must remove some days to align the
372044     // calculation. So: the week days of the first week of 1970
372045     // are not valid! After 1970-01-04 calculations are right.
372046     //
372047     tms.tm_wday = (days - 3) & 7;
372048     //
372049     // Find the year: the field 'tm_yday' will be reduced to the days
372050     // of current year.
372051     //
372052     for (tms.tm_year = 1970; days > 0; tms.tm_year++)
372053     {
372054         if (leap_year (tms.tm_year))
372055         {
372056             if (days >= 366)
372057             {
372058                 days -= 366;
372059                 continue;
372060             }
372061             else
372062             {
372063                 break;
372064             }
372065         }
372066         else
372067         {

```

1828

```

372068         if (days >= 365)
372069         {
372070             days -= 365;
372071             continue;
372072         }
372073         else
372074         {
372075             break;
372076         }
372077     }
372078     }
372079     //
372080     // Day of the year.
372081     //
372082     tms.tm_yday = days + 1;
372083     //
372084     // Find the month.
372085     //
372086     tms.tm_mday = days + 1;
372087     //
372088     for (tms.tm_mon = 0, loop = 1; tms.tm_mon <= 12 && loop;)
372089     {
372090         tms.tm_mon++;
372091         //
372092         switch (tms.tm_mon)
372093         {
372094             case 1:
372095             case 3:
372096             case 5:
372097             case 7:
372098             case 8:
372099             case 10:
372100             case 12:
372101                 if (tms.tm_mday >= 31)
372102                 {
372103                     tms.tm_mday -= 31;
372104                 }
372105                 else
372106                 {
372107                     loop = 0;
372108                 }
372109                 break;
372110             case 4:
372111             case 6:
372112             case 9:
372113             case 11:
372114                 if (tms.tm_mday >= 30)
372115                 {
372116                     tms.tm_mday -= 30;
372117                 }
372118                 else
372119                 {
372120                     loop = 0;
372121                 }
372122                 break;
372123             case 2:
372124                 if (leap_year (tms.tm_year))
372125                 {
372126                     if (tms.tm_mday >= 29)
372127                     {
372128                         tms.tm_mday -= 29;
372129                     }
372130                     else
372131                     {
372132                         loop = 0;
372133                     }
372134                 }
372135                 else
372136                 {
372137                     if (tms.tm_mday >= 28)
372138                     {
372139                         tms.tm_mday -= 28;
372140                     }
372141                     else
372142                     {
372143                         loop = 0;
372144                     }
372145                 }
372146                 break;
372147             }
372148         }
372149     }
372150     // No check for day light saving time.
372151     //
372152     tms.tm_isdst = 0;
372153     //
372154     // Return.
372155     //
372156     return (&tms);
372157 }
372158 //-----
372159 static int
372160 leap_year (int year)
372161 {
372162     if ((year % 4) == 0)
372163     {
372164         if ((year % 100) == 0)
372165         {
372166             if ((year % 400) == 0)
372167             {
372168                 return (1);

```

1829

```

3720169     }
3720170     else
3720171     {
3720172         return (0);
3720173     }
3720174     }
3720175     else
3720176     {
3720177         return (1);
3720178     }
3720179     }
3720180     else
3720181     {
3720182         return (0);
3720183     }
3720184 }

```

lib/time/mktime.c

« Si veda la sezione u0.13.

```

3730001 #include <time.h>
3730002 #include <string.h>
3730003 #include <stdio.h>
3730004 //-----
3730005 static int leap_year (int year);
3730006 //-----
3730007 time_t
3730008 mktime (const struct tm *timeptr)
3730009 {
3730010     time_t timer_total;
3730011     time_t timer_aux;
3730012     int days;
3730013     int month;
3730014     int year;
3730015     //
3730016     // From seconds to days.
3730017     //
3730018     timer_total = timeptr->tm_sec;
3730019     //
3730020     timer_aux = timeptr->tm_min;
3730021     timer_aux *= 60;
3730022     timer_total += timer_aux;
3730023     //
3730024     timer_aux = timeptr->tm_hour;
3730025     timer_aux *= (60 * 60);
3730026     timer_total += timer_aux;
3730027     //
3730028     timer_aux = timeptr->tm_mday;
3730029     timer_aux *= 24;
3730030     timer_aux *= (60 * 60);
3730031     timer_total += timer_aux;
3730032     //
3730033     // Month: add the days of months.
3730034     // Will scan the months, from the first, but before the
3730035     // months of the value inside field 'tm_mon'.
3730036     //
3730037     for (month = 1, days = 0; month < timeptr->tm_mon; month++)
3730038     {
3730039         switch (month)
3730040         {
3730041             case 1:
3730042             case 3:
3730043             case 5:
3730044             case 7:
3730045             case 8:
3730046                 //
3730047                 // There is no December, because the scan can go up to
3730048                 // the month before the value inside field 'tm_mon'.
3730049                 //
3730050                 days += 31;
3730051                 break;
3730052             case 4:
3730053             case 6:
3730054             case 9:
3730055             case 11:
3730056                 days += 30;
3730057                 break;
3730058             case 2:
3730059                 if (leap_year (timeptr->tm_year))
3730060                 {
3730061                     days += 29;
3730062                 }
3730063                 else
3730064                 {
3730065                     days += 28;
3730066                 }
3730067                 break;
3730068         }
3730069     }
3730070     //
3730071     //
3730072     timer_aux = days;
3730073     timer_aux *= 24;
3730074     timer_aux *= (60 * 60);
3730075     timer_total += timer_aux;
3730076     //
3730077     // Year. The work is similar to the one of months: days of
3730078     // years are counted, up to the year before the one reported
3730079     // by the field 'tm_year'.
3730080     //

```

1830

```

3730081     for (year = 1970, days = 0; year < timeptr->tm_year; year++)
3730082     {
3730083         if (leap_year (year))
3730084         {
3730085             days += 366;
3730086         }
3730087         else
3730088         {
3730089             days += 365;
3730090         }
3730091     }
3730092     //
3730093     // After all, must subtract a day from the total.
3730094     //
3730095     days--;
3730096     //
3730097     timer_aux = days;
3730098     timer_aux *= 24;
3730099     timer_aux *= (60 * 60);
3730100     timer_total += timer_aux;
3730101     //
3730102     // That's all.
3730103     //
3730104     return (timer_total);
3730105 }
3730106 //-----
3730107 int
3730108 leap_year (int year)
3730109 {
3730110     if ((year % 4) == 0)
3730111     {
3730112         if ((year % 100) == 0)
3730113         {
3730114             if ((year % 400) == 0)
3730115             {
3730116                 return (1);
3730117             }
3730118             else
3730119             {
3730120                 return (0);
3730121             }
3730122         }
3730123         else
3730124         {
3730125             return (1);
3730126         }
3730127     }
3730128     else
3730129     {
3730130         return (0);
3730131     }
3730132 }

```

lib/time/stime.c

« Si veda la sezione u0.39.

```

3740001 #include <time.h>
3740002 #include <sys/os16.h>
3740003 //-----
3740004 int
3740005 stime (time_t *timer)
3740006 {
3740007     sysmsg_time_t msg;
3740008     msg.timer = *timer;
3740009     msg.ret = 0;
3740010     sys (SYS_TIME, &msg, (sizeof msg));
3740011     return (msg.ret);
3740012 }

```

lib/time/time.c

« Si veda la sezione u0.39.

```

3750001 #include <time.h>
3750002 #include <sys/os16.h>
3750003 //-----
3750004 time_t
3750005 time (time_t *timer)
3750006 {
3750007     sysmsg_time_t msg;
3750008     msg.ret = ((time_t) 0);
3750009     sys (SYS_TIME, &msg, (sizeof msg));
3750010     if (timer != NULL)
3750011     {
3750012         *timer = msg.ret;
3750013     }
3750014     return (msg.ret);
3750015 }

```

os16: «lib/unistd.h»

« Si veda la sezione u0.2.

```

3760001 #ifndef _UNISTD_H
3760002 #define _UNISTD_H    1
3760003
3760004 #include <const.h>

```

1831

```

376005 #include <sys/stat.h>
376006 #include <sys/osl6.h>
376007 #include <sys/types.h> // size_t, ssize_t, uid_t, gid_t, off_t, pid_t
376008 #include <inttypes.h> // intptr_t
376009 #include <SEEK.h> // SEEK_CUR, SEEK_SET, SEEK_END
376010 //-----
376011 extern char **environ; // Variable 'environ' is used by functions like
376012 // 'execv()' in replacement for 'envp[1]'.
376013 //-----
376014 extern char *optarg; // Used by 'optarg()'.
376015 extern int optind; //
376016 extern int opterr; //
376017 extern int optopt; //
376018 //-----
376019 #define STDIN_FILENO 0 //
376020 #define STDOUT_FILENO 1 // Standard file descriptors.
376021 #define STDERR_FILENO 2 //
376022 //-----
376023 #define R_OK 4 // Read permission.
376024 #define W_OK 2 // Write permission.
376025 #define X_OK 1 // Execute or traverse permission.
376026 #define F_OK 0 // File exists.
376027 //-----
376028
376029 int access (const char *path, int mode);
376030 int chdir (const char *path);
376031 int chown (const char *path, uid_t uid, gid_t gid);
376032 int close (int fdn);
376033 int dup (int fdn_old);
376034 int dup2 (int fdn_old, int fdn_new);
376035 int execl (const char *path, const char *arg, ...);
376036 int execlp (const char *path, const char *arg, ...);
376037 int execlp (const char *path, const char *arg, ...);
376038 int execv (const char *path, char *const argv[]);
376039 int execve (const char *path, char *const argv[],
376040 char *const envp[]);
376041 int execvp (const char *path, char *const argv[]);
376042 void _exit (int status);
376043 int fchown (int fdn, uid_t uid, gid_t gid);
376044 pid_t fork (void);
376045 char *getcwd (char *buffer, size_t size);
376046 uid_t geteuid (void);
376047 int getopt (int argc, char *const argv[],
376048 const char *optstring);
376049 pid_t getpgpr (void);
376050 pid_t getppid (void);
376051 pid_t getpid (void);
376052 uid_t getuid (void);
376053 int isatty (int fdn);
376054 int link (const char *path_old, const char *path_new);
376055 off_t lseek (int fdn, off_t offset, int whence);
376056 #define nice(n) (0)
376057 ssize_t read (int fdn, void *buffer, size_t count);
376058 #define readlink(p,b,s) ((ssize_t) -1)
376059 int rmdir (const char *path);
376060 int seteuid (uid_t uid);
376061 int setpgpr (void);
376062 int setuid (uid_t uid);
376063 unsigned int sleep (unsigned int s);
376064 #define sync() /* */
376065 char *tname (int fdn);
376066 int unlink (const char *path);
376067 ssize_t write (int fdn, const void *buffer, size_t count);
376068
376069 #endif

```

lib/unistd/_exit.c

Si veda la sezione u0.2.

```

377001 #include <unistd.h>
377002 #include <sys/osl6.h>
377003 //-----
377004 void
377005 _exit (int status)
377006 {
377007     sysmsg_exit_t msg;
377008     //
377009     // Only the low eight bit are returned.
377010     //
377011     msg.status = (status & 0xFF);
377012     //
377013     //
377014     //
377015     sys (SYS_EXIT, &msg, (sizeof msg));
377016     //
377017     // Should not return from system call, but if it does, loop
377018     // forever:
377019     //
377020     while (1);
377021 }

```

lib/unistd/access.c

Si veda la sezione u0.1.

```

378001 #include <unistd.h>
378002 #include <sys/stat.h>
378003 #include <errno.h>
378004 //-----

```

1832

```

378005 int
378006 access (const char *path, int mode)
378007 {
378008     struct stat st;
378009     int status;
378010     uid_t euid;
378011     //
378012     status = stat (path, &st);
378013     if (status != 0)
378014     {
378015         return (-1);
378016     }
378017     //
378018     // File exists?
378019     //
378020     if (mode == F_OK)
378021     {
378022         return (0);
378023     }
378024     //
378025     // Some access permissions are requested: get effective user id.
378026     //
378027     euid = geteuid ();
378028     //
378029     // Check owner access permissions.
378030     //
378031     if (st.st_uid == euid && ((st.st_mode & S_IRWXU) == (mode << 6)))
378032     {
378033         return (0);
378034     }
378035     //
378036     // Check others access permissions.
378037     //
378038     if ((st.st_mode & S_IRWXO) == (mode))
378039     {
378040         return (0);
378041     }
378042     //
378043     // Otherwise there are no access permissions.
378044     //
378045     errset (EACCES); // Permission denied.
378046     return (-1);
378047 }

```

lib/unistd/chdir.c

Si veda la sezione u0.3.

```

379001 #include <unistd.h>
379002 #include <string.h>
379003 #include <const.h>
379004 #include <sys/osl6.h>
379005 #include <errno.h>
379006 #include <limits.h>
379007 //-----
379008 int
379009 chdir (const char *path)
379010 {
379011     sysmsg_chdir_t msg;
379012     //
379013     msg.ret = 0;
379014     msg.errno = 0;
379015     //
379016     strncpy (msg.path, path, PATH_MAX);
379017     //
379018     sys (SYS_CHDIR, &msg, (sizeof msg));
379019     //
379020     errno = msg.errno;
379021     errln = msg.errln;
379022     strncpy (errfn, msg.errfn, PATH_MAX);
379023     return (msg.ret);
379024 }

```

lib/unistd/chown.c

Si veda la sezione u0.5.

```

380001 #include <unistd.h>
380002 #include <string.h>
380003 #include <const.h>
380004 #include <sys/osl6.h>
380005 #include <errno.h>
380006 #include <limits.h>
380007 //-----
380008 int
380009 chown (const char *path, uid_t uid, gid_t gid)
380010 {
380011     sysmsg_chown_t msg;
380012     //
380013     strncpy (msg.path, path, PATH_MAX);
380014     msg.uid = uid;
380015     msg.gid = gid;
380016     //
380017     sys (SYS_CHOWN, &msg, (sizeof msg));
380018     //
380019     errno = msg.errno;
380020     errln = msg.errln;
380021     strncpy (errfn, msg.errfn, PATH_MAX);
380022     return (msg.ret);
380023 }

```

1833

lib/unistd/close.c

<<

Si veda la sezione [u0.7](#).

```
3830001 #include <unistd.h>
3830002 #include <errno.h>
3830003 #include <sys/osl6.h>
3830004 #include <string.h>
3830005 //-----
3830006 int
3830007 close (int fdn)
3830008 {
3830009     sysmsg_close_t msg;
3830010     msg.fdn = fdn;
3830011     sys (SYS_CLOSE, &msg, (sizeof msg));
3830012     errno = msg.errno;
3830013     errln = msg.errln;
3830014     strncpy (errfn, msg.errfn, PATH_MAX);
3830015     return (msg.ret);
3830016 }
```

lib/unistd/dup.c

<<

Si veda la sezione [u0.8](#).

```
3820001 #include <unistd.h>
3820002 #include <sys/osl6.h>
3820003 #include <string.h>
3820004 #include <errno.h>
3820005 //-----
3820006 int
3820007 dup (int fdn_old)
3820008 {
3820009     sysmsg_dup_t msg;
3820010     //
3820011     msg.fdn_old = fdn_old;
3820012     //
3820013     sys (SYS_DUP, &msg, (sizeof msg));
3820014     //
3820015     errno = msg.errno;
3820016     errln = msg.errln;
3820017     strncpy (errfn, msg.errfn, PATH_MAX);
3820018     return (msg.ret);
3820019 }
```

lib/unistd/dup2.c

<<

Si veda la sezione [u0.8](#).

```
3830001 #include <unistd.h>
3830002 #include <sys/osl6.h>
3830003 #include <string.h>
3830004 #include <errno.h>
3830005 //-----
3830006 int
3830007 dup2 (int fdn_old, int fdn_new)
3830008 {
3830009     sysmsg_dup2_t msg;
3830010     //
3830011     msg.fdn_old = fdn_old;
3830012     msg.fdn_new = fdn_new;
3830013     //
3830014     sys (SYS_DUP2, &msg, (sizeof msg));
3830015     //
3830016     errno = msg.errno;
3830017     errln = msg.errln;
3830018     strncpy (errfn, msg.errfn, PATH_MAX);
3830019     return (msg.ret);
3830020 }
```

lib/unistd/envron.c

<<

Si veda la sezione [u0.1](#).

```
3840001 #include <unistd.h>
3840002 //-----
3840003 char **environ;
```

lib/unistd/execl.c

<<

Si veda la sezione [u0.20](#).

```
3850001 #include <unistd.h>
3850002 //-----
3850003 int
3850004 execl (const char *path, const char *arg, ...)
3850005 {
3850006     int argc;
3850007     char *arg_next;
3850008     char *argv[ARG_MAX/2];
3850009     //
3850010     va_list ap;
3850011     va_start (ap, arg);
3850012     //
3850013     arg_next = arg;
3850014     //
3850015     for (argc = 0; argc < ARG_MAX/2; argc++)
3850016     {
```

```
3850017     argv[argc] = arg_next;
3850018     if (argv[argc] == NULL)
3850019     {
3850020         break; // End of arguments.
3850021     }
3850022     arg_next = va_arg (ap, char *);
3850023     //
3850024     //
3850025     return (execve (path, argv, environ)); // [1]
3850026 }
3850027 //
3850028 // The variable 'environ' is declared as 'char **environ' and is
3850029 // included from <unistd.h>.
3850030 //
```

lib/unistd/execl.c

Si veda la sezione [u0.20](#).

<<

```
3860001 #include <unistd.h>
3860002 //-----
3860003 int
3860004 execl (const char *path, const char *arg, ...)
3860005 {
3860006     int argc;
3860007     char *arg_next;
3860008     char *argv[ARG_MAX/2];
3860009     char **envp;
3860010     //
3860011     va_list ap;
3860012     va_start (ap, arg);
3860013     //
3860014     arg_next = arg;
3860015     //
3860016     for (argc = 0; argc < ARG_MAX/2; argc++)
3860017     {
3860018         argv[argc] = arg_next;
3860019         if (argv[argc] == NULL)
3860020         {
3860021             break; // End of arguments.
3860022         }
3860023         arg_next = va_arg (ap, char *);
3860024     }
3860025     //
3860026     envp = va_arg (ap, char **);
3860027     //
3860028     return (execve (path, argv, envp));
3860029 }
```

lib/unistd/execlp.c

Si veda la sezione [u0.20](#).

<<

```
3870001 #include <unistd.h>
3870002 #include <string.h>
3870003 #include <stdlib.h>
3870004 #include <errno.h>
3870005 #include <sys/osl6.h>
3870006 //-----
3870007 int
3870008 execlp (const char *path, const char *arg, ...)
3870009 {
3870010     int argc;
3870011     char *arg_next;
3870012     char *argv[ARG_MAX/2];
3870013     char command[PATH_MAX];
3870014     int status;
3870015     //
3870016     va_list ap;
3870017     va_start (ap, arg);
3870018     //
3870019     arg_next = arg;
3870020     //
3870021     for (argc = 0; argc < ARG_MAX/2; argc++)
3870022     {
3870023         argv[argc] = arg_next;
3870024         if (argv[argc] == NULL)
3870025         {
3870026             break; // End of arguments.
3870027         }
3870028         arg_next = va_arg (ap, char *);
3870029     }
3870030     //
3870031     // Get a full command path if necessary.
3870032     //
3870033     status = namep (path, command, (size_t) PATH_MAX);
3870034     if (status != 0)
3870035     {
3870036         //
3870037         // Variable 'errno' is already set by 'commandp()'.
3870038         //
3870039         return (-1);
3870040     }
3870041     //
3870042     // Return calling 'execve()'
3870043     //
3870044     return (execve (command, argv, environ)); // [1]
3870045 }
3870046 //
3870047 // The variable 'environ' is declared as 'char **environ' and is
```



```

387048 // included from <unistd.h>.
387049 //

```

lib/unistd/execv.c

« Si veda la sezione u0.20.

```

388001 #include <unistd.h>
388002 //-----
388003 int
388004 execv (const char *path, char *const argv[])
388005 {
388006     return (execve (path, argv, environ)); // [1]
388007 }
388008 //
388009 // The variable 'environ' is declared as 'char **environ' and is
388010 // included from <unistd.h>.
388011 //

```

lib/unistd/execve.c

« Si veda la sezione u0.10.

```

389001 #include <unistd.h>
389002 #include <sys/types.h>
389003 #include <sys/osi6.h>
389004 #include <errno.h>
389005 #include <string.h>
389006 #include <string.h>
389007 //-----
389008 int
389009 execve (const char *path, char *const argv[], char *const envp[])
389010 {
389011     sysmsg_exec_t msg;
389012     size_t
389013     size_t arg_size;
389014     int
389015     size_t env_size;
389016     int
389017     char *arg_data = msg.arg_data;
389018     char *env_data = msg.env_data;
389019     //
389020     msg.ret = 0;
389021     msg.errno = 0;
389022     //
389023     strncpy (msg.path, path, PATH_MAX);
389024     //
389025     // Copy 'argv[]' inside a the message buffer 'msg.arg_data',
389026     // separating each string with a null character and counting the
389027     // number of strings inside 'argc'.
389028     //
389029     for (argc = 0, arg_size = 0, size = 0;
389030          argv != NULL &&
389031          argc < (ARG_MAX/16) &&
389032          arg_size < ARG_MAX/2 &&
389033          argv[argc] != NULL;
389034          argc++, arg_size += size)
389035     {
389036         size = strlen (argv[argc]);
389037         size++; // Count also the final null character.
389038         if (size > (ARG_MAX/2 - arg_size))
389039             {
389040                 errset (E2BIG); // Argument list too long.
389041                 return (-1);
389042             }
389043         strncpy (arg_data, argv[argc], size);
389044         arg_data += size;
389045     }
389046     msg.argc = argc;
389047     //
389048     // Copy 'envp[]' inside a the message buffer 'msg.env_data',
389049     // separating each string with a null character and counting the
389050     // number of strings inside 'envc'.
389051     //
389052     for (envc = 0, env_size = 0, size = 0;
389053          envp != NULL &&
389054          envc < (ARG_MAX/16) &&
389055          env_size < ARG_MAX/2 &&
389056          envp[envc] != NULL;
389057          envc++, env_size += size)
389058     {
389059         size = strlen (envp[envc]);
389060         size++; // Count also the final null character.
389061         if (size > (ARG_MAX/2 - env_size))
389062             {
389063                 errset (E2BIG); // Argument list too long.
389064                 return (-1);
389065             }
389066         strncpy (env_data, envp[envc], size);
389067         env_data += size;
389068     }
389069     msg.envc = envc;
389070     //
389071     // System call.
389072     //
389073     sys (SYS_EXEC, &msg, (sizeof msg));
389074     //
389075     // Should not return, but if it does, then there is an error.
389076     //
389077     errno = msg.errno;

```

1836

```

389078     errln = msg.errln;
389079     strncpy (errfn, msg.errfn, PATH_MAX);
389080     return (msg.ret);
389081 }

```

lib/unistd/execvp.c

« Si veda la sezione u0.20.

```

390001 #include <unistd.h>
390002 #include <string.h>
390003 #include <stdlib.h>
390004 #include <errno.h>
390005 #include <sys/osi6.h>
390006 //-----
390007 int
390008 execvp (const char *path, char *const argv[])
390009 {
390010     char command[PATH_MAX];
390011     int status;
390012     //
390013     // Get a full command path if necessary.
390014     //
390015     status = namep (path, command, (size_t) PATH_MAX);
390016     if (status != 0)
390017         {
390018             //
390019             // Variable 'errno' is already set by 'namep()'.
390020             //
390021             return (-1);
390022         }
390023     //
390024     // Return calling 'execve()'
390025     //
390026     return (execve (command, argv, environ)); // [1]
390027 }
390028 //
390029 // The variable 'environ' is declared as 'char **environ' and is
390030 // included from <unistd.h>.
390031 //

```

lib/unistd/fchdir.c

« Si veda la sezione u0.2.

```

391001 #include <unistd.h>
391002 #include <errno.h>
391003 //-----
391004 int
391005 fchdir (int fdn)
391006 {
391007     //
391008     // osl6 requires to keep track of the path for the current working
391009     // directory. The standard function 'fchdir()' is not applicable.
391010     //
391011     errset (E_NOT_IMPLEMENTED);
391012     return (-1);
391013 }

```

lib/unistd/fchown.c

« Si veda la sezione u0.5.

```

392001 #include <unistd.h>
392002 #include <string.h>
392003 #include <const.h>
392004 #include <sys/osi6.h>
392005 #include <errno.h>
392006 #include <limits.h>
392007 //-----
392008 int
392009 fchown (int fdn, uid_t uid, gid_t gid)
392010 {
392011     sysmsg_fchown_t msg;
392012     //
392013     msg.fdn = fdn;
392014     msg.uid = uid;
392015     msg.gid = gid;
392016     //
392017     sys (SYS_FCHOWN, &msg, (sizeof msg));
392018     //
392019     errno = msg.errno;
392020     errln = msg.errln;
392021     strncpy (errfn, msg.errfn, PATH_MAX);
392022     return (msg.ret);
392023 }

```

lib/unistd/fork.c

« Si veda la sezione u0.14.

```

393001 #include <unistd.h>
393002 #include <sys/types.h>
393003 #include <sys/osi6.h>
393004 #include <errno.h>
393005 #include <string.h>
393006 //-----

```

1837

```

3930007 pid_t
3930008 fork (void)
3930009 {
3930010     sysmsg_fork_t msg;
3930011     //
3930012     // Set the return value for the child process.
3930013     //
3930014     msg.ret = 0;
3930015     //
3930016     // Do the system call.
3930017     //
3930018     sys (SYS_FORK, &msg, (sizeof msg));
3930019     //
3930020     // If the system call has successfully generated a copy of
3930021     // the original process, the following code is executed from
3930022     // the parent and the child. But the child has the 'msg'
3930023     // structure untouched, while the parent has, at least, the
3930024     // pid number inside 'msg.ret'.
3930025     // If the system call fails, there is no child, and the
3930026     // parent finds the return value equal to -1, with an
3930027     // error number.
3930028     //
3930029     errno = msg.errno;
3930030     errln = msg.errln;
3930031     strncpy (errfn, msg.errfn, PATH_MAX);
3930032     return (msg.ret);
3930033 }

```

lib/unistd/getcwd.c

Si veda la sezione u0.16.

```

3940001 #include <unistd.h>
3940002 #include <sys/types.h>
3940003 #include <sys/osl6.h>
3940004 #include <errno.h>
3940005 #include <stddef.h>
3940006 #include <string.h>
3940007 //-----
3940008 char *
3940009 getcwd (char *buffer, size_t size)
3940010 {
3940011     sysmsg_uarea_t msg;
3940012     //
3940013     // Check arguments: the buffer must be given.
3940014     //
3940015     if (buffer == NULL)
3940016     {
3940017         errset (EINVAL);
3940018         return (NULL);
3940019     }
3940020     //
3940021     // Make shure that the last character, inside the working directory
3940022     // path is a null character.
3940023     //
3940024     msg.path_cwd[PATH_MAX-1] = 0;
3940025     //
3940026     // Just get the user area data.
3940027     //
3940028     sys (SYS_UAREA, &msg, (sizeof msg));
3940029     //
3940030     // Check that the path is still correctly terminated. If isn't,
3940031     // the path is longer than the implementation limits, and it is
3940032     // really *bad*.
3940033     //
3940034     if (msg.path_cwd[PATH_MAX-1] != 0)
3940035     {
3940036         errset (E_LIMIT); // Exceeded implementation limits.
3940037         return (NULL);
3940038     }
3940039     //
3940040     // If the path is larger than the buffer size, return an error.
3940041     // Please note that the parameter 'size' must include the
3940042     // terminating null character, so, if the string is equal to
3940043     // the size, it is already beyond the size limit.
3940044     //
3940045     if (strlen (msg.path_cwd) >= size)
3940046     {
3940047         errset (ERANGE); // Result too large.
3940048         return (NULL);
3940049     }
3940050     //
3940051     // Everything is fine, so, copy the path to the buffer and return.
3940052     //
3940053     strncpy (buffer, msg.path_cwd, size);
3940054     return (buffer);
3940055 }

```

lib/unistd/geteuid.c

Si veda la sezione u0.18.

```

3950001 #include <unistd.h>
3950002 #include <sys/types.h>
3950003 #include <sys/osl6.h>
3950004 #include <errno.h>
3950005 //-----
3950006 uid_t
3950007 geteuid (void)
3950008 {

```

1838

```

3950009 sysmsg_uarea_t msg;
3950010 sys (SYS_UAREA, &msg, (sizeof msg));
3950011 return (msg.euid);
3950012 }

```

lib/unistd/getopt.c

Si veda la sezione u0.52.

```

3960001 #include <unistd.h>
3960002 #include <sys/types.h>
3960003 #include <sys/osl6.h>
3960004 #include <errno.h>
3960005 //-----
3960006 char *optarg;
3960007 int optind = 1;
3960008 int opterr = 1;
3960009 int optopt = 0;
3960010 //-----
3960011 static void getopt_no_argument (int opt);
3960012 //-----
3960013 int
3960014 getopt (int argc, char *const argv[], const char *optstring)
3960015 {
3960016     static int o = 0; // Index to scan grouped options.
3960017     int s; // Index to scan 'optstring'
3960018     int opt; // Current option letter.
3960019     int flag_argument; // If there should be an argument.
3960020     //
3960021     // Entering the function, 'flag_argument' is zero. Just to make
3960022     // it clear:
3960023     //
3960024     flag_argument = 0;
3960025     //
3960026     // Scan 'argv[]' elements, starting form the value that 'optind'
3960027     // already have.
3960028     //
3960029     for (; optind < argc; optind++)
3960030     {
3960031         //
3960032         // If an option is expected, some check must be done at
3960033         // the beginning.
3960034         //
3960035         if (!flag_argument)
3960036         {
3960037             //
3960038             // Check if the scan is finished and 'optind' should be kept
3960039             // untouched:
3960040             // 'argv[optind]' is a null pointer;
3960041             // 'argv[optind][0]' is not the character '-';
3960042             // 'argv[optind]' points to the string "--";
3960043             // all 'argv[]' elements are parsed.
3960044             //
3960045             if (argv[optind] == NULL
3960046                 || argv[optind][0] != '-'
3960047                 || argv[optind][1] == 0
3960048                 || optind >= argc)
3960049             {
3960050                 return (-1);
3960051             }
3960052             //
3960053             // Check if the scan is finished and 'optind' is to be
3960054             // incremented:
3960055             // 'argv[optind]' points to the string "--".
3960056             //
3960057             if (argv[optind][0] == '-'
3960058                 && argv[optind][1] == '-'
3960059                 && argv[optind][2] == 0)
3960060             {
3960061                 optind++;
3960062                 return (-1);
3960063             }
3960064             //
3960065             // Scan 'argv[optind]' using the static index 'o'.
3960066             //
3960067             for (; o < strlen (argv[optind]); o++)
3960068             {
3960069                 //
3960070                 // If there should be an option, index 'o' should
3960071                 // start from 1, because 'argv[optind][0]' must
3960072                 // be equal to '-'.
3960073                 //
3960074                 if (!flag_argument && (o == 0))
3960075                 {
3960076                     //
3960077                     // As there is no options, 'o' cannot start
3960078                     // from zero, so a new loop is done.
3960079                     //
3960080                     continue;
3960081                 }
3960082                 //
3960083                 if (flag_argument)
3960084                 {
3960085                     //
3960086                     // There should be an argument, starting from
3960087                     // 'argv[optind][o]'.
3960088                     //
3960089                     if ((o == 0) && (argv[optind][o] == '-'))
3960090                     {
3960091                         //

```

1839

```

3960091 // 'argv[optind][0]' is equal to '-', but there
3960092 // should be an argument instead: the argument
3960093 // is missing.
3960094 //
3960095 optarg = NULL;
3960096 //
3960097 if (optstring[0] == ':')
3960098 {
3960099 //
3960100 // As the option string starts with ':' the
3960101 // function must return ':'.
3960102 //
3960103 optopt = opt;
3960104 opt = ':';
3960105 }
3960106 else
3960107 {
3960108 //
3960109 // As the option string does not start with ':'
3960110 // the function must return '?'.
3960111 //
3960112 getopt_no_argument (opt);
3960113 optopt = opt;
3960114 opt = '?';
3960115 }
3960116 //
3960117 // 'optind' is left untouched.
3960118 //
3960119 }
3960120 }
3960121 else
3960122 {
3960123 //
3960124 // The argument is found: 'optind' is to be
3960125 // incremented and 'o' is reset.
3960126 //
3960127 optarg = &argv[optind][o];
3960128 optind++;
3960129 o = 0;
3960130 }
3960131 //
3960132 // Return the option, or ':', or '?'.
3960133 //
3960134 return (opt);
3960135 }
3960136 }
3960137 else
3960138 {
3960139 //
3960140 // It should be an option: 'optstring[]' must be
3960141 // scanned.
3960142 //
3960143 opt = argv[optind][o];
3960144 //
3960145 for (s = 0, optopt = 0; s < strlen (optstring); s++)
3960146 {
3960147 //
3960148 // If 'optstring[0]' is equal to ':', index 's' must
3960149 // start at 1.
3960150 //
3960151 if ((s == 0) && (optstring[0] == ':'))
3960152 {
3960153 continue;
3960154 }
3960155 //
3960156 if (opt == optstring[s])
3960157 {
3960158 //
3960159 if (optstring[s+1] == ':')
3960160 {
3960161 //
3960162 // There is an argument.
3960163 //
3960164 flag_argument = 1;
3960165 break;
3960166 }
3960167 else
3960168 {
3960169 //
3960170 // There is no argument.
3960171 //
3960172 o++;
3960173 return (opt);
3960174 }
3960175 }
3960176 }
3960177 //
3960178 if (s >= strlen (optstring))
3960179 {
3960180 //
3960181 // The 'optstring' scan is concluded with no
3960182 // match.
3960183 //
3960184 o++;
3960185 optopt = opt;
3960186 return ('?');
3960187 }
3960188 //
3960189 // Otherwise the loop was broken.
3960190 //
3960191 }
3960192 }
3960193 //

```

1840

```

3960194 // Check index 'o'.
3960195 //
3960196 if (o >= strlen (argv[optind]))
3960197 {
3960198 //
3960199 // There are no more options or there is no argument
3960200 // inside current 'argv[optind]' string. Index 'o' is
3960201 // reset before the next loop.
3960202 //
3960203 o = 0;
3960204 }
3960205 }
3960206 //
3960207 // No more elements inside 'argv' or loop broken: there might be a
3960208 // missing argument.
3960209 //
3960210 if (flag_argument)
3960211 {
3960212 //
3960213 // Missing option argument.
3960214 //
3960215 optarg = NULL;
3960216 //
3960217 if (optstring[0] == ':')
3960218 {
3960219 return (':');
3960220 }
3960221 else
3960222 {
3960223 getopt_no_argument (opt);
3960224 return ('?');
3960225 }
3960226 }
3960227 //
3960228 return (-1);
3960229 }
3960230
3960231 static void
3960232 getopt_no_argument (int opt)
3960233 {
3960234 if (opterr)
3960235 {
3960236 fprintf (stderr, "Missing argument for option '-%c'\n", opt);
3960237 }
3960238 }

```

lib/unistd/getpgrp.c

Si veda la sezione u0.20.

```

3970001 #include <unistd.h>
3970002 #include <sys/types.h>
3970003 #include <sys/unistd.h>
3970004 #include <errno.h>
3970005 //-----
3970006 pid_t
3970007 getpgrp (void)
3970008 {
3970009     sysmsg_uarea_t msg;
3970010     sys (SYS_UAREA, &msg, (sizeof msg));
3970011     return (msg.pgrp);
3970012 }

```

«

lib/unistd/getpid.c

Si veda la sezione u0.20.

```

3980001 #include <unistd.h>
3980002 #include <sys/types.h>
3980003 #include <sys/unistd.h>
3980004 #include <errno.h>
3980005 //-----
3980006 pid_t
3980007 getpid (void)
3980008 {
3980009     sysmsg_uarea_t msg;
3980010     sys (SYS_UAREA, &msg, (sizeof msg));
3980011     return (msg.pid);
3980012 }

```

«

lib/unistd/getppid.c

Si veda la sezione u0.20.

```

3990001 #include <unistd.h>
3990002 #include <sys/types.h>
3990003 #include <sys/unistd.h>
3990004 #include <errno.h>
3990005 //-----
3990006 pid_t
3990007 getppid (void)
3990008 {
3990009     sysmsg_uarea_t msg;
3990010     sys (SYS_UAREA, &msg, (sizeof msg));
3990011     return (msg.ppid);
3990012 }

```

«

1841

lib/unistd/getuid.c

«

Si veda la sezione [u0.18](#).

```
400001 #include <unistd.h>
400002 #include <sys/types.h>
400003 #include <sys/unistd.h>
400004 #include <errno.h>
400005 //-----
400006 uid_t
400007 getuid (void)
400008 {
400009     struct sysmsg_uarea_t msg;
400010     sys (SYS_UAREA, &msg, (sizeof msg));
400011     return (msg.uid);
400012 }
```

```
400004 #include <errno.h>
400005 #include <string.h>
400006 //-----
400007 off_t
400008 lseek (int fdn, off_t offset, int whence)
400009 {
400010     struct sysmsg_lseek_t msg;
400011     msg.fdn = fdn;
400012     msg.offset = offset;
400013     msg.whence = whence;
400014     sys (SYS_LSEEK, &msg, (sizeof msg));
400015     errno = msg.errno;
400016     errln = msg.errln;
400017     strncpy (errfn, msg.errfn, PATH_MAX);
400018     return (msg.ret);
400019 }
```

lib/unistd/isatty.c

«

Si veda la sezione [u0.61](#).

```
400001 #include <sys/stat.h>
400002 #include <sys/unistd.h>
400003 #include <sys/types.h>
400004 #include <errno.h>
400005 //-----
400006 int
400007 isatty (int fdn)
400008 {
400009     struct stat file_status;
400010     //
400011     // Verify to have valid input data.
400012     //
400013     if (fdn < 0)
400014     {
400015         errset (EBADF);
400016         return (0);
400017     }
400018     //
400019     // Verify the standard input.
400020     //
400021     if (fstat (fdn, &file_status) == 0)
400022     {
400023         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
400024         {
400025             return (1); // Meaning it is ok!
400026         }
400027         if (major (file_status.st_rdev) == DEV_TTY_MAJOR)
400028         {
400029             return (1); // Meaning it is ok!
400030         }
400031     }
400032     else
400033     {
400034         errset (errno);
400035         return (0);
400036     }
400037     //
400038     // If here, it is not a terminal of any kind.
400039     //
400040     errset (EINVAL);
400041     return (0);
400042 }
```

lib/unistd/link.c

«

Si veda la sezione [u0.23](#).

```
402001 #include <unistd.h>
402002 #include <string.h>
402003 #include <const.h>
402004 #include <sys/unistd.h>
402005 #include <errno.h>
402006 #include <limits.h>
402007 //-----
402008 int
402009 link (const char *path_old, const char *path_new)
402010 {
402011     struct sysmsg_link_t msg;
402012     //
402013     strncpy (msg.path_old, path_old, PATH_MAX);
402014     strncpy (msg.path_new, path_new, PATH_MAX);
402015     //
402016     sys (SYS_LINK, &msg, (sizeof msg));
402017     //
402018     errno = msg.errno;
402019     errln = msg.errln;
402020     strncpy (errfn, msg.errfn, PATH_MAX);
402021     return (msg.ret);
402022 }
```

lib/unistd/read.c

«

Si veda la sezione [u0.29](#).

```
404001 #include <unistd.h>
404002 #include <sys/unistd.h>
404003 #include <errno.h>
404004 #include <string.h>
404005 #include <stdio.h>
404006 //-----
404007 ssize_t
404008 read (int fdn, void *buffer, size_t count)
404009 {
404010     struct sysmsg_read_t msg;
404011     //
404012     // Reduce size of read if necessary.
404013     //
404014     if (count > BUFSIZ)
404015     {
404016         count = BUFSIZ;
404017     }
404018     //
404019     // Fill the message.
404020     //
404021     msg.fdn = fdn;
404022     msg.count = count;
404023     msg.eof = 0;
404024     msg.ret = 0;
404025     //
404026     // Repeat syscall, until something is received or end of file is
404027     // reached.
404028     //
404029     while (1)
404030     {
404031         sys (SYS_READ, &msg, (sizeof msg));
404032         if (msg.ret != 0 || msg.eof)
404033         {
404034             break;
404035         }
404036     }
404037     //
404038     // Before return: be careful with the 'msg.buffer' copy, because
404039     // it cannot be longer than 'count', otherwise, some unexpected
404040     // memory will be overwritten!
404041     //
404042     if (msg.ret < 0)
404043     {
404044         //
404045         // No valid read, no change inside the buffer.
404046         //
404047         errno = msg.errno;
404048         return (msg.ret);
404049     }
404050     //
404051     if (msg.ret > count)
404052     {
404053         //
404054         // A strange value was returned. Considering it a read error.
404055         //
404056         errset (EIO); // I/O error.
404057         return (-1);
404058     }
404059     //
404060     // A valid read: fill the buffer with 'msg.ret' bytes.
404061     //
404062     memcpy (buffer, msg.buffer, msg.ret);
404063     //
404064     // Return.
404065     //
404066     return (msg.ret);
404067 }
```

lib/unistd/lseek.c

«

Si veda la sezione [u0.24](#).

```
403001 #include <unistd.h>
403002 #include <sys/types.h>
403003 #include <sys/unistd.h>
```

lib/unistd/rmdir.c

«

Si veda la sezione [u0.30](#).

```
405001 #include <unistd.h>
405002 #include <string.h>
405003 #include <const.h>
405004 #include <sys/unistd.h>
405005 #include <errno.h>
405006 #include <limits.h>
405007 //-----
```

```

405008 int
405009 rmdir (const char *path)
405010 {
405011     sysmsg_stat_t msg_stat;
405012     sysmsg_unlink_t msg_unlink;
405013     //
405014     if (path == NULL)
405015     {
405016         errset (EINVAL);
405017         return (-1);
405018     }
405019     //
405020     strncpy (msg_stat.path, path, PATH_MAX);
405021     //
405022     sys (SYS_STAT, &msg_stat, (sizeof msg_stat));
405023     //
405024     if (msg_stat.ret != 0)
405025     {
405026         errno = msg_stat.errno;
405027         errln = msg_stat.errln;
405028         strncpy (errfn, msg_stat.errfn, PATH_MAX);
405029         return (msg_stat.ret);
405030     }
405031     //
405032     if (!S_ISDIR (msg_stat.stat.st_mode))
405033     {
405034         errset (ENOTDIR); // Not a directory.
405035         return (-1);
405036     }
405037     //
405038     strncpy (msg_unlink.path, path, PATH_MAX);
405039     //
405040     sys (SYS_UNLINK, &msg_unlink, (sizeof msg_unlink));
405041     //
405042     errno = msg_unlink.errno;
405043     errln = msg_unlink.errln;
405044     strncpy (errfn, msg_unlink.errfn, PATH_MAX);
405045     return (msg_unlink.ret);
405046 }

```

```

406015     errno = msg.errno;
406016     errln = msg.errln;
406017     strncpy (errfn, msg.errfn, PATH_MAX);
406018     return (msg.ret);
406019 }

```

lib/unistd/sleep.c

Si veda la sezione [u0.35](#).

```

406001 #include <unistd.h>
406002 #include <sys/types.h>
406003 #include <sys/osl6.h>
406004 #include <errno.h>
406005 #include <time.h>
406006 //-----
406007 unsigned int
406008 sleep (unsigned int seconds)
406009 {
406010     sysmsg_sleep_t msg;
406011     time_t start;
406012     time_t end;
406013     int slept;
406014     //
406015     if (seconds == 0)
406016     {
406017         return (0);
406018     }
406019     //
406020     msg.events = WAKEUP_EVENT_TIMER;
406021     msg.seconds = seconds;
406022     sys (SYS_SLEEP, &msg, (sizeof msg));
406023     start = msg.ret;
406024     end = time (NULL);
406025     slept = end - msg.ret;
406026     //
406027     if (slept < 0)
406028     {
406029         return (seconds);
406030     }
406031     else if (slept < seconds)
406032     {
406033         return (seconds - slept);
406034     }
406035     else
406036     {
406037         return (0);
406038     }
406039 }

```

lib/unistd/seteuid.c

Si veda la sezione [u0.33](#).

```

406001 #include <unistd.h>
406002 #include <sys/types.h>
406003 #include <sys/osl6.h>
406004 #include <errno.h>
406005 #include <string.h>
406006 //-----
406007 int
406008 seteuid (uid_t uid)
406009 {
406010     sysmsg_seteuid_t msg;
406011     msg.ret = 0;
406012     msg.errno = 0;
406013     msg.euid = uid;
406014     sys (SYS_SETEUID, &msg, (sizeof msg));
406015     errno = msg.errno;
406016     errln = msg.errln;
406017     strncpy (errfn, msg.errfn, PATH_MAX);
406018     return (msg.ret);
406019 }
406020

```

lib/unistd/ttyname.c

Si veda la sezione [u0.124](#).

```

410001 #include <sys/osl6.h>
410002 #include <sys/stat.h>
410003 #include <unistd.h>
410004 #include <sys/types.h>
410005 #include <errno.h>
410006 #include <limits.h>
410007 //-----
410008 char *
410009 ttyname (int fdn)
410010 {
410011     int dev_minor;
410012     struct stat file_status;
410013     static char name[PATH_MAX];
410014     //
410015     // Verify to have valid input data.
410016     //
410017     if (fdn < 0)
410018     {
410019         errset (EBADF);
410020         return (NULL);
410021     }
410022     //
410023     // Verify the file descriptor.
410024     //
410025     if (fstat (fdn, &file_status) == 0)
410026     {
410027         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
410028         {
410029             dev_minor = minor (file_status.st_rdev);
410030             //
410031             // If minor is equal to 0xFF, it is '/dev/console'.
410032             //
410033             if (dev_minor < 0xFF)
410034             {
410035                 sprintf (name, "/dev/console%i", dev_minor);
410036             }
410037             else
410038             {
410039                 strcpy (name, "/dev/console*");
410040             }
410041             return (name);
410042         }
410043         else if (file_status.st_rdev == DEV_TTY)
410044         {
410045             strcpy (name, "/dev/tty*");
410046             return (name);
410047         }
410048     }

```

```

410047     }
410048     else
410049     {
410050         errset (ENOTTY);
410051         return (NULL);
410052     }
410053     }
410054     else
410055     {
410056         errset (errno);
410057         return (NULL);
410058     }
410059 }

```

lib/unistd/unlink.c

Si veda la sezione u0.42.

```

410001 #include <unistd.h>
410002 #include <string.h>
410003 #include <const.h>
410004 #include <sys/os16.h>
410005 #include <errno.h>
410006 #include <limits.h>
410007 //-----
410008 int
410009 unlink (const char *path)
410010 {
410011     sysmsg_unlink_t msg;
410012     //
410013     strncpy (msg.path, path, PATH_MAX);
410014     //
410015     sys (SYS_UNLINK, &msg, (sizeof msg));
410016     //
410017     errno = msg.errno;
410018     errln = msg.errln;
410019     strncpy (errfn, msg.errfn, PATH_MAX);
410020     return (msg.ret);
410021 }

```

lib/unistd/write.c

Si veda la sezione u0.44.

```

412001 #include <unistd.h>
412002 #include <sys/os16.h>
412003 #include <errno.h>
412004 #include <string.h>
412005 #include <const.h>
412006 #include <stdio.h>
412007 //-----
412008 ssize_t
412009 write (int fdn, const void *buffer, size_t count)
412010 {
412011     sysmsg_write_t msg;
412012     //
412013     // Reduce size of write if necessary.
412014     //
412015     if (count > BUFSIZ)
412016     {
412017         count = BUFSIZ;
412018     }
412019     //
412020     // Fill the message.
412021     //
412022     msg.fdn = fdn;
412023     msg.count = count;
412024     memcpy (msg.buffer, buffer, count);
412025     //
412026     // Syscall.
412027     //
412028     sys (SYS_WRITE, &msg, (sizeof msg));
412029     //
412030     // Check result and return.
412031     //
412032     if (msg.ret < 0)
412033     {
412034         //
412035         // No valid read, no change inside the buffer.
412036         //
412037         errno = msg.errno;
412038         errln = msg.errln;
412039         strncpy (errfn, msg.errfn, PATH_MAX);
412040         return (msg.ret);
412041     }
412042     //
412043     if (msg.ret > count)
412044     {
412045         //
412046         // A strange value was returned. Considering it a read error.
412047         //
412048         errset (EIO); // I/O error.
412049         return (-1);
412050     }
412051     //
412052     // A valid write return.
412053     //
412054     return (msg.ret);
412055 }

```

os16: «lib/utime.h»

Si veda la sezione u0.2.

```

413001 #ifndef _UTIME_H
413002 #define _UTIME_H 1
413003
413004 #include <const.h>
413005 #include <restrict.h>
413006 #include <sys/types.h> // time_t
413007
413008 //-----
413009 struct utimbuf {
413010     time_t actime;
413011     time_t modtime;
413012 };
413013 //-----
413014 int utime (const char *path, const struct utimbuf *times);
413015 //-----
413016
413017 #endif

```

lib/utime/utime.c

Si veda la sezione u0.2.

```

414001 #include <utime.h>
414002 #include <errno.h>
414003 //-----
414004 int
414005 utime (const char *path, const struct utimbuf *times)
414006 {
414007     //
414008     // Currently not implemented.
414009     //
414010     return (0);
414011 }

```

os16: directory «applic/» 1850

- applic/MAKEDEV.c 1850
- applic/aaa.c 1850
- applic/bbb.c 1851
- applic/cat.c 1851
- applic/ccc.c 1852
- applic/chmod.c 1852
- applic/chown.c 1853
- applic/cp.c 1854
- applic/crt0.s 1856
- applic/date.c 1857
- applic/ed.c 1859
- applic/getty.c 1873
- applic/init.c 1874
- applic/kill.c 1876
- applic/ln.c 1879
- applic/login.c 1880
- applic/ls.c 1882
- applic/man.c 1885
- applic/mkdir.c 1888
- applic/more.c 1890
- applic/mount.c 1892
- applic/ps.c 1893
- applic/rm.c 1894
- applic/shell.c 1894
- applic/touch.c 1897
- applic/tty.c 1898
- applic/umount.c 1898

aaa.c 1850 bbb.c 1851 cat.c 1851 ccc.c 1852 chmod.c 1852
 chown.c 1853 cp.c 1854 crt0.s 1856 date.c 1857 ed.c 1859
 getty.c 1873 init.c 1874 kill.c 1876 ln.c 1879 login.c 1880
 ls.c 1882 MAKEDEV.c 1850 man.c 1885 mkdir.c 1888
 more.c 1890 mount.c 1892 ps.c 1893 rm.c 1894 shell.c 1894
 touch.c 1897 tty.c 1898 umount.c 1898

os16: directory «applic/» 1850

- applic/MAKEDEV.c 1850
- applic/aaa.c 1850
- applic/bbb.c 1851
- applic/cat.c 1851
- applic/ccc.c 1852
- applic/chmod.c 1852
- applic/chown.c 1853
- applic/cp.c 1854
- applic/crt0.s 1856
- applic/date.c 1857
- applic/ed.c 1859
- applic/getty.c 1873
- applic/init.c 1874
- applic/kill.c 1876
- applic/ln.c 1879
- applic/login.c 1880
- applic/ls.c 1882
- applic/man.c 1885
- applic/mkdir.c 1888
- applic/more.c 1890

applic/mount.c	1892
applic/ps.c	1893
applic/rm.c	1894
applic/shell.c	1894
applic/touch.c	1897
applic/tty.c	1898
applic/umount.c	1898

os16: directory «applic/»

applic/MAKEDEV.c

Si veda la sezione u0.3.

```

4150001 #include <unistd.h>
4150002 #include <stdlib.h>
4150003 #include <sys/stat.h>
4150004 #include <fcntl.h>
4150005 #include <kernel/devices.h>
4150006 #include <stdio.h>
4150007 //-----
4150008 int
4150009 main (void)
4150010 {
4150011     int status;
4150012     status = mknod ("mem",      (mode_t) (S_IFCHR | 0444),
4150013                  (dev_t) DEV_MEM);
4150014     if (status) perror (NULL);
4150015     status = mknod ("null",    (mode_t) (S_IFCHR | 0666),
4150016                  (dev_t) DEV_NULL);
4150017     if (status) perror (NULL);
4150018     status = mknod ("port",    (mode_t) (S_IFCHR | 0644),
4150019                  (dev_t) DEV_PORT);
4150020     if (status) perror (NULL);
4150021     status = mknod ("zero",    (mode_t) (S_IFCHR | 0666),
4150022                  (dev_t) DEV_ZERO);
4150023     if (status) perror (NULL);
4150024     status = mknod ("tty",     (mode_t) (S_IFCHR | 0666),
4150025                  (dev_t) DEV_TTY);
4150026     if (status) perror (NULL);
4150027     status = mknod ("disk0",   (mode_t) (S_IFBLK | 0644),
4150028                  (dev_t) DEV_DSK0);
4150029     if (status) perror (NULL);
4150030     status = mknod ("disk1",   (mode_t) (S_IFBLK | 0644),
4150031                  (dev_t) DEV_DSK1);
4150032     if (status) perror (NULL);
4150033     status = mknod ("disk2",   (mode_t) (S_IFBLK | 0644),
4150034                  (dev_t) DEV_DSK2);
4150035     if (status) perror (NULL);
4150036     status = mknod ("disk3",   (mode_t) (S_IFBLK | 0644),
4150037                  (dev_t) DEV_DSK3);
4150038     if (status) perror (NULL);
4150039     status = mknod ("kmem_ps",  (mode_t) (S_IFCHR | 0444),
4150040                  (dev_t) DEV_KMEM_PS);
4150041     if (status) perror (NULL);
4150042     status = mknod ("kmem_mmp", (mode_t) (S_IFCHR | 0444),
4150043                  (dev_t) DEV_KMEM_MMP);
4150044     if (status) perror (NULL);
4150045     status = mknod ("kmem_sb",  (mode_t) (S_IFCHR | 0444),
4150046                  (dev_t) DEV_KMEM_SB);
4150047     if (status) perror (NULL);
4150048     status = mknod ("kmem_inode", (mode_t) (S_IFCHR | 0444),
4150049                  (dev_t) DEV_KMEM_INODE);
4150050     if (status) perror (NULL);
4150051     status = mknod ("kmem_file", (mode_t) (S_IFCHR | 0444),
4150052                  (dev_t) DEV_KMEM_FILE);
4150053     if (status) perror (NULL);
4150054     status = mknod ("console",  (mode_t) (S_IFCHR | 0644),
4150055                  (dev_t) DEV_CONSOLE);
4150056     if (status) perror (NULL);
4150057     status = mknod ("console0", (mode_t) (S_IFCHR | 0644),
4150058                  (dev_t) DEV_CONSOLE0);
4150059     if (status) perror (NULL);
4150060     status = mknod ("console1", (mode_t) (S_IFCHR | 0644),
4150061                  (dev_t) DEV_CONSOLE1);
4150062     if (status) perror (NULL);
4150063     status = mknod ("console2", (mode_t) (S_IFCHR | 0644),
4150064                  (dev_t) DEV_CONSOLE2);
4150065     if (status) perror (NULL);
4150066     status = mknod ("console3", (mode_t) (S_IFCHR | 0644),
4150067                  (dev_t) DEV_CONSOLE3);
4150068     if (status) perror (NULL);
4150069
4150070     return (0);
4150071 }

```

applic/aaa.c

Si veda la sezione u0.1.

```

4160001 #include <unistd.h>
4160002 #include <stdio.h>
4160003 //-----
4160004 int

```

```

4160005 main (void)
4160006 {
4160007     unsigned int count;
4160008     for (count = 0; count < 60; count++)
4160009     {
4160010         printf ("a");
4160011         sleep (1);
4160012     }
4160013     return (8);
4160014 }

```

applic/bbb.c

Si veda la sezione u0.1.

```

4170001 #include <unistd.h>
4170002 #include <stdio.h>
4170003 #include <stdlib.h>
4170004 //-----
4170005 int
4170006 main (void)
4170007 {
4170008     unsigned int count;
4170009     for (count = 0; count < 30; count++)
4170010     {
4170011         printf ("b");
4170012         sleep (2);
4170013     }
4170014     exit (0);
4170015     return (0);
4170016 }

```

applic/cat.c

Si veda la sezione u0.3.

```

4180001 #include <fcntl.h>
4180002 #include <sys/stat.h>
4180003 #include <stddef.h>
4180004 #include <unistd.h>
4180005 #include <stdio.h>
4180006 #include <stdlib.h>
4180007 #include <errno.h>
4180008 //-----
4180009 static void cat_file_descriptor (int fd);
4180010 //-----
4180011 int
4180012 main (int argc, char *argv[], char *envp[])
4180013 {
4180014     int i;
4180015     int fd;
4180016     struct stat file_status;
4180017     //
4180018     // Check if the input comes from standard input.
4180019     //
4180020     if (argc < 2)
4180021     {
4180022         cat_file_descriptor (STDIN_FILENO);
4180023         exit (0);
4180024     }
4180025     //
4180026     // There is at least an argument: scan them.
4180027     //
4180028     for(i = 1; i < argc; i++)
4180029     {
4180030         //
4180031         // Verify if the file exists.
4180032         //
4180033         if (stat(argv[i], &file_status) != 0)
4180034         {
4180035             fprintf (stderr, "File \"%s\" does not exist!\n",
4180036                     argv[i]);
4180037             continue;
4180038         }
4180039         //
4180040         // File exists: check the file type.
4180041         //
4180042         if (S_ISDIR (file_status.st_mode))
4180043         {
4180044             fprintf (stderr, "Cannot \"cat\" \"
4180045                     \"%s\": it is a directory!\n",
4180046                     argv[i]);
4180047             continue;
4180048         }
4180049         //
4180050         // File exists and can be "cat"ed.
4180051         //
4180052         fd = open (argv[i], O_RDONLY);
4180053         if (fd >= 0)
4180054         {
4180055             cat_file_descriptor (fd);
4180056             close (fd);
4180057         }
4180058         else
4180059         {
4180060             perror (NULL);
4180061             exit (1);
4180062         }
4180063     }
4180064     return (0);

```



```

418065 }
418066 //-----
418067 static void
418068 cat_file_descriptor (int fd)
418069 {
418070     ssize_t count;
418071     char buffer[BUFSIZ];
418072
418073     for (;;)
418074     {
418075         count = read (fd, buffer, (size_t) BUFSIZ);
418076         if (count > 0)
418077         {
418078             write (STDOUT_FILENO, buffer, (size_t) count);
418079         }
418080         else
418081         {
418082             break;
418083         }
418084     }
418085 }
418086 }

```

applic/ccc.c

« Si veda la sezione u0.1.

```

419081 #include <unistd.h>
419082 #include <stdlib.h>
419083 #include <signal.h>
419084 //-----
419085 int
419086 main (void)
419087 {
419088     pid_t pid;
419089     //-----
419090     pid = fork ();
419091     if (pid == 0)
419092     {
419093         setuid ((uid_t) 10);
419094         exece ("/bin/aaa", NULL, NULL);
419095         exit (0);
419096     }
419097     //-----
419098     pid = fork ();
419099     if (pid == 0)
419100     {
419101         setuid ((uid_t) 11);
419102         exece ("/bin/bbb", NULL, NULL);
419103         exit (0);
419104     }
419105     //-----
419106     while (1)
419107     {
419108         ; // Just loop, to consume CPU time: it must be killed manually.
419109     }
419110     return (0);
419111 }

```

applic/chmod.c

« Si veda la sezione u0.5.

```

420001 #include <unistd.h>
420002 #include <stdlib.h>
420003 #include <sys/stat.h>
420004 #include <sys/types.h>
420005 #include <fcntl.h>
420006 #include <errno.h>
420007 #include <signal.h>
420008 #include <stdio.h>
420009 #include <sys/wait.h>
420010 #include <string.h>
420011 #include <limits.h>
420012 #include <sys/osl6.h>
420013 //-----
420014 static void usage (void);
420015 //-----
420016 int
420017 main (int argc, char *argv[], char *envp[])
420018 {
420019     int status;
420020     mode_t mode;
420021     char *m; // Pointer inside the octal mode string.
420022     int digit;
420023     int a; // Argument index.
420024     //
420025     //
420026     //
420027     //
420028     if (argc < 3)
420029     {
420030         usage ();
420031         return (1);
420032     }
420033     //
420034     // Get mode: must be the first argument.
420035     //
420036     for (m = argv[1]; *m != 0; m++)
420037     {

```

1852

```

420038     digit = (*m - '0');
420039     if (digit < 0 || digit > 7)
420040     {
420041         usage ();
420042         return (2);
420043     }
420044     mode = mode * 8 + digit;
420045     }
420046     //
420047     // System call for all the remaining arguments.
420048     //
420049     for (a = 2; a < argc; a++)
420050     {
420051         status = chmod (argv[a], mode);
420052         if (status != 0)
420053         {
420054             perror (argv[a]);
420055             return (3);
420056         }
420057     }
420058     //
420059     // All done.
420060     //
420061     return (0);
420062 }
420063 //-----
420064 static void
420065 usage (void)
420066 {
420067     fprintf (stderr, "Usage:  chmod OCTAL_MODE FILE...\n");
420068     fprintf (stderr, "Example: chmod 0640 my_file\n");
420069 }

```

applic/chown.c

« Si veda la sezione u0.6.

```

421001 #include <unistd.h>
421002 #include <stdlib.h>
421003 #include <sys/stat.h>
421004 #include <sys/types.h>
421005 #include <fcntl.h>
421006 #include <errno.h>
421007 #include <stdio.h>
421008 #include <ctype.h>
421009 #include <pwd.h>
421010 //-----
421011 static void usage (void);
421012 //-----
421013 int
421014 main (int argc, char *argv[], char *envp[])
421015 {
421016     char *user;
421017     int uid;
421018     struct passwd *pws;
421019     struct stat file_status;
421020     int a; // Argument index.
421021     int status;
421022     //
421023     //
421024     //
421025     if (argc < 3)
421026     {
421027         usage ();
421028         return (1);
421029     }
421030     //
421031     // Get user id number.
421032     //
421033     user = argv[1];
421034     if (isdigit (*user))
421035     {
421036         uid = atoi (user);
421037     }
421038     else
421039     {
421040         pws = getpwnam (user);
421041         if (pws == NULL)
421042         {
421043             fprintf (stderr, "Unknown user \"%s\"\n", user);
421044             return (2);
421045         }
421046         uid = pws->pw_uid;
421047     }
421048     //
421049     // Now we have the user id. Start scanning file names.
421050     //
421051     for (a = 2; a < argc; a++)
421052     {
421053         //
421054         // Verify if the file exists, through the return value of
421055         // 'stat()'. No other checks are made.
421056         //
421057         if (stat(argv[a], &file_status) == 0)
421058         {
421059             //
421060             // Try to change ownership.
421061             //
421062             status = chown (argv[a], uid, file_status.st_gid);
421063             if (status != 0)
421064             {

```

1853

```

4210065         perror (NULL);
4210066         return (3);
4210067     }
4210068     }
4210069     else
4210070     {
4210071         fprintf (stderr, "File \"%s\" does not exist!\n",
4210072                 argv[a]);
4210073         continue;
4210074     }
4210075     }
4210076     //
4210077     // All done.
4210078     //
4210079     return (0);
4210080 }
4210081 //-----
4210082 static void
4210083 usage (void)
4210084 {
4210085     fprintf (stderr, "Usage:  chown USER[UID FILE...\n");
4210086     fprintf (stderr, "Example: chown user my_file\n");
4210087 }

```

applic/cp.c

<

Si veda la sezione u0.7.

```

4220001 #include <sys/osl6.h>
4220002 #include <sys/stat.h>
4220003 #include <sys/types.h>
4220004 #include <unistd.h>
4220005 #include <stdlib.h>
4220006 #include <fcntl.h>
4220007 #include <errno.h>
4220008 #include <signal.h>
4220009 #include <stdio.h>
4220010 #include <string.h>
4220011 #include <limits.h>
4220012 #include <libgen.h>
4220013 //-----
4220014 static void usage (void);
4220015 //-----
4220016 int
4220017 main (int argc, char *argv[], char *envp[])
4220018 {
4220019     char      *source;
4220020     char      *destination;
4220021     char      *destination_full;
4220022     struct stat file_status;
4220023     int       dest_is_a_dir = 0;
4220024     int       a;                // Argument index.
4220025     char      path[PATH_MAX];
4220026     int       fd_source = -1;
4220027     int       fd_destination = -1;
4220028     char      buffer_in[BUFSIZ];
4220029     char      *buffer_out;
4220030     ssize_t   count_in;        // Read counter.
4220031     ssize_t   count_out;      // Write counter.
4220032     //
4220033     // There must be at least two arguments, plus the program name.
4220034     //
4220035     if (argc < 3)
4220036     {
4220037         usage ();
4220038         return (1);
4220039     }
4220040     //
4220041     // Select the last argument as the destination.
4220042     //
4220043     destination = argv[argc-1];
4220044     //
4220045     // Check if it is a directory and save it in a flag.
4220046     //
4220047     if (stat (destination, &file_status) == 0)
4220048     {
4220049         if (S_ISDIR (file_status.st_mode))
4220050         {
4220051             dest_is_a_dir = 1;
4220052         }
4220053     }
4220054     //
4220055     // If there are more than two arguments, verify that the last
4220056     // one is a directory.
4220057     //
4220058     if (argc > 3)
4220059     {
4220060         if (!dest_is_a_dir)
4220061         {
4220062             usage ();
4220063             fprintf (stderr, "The destination \"%s\" ",
4220064                     destination);
4220065             fprintf (stderr, "is not a directory!\n");
4220066             return (1);
4220067         }
4220068     }
4220069     //
4220070     // Scan the arguments, excluded the last, that is the destination.
4220071     //
4220072     for (a = 1; a < (argc - 1); a++)
4220073     {

```

1854

```

4220074     //
4220075     // Source.
4220076     //
4220077     source = argv[a];
4220078     //
4220079     // Verify access permissions.
4220080     //
4220081     if (access (source, R_OK) < 0)
4220082     {
4220083         perror (source);
4220084         continue;
4220085     }
4220086     //
4220087     // Destination.
4220088     //
4220089     // If it is a directory, the destination path
4220090     // must be corrected.
4220091     //
4220092     if (dest_is_a_dir)
4220093     {
4220094         path[0] = 0;
4220095         strcat (path, destination);
4220096         strcat (path, "/");
4220097         strcat (path, basename (source));
4220098         //
4220099         // Update the destination path.
4220100         //
4220101         destination_full = path;
4220102     }
4220103     else
4220104     {
4220105         destination_full = destination;
4220106     }
4220107     //
4220108     // Check if destination file exists.
4220109     //
4220110     if (stat (destination_full, &file_status) == 0)
4220111     {
4220112         fprintf (stderr, "The destination file, \"%s\", ",
4220113                 destination_full);
4220114         fprintf (stderr, "already exists!\n");
4220115         continue;
4220116     }
4220117     //
4220118     // Everything is ready for the copy.
4220119     //
4220120     fd_source = open (source, O_RDONLY);
4220121     if (fd_source < 0)
4220122     {
4220123         perror (source);
4220124         //
4220125         // Continue with the next file.
4220126         //
4220127         continue;
4220128     }
4220129     //
4220130     fd_destination = creat (destination_full, 0777);
4220131     if (fd_destination < 0)
4220132     {
4220133         perror (destination);
4220134         close (fd_source);
4220135         //
4220136         // Continue with the next file.
4220137         //
4220138         continue;
4220139     }
4220140     //
4220141     // Copy the data.
4220142     //
4220143     while (1)
4220144     {
4220145         count_in = read (fd_source, buffer_in, (size_t) BUFSIZ);
4220146         if (count_in > 0)
4220147         {
4220148             for (buffer_out = buffer_in; count_in > 0;)
4220149             {
4220150                 count_out = write (fd_destination, buffer_out,
4220151                                   (size_t) count_in);
4220152                 if (count_out < 0)
4220153                 {
4220154                     perror (destination);
4220155                     close (fd_source);
4220156                     close (fd_destination);
4220157                     return (3);
4220158                 }
4220159                 //
4220160                 // If not all data is written, continue writing,
4220161                 // but change the buffer start position and the
4220162                 // amount to be written.
4220163                 //
4220164                 buffer_out += count_out;
4220165                 count_in -= count_out;
4220166             }
4220167         }
4220168         else if (count_in < 0)
4220169         {
4220170             perror (source);
4220171             close (fd_source);
4220172             close (fd_destination);
4220173         }
4220174         else

```

1855

```

4220175     {
4220176         break;
4220177     }
4220178     }
4220179     //
4220180     if (close (fd_source))
4220181     {
4220182         perror (source);
4220183     }
4220184     if (close (fd_destination))
4220185     {
4220186         perror (destination);
4220187         return (4);
4220188     }
4220189     }
4220190     //
4220191     // All done.
4220192     //
4220193     return (0);
4220194 }
4220195 //-----
4220196 static void
4220197 usage (void)
4220198 {
4220199     fprintf (stderr, "Usage: cp OLD_NAME NEW_NAME\n");
4220200     fprintf (stderr, "      cp FILE... DIRECTORY\n");
4220201 }

```

applic/crt0.s

Si veda la sezione u0.2.

```

4230001 .extern _main
4230002 .extern __stdio_stream_setup
4230003 .extern __dirent_directory_stream_setup
4230004 .extern __atexit_setup
4230005 .extern __environment_setup
4230006 .global __mkargv
4230007 ;-----
4230008 ; Please note that, all segments are already set from the scheduler,
4230009 ; and there is also data inside the stack, so that the call to 'main()'
4230010 ; function will result as expected.
4230011 ;
4230012 ; This is a modified version of 'crt0.s' with a smaller stack size.
4230013 ;-----
4230014 ; The following statement says that the code will start at "startup"
4230015 ; label.
4230016 ;-----
4230017 entry startup
4230018 ;-----
4230019 .text
4230020 ;-----
4230021 startup:
4230022 ;
4230023 ; Jump after initial data.
4230024 ;
4230025 jmp startup_code
4230026 ;
4230027 filler:
4230028 ;
4230029 ; After four bytes, from the start, there is the
4230030 ; magic number and other data.
4230031 ;
4230032 .space (0x0004 - (filler - startup))
4230033 ;
4230034 magic:
4230035 .data4 0x6F733136 ; os16
4230036 .data4 0x6170706C ; appl
4230037 ;
4230038 segoff:
4230039 .data2 __segoff ; Data segment offset.
4230040 etext:
4230041 .data2 __etext ; End of code
4230042 edata:
4230043 .data2 __edata ; End of initialized data.
4230044 ebss:
4230045 .data2 __end ; End of not initialized data.
4230046 stack_size:
4230047 .data2 0x2000 ; Requested stack size. Every single application
4230048 ; might change this value.
4230049 ;
4230050 ; At the next label, the work begins.
4230051 ;
4230052 .align 2
4230053 startup_code:
4230054 ;
4230055 ; Before the call to the main function, it is necessary to extract
4230056 ; the value to assign to the global variable 'environ'. It is
4230057 ; described as 'char **environ' and should contain the same address
4230058 ; pointed by 'envp'. To get this value, the stack is popped and then
4230059 ; pushed again. Please recall that the stack was prepared from
4230060 ; the process management, at the 'exec()' system call.
4230061 ;
4230062 pop ax ; argc
4230063 pop bx ; argv
4230064 pop cx ; envp
4230065 mov _environ, cx ; Variable 'environ' comes from <unistd.h>.
4230066 push cx
4230067 push bx
4230068 push ax
4230069 ;

```

1856

```

4230070 ; Could it be enough? Of course not! To be able to handle the
4230071 ; environment, it must be copied inside the table
4230072 ; '_environment_table[[]]', that is defined inside <stdlib.h>.
4230073 ; To copy the environment it is used the function
4230074 ; '_environment_setup()', passing the 'envp' pointer.
4230075 ;
4230076 push cx
4230077 call __environment_setup
4230078 add sp, #2
4230079 ;
4230080 ; After the environment copy is done, the value for the traditional
4230081 ; variable 'environ' is updated, to point to the new array of
4230082 ; pointer. The updated value comes from variable '_environment',
4230083 ; defined inside <stdlib.h>. Then, also the 'argv' contained inside
4230084 ; the stack is replaced with the new value.
4230085 ;
4230086 mov ax, __environment
4230087 mov _environ, ax
4230088 ;
4230089 pop ax ; argc
4230090 pop bx ; argv[[]]
4230091 pop cx ; envp[[]]
4230092 mov cx, __environment
4230093 push cx
4230094 push bx
4230095 push ax
4230096 ;
4230097 ; Setup standard I/O streams and at-exit table.
4230098 ;
4230099 call __stdio_stream_setup
4230100 call __dirent_directory_stream_setup
4230101 call __atexit_setup
4230102 ;
4230103 ; Call the main function. The arguments are already pushed inside
4230104 ; the stack.
4230105 ;
4230106 call _main
4230107 ;
4230108 ; Save the return value at the symbol 'exit_value'.
4230109 ;
4230110 mov exit_value, ax
4230111 ;
4230112 .align 2
4230113 halt:
4230114 ;
4230115 push #2 ; Size of message.
4230116 push #exit_value ; Pointer to the message.
4230117 push #6 ; SYS_EXIT
4230118 call _sys
4230119 add sp, #2
4230120 add sp, #2
4230121 add sp, #2
4230122 ;
4230123 jmp halt
4230124 ;
4230125 ;-----
4230126 .align 2
4230127 __mkargv:
4230128 ; Symbol '__mkargv' is used by Bcc inside the function
4230129 ; 'main()' and must be present for a successful
4230130 ; compilation.
4230131 ;-----
4230132 .align 2
4230133 .data
4230134 exit_value:
4230135 .data2 0x0000
4230136 ;-----
4230137 .align 2
4230138 .bss

```

applic/date.c

Si veda la sezione u0.8.

```

4240001 #include <unistd.h>
4240002 #include <stdlib.h>
4240003 #include <errno.h>
4240004 #include <time.h>
4240005 #include <ctype.h>
4240006 //-----
4240007 static void usage (void);
4240008 //-----
4240009 int
4240010 main (int argc, char *argv[], char *envp[])
4240011 {
4240012     struct tm *timeptr;
4240013     char string[5];
4240014     time_t timer;
4240015     int length;
4240016     char *input;
4240017     int i;
4240018     //
4240019     // There can be at most an argument.
4240020     //
4240021     if (argc > 2)
4240022     {
4240023         usage ();
4240024         return (1);
4240025     }
4240026     //
4240027     // Check if there is no argument: must show the date.

```

1857

```

4240028 //
4240029 if (argc == 1)
4240030 {
4240031     timer = time (NULL);
4240032     printf ("%s\n", ctime (&timer));
4240033     return (0);
4240034 }
4240035 //
4240036 // There is one argument and must be the date do set.
4240037 //
4240038 input = argv[1];
4240039 //
4240040 // First get current date, for default values.
4240041 //
4240042 timer = time (NULL);
4240043 timeptr = gmtime (&timer);
4240044 //
4240045 // Verify to have a correct input.
4240046 //
4240047 length = (int) strlen (input);
4240048 if (length == 8 || length == 10 || length == 12)
4240049 {
4240050     for (i = 0; i < length; i++)
4240051     {
4240052         if (!isdigit (input[i]))
4240053         {
4240054             usage ();
4240055             return (2);
4240056         }
4240057     }
4240058 }
4240059 else
4240060 {
4240061     printf ("input: \"%s\": length: %i\n", input, length);
4240062     usage ();
4240063     return (3);
4240064 }
4240065 //
4240066 // Select the month.
4240067 //
4240068 string[0] = input[0];
4240069 string[1] = input[1];
4240070 string[2] = '\0';
4240071 timeptr->tm_mon = atoi (string);
4240072 //
4240073 // Select the day.
4240074 //
4240075 string[0] = input[2];
4240076 string[1] = input[3];
4240077 string[2] = '\0';
4240078 timeptr->tm_mday = atoi (string);
4240079 //
4240080 // Select the hour.
4240081 //
4240082 string[0] = input[4];
4240083 string[1] = input[5];
4240084 string[2] = '\0';
4240085 timeptr->tm_hour = atoi (string);
4240086 //
4240087 // Select the minute.
4240088 //
4240089 string[0] = input[6];
4240090 string[1] = input[7];
4240091 string[2] = '\0';
4240092 timeptr->tm_min = atoi (string);
4240093 //
4240094 // Select the year: must verify if there is a century.
4240095 //
4240096 if (length == 12)
4240097 {
4240098     string[0] = input[8];
4240099 string[1] = input[9];
4240100 string[2] = input[10];
4240101 string[3] = input[11];
4240102 string[4] = '\0';
4240103 timeptr->tm_year = atoi (string);
4240104 }
4240105 else if (length == 10)
4240106 {
4240107     sprintf (string, "%04i", timeptr->tm_year);
4240108 string[2] = input[8];
4240109 string[3] = input[9];
4240110 string[4] = '\0';
4240111 timeptr->tm_year = atoi (string);
4240112 }
4240113 //
4240114 // Now convert to 'time_t'.
4240115 //
4240116 timer = mktime (timeptr);
4240117 //
4240118 // Save to the system.
4240119 //
4240120 stime (&timer);
4240121 //
4240122 return (0);
4240123 }
4240124 //-----
4240125 static void
4240126 usage (void)
4240127 {
4240128     fprintf (stderr, "Usage: date [MDDHMM[CC]YY]\n");

```

1858

```

4240129 }

```

applic/ed.c

Si veda la sezione u0.9.

```

4250001 //-----
4250002 // 2009.08.18
4250003 // Modified by Daniele Giacomini for 'os16', to harmonize with it,
4250004 // even, when possible, on coding style.
4250005 //
4250006 // The original was taken form ELKS sources: 'elkscmd/misc_utils/ed.c'.
4250007 //-----
4250008 //
4250009 // Copyright (c) 1993 by David I. Bell
4250010 // Permission is granted to use, distribute, or modify this source,
4250011 // provided that this copyright notice remains intact.
4250012 //
4250013 // The "ed" built-in command (much simplified)
4250014 //
4250015 //-----
4250016 #include <stdio.h>
4250017 #include <ctype.h>
4250018 #include <unistd.h>
4250019 #include <stdbool.h>
4250020 #include <string.h>
4250021 #include <stdlib.h>
4250022 #include <fcntl.h>
4250023 //-----
4250024 #define isoctal(ch) (((ch) >= '0') && ((ch) <= '7'))
4250025 #define USERSIZE 1024 /* max line length typed in by user */
4250026 #define INITBUFSIZE 1024 /* initial buffer size */
4250027 //-----
4250028 typedef int num_t;
4250029 typedef int len_t;
4250030 //
4250031 //
4250032 // The following is the type definition of structure 'line_t', but the
4250033 // structure contains pointers to the same kind of type. With the
4250034 // compiler Bcc, it is the only way to declare it.
4250035 //
4250036 typedef struct line line_t;
4250037 //
4250038 struct line {
4250039     line_t *next;
4250040     line_t *prev;
4250041     len_t len;
4250042     char data[1];
4250043 };
4250044 //
4250045 static line_t lines;
4250046 static line_t *curline;
4250047 static num_t curnum;
4250048 static num_t lastnum;
4250049 static num_t marks[26];
4250050 static bool dirty;
4250051 static char *filename;
4250052 static char searchstring[USERSIZE];
4250053 //
4250054 static char *bufbase;
4250055 static char *bufptr;
4250056 static len_t bufused;
4250057 static len_t bufsize;
4250058 //-----
4250059 static void docommands (void);
4250060 static void subcommand (char *cp, num_t num1, num_t num2);
4250061 static bool getnum (char **retcp, bool *rethavenum,
4250062                  num_t *retnum);
4250063 static bool setcurnum (num_t num);
4250064 static bool initedit (void);
4250065 static void termedit (void);
4250066 static void addlines (num_t num);
4250067 static bool insertline (num_t num, char *data, len_t len);
4250068 static bool deletelines (num_t num1, num_t num2);
4250069 static bool printlines (num_t num1, num_t num2, bool expandflag);
4250070 static bool writelines (char *file, num_t num1, num_t num2);
4250071 static bool readlines (char *file, num_t num);
4250072 static num_t searchlines (char *str, num_t num1, num_t num2);
4250073 static len_t findstring (line_t *lp, char *str, len_t len,
4250074                       len_t offset);
4250075 static line_t *findline (num_t num);
4250076 //-----
4250077 // Main.
4250078 //-----
4250079 int
4250080 main (int argc, char *argv[], char *envp[])
4250081 {
4250082     if (!initedit ()) return (2);
4250083     //
4250084     if (argc > 1)
4250085     {
4250086         filename = strdup (argv[1]);
4250087         if (filename == NULL)
4250088         {
4250089             fprintf (stderr, "No memory\n");
4250090             termedit ();
4250091             return (1);
4250092         }
4250093         //
4250094         if (!readlines (filename, 1))
4250095         {

```

1859

```

425096         termit ();
425097         return (0);
425098     }
425099     //
425100     if (lastnum) setcurnum(1);
425101     //
425102     dirty = false;
425103 }
425104 //
425105 docommands ();
425106 //
425107 termit ();
425108 return (0);
425109 }
425110 -----
425111 // Read commands until we are told to stop.
425112 //-----
425113 void
425114 docommands (void)
425115 {
425116     char *cp;
425117     int len;
425118     num_t num1;
425119     num_t num2;
425120     bool have1;
425121     bool have2;
425122     char buf[USERSIZE];
425123     //
425124     while (true)
425125     {
425126         printf(": ");
425127         fflush (stdout);
425128         //
425129         if (fgets (buf, sizeof(buf), stdin) == NULL)
425130         {
425131             return;
425132         }
425133         //
425134         len = strlen (buf);
425135         if (len == 0)
425136         {
425137             return;
425138         }
425139         //
425140         cp = &buf[len - 1];
425141         if (*cp != '\n')
425142         {
425143             fprintf(stderr, "Command line too long\n");
425144             do
425145             {
425146                 len = fgetc(stdin);
425147             }
425148             while ((len != EOF) && (len != '\n'));
425149             //
425150             continue;
425151         }
425152         //
425153         while ((cp > buf) && isblank (cp[-1]))
425154         {
425155             cp--;
425156         }
425157         //
425158         *cp = '\0';
425159         //
425160         cp = buf;
425161         //
425162         while (isblank (*cp))
425163         {
425164             //*cp++;
425165             cp++;
425166         }
425167         //
425168         have1 = false;
425169         have2 = false;
425170         //
425171         if ((curnum == 0) && (lastnum > 0))
425172         {
425173             curnum = 1;
425174             curline = lines.next;
425175         }
425176         //
425177         if (!getnum (&cp, &have1, &num1))
425178         {
425179             continue;
425180         }
425181         //
425182         while (isblank (*cp))
425183         {
425184             cp++;
425185         }
425186         //
425187         if (*cp == ',')
425188         {
425189             cp++;
425190             if (!getnum (&cp, &have2, &num2))
425191             {
425192                 continue;
425193             }
425194             //
425195             if (!have1)
425196             {

```

1860

```

425197         num1 = 1;
425198     }
425199     if (!have2)
425200     {
425201         num2 = lastnum;
425202     }
425203     have1 = true;
425204     have2 = true;
425205 }
425206 //
425207 if (!have1)
425208 {
425209     num1 = curnum;
425210 }
425211 if (!have2)
425212 {
425213     num2 = num1;
425214 }
425215 //
425216 // Command interpretation switch.
425217 //
425218 switch (*cp++)
425219 {
425220     case 'a':
425221         addlines (num1 + 1);
425222         break;
425223     //
425224     case 'c':
425225         deletelines (num1, num2);
425226         addlines (num1);
425227         break;
425228     //
425229     case 'd':
425230         deletelines (num1, num2);
425231         break;
425232     //
425233     case 'f':
425234         if (*cp && !isblank (*cp))
425235         {
425236             fprintf (stderr, "Bad file command\n");
425237             break;
425238         }
425239         //
425240         while (isblank (*cp))
425241         {
425242             cp++;
425243         }
425244         if (*cp == '\0')
425245         {
425246             if (filename)
425247             {
425248                 printf ("%s\n", filename);
425249             }
425250             else
425251             {
425252                 printf ("No filename\n");
425253             }
425254             break;
425255         }
425256         //
425257         cp = strdup (cp);
425258         //
425259         if (cp == NULL)
425260         {
425261             fprintf (stderr, "No memory for filename\n");
425262             break;
425263         }
425264         //
425265         if (filename)
425266         {
425267             free(filename);
425268         }
425269         //
425270         filename = cp;
425271         break;
425272     //
425273     case 'i':
425274         addlines (num1);
425275         break;
425276     //
425277     case 'k':
425278         while (isblank(*cp))
425279         {
425280             cp++;
425281         }
425282         //
425283         if ((*cp < 'a') || (*cp > 'a') || cp[1])
425284         {
425285             fprintf (stderr, "Bad mark name\n");
425286             break;
425287         }
425288         //
425289         marks[*cp - 'a'] = num2;
425290         break;
425291     //
425292     case 'l':
425293         printlines (num1, num2, true);
425294         break;
425295     //
425296     case 'p':
425297         printlines (num1, num2, false);

```

1861

```

4250298     break;
4250299     //
4250300 case 'q':
4250301     while (isblank(*cp))
4250302     {
4250303         cp++;
4250304     }
4250305     //
4250306     if (havel || *cp)
4250307     {
4250308         fprintf (stderr, "Bad quit command\n");
4250309         break;
4250310     }
4250311     //
4250312     if (!dirty)
4250313     {
4250314         return;
4250315     }
4250316     //
4250317     printf ("Really quit? ");
4250318     fflush (stdout);
4250319     //
4250320     buf[0] = '\0';
4250321     fgets (buf, sizeof(buf), stdin);
4250322     cp = buf;
4250323     //
4250324     while (isblank (*cp))
4250325     {
4250326         cp++;
4250327     }
4250328     //
4250329     if ((*cp == 'y' || *cp == 'Y'))
4250330     {
4250331         return;
4250332     }
4250333     //
4250334     break;
4250335     //
4250336 case 'x':
4250337     if (*cp && !isblank(*cp))
4250338     {
4250339         fprintf (stderr, "Bad read command\n");
4250340         break;
4250341     }
4250342     //
4250343     while (isblank(*cp))
4250344     {
4250345         cp++;
4250346     }
4250347     //
4250348     if (*cp == '\0')
4250349     {
4250350         fprintf (stderr, "No filename\n");
4250351         break;
4250352     }
4250353     //
4250354     if (!havel)
4250355     {
4250356         num1 = lastnum;
4250357     }
4250358     //
4250359     // Open the file and add to the buffer
4250360     // at the next line.
4250361     //
4250362     if (readlines (cp, num1 + 1))
4250363     {
4250364         //
4250365         // If the file open fails, just
4250366         // break the command.
4250367         //
4250368         break;
4250369     }
4250370     //
4250371     // Set the default file name, if no
4250372     // previous name is available.
4250373     //
4250374     if (filename == NULL)
4250375     {
4250376         filename = strdup (cp);
4250377     }
4250378     //
4250379     break;
4250380
4250381 case 's':
4250382     subcommand (cp, num1, num2);
4250383     break;
4250384     //
4250385 case 'w':
4250386     if (*cp && !isblank(*cp))
4250387     {
4250388         fprintf(stderr, "Bad write command\n");
4250389         break;
4250390     }
4250391     //
4250392     while (isblank(*cp))
4250393     {
4250394         cp++;
4250395     }
4250396     //
4250397     if (!havel)
4250398     {

```

```

4250399         num1 = 1;
4250400         num2 = lastnum;
4250401     }
4250402     //
4250403     // If the file name is not specified, use the
4250404     // default one.
4250405     //
4250406     if (*cp == '\0')
4250407     {
4250408         cp = filename;
4250409     }
4250410     //
4250411     // If even the default file name is not specified,
4250412     // tell it.
4250413     //
4250414     if (cp == NULL)
4250415     {
4250416         fprintf (stderr, "No file name specified\n");
4250417         break;
4250418     }
4250419     //
4250420     // Write the file.
4250421     //
4250422     writelines (cp, num1, num2);
4250423     //
4250424     break;
4250425     //
4250426 case 'z':
4250427     switch (*cp)
4250428     {
4250429         case '-':
4250430             printlines (curnum-21, curnum, false);
4250431             break;
4250432         case '.':
4250433             printlines (curnum-11, curnum+10, false);
4250434             break;
4250435         default:
4250436             printlines (curnum, curnum+21, false);
4250437             break;
4250438     }
4250439     break;
4250440     //
4250441 case '.':
4250442     if (havel)
4250443     {
4250444         fprintf (stderr, "No arguments allowed\n");
4250445         break;
4250446     }
4250447     printlines (curnum, curnum, false);
4250448     break;
4250449     //
4250450 case '-':
4250451     if (setcurnum (curnum - 1))
4250452     {
4250453         printlines (curnum, curnum, false);
4250454     }
4250455     break;
4250456     //
4250457 case '=':
4250458     printf ("%d\n", num1);
4250459     break;
4250460     //
4250461 case '\0':
4250462     if (havel)
4250463     {
4250464         printlines (num2, num2, false);
4250465         break;
4250466     }
4250467     //
4250468     if (setcurnum (curnum + 1))
4250469     {
4250470         printlines (curnum, curnum, false);
4250471     }
4250472     break;
4250473     //
4250474     default:
4250475         fprintf (stderr, "Unimplemented command\n");
4250476         break;
4250477     }
4250478     }
4250479 }
4250480 //-----
4250481 // Do the substitute command.
4250482 // The current line is set to the last substitution done.
4250483 //-----
4250484 void
4250485 subcommand (char *cp, num_t num1, num_t num2)
4250486 {
4250487     int    delim;
4250488     char  *oldstr;
4250489     char  *newstr;
4250490     len_t  oldlen;
4250491     len_t  newlen;
4250492     len_t  deltalen;
4250493     len_t  offset;
4250494     line_t *lp;
4250495     line_t *nlp;
4250496     bool  globalflag;
4250497     bool  printflag;
4250498     bool  didsub;
4250499     bool  needprint;

```

```

425000     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
425001     {
425002         fprintf(stderr, "Bad line range for substitute\n");
425003         return;
425004     }
425005     //
425006     globalflag = false;
425007     printflag = false;
425008     didsub = false;
425009     needprint = false;
425010     //
425011     if (isblank(*cp) || (*cp == '\0'))
425012     {
425013         fprintf(stderr, "Bad delimiter for substitute\n");
425014         return;
425015     }
425016     //
425017     delim = *cp++;
425018     oldstr = cp;
425019     //
425020     cp = strchr(cp, delim);
425021     //
425022     if (cp == NULL)
425023     {
425024         fprintf(stderr, "Missing 2nd delimiter for substitute\n");
425025         return;
425026     }
425027     //
425028     *cp++ = '\0';
425029     //
425030     newstr = cp;
425031     cp = strchr(cp, delim);
425032     //
425033     if (cp)
425034     {
425035         *cp++ = '\0';
425036     }
425037     else
425038     {
425039         cp = "";
425040     }
425041     while (*cp)
425042     {
425043         switch (*cp++)
425044         {
425045             case 'g':
425046                 globalflag = true;
425047                 break;
425048             //
425049             case 'p':
425050                 printflag = true;
425051                 break;
425052             //
425053             default:
425054                 fprintf(stderr, "Unknown option for substitute\n");
425055                 return;
425056         }
425057     }
425058     //
425059     if (*oldstr == '\0')
425060     {
425061         if (searchstring[0] == '\0')
425062         {
425063             fprintf(stderr, "No previous search string\n");
425064             return;
425065         }
425066         oldstr = searchstring;
425067     }
425068     //
425069     if (oldstr != searchstring)
425070     {
425071         strcpy(searchstring, oldstr);
425072     }
425073     //
425074     lp = findline(num1);
425075     if (lp == NULL)
425076     {
425077         return;
425078     }
425079     //
425080     oldlen = strlen(oldstr);
425081     newlen = strlen(newstr);
425082     deltalen = newlen - oldlen;
425083     offset = 0;
425084     //
425085     while (num1 <= num2)
425086     {
425087         offset = findstring(lp, oldstr, oldlen, offset);
425088         if (offset < 0)
425089         {
425090             if (needprint)
425091             {
425092                 printlines(num1, num1, false);
425093                 needprint = false;
425094             }
425095             //
425096             offset = 0;
425097             lp = lp->nnext;
425098             num1++;
425099             continue;
425000

```

1864

```

425001     }
425002     //
425003     needprint = printflag;
425004     didsub = true;
425005     dirty = true;
425006     //-----
425007     // If the replacement string is the same size or shorter
425008     // than the old string, then the substitution is easy.
425009     //-----
425010     //
425011     if (deltalen <= 0)
425012     {
425013         memcpy(&lp->data[offset], newstr, newlen);
425014         //
425015         if (deltalen)
425016         {
425017             memcpy(&lp->data[offset + newlen],
425018                 &lp->data[offset + oldlen],
425019                 lp->len - offset - oldlen);
425020             //
425021             lp->len += deltalen;
425022         }
425023         //
425024         offset += newlen;
425025         //
425026         if (globalflag)
425027         {
425028             continue;
425029         }
425030         //
425031         if (needprint)
425032         {
425033             printlines(num1, num1, false);
425034             needprint = false;
425035         }
425036         //
425037         lp = lp->nnext;
425038         num1++;
425039         continue;
425040     }
425041     //-----
425042     // The new string is larger, so allocate a new line
425043     // structure and use that. Link it in in place of
425044     // the old line structure.
425045     //-----
425046     nlp = (line_t *) malloc(sizeof(line_t) + lp->len + deltalen);
425047     //
425048     if (nlp == NULL)
425049     {
425050         fprintf(stderr, "Cannot get memory for line\n");
425051         return;
425052     }
425053     //
425054     nlp->len = lp->len + deltalen;
425055     //
425056     memcpy(nlp->data, lp->data, offset);
425057     //
425058     memcpy(&nlp->data[offset], newstr, newlen);
425059     //
425060     memcpy(&nlp->data[offset + newlen],
425061         &lp->data[offset + oldlen],
425062         lp->len - offset - oldlen);
425063     //
425064     nlp->nnext = lp->nnext;
425065     nlp->prev = lp->prev;
425066     nlp->prev->nnext = nlp;
425067     nlp->nnext->prev = nlp;
425068     //
425069     if (curline == lp)
425070     {
425071         curline = nlp;
425072     }
425073     //
425074     free(lp);
425075     lp = nlp;
425076     //
425077     offset += newlen;
425078     //
425079     if (globalflag)
425080     {
425081         continue;
425082     }
425083     //
425084     if (needprint)
425085     {
425086         printlines(num1, num1, false);
425087         needprint = false;
425088     }
425089     //
425090     lp = lp->nnext;
425091     num1++;
425092     }
425093     //
425094     if (!didsub)
425095     {
425096         fprintf(stderr, "No substitutions found for \"%s\"\n", oldstr);
425097     }
425098 }

```

1865

```

425002 //-----
425003 // Search a line for the specified string starting at the specified
425004 // offset in the line. Returns the offset of the found string, or -1.
425005 //-----
425006 len_t
425007 findstring (line_t *lp, char *str, len_t len, len_t offset)
425008 {
425009     len_t left;
425010     char *cp;
425011     char *ncp;
425012     //
425013     cp = &lp->data[offset];
425014     left = lp->len - offset;
425015     //
425016     while (left >= len)
425017     {
425018         ncp = strchr(cp, *str, left);
425019         if (ncp == NULL)
425020         {
425021             return (len_t) -1;
425022         }
425023         //
425024         left -= (ncp - cp);
425025         if (left < len)
425026         {
425027             return (len_t) -1;
425028         }
425029         //
425030         cp = ncp;
425031         if (memcmp(cp, str, len) == 0)
425032         {
425033             return (len_t) (cp - lp->data);
425034         }
425035         //
425036         cp++;
425037         left--;
425038     }
425039     //
425040     return (len_t) -1;
425041 }
425042 //-----
425043 // Add lines which are typed in by the user.
425044 // The lines are inserted just before the specified line number.
425045 // The lines are terminated by a line containing a single dot (ugly!),
425046 // or by an end of file.
425047 //-----
425048 void
425049 addlines (num_t num)
425050 {
425051     int len;
425052     char buf[USERSIZE + 1];
425053     //
425054     while (fgets (buf, sizeof (buf), stdin))
425055     {
425056         if ((buf[0] == '.' && (buf[1] == '\n') && (buf[2] == '\0'))
425057         {
425058             return;
425059         }
425060         //
425061         len = strlen (buf);
425062         //
425063         if (len == 0)
425064         {
425065             return;
425066         }
425067         //
425068         if (buf[len - 1] != '\n')
425069         {
425070             fprintf (stderr, "Line too long\n");
425071             //
425072             do
425073             {
425074                 len = fgetc(stdin);
425075             }
425076             while ((len != EOF) && (len != '\n'));
425077             //
425078             return;
425079         }
425080         //
425081         if (!insertline (num++, buf, len))
425082         {
425083             return;
425084         }
425085     }
425086 }
425087 //-----
425088 // Parse a line number argument if it is present. This is a sum
425089 // or difference of numbers, '.', '$', 'x', or a search string.
425090 // Returns true if successful (whether or not there was a number).
425091 // Returns false if there was a parsing error, with a message output.
425092 // Whether there was a number is returned indirectly, as is the number.
425093 // The character pointer which stopped the scan is also returned.
425094 //-----
425095 static bool
425096 getnum (char **retcp, bool *rethavenum, num_t *retnum)
425097 {
425098     char *cp;
425099     char *str;
425100     bool havenum;
425101     num_t value;
425102     num_t num;

```

```

425003 num_t sign;
425004 //
425005 cp = *retcp;
425006 havenum = false;
425007 value = 0;
425008 sign = 1;
425009 //
425010 while (true)
425011 {
425012     while (isblank(*cp))
425013     {
425014         cp++;
425015     }
425016     //
425017     switch (*cp)
425018     {
425019         case '.':
425020             havenum = true;
425021             num = curnum;
425022             cp++;
425023             break;
425024         //
425025         case '$':
425026             havenum = true;
425027             num = lastnum;
425028             cp++;
425029             break;
425030         //
425031         case '\':
425032             cp++;
425033             if ((*cp < 'a' || (*cp > 'z'))
425034             {
425035                 fprintf (stderr, "Bad mark name\n");
425036                 return false;
425037             }
425038             //
425039             havenum = true;
425040             num = marks[*cp++ - 'a'];
425041             break;
425042         //
425043         case '/':
425044             str = ++cp;
425045             cp = strchr (str, '/');
425046             if (cp)
425047             {
425048                 *cp++ = '\0';
425049             }
425050             else
425051             {
425052                 cp = "";
425053             }
425054             num = searchlines (str, curnum, lastnum);
425055             if (num == 0)
425056             {
425057                 return false;
425058             }
425059             //
425060             havenum = true;
425061             break;
425062         //
425063         default:
425064             if (!isdigit (*cp))
425065             {
425066                 *retcp = cp;
425067                 *rethavenum = havenum;
425068                 *retnum = value;
425069                 return true;
425070             }
425071             //
425072             num = 0;
425073             while (isdigit(*cp))
425074             {
425075                 num = num * 10 + *cp++ - '0';
425076             }
425077             havenum = true;
425078             break;
425079     }
425080     //
425081     value += num * sign;
425082     //
425083     while (isblank(*cp))
425084     {
425085         cp++;
425086     }
425087     //
425088     switch (*cp)
425089     {
425090         case '-':
425091             sign = -1;
425092             cp++;
425093             break;
425094         //
425095         case '+':
425096             sign = 1;
425097             cp++;
425098             break;
425099         //
425100         default:
425101             *retcp = cp;
425102             *rethavenum = havenum;
425103             *retnum = value;

```



```

425004         return true;
425005     }
425006 }
425007 }
425008 //-----
425009 // Initialize everything for editing.
425010 //-----
425011 bool
425012 initedit (void)
425013 {
425014     int i;
425015     //
425016     bufsize = INITBUFSIZE;
425017     bufbase = malloc (bufsize);
425018     //
425019     if (bufbase == NULL)
425020     {
425021         fprintf (stderr, "No memory for buffer\n");
425022         return false;
425023     }
425024     //
425025     bufptr = bufbase;
425026     bufused = 0;
425027     //
425028     lines.next = &lines;
425029     lines.prev = &lines;
425030     //
425031     curline = NULL;
425032     curnum = 0;
425033     lastnum = 0;
425034     dirty = false;
425035     filename = NULL;
425036     searchstring[0] = '\0';
425037     //
425038     for (i = 0; i < 26; i++)
425039     {
425040         marks[i] = 0;
425041     }
425042     //
425043     return true;
425044 }
425045 //-----
425046 // Finish editing.
425047 //-----
425048 void
425049 termdit (void)
425050 {
425051     if (bufbase) free(bufbase);
425052     bufbase = NULL;
425053     //
425054     bufptr = NULL;
425055     bufsize = 0;
425056     bufused = 0;
425057     //
425058     if (filename) free(filename);
425059     filename = NULL;
425060     //
425061     searchstring[0] = '\0';
425062     //
425063     if (lastnum) deletelines (1, lastnum);
425064     //
425065     lastnum = 0;
425066     curnum = 0;
425067     curline = NULL;
425068 }
425069 //-----
425070 // Read lines from a file at the specified line number.
425071 // Returns true if the file was successfully read.
425072 //-----
425073 bool
425074 readlines (char *file, num_t num)
425075 {
425076     int fd;
425077     int cc;
425078     len_t len;
425079     len_t linecount;
425080     len_t charcount;
425081     char *cp;
425082     //
425083     if ((num < 1) || (num > lastnum + 1))
425084     {
425085         fprintf (stderr, "Bad line for read\n");
425086         return false;
425087     }
425088     //
425089     fd = open (file, O_RDONLY);
425090     if (fd < 0)
425091     {
425092         perror (file);
425093         return false;
425094     }
425095     //
425096     bufptr = bufbase;
425097     bufused = 0;
425098     linecount = 0;
425099     charcount = 0;
425100     //
425101     printf ("\n%s\n", file);
425102     fflush(stdout);
425103     //
425104     do

```

1868

```

425105     {
425106         cp = memchr(bufptr, '\n', bufused);
425107         if (cp)
425108         {
425109             len = (cp - bufptr) + 1;
425110             //
425111             if (!insertline (num, bufptr, len))
425112             {
425113                 close (fd);
425114                 return false;
425115             }
425116             //
425117             bufptr += len;
425118             bufused -= len;
425119             charcount += len;
425120             linecount++;
425121             num++;
425122             continue;
425123         }
425124         //
425125         if (bufptr != bufbase)
425126         {
425127             memcpy (bufbase, bufptr, bufused);
425128             bufptr = bufbase + bufused;
425129         }
425130         //
425131         if (bufused >= bufsize)
425132         {
425133             len = (bufsize * 3) / 2;
425134             cp = realloc (bufbase, len);
425135             if (cp == NULL)
425136             {
425137                 fprintf (stderr, "No memory for buffer\n");
425138                 close (fd);
425139                 return false;
425140             }
425141             //
425142             bufbase = cp;
425143             bufptr = bufbase + bufused;
425144             bufsize = len;
425145         }
425146         //
425147         cc = read (fd, bufptr, bufsize - bufused);
425148         bufused += cc;
425149         bufptr = bufbase;
425150     }
425151     while (cc > 0);
425152     //
425153     if (cc < 0)
425154     {
425155         perror (file);
425156         close (fd);
425157         return false;
425158     }
425159     //
425160     if (bufused)
425161     {
425162         if (!insertline (num, bufptr, bufused))
425163         {
425164             close (fd);
425165             return -1;
425166         }
425167         linecount++;
425168         charcount += bufused;
425169     }
425170     //
425171     close (fd);
425172     //
425173     printf ("%d lines%s, %d chars\n",
425174             linecount,
425175             (bufused ? " (incomplete)" : ""),
425176             charcount);
425177     //
425178     return true;
425179 }
425180 //-----
425181 // Write the specified lines out to the specified file.
425182 // Returns true if successful, or false on an error with a message
425183 // output.
425184 //-----
425185 bool
425186 writelines (char *file, num_t num1, num_t num2)
425187 {
425188     int fd;
425189     line_t *lp;
425190     len_t linecount;
425191     len_t charcount;
425192     //
425193     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
425194     {
425195         fprintf (stderr, "Bad line range for write\n");
425196         return false;
425197     }
425198     //
425199     linecount = 0;
425200     charcount = 0;
425201     //
425202     fd = creat (file, 0666);
425203     if (fd < 0)
425204     {
425205         perror (file);

```

1869

```

425106     return false;
425107     }
425108     //
425109     printf("\n\n", file);
425110     fflush (stdout);
425111     //
425112     lp = findline (num1);
425113     if (lp == NULL)
425114     {
425115         close (fd);
425116         return false;
425117     }
425118     //
425119     while (num1++ <= num2)
425120     {
425121         if (write (fd, lp->data, lp->len) != lp->len)
425122         {
425123             perror (file);
425124             close (fd);
425125             return false;
425126         }
425127         //
425128         charcount += lp->len;
425129         linecount++;
425130         lp = lp->next;
425131     }
425132     //
425133     if (close (fd) < 0)
425134     {
425135         perror (file);
425136         return false;
425137     }
425138     //
425139     printf ("%d lines, %d chars\n", linecount, charcount);
425140     //
425141     return true;
425142 }
425143 //-----
425144 // Print lines in a specified range.
425145 // The last line printed becomes the current line.
425146 // If expandflag is true, then the line is printed specially to
425147 // show magic characters.
425148 //-----
425149 bool
425150 printlines (num_t num1, num_t num2, bool expandflag)
425151 {
425152     line_t *lp;
425153     unsigned char *cp;
425154     int ch;
425155     len_t count;
425156     //
425157     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
425158     {
425159         fprintf (stderr, "Bad line range for print\n");
425160         return false;
425161     }
425162     //
425163     lp = findline (num1);
425164     if (lp == NULL)
425165     {
425166         return false;
425167     }
425168     //
425169     while (num1 <= num2)
425170     {
425171         if (!expandflag)
425172         {
425173             write (STDOUT_FILENO, lp->data, lp->len);
425174             setcurnum (num1++);
425175             lp = lp->next;
425176             continue;
425177         }
425178         //-----
425179         // Show control characters and characters with the
425180         // high bit set specially.
425181         //-----
425182         //-----
425183         cp = (unsigned char *) lp->data;
425184         count = lp->len;
425185         //
425186         if ((count > 0) && (cp[count - 1] == '\n'))
425187         {
425188             count--;
425189         }
425190         //
425191         while (count-- > 0)
425192         {
425193             ch = *cp++;
425194             if (ch & 0x80)
425195             {
425196                 fputs ("M-", stdout);
425197                 ch &= 0x7F;
425198             }
425199             if (ch < ' ')
425200             {
425201                 fputc ('^', stdout);
425202                 ch += '@';
425203             }
425204             if (ch == 0x7F)
425205             {

```

1870

```

425207         fputc ('^', stdout);
425208         ch = '?';
425209     }
425210     fputs (ch, stdout);
425211     }
425212     //
425213     fputs ("\n", stdout);
425214     //
425215     setcurnum (num1++);
425216     lp = lp->next;
425217     }
425218     //
425219     return true;
425220 }
425221 //-----
425222 // Insert a new line with the specified text.
425223 // The line is inserted so as to become the specified line,
425224 // thus pushing any existing and further lines down one.
425225 // The inserted line is also set to become the current line.
425226 // Returns true if successful.
425227 //-----
425228 bool
425229 insertline (num_t num, char *data, len_t len)
425230 {
425231     line_t *newlp;
425232     line_t *lp;
425233     //
425234     if ((num < 1) || (num > lastnum + 1))
425235     {
425236         fprintf (stderr, "Inserting at bad line number\n");
425237         return false;
425238     }
425239     //
425240     newlp = (line_t *) malloc (sizeof (line_t) + len - 1);
425241     if (newlp == NULL)
425242     {
425243         fprintf (stderr, "Failed to allocate memory for line\n");
425244         return false;
425245     }
425246     //
425247     memcpy (newlp->data, data, len);
425248     newlp->len = len;
425249     //
425250     if (num > lastnum)
425251     {
425252         lp = &lines;
425253     }
425254     else
425255     {
425256         lp = findline (num);
425257         if (lp == NULL)
425258         {
425259             free ((char *) newlp);
425260             return false;
425261         }
425262     }
425263     //
425264     newlp->next = lp;
425265     newlp->prev = lp->prev;
425266     lp->prev->next = newlp;
425267     lp->prev = newlp;
425268     //
425269     lastnum++;
425270     dirty = true;
425271     //
425272     return setcurnum (num);
425273 }
425274 //-----
425275 // Delete lines from the given range.
425276 //-----
425277 bool
425278 deletelines (num_t num1, num_t num2)
425279 {
425280     line_t *lp;
425281     line_t *nlp;
425282     line_t *plp;
425283     num_t count;
425284     //
425285     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
425286     {
425287         fprintf (stderr, "Bad line numbers for delete\n");
425288         return false;
425289     }
425290     //
425291     lp = findline (num1);
425292     if (lp == NULL)
425293     {
425294         return false;
425295     }
425296     //
425297     if ((curnum >= num1) && (curnum <= num2))
425298     {
425299         if (num2 < lastnum)
425300         {
425301             setcurnum (num2 + 1);
425302         }
425303         else if (num1 > 1)
425304         {
425305             setcurnum (num1 - 1);
425306         }
425307         else

```

1871

```

4251308     {
4251309         curnum = 0;
4251310     }
4251311     }
4251312     //
4251313     count = num2 - num1 + 1;
4251314     //
4251315     if (curnum > num2)
4251316     {
4251317         curnum -= count;
4251318     }
4251319     //
4251320     lastnum -= count;
4251321     //
4251322     while (count-- > 0)
4251323     {
4251324         nlp = lp->next;
4251325         plp = lp->prev;
4251326         plp->next = nlp;
4251327         nlp->prev = plp;
4251328         lp->next = NULL;
4251329         lp->prev = NULL;
4251330         lp->len = 0;
4251331         free(lp);
4251332         lp = nlp;
4251333     }
4251334     //
4251335     dirty = true;
4251336     //
4251337     return true;
4251338 }
4251339 //-----
4251340 // Search for a line which contains the specified string.
4251341 // If the string is NULL, then the previously searched for string
4251342 // is used. The currently searched for string is saved for future use.
4251343 // Returns the line number which matches, or 0 if there was no match
4251344 // with an error printed.
4251345 //-----
4251346 num_t
4251347 searchlines (char *str, num_t num1, num_t num2)
4251348 {
4251349     line_t *lp;
4251350     int len;
4251351     //
4251352     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4251353     {
4251354         fprintf (stderr, "Bad line numbers for search\n");
4251355         return 0;
4251356     }
4251357     //
4251358     if (*str == '\0')
4251359     {
4251360         if (searchstring[0] == '\0')
4251361         {
4251362             fprintf(stderr, "No previous search string\n");
4251363             return 0;
4251364         }
4251365         str = searchstring;
4251366     }
4251367     //
4251368     if (str != searchstring)
4251369     {
4251370         strcpy(searchstring, str);
4251371     }
4251372     //
4251373     len = strlen(str);
4251374     //
4251375     lp = findline (num1);
4251376     if (lp == NULL)
4251377     {
4251378         return 0;
4251379     }
4251380     //
4251381     while (num1 <= num2)
4251382     {
4251383         if (findstring(lp, str, len, 0) >= 0)
4251384         {
4251385             return num1;
4251386         }
4251387         //
4251388         num1++;
4251389         lp = lp->next;
4251390     }
4251391     //
4251392     fprintf (stderr, "Cannot find string \"%s\"\n", str);
4251393     //
4251394     return 0;
4251395 }
4251396 //-----
4251397 // Return a pointer to the specified line number.
4251398 //-----
4251399 line_t *
4251400 findline (num_t num)
4251401 {
4251402     line_t *lp;
4251403     num_t lnum;
4251404     //
4251405     if ((num < 1) || (num > lastnum))
4251406     {
4251407         fprintf (stderr, "Line number %d does not exist\n", num);
4251408         return NULL;

```

1872

```

4251409     }
4251410     //
4251411     if (curnum <= 0)
4251412     {
4251413         curnum = 1;
4251414         curline = lines.next;
4251415     }
4251416     //
4251417     if (num == curnum)
4251418     {
4251419         return curline;
4251420     }
4251421     //
4251422     lp = curline;
4251423     lnum = curnum;
4251424     //
4251425     if (num < (curnum / 2))
4251426     {
4251427         lp = lines.next;
4251428         lnum = 1;
4251429     }
4251430     else if (num > ((curnum + lastnum) / 2))
4251431     {
4251432         lp = lines.prev;
4251433         lnum = lastnum;
4251434     }
4251435     //
4251436     while (lnum < num)
4251437     {
4251438         lp = lp->next;
4251439         lnum++;
4251440     }
4251441     //
4251442     while (lnum > num)
4251443     {
4251444         lp = lp->prev;
4251445         lnum--;
4251446     }
4251447     //
4251448     return lp;
4251449 }
4251450 //-----
4251451 // Set the current line number.
4251452 // Returns true if successful.
4251453 //-----
4251454 bool
4251455 setcurnum (num_t num)
4251456 {
4251457     line_t *lp;
4251458     //
4251459     lp = findline (num);
4251460     if (lp == NULL)
4251461     {
4251462         return false;
4251463     }
4251464     //
4251465     curnum = num;
4251466     curline = lp;
4251467     //
4251468     return true;
4251469 }
4251470
4251471 /* END CODE */

```

applic/getty.c

Si veda la sezione u0.1.

«

```

4260001 #include <unistd.h>
4260002 #include <stdio.h>
4260003 #include <stdlib.h>
4260004 #include <signal.h>
4260005 #include <sys/wait.h>
4260006 #include <limits.h>
4260007 #include <sys/unistd.h>
4260008 #include <fcntl.h>
4260009 #include <stdio.h>
4260010 //-----
4260011 int
4260012 main (int argc, char *argv[], char *envp[])
4260013 {
4260014     char *device_name;
4260015     int fdn;
4260016     char *exec_argv[2];
4260017     char **exec_envp;
4260018     char buffer[BUFSIZ];
4260019     ssize_t size_read;
4260020     int status;
4260021     //
4260022     // The first argument is mandatory and must be a console terminal.
4260023     //
4260024     device_name = argv[1];
4260025     //
4260026     // A console terminal is correctly selected (but it is not checked
4260027     // if it is a really available one).
4260028     // Set as a process group leader.
4260029     //
4260030     setpgid (0);
4260031     //
4260032     // Open the terminal, that should become the controlling terminal:
4260033     // close the standard input and open the new terminal (r/w).

```

1873

```

426004 //
426005 close (0);
426006 fdn = open (device_name, O_RDWR);
426007 if (fdn < 0)
426008 {
426009 //
426010 // Cannot open terminal. A message should appear, at least
426011 // to the current console.
426012 //
426013 perror (NULL);
426014 return (-1);
426015 }
426016 //
426017 // Reset terminal device permissions and ownership.
426018 //
426019 status = fchown (fdn, (uid_t) 0, (gid_t) 0);
426020 if (status != 0)
426021 {
426022 perror (NULL);
426023 }
426024 status = fchmod (fdn, 0644);
426025 if (status != 0)
426026 {
426027 perror (NULL);
426028 }
426029 //
426030 // The terminal is open and it should be already the controlling
426031 // one: show '/etc/issue'. The same variable 'fdn' is used, because
426032 // the controlling terminal will never be closed (the exit syscall
426033 // will do it).
426034 //
426035 fdn = open ("/etc/issue", O_RDONLY);
426036 if (fdn > 0)
426037 {
426038 //
426039 // The file is present and is shown.
426040 //
426041 for (size_read = 1; size_read > 0;)
426042 {
426043 size_read = read (fdn, buffer, (size_t) (BUFSIZ - 1));
426044 if (size_read < 0)
426045 {
426046 break;
426047 }
426048 buffer[size_read] = '\0';
426049 printf ("%s", buffer);
426050 }
426051 close (fdn);
426052 }
426053 //
426054 // Show the terminal.
426055 //
426056 printf ("This is terminal %s\n", device_name);
426057 //
426058 // It is time to exec login: the environment is inherited directly
426059 // from 'init'.
426060 //
426061 exec_argv[0] = "login";
426062 exec_argv[1] = NULL;
426063 exec_envp = envp;
426064 execve ("/bin/login", exec_argv, exec_envp);
426065 //
426066 // If 'execve()' returns, it is an error.
426067 //
426068 //
426069 exit (-1);
426070 }

```

applic/init.c

Si veda la sezione u0.2.

```

427001 #include <unistd.h>
427002 #include <stdio.h>
427003 #include <stdlib.h>
427004 #include <signal.h>
427005 #include <sys/wait.h>
427006 #include <limits.h>
427007 #include <sys/osi6.h>
427008 #include <fcntl.h>
427009 #include <string.h>
427010 //-----
427011 #define RESPAWN_MAX      7
427012 #define COMMAND_MAX     100
427013 #define LINE_MAX        1024
427014 //-----
427015 int
427016 main (int argc, char *argv[], char *envp[])
427017 {
427018 //
427019 // 'init.c' has its own 'init.crt0.s' with a very small stack
427020 // size. Remember to verify to have enough room for the stack.
427021 //
427022 pid_t pid;
427023 int status;
427024 char *exec_argv[3];
427025 char *exec_envp[3];
427026 char buffer[LINE_MAX];
427027 int r; // Respawn table index.
427028 int b; // Buffer index.
427029 size_t size_read;
427030 char *inittab_id;

```

1874

```

427031 char *inittab_runlevels;
427032 char *inittab_action;
427033 char *inittab_process;
427034 int eof;
427035 int fd;
427036 //
427037 // It follows a table for commands to be respawn.
427038 //
427039 struct {
427040 pid_t pid;
427041 char command[COMMAND_MAX];
427042 } respawn[RESPAWN_MAX];
427043 //-----
427044 signal (SIGHUP, SIG_IGN);
427045 signal (SIGINT, SIG_IGN);
427046 signal (SIGQUIT, SIG_IGN);
427047 signal (SIGILL, SIG_IGN);
427048 signal (SIGABRT, SIG_IGN);
427049 signal (SIGFPE, SIG_IGN);
427050 // signal (SIGKILL, SIG_IGN); Cannot ignore SIGKILL.
427051 signal (SIGSEGV, SIG_IGN);
427052 signal (SIGPIPE, SIG_IGN);
427053 signal (SIGALRM, SIG_IGN);
427054 signal (SIGTERM, SIG_IGN);
427055 // signal (SIGSTOP, SIG_IGN); Cannot ignore SIGSTOP.
427056 signal (SIGTSTP, SIG_IGN);
427057 signal (SIGCONT, SIG_IGN);
427058 signal (SIGTTIN, SIG_IGN);
427059 signal (SIGTTOU, SIG_IGN);
427060 signal (SIGUSR1, SIG_IGN);
427061 signal (SIGUSR2, SIG_IGN);
427062 //-----
427063 printf ("init\n");
427064 // heap_clear ();
427065 // process_info ();
427066 //-----
427067 //
427068 // Reset the 'respawn' table.
427069 //
427070 for (r = 0; r < RESPAWN_MAX; r++)
427071 {
427072 respawn[r].pid = 0;
427073 respawn[r].command[0] = 0;
427074 respawn[r].command[COMMAND_MAX-1] = 0;
427075 }
427076 //
427077 // Read the '/etc/inittab' file.
427078 //
427079 fd = open ("/etc/inittab", O_RDONLY);
427080 //
427081 if (fd < 0)
427082 {
427083 perror ("Cannot open file '/etc/inittab'");
427084 exit (-1);
427085 }
427086 //
427087 //
427088 //
427089 //
427090 for (eof = 0, r = 0; !eof && r < RESPAWN_MAX; r++)
427091 {
427092 for (b = 0; b < LINE_MAX; b++)
427093 {
427094 size_read = read (fd, &buffer[b], (size_t) 1);
427095 if (size_read <= 0)
427096 {
427097 buffer[b] = 0;
427098 eof = 1; // Close the read loop.
427099 break;
427100 }
427101 if (buffer[b] == '\n')
427102 {
427103 buffer[b] = 0;
427104 break;
427105 }
427106 }
427107 //
427108 // Remove comments: just replace '#' with '\0'.
427109 //
427110 for (b = 0; b < LINE_MAX; b++)
427111 {
427112 if (buffer[b] == '#')
427113 {
427114 buffer[b] = 0;
427115 break;
427116 }
427117 }
427118 //
427119 // If the buffer is an empty string, just loop to next
427120 // record.
427121 //
427122 if (strlen (buffer) == 0)
427123 {
427124 r--;
427125 continue;
427126 }
427127 //
427128 //
427129 //
427130 inittab_id = strtok (buffer, ":");
427131 inittab_runlevels = strtok (NULL, ":");

```

1875

```

4270132     inittab_action = strtok (NULL, "*");
4270133     inittab_process = strtok (NULL, "*");
4270134     //
4270135     // Only action 'respawn' is used.
4270136     //
4270137     if (strcmp (inittab_action, "respawn") == 0)
4270138     {
4270139         strncpy (respawn[r].command, inittab_process, COMMAND_MAX);
4270140     }
4270141     else
4270142     {
4270143         r--;
4270144     }
4270145     }
4270146     //
4270147     //
4270148     //
4270149 close (fd);
4270150 //
4270151 // Define common environment.
4270152 //
4270153 exec_envp[0] = "PATH=/bin:/usr/bin:/sbin:/usr/sbin";
4270154 exec_envp[1] = "CONSOLE=/dev/console";
4270155 exec_envp[2] = NULL;
4270156 //
4270157 // Start processes.
4270158 //
4270159 for (r = 0; r < RESPAWN_MAX; r++)
4270160 {
4270161     if (strlen (respawn[r].command) > 0)
4270162     {
4270163         respawn[r].pid = fork ();
4270164         if (respawn[r].pid == 0)
4270165         {
4270166             exec_argv[0] = strtok (respawn[r].command, " \\t");
4270167             exec_argv[1] = strtok (NULL, " \\t");
4270168             exec_argv[2] = NULL;
4270169             execve (exec_argv[0], exec_argv, exec_envp);
4270170             perror (NULL);
4270171             exit (0);
4270172         }
4270173     }
4270174 }
4270175 //
4270176 // Wait for the death of child.
4270177 //
4270178 while (1)
4270179 {
4270180     pid = wait (&status);
4270181     for (r = 0; r < RESPAWN_MAX; r++)
4270182     {
4270183         if (pid == respawn[r].pid)
4270184         {
4270185             //
4270186             // Run it again.
4270187             //
4270188             respawn[r].pid = fork ();
4270189             if (respawn[r].pid == 0)
4270190             {
4270191                 exec_argv[0] = strtok (respawn[r].command, " \\t");
4270192                 exec_argv[1] = strtok (NULL, " \\t");
4270193                 exec_argv[2] = NULL;
4270194                 execve (exec_argv[0], exec_argv, exec_envp);
4270195                 exit (0);
4270196             }
4270197             break;
4270198         }
4270199     }
4270200 }
4270201 }

```

```

4280027 //
4280028 // There must be at least an option, plus the program name.
4280029 //
4280030 if (argc < 2)
4280031 {
4280032     usage ();
4280033     return (1);
4280034 }
4280035 //
4280036 // Check for options.
4280037 //
4280038 while ((opt = getopt (argc, argv, "l:s:")) != -1)
4280039 {
4280040     switch (opt)
4280041     {
4280042     case 'l':
4280043         option_l = 1;
4280044         break;
4280045     case 's':
4280046         option_s = 1;
4280047         //
4280048         // In that case, there must be at least three arguments:
4280049         // the option, the signal and the process id.
4280050         //
4280051         if (argc < 4)
4280052         {
4280053             usage ();
4280054             return (1);
4280055         }
4280056         //
4280057         // Argument numbers are ok. Check the signal.
4280058         //
4280059         if (strcmp (optarg, "HUP") == 0)
4280060         {
4280061             signal = SIGHUP;
4280062         }
4280063         else if (strcmp (optarg, "INT") == 0)
4280064         {
4280065             signal = SIGINT;
4280066         }
4280067         else if (strcmp (optarg, "QUIT") == 0)
4280068         {
4280069             signal = SIGQUIT;
4280070         }
4280071         else if (strcmp (optarg, "ILL") == 0)
4280072         {
4280073             signal = SIGILL;
4280074         }
4280075         else if (strcmp (optarg, "ABRT") == 0)
4280076         {
4280077             signal = SIGABRT;
4280078         }
4280079         else if (strcmp (optarg, "FPE") == 0)
4280080         {
4280081             signal = SIGFPE;
4280082         }
4280083         else if (strcmp (optarg, "KILL") == 0)
4280084         {
4280085             signal = SIGKILL;
4280086         }
4280087         else if (strcmp (optarg, "SEGV") == 0)
4280088         {
4280089             signal = SIGSEGV;
4280090         }
4280091         else if (strcmp (optarg, "PIPE") == 0)
4280092         {
4280093             signal = SIGPIPE;
4280094         }
4280095         else if (strcmp (optarg, "ALRM") == 0)
4280096         {
4280097             signal = SIGALRM;
4280098         }
4280099         else if (strcmp (optarg, "TERM") == 0)
4280100         {
4280101             signal = SIGTERM;
4280102         }
4280103         else if (strcmp (optarg, "STOP") == 0)
4280104         {
4280105             signal = SIGSTOP;
4280106         }
4280107         else if (strcmp (optarg, "TSTP") == 0)
4280108         {
4280109             signal = SIGTSTP;
4280110         }
4280111         else if (strcmp (optarg, "CONT") == 0)
4280112         {
4280113             signal = SIGCONT;
4280114         }
4280115         else if (strcmp (optarg, "CHLD") == 0)
4280116         {
4280117             signal = SIGCHLD;
4280118         }
4280119         else if (strcmp (optarg, "TTIN") == 0)
4280120         {
4280121             signal = SIGTTIN;
4280122         }
4280123         else if (strcmp (optarg, "TTOU") == 0)
4280124         {
4280125             signal = SIGTTOU;
4280126         }
4280127         else if (strcmp (optarg, "USR1") == 0)

```

applic/kill.c

« Si veda la sezione u0.10.

```

4280001 #include <sys/os16.h>
4280002 #include <sys/stat.h>
4280003 #include <sys/types.h>
4280004 #include <unistd.h>
4280005 #include <stdlib.h>
4280006 #include <fcntl.h>
4280007 #include <errno.h>
4280008 #include <signal.h>
4280009 #include <stdio.h>
4280010 #include <string.h>
4280011 #include <limits.h>
4280012 #include <libgen.h>
4280013 //-----
4280014 static void usage (void);
4280015 //-----
4280016 int
4280017 main (int argc, char *argv[], char *envp[])
4280018 {
4280019     int     signal;
4280020     int     pid;
4280021     int     a; // Index inside arguments.
4280022     int     option_s = 0;
4280023     int     option_l = 0;
4280024     int     opt;
4280025     extern char *optarg;
4280026     extern int optopt;

```

```

4280128     {
4280129         signal = SIGUSR1;
4280130     }
4280131     else if (strcmp (optarg, "USR2") == 0)
4280132     {
4280133         signal = SIGUSR2;
4280134     }
4280135     else
4280136     {
4280137         fprintf (stderr, "Unknown signal %s.\n", optarg);
4280138         return (1);
4280139     }
4280140     break;
4280141     case '?':
4280142         fprintf (stderr, "Unknown option -%c.\n", optopt);
4280143         usage ();
4280144         return (1);
4280145     break;
4280146     case ':':
4280147         fprintf (stderr, "Missing argument for option -%c\n",
4280148                 optopt);
4280149         usage ();
4280150         return (1);
4280151     break;
4280152     default:
4280153         fprintf (stderr, "Getopt problem: unknown option %c\n",
4280154                 opt);
4280155         return (1);
4280156     }
4280157 }
4280158 //
4280159 //
4280160 //
4280161 if (option_l && option_s)
4280162 {
4280163     fprintf (stderr, "Options \"-l\" and \"-s\" together ");
4280164     fprintf (stderr, "are incompatible.\n");
4280165     usage ();
4280166     return (1);
4280167 }
4280168 //
4280169 // Option "-l".
4280170 //
4280171 if (option_l)
4280172 {
4280173     printf ("HUP ");
4280174     printf ("INT ");
4280175     printf ("QUIT ");
4280176     printf ("ILL ");
4280177     printf ("ABRT ");
4280178     printf ("FPE ");
4280179     printf ("KILL ");
4280180     printf ("SEGV ");
4280181     printf ("PIPE ");
4280182     printf ("ALRM ");
4280183     printf ("TERM ");
4280184     printf ("STOP ");
4280185     printf ("TSTP ");
4280186     printf ("CONT ");
4280187     printf ("CHLD ");
4280188     printf ("TTIN ");
4280189     printf ("TTOU ");
4280190     printf ("USR1 ");
4280191     printf ("USR2 ");
4280192     printf ("\n");
4280193 }
4280194 //
4280195 // Option "-s".
4280196 //
4280197 if (option_s)
4280198 {
4280199     //
4280200     // Scan arguments.
4280201     //
4280202     for (a = 3; a < argc; a++)
4280203     {
4280204         //
4280205         // Get PID.
4280206         //
4280207         pid = atoi (argv[a]);
4280208         if (pid > 0)
4280209         {
4280210             //
4280211             // Kill.
4280212             //
4280213             if (kill (pid, signal) < 0)
4280214             {
4280215                 perror (argv[a]);
4280216             }
4280217         }
4280218         else
4280219         {
4280220             fprintf (stderr, "Invalid PID %s.", argv[a]);
4280221         }
4280222     }
4280223 }
4280224 //
4280225 // All done.
4280226 //
4280227 return (0);
4280228 }

```

1878

```

4280229 //-----
4280230 static void
4280231 usage (void)
4280232 {
4280233     fprintf (stderr, "Usage: kill -s SIGNAL_NAME PID...\n");
4280234     fprintf (stderr, "        kill -l\n");
4280235 }

```

applic/ln.c

Si veda la sezione u0.11.

«

```

4280001 #include <sys/osal6.h>
4280002 #include <sys/stat.h>
4280003 #include <sys/types.h>
4280004 #include <unistd.h>
4280005 #include <stdlib.h>
4280006 #include <fcntl.h>
4280007 #include <errno.h>
4280008 #include <signal.h>
4280009 #include <stdio.h>
4280010 #include <string.h>
4280011 #include <limits.h>
4280012 #include <libgen.h>
4280013 //-----
4280014 static void usage (void);
4280015 //-----
4280016 int
4280017 main (int argc, char *argv[], char *envp[])
4280018 {
4280019     char *source;
4280020     char *destination;
4280021     char *new_destination;
4280022     struct stat file_status;
4280023     int dest_is_a_dir = 0;
4280024     int a; // Argument index.
4280025     char path[PATH_MAX];
4280026     //
4280027     // There must be at least two arguments, plus the program name.
4280028     //
4280029     if (argc < 3)
4280030     {
4280031         usage ();
4280032         return (1);
4280033     }
4280034     //
4280035     // Select the last argument as the destination.
4280036     //
4280037     destination = argv[argc-1];
4280038     //
4280039     // Check if it is a directory and save it in a flag.
4280040     //
4280041     if (stat (destination, &file_status) == 0)
4280042     {
4280043         if (S_ISDIR (file_status.st_mode))
4280044         {
4280045             dest_is_a_dir = 1;
4280046         }
4280047     }
4280048     //
4280049     // If there are more than two arguments, verify that the last
4280050     // one is a directory.
4280051     //
4280052     if (argc > 3)
4280053     {
4280054         if (!dest_is_a_dir)
4280055         {
4280056             usage ();
4280057             fprintf (stderr, "The destination \"%s\" ",
4280058                     destination);
4280059             fprintf (stderr, "is not a directory!\n");
4280060             return (1);
4280061         }
4280062     }
4280063     //
4280064     // Scan the arguments, excluded the last, that is the destination.
4280065     //
4280066     for (a = 1; a < (argc - 1); a++)
4280067     {
4280068         //
4280069         // Source.
4280070         //
4280071         source = argv[a];
4280072         //
4280073         // Verify access permissions.
4280074         //
4280075         if (access (source, R_OK) < 0)
4280076         {
4280077             perror (source);
4280078             continue;
4280079         }
4280080         //
4280081         // Destination.
4280082         //
4280083         // If it is a directory, the destination path
4280084         // must be corrected.
4280085         //
4280086         if (dest_is_a_dir)
4280087         {
4280088             path[0] = 0;
4280089             strcat (path, destination);

```

1879

```

428090     strcat (path, "/");
428091     strcat (path, basename (source));
428092     //
428093     // Update the destination path.
428094     //
428095     new_destination = path;
428096     }
428097     else
428098     {
428099         new_destination = destination;
428100     }
428101     //
428102     // Check if destination file exists.
428103     //
428104     if (stat (new_destination, &file_status) == 0)
428105     {
428106         fprintf (stderr, "The destination file, \"%s\", ",
428107                 new_destination);
428108         fprintf (stderr, "already exists!\n");
428109         continue;
428110     }
428111     //
428112     // Everything is ready for the link.
428113     //
428114     if (link (source, new_destination) < 0)
428115     {
428116         perror (new_destination);
428117         continue;
428118     }
428119     }
428120     //
428121     // All done.
428122     //
428123     return (0);
428124 }
428125 //-----
428126 static void
428127 usage (void)
428128 {
428129     fprintf (stderr, "Usage: ln OLD_NAME NEW_NAME\n");
428130     fprintf (stderr, "        ln FILE... DIRECTORY\n");
428131 }

```

applic/login.c

« Si veda la sezione u0.12.

```

430001 #include <unistd.h>
430002 #include <stdlib.h>
430003 #include <sys/stat.h>
430004 #include <sys/types.h>
430005 #include <fcntl.h>
430006 #include <errno.h>
430007 #include <unistd.h>
430008 #include <signal.h>
430009 #include <stdio.h>
430010 #include <sys/wait.h>
430011 #include <stdio.h>
430012 #include <string.h>
430013 #include <limits.h>
430014 #include <stdint.h>
430015 #include <sys/osl6.h>
430016 //-----
430017 #define LOGIN_MAX      64
430018 #define PASSWORD_MAX  64
430019 #define HOME_MAX      64
430020 #define LINE_MAX      1024
430021 //-----
430022 int
430023 main (int argc, char *argv[], char *envp[])
430024 {
430025     char    login[LOGIN_MAX];
430026     char    password[PASSWORD_MAX];
430027     char    buffer[LINE_MAX];
430028     char    *user_name;
430029     char    *user_password;
430030     char    *user_uid;
430031     char    *user_gid;
430032     char    *user_description;
430033     char    *user_home;
430034     char    *user_shell;
430035     uid_t   uid;
430036     uid_t   euid;
430037     int     fd;
430038     ssize_t size_read;
430039     int     b;           // Index inside buffer.
430040     int     loop;
430041     char    *exec_argv[2];
430042     int     status;
430043     char    *tty_path;
430044     //
430045     // Check if login is running correctly.
430046     //
430047     euid = geteuid ();
430048     uid  = geteuid ();
430049     // //
430050     // // Show process info.
430051     // //
430052     // heap_clear ();
430053     // process_info ();
430054     //

```

1880

```

430055 // Check privileges.
430056 //
430057 if (!(uid == 0 && euid == 0))
430058 {
430059     printf ("%s: can only run with root privileges!\n", argv[0]);
430060     exit (-1);
430061 }
430062 //
430063 // Prepare arguments for the shell call.
430064 //
430065 exec_argv[0] = "-";
430066 exec_argv[1] = NULL;
430067 //
430068 // Login.
430069 //
430070 while (1)
430071 {
430072     fd = open ("/etc/passwd", O_RDONLY);
430073     //
430074     if (fd < 0)
430075     {
430076         perror ("Cannot open file '/etc/passwd'");
430077         exit (-1);
430078     }
430079     //
430080     printf ("Log in as \"root\" or \"user\" "
430081            "with password \"ciao\" :-)\n");
430082     input_line (login, "login:", LOGIN_MAX, INPUT_LINE_ECHO);
430083     //
430084     //
430085     //
430086     loop = 1;
430087     while (loop)
430088     {
430089         for (b = 0; b < LINE_MAX; b++)
430090         {
430091             size_read = read (fd, &buffer[b], (size_t) 1);
430092             if (size_read <= 0)
430093             {
430094                 buffer[b] = 0;
430095                 loop = 0;           // Close the middle loop.
430096                 break;
430097             }
430098             if (buffer[b] == '\n')
430099             {
430100                 buffer[b] = 0;
430101                 break;
430102             }
430103         }
430104         //
430105         user_name   = strtok (buffer, ":");
430106         user_password = strtok (NULL, ":");
430107         user_uid     = strtok (NULL, ":");
430108         user_gid     = strtok (NULL, ":");
430109         user_description = strtok (NULL, ":");
430110         user_home    = strtok (NULL, ":");
430111         user_shell   = strtok (NULL, ":");
430112         //
430113         if (strcmp (user_name, login) == 0)
430114         {
430115             input_line (password, "password:", PASSWORD_MAX,
430116                       INPUT_LINE_STARS);
430117             //
430118             // Compare passwords: empty passwords are not allowed.
430119             //
430120             if (strcmp (user_password, password) == 0)
430121             {
430122                 uid = atoi (user_uid);
430123                 euid = uid;
430124                 //
430125                 // Find the controlling terminal and change
430126                 // property and access permissions.
430127                 //
430128                 tty_path = ttyname (STDIN_FILENO);
430129                 if (tty_path != NULL)
430130                 {
430131                     status = chown (tty_path, uid, 0);
430132                     if (status != 0)
430133                     {
430134                         perror (NULL);
430135                     }
430136                     status = chmod (tty_path, 0600);
430137                     if (status != 0)
430138                     {
430139                         perror (NULL);
430140                     }
430141                 }
430142                 //
430143                 // Cd to the home directory, if present.
430144                 //
430145                 status = chdir (user_home);
430146                 if (status != 0)
430147                 {
430148                     perror (NULL);
430149                 }
430150                 //
430151                 // Now change personality.
430152                 //
430153                 setuid (uid);
430154                 seteuid (euid);
430155                 //

```

1881

```

4300156 // Run the shell, replacing the login process; the
4300157 // environment is taken from 'init'.
4300158 //
4300159 execve (user_shell, exec_argv, envp);
4300160 exit (0);
4300161 }
4300162 //
4300163 // Login failed: will try again.
4300164 //
4300165 loop = 0; // Close the middle loop.
4300166 break;
4300167 }
4300168 }
4300169 close (fd);
4300170 }
4300171 }

```

applic/ls.c

« Si veda la sezione u0.13.

```

4310001 #include <sys/osi6.h>
4310002 #include <sys/stat.h>
4310003 #include <sys/types.h>
4310004 #include <unistd.h>
4310005 #include <stdlib.h>
4310006 #include <fcntl.h>
4310007 #include <errno.h>
4310008 #include <signal.h>
4310009 #include <stdio.h>
4310010 #include <string.h>
4310011 #include <limits.h>
4310012 #include <libgen.h>
4310013 #include <dirent.h>
4310014 #include <pwd.h>
4310015 #include <time.h>
4310016
4310017 #define BUFFER_SIZE 16384
4310018 #define LIST_SIZE 256
4310019
4310020 //-----
4310021 static void usage (void);
4310022 //-----
4310023 //-----
4310024 int compare (void *p1, void *p2);
4310025 //-----
4310026 int
4310027 main (int argc, char *argv[], char *envp[])
4310028 {
4310029     int option_a = 0;
4310030     int option_l = 0;
4310031     int opt;
4310032 // extern char *optarg; // not used.
4310033 extern int optind;
4310034 extern int optopt;
4310035 struct stat file_status;
4310036 DIR *dp;
4310037 struct dirent *dir;
4310038 char buffer[BUFFER_SIZE];
4310039 int b; // Buffer index.
4310040 char *list[LIST_SIZE];
4310041 int l; // List index.
4310042 int len; // Name length.
4310043 char *path = NULL;
4310044 char pathname[PATH_MAX];
4310045 struct passwd *pws;
4310046 struct tm *tms;
4310047 //
4310048 // Check for options.
4310049 //
4310050 while ((opt = getopt (argc, argv, "al*")) != -1)
4310051 {
4310052     switch (opt)
4310053     {
4310054     case 'l':
4310055         option_l = 1;
4310056         break;
4310057     case 'a':
4310058         option_a = 1;
4310059         break;
4310060     case '?':
4310061         fprintf (stderr, "Unknown option -%c.\n", optopt);
4310062         usage ();
4310063         return (1);
4310064         break;
4310065     case ':':
4310066         fprintf (stderr, "Missing argument for option -%c\n",
4310067                 optopt);
4310068         usage ();
4310069         return (1);
4310070         break;
4310071     default:
4310072         fprintf (stderr, "Getopt problem: unknown option %c\n",
4310073                 opt);
4310074         return (1);
4310075     }
4310076 }
4310077 //
4310078 // If no arguments are present, at least the current directory is
4310079 // read.
4310080 //

```

1882

```

4310081 if (optind == argc)
4310082 {
4310083     //
4310084     // There are no more arguments. Replace the program name,
4310085     // corresponding to 'argv[0]', with the current directory
4310086     // path string.
4310087     //
4310088     argv[0] = ".";
4310089     argc = 1;
4310090     optind = 0;
4310091 }
4310092 //
4310093 // This is a very simplified 'ls': if there is only a name
4310094 // and it is a directory, the directory content is taken as
4310095 // the new 'argv[]' array.
4310096 //
4310097 if (optind == (argc - 1))
4310098 {
4310099     //
4310100     // There is a request for a single name. Test if it exists
4310101     // and if it is a directory.
4310102     //
4310103     if (stat(argv[optind], &file_status) != 0)
4310104     {
4310105         fprintf (stderr, "File \"%s\" does not exist!\n",
4310106                 argv[optind]);
4310107         return (2);
4310108     }
4310109     //
4310110     if (S_ISDIR (file_status.st_mode))
4310111     {
4310112         //
4310113         // Save the directory inside the 'path' pointer.
4310114         //
4310115         path = argv[optind];
4310116         //
4310117         // Open the directory.
4310118         //
4310119         dp = opendir (argv[optind]);
4310120         if (dp == NULL)
4310121         {
4310122             perror (argv[optind]);
4310123             return (3);
4310124         }
4310125         //
4310126         // Read the directory and fill the buffer with names.
4310127         //
4310128         b = 0;
4310129         l = 0;
4310130         while ((dir = readdir (dp)) != NULL)
4310131         {
4310132             len = strlen (dir->d_name);
4310133             //
4310134             // Check if the buffer can hold it.
4310135             //
4310136             if ((b + len + 1) > BUFFER_SIZE)
4310137             {
4310138                 fprintf (stderr, "not enough memory\n");
4310139                 break;
4310140             }
4310141             //
4310142             // Consider the directory item only if there is
4310143             // a valid name. If it is empty, just ignore it.
4310144             //
4310145             if (len > 0)
4310146             {
4310147                 strcpy (&buffer[b], dir->d_name);
4310148                 list[l] = &buffer[b];
4310149                 b += len + 1;
4310150                 l++;
4310151             }
4310152         }
4310153         //
4310154         // Close the directory.
4310155         //
4310156         closedir (dp);
4310157         //
4310158         // Sort the list.
4310159         //
4310160         qsort (list, (size_t) l, sizeof (char *), compare);
4310161         //
4310162         //
4310163         // Convert the directory list into a new 'argv[]' array,
4310164         // with a valid 'argc'. The variable 'optind' must be
4310165         // reset to the first element index, because there is
4310166         // no program name inside the new 'argv[]' at index zero.
4310167         //
4310168         argv = list;
4310169         argc = l;
4310170         optind = 0;
4310171     }
4310172 }
4310173 //
4310174 // Scan arguments, or list converted into 'argv[]'.
4310175 //
4310176 for (; optind < argc; optind++)
4310177 {
4310178     if (argv[optind][0] == '.')
4310179     {
4310180         //
4310181         // Current name starts with '.'.

```

1883


```

4310182 //
4310183 if (!option_a)
4310184 {
4310185     //
4310186     // Do not show name starting with `.`.
4310187     //
4310188     continue;
4310189 }
4310190 }
4310191 //
4310192 // Build the pathname.
4310193 //
4310194 if (path == NULL)
4310195 {
4310196     strcpy (&pathname[0], argv[optind]);
4310197 }
4310198 else
4310199 {
4310200     strcpy (pathname, path);
4310201     strcat (pathname, "/");
4310202     strcat (pathname, argv[optind]);
4310203 }
4310204 //
4310205 // Check if file exists, reading status.
4310206 //
4310207 if (stat(pathname, &file_status) != 0)
4310208 {
4310209     fprintf (stderr, "File \"%s\" does not exist!\n",
4310210             pathname);
4310211     return (2);
4310212 }
4310213 //
4310214 // Show file name.
4310215 //
4310216 if (option_l)
4310217 {
4310218     //
4310219     // Print the file type.
4310220     //
4310221     if (S_ISBLK (file_status.st_mode)) printf ("b");
4310222     else if (S_ISCHR (file_status.st_mode)) printf ("c");
4310223     else if (S_ISFIFO (file_status.st_mode)) printf ("p");
4310224     else if (S_ISREG (file_status.st_mode)) printf ("-");
4310225     else if (S_ISDIR (file_status.st_mode)) printf ("d");
4310226     else if (S_ISLNK (file_status.st_mode)) printf ("l");
4310227     else if (S_ISOCK (file_status.st_mode)) printf ("s");
4310228     else printf ("?");
4310229     //
4310230     // Print permissions.
4310231     //
4310232     if (S_IRUSR & file_status.st_mode) printf ("r");
4310233     else printf ("-");
4310234     if (S_IWUSR & file_status.st_mode) printf ("w");
4310235     else printf ("-");
4310236     if (S_IXUSR & file_status.st_mode) printf ("x");
4310237     else printf ("-");
4310238     if (S_IRGRP & file_status.st_mode) printf ("r");
4310239     else printf ("-");
4310240     if (S_IWGRP & file_status.st_mode) printf ("w");
4310241     else printf ("-");
4310242     if (S_IXGRP & file_status.st_mode) printf ("x");
4310243     else printf ("-");
4310244     if (S_IROTH & file_status.st_mode) printf ("r");
4310245     else printf ("-");
4310246     if (S_IWOTH & file_status.st_mode) printf ("w");
4310247     else printf ("-");
4310248     if (S_IXOTH & file_status.st_mode) printf ("x");
4310249     else printf ("-");
4310250     //
4310251     // Print links.
4310252     //
4310253     printf (" %3i", (int) file_status.st_nlink);
4310254     //
4310255     // Print owner.
4310256     //
4310257     pws = getpwuid (file_status.st_uid);
4310258     //
4310259     printf (" %s", pws->pw_name);
4310260     //
4310261     // Print group (no group available);
4310262     //
4310263     printf (" (no group)");
4310264     //
4310265     // Print file size or device major-minor.
4310266     //
4310267     if (S_ISBLK (file_status.st_mode)
4310268         || S_ISCHR (file_status.st_mode))
4310269     {
4310270         printf (" %3i", (int) major (file_status.st_rdev));
4310271         printf (" %3i", (int) minor (file_status.st_rdev));
4310272     }
4310273     else
4310274     {
4310275         printf (" %8i", (int) file_status.st_size);
4310276     }
4310277     //
4310278     // Print modification date and time.
4310279     //
4310280     tms = localtime (&(file_status.st_mtime));
4310281     printf (" %4u-%02u-%02u %02u:%02u",
4310282             tms->tm_year, tms->tm_mon, tms->tm_mday,

```

1884

```

4310283             tms->tm_hour, tms->tm_min);
4310284     //
4310285     // Print file name, but with no additional path.
4310286     //
4310287     printf (" %s\n", argv[optind]);
4310288 }
4310289 else
4310290 {
4310291     //
4310292     // Just show the file name and go to the next line.
4310293     //
4310294     printf ("%s\n", argv[optind]);
4310295 }
4310296 }
4310297 //
4310298 // All done.
4310299 //
4310300 return (0);
4310301 }
4310302 //-----
4310303 static void
4310304 usage (void)
4310305 {
4310306     fprintf (stderr, "Usage: ls [OPTION] [FILE]...\n");
4310307 }
4310308 //-----
4310309 int
4310310 compare (void *p1, void *p2)
4310311 {
4310312     char **pp1 = p1;
4310313     char **pp2 = p2;
4310314     //
4310315     return (strcmp (*pp1, *pp2));
4310316 }
4310317

```

applic/man.c

Si veda la sezione u0.14.

«

```

4320001 #include <unistd.h>
4320002 #include <stdlib.h>
4320003 #include <errno.h>
4320004 //-----
4320005 #define MAX_LINES 20
4320006 #define MAX_COLUMNS 80
4320007 //-----
4320008 static char *man_page_directory = "/usr/share/man";
4320009 //-----
4320010 static void usage (void);
4320011 static FILE *open_man_page (int section, char *name);
4320012 static void build_path_name (int section, char *name, char *path);
4320013 //-----
4320014 int
4320015 main (int argc, char *argv[], char *envp[])
4320016 {
4320017     FILE *fp;
4320018     char *name;
4320019     int section;
4320020     int c;
4320021     int line = 1; // Line internal counter.
4320022     int column = 1; // Column internal counter.
4320023     int loop;
4320024     //
4320025     // There must be minimum an argument, and maximum two.
4320026     //
4320027     if (argc < 2 || argc > 3)
4320028     {
4320029         usage ();
4320030         return (1);
4320031     }
4320032     //
4320033     // If there are two arguments, there must be the
4320034     // section number.
4320035     //
4320036     if (argc == 3)
4320037     {
4320038         section = atoi (argv[1]);
4320039         name = argv[2];
4320040     }
4320041     else
4320042     {
4320043         section = 0;
4320044         name = argv[1];
4320045     }
4320046     //
4320047     // Try to open the manual page.
4320048     //
4320049     fp = open_man_page (section, name);
4320050     //
4320051     if (fp == NULL)
4320052     {
4320053         //
4320054         // Error opening file.
4320055         //
4320056         return (1);
4320057     }
4320058     //
4320059     //
4320060     // The following loop continues while the file
4320061     // gives characters, or when a command to change

```

1885

```

432062 // file or to quit is given.
432063 //
432064 for (loop = 1; loop; )
432065 {
432066 //
432067 // Read a single character.
432068 //
432069 c = getc (fp);
432070 //
432071 if (c == EOF)
432072 {
432073     loop = 0;
432074     break;
432075 }
432076 //
432077 // If the character read is a special one,
432078 // the line/column calculation is modified,
432079 // so that it is known when to stop scrolling.
432080 //
432081 switch (c)
432082 {
432083     case '\r':
432084         //
432085         // Displaying this character, the cursor should go
432086         // back to the first column. So the column counter
432087         // is reset.
432088         //
432089         column = 1;
432090         break;
432091     case '\n':
432092         //
432093         // Displaying this character, the cursor should go
432094         // back to the next line, at the first column.
432095         // So the column counter is reset and the line
432096         // counter is incremented.
432097         //
432098         line++;
432099         column = 1;
432100         break;
432101     case '\b':
432102         //
432103         // Displaying this character, the cursor should go
432104         // back one position, unless it is already at the
432105         // beginning.
432106         //
432107         if (column > 1)
432108             {
432109                 column--;
432110             }
432111         break;
432112     default:
432113         //
432114         // Any other character must increase the column
432115         // counter.
432116         //
432117         column++;
432118     }
432119 //
432120 // Display the character, even if it is a special one:
432121 // it is responsibility of the screen device management
432122 // to do something good with special characters.
432123 //
432124 putchar (c);
432125 //
432126 // If the column counter is gone beyond the screen columns,
432127 // then adjust the column counter and increment the line
432128 // counter.
432129 //
432130 if (column > MAX_COLUMNS)
432131 {
432132     column -= MAX_COLUMNS;
432133     line++;
432134 }
432135 //
432136 // Check if there is space for scrolling.
432137 //
432138 if (line < MAX_LINES)
432139 {
432140     continue;
432141 }
432142 //
432143 // Here, displayed lines are MAX_LINES.
432144 //
432145 if (column > 1)
432146 {
432147     //
432148     // Something was printed at the current line: must
432149     // do a new line.
432150     //
432151     putchar ('\n');
432152 }
432153 //
432154 // Show the more prompt.
432155 //
432156 printf ("--More--");
432157 fflush (stdout);
432158 //
432159 // Read a character from standard input.
432160 //
432161 c = getchar ();
432162 //

```

1886

```

432063 // Consider command 'q', but any other character
432064 // can be introduced, to let show the next page.
432065 //
432066 switch (c)
432067 {
432068     case 'Q':
432069     case 'q':
432070         //
432071         // Quit. But must erase the '--More--' prompt.
432072         //
432073         printf ("\b \b\b \b\b \b\b \b\b \b");
432074         printf ("\b \b\b \b\b \b\b \b");
432075         fclose (fp);
432076         return (0);
432077     }
432078 //
432079 // Backspace to overwrite '--More--' and the character
432080 // pressed.
432081 //
432082 printf ("\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b");
432083 //
432084 // Reset line/column counters.
432085 //
432086 column = 1;
432087 line = 1;
432088 }
432089 //
432090 // Close the file pointer if it is still open.
432091 //
432092 if (fp != NULL)
432093 {
432094     fclose (fp);
432095 }
432096 //
432097 return (0);
432098 }
432099 -----
432100 static void
432101 usage (void)
432102 {
432103     fprintf (stderr, "Usage: man [SECTION] NAME\n");
432104 }
432105 -----
432106 FILE *
432107 open_man_page (int section, char *name)
432108 {
432109     FILE *fp;
432110     char path[PATH_MAX];
432111     struct stat file_status;
432112     //
432113     //
432114     //
432115     if (section > 0)
432116     {
432117         build_path_name (section, name, path);
432118         //
432119         // Check if file exists.
432120         //
432121         if (stat (path, &file_status) != 0)
432122             {
432123                 fprintf (stderr, "Man page %s(%i) does not exist!\n",
432124                     name, section);
432125                 return (NULL);
432126             }
432127     }
432128     else
432129     {
432130         //
432131         // Must try a section.
432132         //
432133         for (section = 1; section < 9; section++)
432134             {
432135                 build_path_name (section, name, path);
432136                 //
432137                 // Check if file exists.
432138                 //
432139                 if (stat (path, &file_status) == 0)
432140                     {
432141                         //
432142                         // Found.
432143                         //
432144                         break;
432145                     }
432146             }
432147     }
432148     //
432149     // Check if a file was found.
432150     //
432151     if (section < 9)
432152     {
432153         fp = fopen (path, "r");
432154         //
432155         if (fp == NULL)
432156             {
432157                 //
432158                 // Error opening file.
432159                 //
432160                 perror (path);
432161                 return (NULL);
432162             }
432163     }
432164     else

```

1887

```

4320264     {
4320265         //
4320266         // Opened right.
4320267         //
4320268         return (fp);
4320269     }
4320270 }
4320271 else
4320272 {
4320273     fprintf (stderr, "Man page %s does not exist!\n",
4320274             name);
4320275     return (NULL);
4320276 }
4320277 }
4320278 //-----
4320279 void
4320280 build_path_name (int section, char *name, char *path)
4320281 {
4320282     char string_section[10];
4320283     //
4320284     // Convert the section number into a string.
4320285     //
4320286     sprintf (string_section, "%i", section);
4320287     //
4320288     // Prepare the path to the man file.
4320289     //
4320290     path[0] = 0;
4320291     strcat (path, man_page_directory);
4320292     strcat (path, "/");
4320293     strcat (path, name);
4320294     strcat (path, ".");
4320295     strcat (path, string_section);
4320296 }

```

applic/mkdir.c

« Si veda la sezione u0.15.

```

4330001 #include <sys/osal6.h>
4330002 #include <sys/stat.h>
4330003 #include <sys/types.h>
4330004 #include <unistd.h>
4330005 #include <stdlib.h>
4330006 #include <fcntl.h>
4330007 #include <errno.h>
4330008 #include <signal.h>
4330009 #include <stdio.h>
4330010 #include <string.h>
4330011 #include <limits.h>
4330012 #include <libgen.h>
4330013 //-----
4330014 static int mkdir_parents (const char *path, mode_t mode);
4330015 static void usage (void);
4330016 //-----
4330017 int
4330018 main (int argc, char *argv[], char *envp[])
4330019 {
4330020     sysmsg_uarea_t msg;
4330021     int status;
4330022     mode_t mode = 0;
4330023     int m; // Index inside mode argument.
4330024     int digit;
4330025     char **dir;
4330026     int d; // Directory index.
4330027     int option_p = 0;
4330028     int option_m = 0;
4330029     int opt;
4330030     extern char *optarg;
4330031     extern int optind;
4330032     extern int optopt;
4330033     //
4330034     // There must be at least an argument, plus the program name.
4330035     //
4330036     if (argc < 2)
4330037     {
4330038         usage ();
4330039         return (1);
4330040     }
4330041     //
4330042     // Check for options, starting from 'p'. The 'dir' pointer is used
4330043     // to calculate the argument pointer to the first directory [1].
4330044     // The macro-instruction 'max()' is declared inside <sys/osal6.h>
4330045     // and does the expected thing.
4330046     //
4330047     while ((opt = getopt (argc, argv, "pm:")) != -1)
4330048     {
4330049         switch (opt)
4330050         {
4330051             case 'm':
4330052                 option_m = 1;
4330053                 for (m = 0; m < strlen (optarg); m++)
4330054                 {
4330055                     digit = (optarg[m] - '0');
4330056                     if (digit < 0 || digit > 7)
4330057                     {
4330058                         usage ();
4330059                         return (2);
4330060                     }
4330061                     mode = mode * 8 + digit;
4330062                 }
4330063                 break;

```

1888

```

4330064         case 'p':
4330065             option_p = 1;
4330066             break;
4330067         case '?':
4330068             printf ("Unknown option -%c.\n", optopt);
4330069             usage ();
4330070             return (1);
4330071             break;
4330072         case ':':
4330073             printf ("Missing argument for option -%c.\n", optopt);
4330074             usage ();
4330075             return (2);
4330076             break;
4330077         default:
4330078             printf ("Getopt problem: unknown option %c.\n", opt);
4330079             return (3);
4330080     }
4330081 }
4330082 //
4330083 dir = argv + optind;
4330084 //
4330085 // Check if the mode is to be set to a default value.
4330086 //
4330087 if (!option_m)
4330088 {
4330089     //
4330090     // Default mode.
4330091     //
4330092     sys (SYS_UAREA, &msg, (sizeof msg));
4330093     mode = 0777 && ~msg.umask;
4330094 }
4330095 //
4330096 // Directory creation.
4330097 //
4330098 for (d = 0; dir[d] != NULL; d++)
4330099 {
4330100     if (option_p)
4330101     {
4330102         status = mkdir_parents (dir[d], mode);
4330103         if (status != 0)
4330104         {
4330105             perror (dir[d]);
4330106             return (3);
4330107         }
4330108     }
4330109     else
4330110     {
4330111         status = mkdir (dir[d], mode);
4330112         if (status != 0)
4330113         {
4330114             perror (dir[d]);
4330115             return (4);
4330116         }
4330117     }
4330118 }
4330119 //
4330120 // All done.
4330121 //
4330122 return (0);
4330123 }
4330124 //-----
4330125 static int
4330126 mkdir_parents (const char *path, mode_t mode)
4330127 {
4330128     char path_copy[PATH_MAX];
4330129     char *path_parent;
4330130     struct stat fst;
4330131     int status;
4330132     //
4330133     // Check if the path is empty.
4330134     //
4330135     if (path == NULL || strlen (path) == 0)
4330136     {
4330137         //
4330138         // Recursion ends here.
4330139         //
4330140         return (0);
4330141     }
4330142     //
4330143     // Check if it does already exist.
4330144     //
4330145     status = stat (path, &fst);
4330146     if (status == 0 && fst.st_mode & S_IFDIR)
4330147     {
4330148         //
4330149         // The path exists and is a directory.
4330150         //
4330151         return (0);
4330152     }
4330153     else if (status == 0 && !(fst.st_mode & S_IFDIR))
4330154     {
4330155         //
4330156         // The path exists but is not a directory.
4330157         //
4330158         errno = ENOTDIR; // Not a directory.
4330159         return (-1);
4330160     }
4330161     //
4330162     // Get the directory path.
4330163     //
4330164     strncpy (path_copy, path, PATH_MAX);

```

1889

```

4330165 path_parent = dirname (path_copy);
4330166 //
4330167 // If it is '.', or '/', the recursion is terminated.
4330168 //
4330169 if (strncmp (path_parent, ".", PATH_MAX) == 0 ||
4330170     strncmp (path_parent, "/", PATH_MAX) == 0)
4330171     {
4330172         return (0);
4330173     }
4330174 //
4330175 // Otherwise, continue the recursion.
4330176 //
4330177 status = mkdir_parents (path_parent, mode);
4330178 if (status != 0)
4330179     {
4330180         return (-1);
4330181     }
4330182 //
4330183 // Previous directories are there: create the current one.
4330184 //
4330185 status = mkdir (path, mode);
4330186 if (status)
4330187     {
4330188         perror (path);
4330189         return (-1);
4330190     }
4330191
4330192 return (0);
4330193 }
4330194 //-----
4330195 static void
4330196 usage (void)
4330197 {
4330198     fprintf (stderr, "Usage: mkdir [-p] [-m OCTAL_MODE] DIR...\n");
4330199 }

```

applic/more.c

« Si veda la sezione u0.16.

```

4340001 #include <unistd.h>
4340002 #include <errno.h>
4340003 //-----
4340004 #define MAX_LINES 20
4340005 #define MAX_COLUMNS 80
4340006 //-----
4340007 static void usage (void);
4340008 //-----
4340009 int
4340010 main (int argc, char *argv[], char *envp[])
4340011 {
4340012     FILE *fp;
4340013     char *name;
4340014     int c;
4340015     int line = 1; // Line internal counter.
4340016     int column = 1; // Column internal counter.
4340017     int a; // Index inside arguments.
4340018     int loop;
4340019 //
4340020 // There must be at least an argument, plus the program name.
4340021 //
4340022 if (argc < 2)
4340023     {
4340024         usage ();
4340025         return (1);
4340026     }
4340027 //
4340028 // No options are allowed.
4340029 //
4340030 for (a = 1; a < argc; a++)
4340031     {
4340032         //
4340033         // Get next name from arguments.
4340034         //
4340035         name = argv[a];
4340036         //
4340037         // Try to open the file, read only.
4340038         //
4340039         fp = fopen (name, "r");
4340040         //
4340041         if (fp == NULL)
4340042             {
4340043                 //
4340044                 // Error opening file.
4340045                 //
4340046                 perror (name);
4340047                 return (1);
4340048             }
4340049         //
4340050         // Print the file name to be displayed.
4340051         //
4340052         printf ("== %s ==\n", name);
4340053         line++;
4340054         //
4340055         // The following loop continues while the file
4340056         // gives characters, or when a command to change
4340057         // file or to quit is given.
4340058         //
4340059         for (loop = 1; loop; )
4340060             {
4340061                 //

```

1890

```

4340062 // Read a single character.
4340063 //
4340064 c =getc (fp);
4340065 //
4340066 if (c == EOF)
4340067     {
4340068         loop = 0;
4340069         break;
4340070     }
4340071 //
4340072 // If the character read is a special one,
4340073 // the line/column calculation is modified,
4340074 // so that it is known when to stop scrolling.
4340075 //
4340076 switch (c)
4340077     {
4340078     case '\r':
4340079         //
4340080         // Displaying this character, the cursor should go
4340081         // back to the first column. So the column counter
4340082         // is reset.
4340083         //
4340084         column = 1;
4340085         break;
4340086     case '\n':
4340087         //
4340088         // Displaying this character, the cursor should go
4340089         // back to the next line, at the first column.
4340090         // So the column counter is reset and the line
4340091         // counter is incremented.
4340092         //
4340093         line++;
4340094         column = 1;
4340095         break;
4340096     case '\b':
4340097         //
4340098         // Displaying this character, the cursor should go
4340099         // back one position, unless it is already at the
4340100         // beginning.
4340101         //
4340102         if (column > 1)
4340103             {
4340104                 column--;
4340105             }
4340106         break;
4340107     default:
4340108         //
4340109         // Any other character must increase the column
4340110         // counter.
4340111         //
4340112         column++;
4340113     }
4340114 //
4340115 // Display the character, even if it is a special one:
4340116 // it is responsibility of the screen device management
4340117 // to do something good with special characters.
4340118 //
4340119 putchar (c);
4340120 //
4340121 // If the column counter is gone beyond the screen columns,
4340122 // then adjust the column counter and increment the line
4340123 // counter.
4340124 //
4340125 if (column > MAX_COLUMNS)
4340126     {
4340127         column = MAX_COLUMNS;
4340128         line++;
4340129     }
4340130 //
4340131 // Check if there is space for scrolling.
4340132 //
4340133 if (line < MAX_LINES)
4340134     {
4340135         continue;
4340136     }
4340137 //
4340138 // Here, displayed lines are MAX_LINES.
4340139 //
4340140 if (column > 1)
4340141     {
4340142         //
4340143         // Something was printed at the current line: must
4340144         // do a new line.
4340145         //
4340146         putchar ('\n');
4340147     }
4340148 //
4340149 // Show the more prompt.
4340150 //
4340151 printf ("--More--");
4340152 fflush (stdout);
4340153 //
4340154 // Read a character from standard input.
4340155 //
4340156 c = getchar ();
4340157 //
4340158 // Consider commands 'n' and 'q', but any other character
4340159 // can be introduced, to let show the next page.
4340160 //
4340161 switch (c)
4340162     {

```

1891

```

4340163         case 'N':
4340164         case 'n':
4340165             //
4340166             // Go to the next file, if any.
4340167             //
4340168             fclose (fp);
4340169             fp = NULL;
4340170             loop = 0;
4340171             break;
4340172         case 'Q':
4340173         case 'q':
4340174             //
4340175             // Quit. But must erase the "--More--" prompt.
4340176             //
4340177             printf ("\b \b\b \b\b \b\b \b\b \b");
4340178             printf ("\b \b\b \b\b \b\b \b\b \b");
4340179             fclose (fp);
4340180             return (0);
4340181         }
4340182         //
4340183         // Backspace to overwrite "--More--" and the character
4340184         // pressed.
4340185         //
4340186         printf ("\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b");
4340187         //
4340188         // Reset line/column counters.
4340189         //
4340190         column = 1;
4340191         line = 1;
4340192     }
4340193     //
4340194     // Close the file pointer if it is still open.
4340195     //
4340196     if (fp != NULL)
4340197     {
4340198         fclose (fp);
4340199     }
4340200 }
4340201 //
4340202 return (0);
4340203 }
4340204 //-----
4340205 static void
4340206 usage (void)
4340207 {
4340208     fprintf (stderr, "Usage: more FILE...\n");
4340209 }

```

applic/mount.c

Si veda la sezione u0.4.

```

4350001 #include <unistd.h>
4350002 #include <stdlib.h>
4350003 #include <sys/stat.h>
4350004 #include <sys/types.h>
4350005 #include <fcntl.h>
4350006 #include <errno.h>
4350007 #include <signal.h>
4350008 #include <stdio.h>
4350009 #include <sys/wait.h>
4350010 #include <stdio.h>
4350011 #include <string.h>
4350012 #include <limits.h>
4350013 #include <sys/osi6.h>
4350014 //-----
4350015 static void usage (void);
4350016 //-----
4350017 int
4350018 main (int argc, char *argv[], char *envp[])
4350019 {
4350020     int options;
4350021     int status;
4350022     //
4350023     //
4350024     //
4350025     if (argc < 3 || argc > 4)
4350026     {
4350027         usage ();
4350028         return (1);
4350029     }
4350030     //
4350031     // Set options.
4350032     //
4350033     if (argc == 4)
4350034     {
4350035         if (strcmp (argv[3], "rw") == 0)
4350036         {
4350037             options = MOUNT_DEFAULT;
4350038         }
4350039         else if (strcmp (argv[3], "ro") == 0)
4350040         {
4350041             options = MOUNT_RO;
4350042         }
4350043         else
4350044         {
4350045             printf ("Invalid mount option: only \"ro\" or \"rw\" "
4350046                 "are allowed\n");
4350047             return (2);
4350048         }
4350049     }

```

```

4350050     else
4350051     {
4350052         options = MOUNT_DEFAULT;
4350053     }
4350054     //
4350055     // System call.
4350056     //
4350057     status = mount (argv[1], argv[2], options);
4350058     if (status != 0)
4350059     {
4350060         perror (NULL);
4350061         return (2);
4350062     }
4350063     //
4350064     return (0);
4350065 }
4350066 //-----
4350067 static void
4350068 usage (void)
4350069 {
4350070     fprintf (stderr, "Usage: mount DEVICE MOUNT_POINT "
4350071             "[MOUNT_OPTIONS]\n");
4350072 }

```

applic/ps.c

Si veda la sezione u0.17.

```

4360001 #include <kernel/proc.h>
4360002 #include <unistd.h>
4360003 #include <stdio.h>
4360004 #include <fcntl.h>
4360005 #include <unistd.h>
4360006 #include <stdlib.h>
4360007 //-----
4360008 void
4360009 print_proc_head (void)
4360010 {
4360011     printf (
4360012         "pp p ps
4360013         *id id rp tty uid euid suid usage s iaddr isiz daddr dsiz sp name\n"
4360014         );
4360015 }
4360016 //-----
4360017 void
4360018 print_proc_pid (proc_t *ps, pid_t pid)
4360019 {
4360020     char stat;
4360021     switch (ps->status)
4360022     {
4360023         case PROC_EMPTY : stat = '-'; break;
4360024         case PROC_CREATED : stat = 'c'; break;
4360025         case PROC_READY : stat = 'r'; break;
4360026         case PROC_RUNNING : stat = 'R'; break;
4360027         case PROC_SLEEPING : stat = 's'; break;
4360028         case PROC_ZOMBIE : stat = 'z'; break;
4360029         default : stat = '?'; break;
4360030     }
4360031 }
4360032 printf ("%2i %2i %2i %04x %4i %4i %4i %02i.%02i %c %05lx %04x ",
4360033         (unsigned int) ps->ppid,
4360034         (unsigned int) pid,
4360035         (unsigned int) ps->pgrp,
4360036         (unsigned int) ps->xdevice_tty,
4360037         (unsigned int) ps->uid,
4360038         (unsigned int) ps->euid,
4360039         (unsigned int) ps->suid,
4360040         (unsigned int) ((ps->usage / CLOCKS_PER_SEC) / 60),
4360041         (unsigned int) ((ps->usage / CLOCKS_PER_SEC) % 60),
4360042         stat,
4360043         (unsigned long int) ps->address_i,
4360044         (unsigned int) ps->size_i);
4360045 }
4360046 printf ("%05lx %04x %04x %s",
4360047         (unsigned long int) ps->address_d,
4360048         (unsigned int) ps->size_d,
4360049         (unsigned int) ps->sp,
4360050         ps->xname);
4360051 }
4360052 printf ("\n");
4360053 }
4360054 //-----
4360055 int
4360056 main (void)
4360057 {
4360058     pid_t pid;
4360059     proc_t *ps;
4360060     int fd;
4360061     ssize_t size_read;
4360062     char buffer[sizeof (proc_t)];
4360063 }
4360064 fd = open ("/dev/kmem_ps", O_RDONLY);
4360065 if (fd < 0)
4360066 {
4360067     perror ("ps: cannot open \"/dev/kmem_ps\"");
4360068     exit (0);
4360069 }
4360070 print_proc_head ();
4360071 for (pid = 0; pid < PROCESS_MAX; pid++)
4360072 {

```

```

436074     lseek (fd, (off_t) pid, SEEK_SET);
436075     size_read = read (fd, buffer, sizeof (proc_t));
436076     if (size_read < sizeof (proc_t))
436077     {
436078         printf ("ps: cannot read \"dev/kmem_ps\" pid %i", pid);
436079         perror (NULL);
436080         continue;
436081     }
436082     ps = (proc_t *) buffer;
436083     if (ps->status > 0)
436084     {
436085         ps->name[PATH_MAX-1] = 0; // Terminated string.
436086         print_proc_pid (ps, pid);
436087     }
436088 }
436089
436090     close (fd);
436091     return (0);
436092 }

```

applic/rm.c

<

Si veda la sezione u0.18.

```

437001 #include <fcntl.h>
437002 #include <sys/stat.h>
437003 #include <stddef.h>
437004 #include <unistd.h>
437005 #include <errno.h>
437006 //-----
437007 static void usage (void);
437008 //-----
437009 int
437010 main (int argc, char *argv[], char *envp[])
437011 {
437012     int a; // Argument index.
437013     int status;
437014     struct stat file_status;
437015     //
437016     // No options are known, but at least an argument must be given.
437017     //
437018     if (argc < 2)
437019     {
437020         usage ();
437021         return (1);
437022     }
437023     //
437024     // Scan arguments.
437025     //
437026     for(a = 1; a < argc; a++)
437027     {
437028         //
437029         // Verify if the file exists.
437030         //
437031         if (stat(argv[a], &file_status) != 0)
437032         {
437033             fprintf (stderr, "File \"%s\" does not exist!\n",
437034                     argv[a]);
437035             continue;
437036         }
437037         //
437038         // File exists: check the file type.
437039         //
437040         if (S_ISDIR (file_status.st_mode))
437041         {
437042             fprintf (stderr, "Cannot remove directory \"%s\"!\n",
437043                     argv[a]);
437044             continue;
437045         }
437046         //
437047         // Can remove it.
437048         //
437049         status = unlink (argv[a]);
437050         if (status != 0)
437051         {
437052             perror (NULL);
437053             return (2);
437054         }
437055     }
437056     return (0);
437057 }
437058 //-----
437059 static void
437060 usage (void)
437061 {
437062     fprintf (stderr, "Usage: rm FILE...\n");
437063 }

```

applic/shell.c

<

Si veda la sezione u0.19.

```

438001 #include <unistd.h>
438002 #include <stdlib.h>
438003 #include <sys/stat.h>
438004 #include <sys/types.h>
438005 #include <fcntl.h>
438006 #include <errno.h>
438007 #include <unistd.h>
438008 #include <signal.h>

```

```

438009 #include <stdio.h>
438010 #include <sys/wait.h>
438011 #include <stdio.h>
438012 #include <string.h>
438013 #include <limits.h>
438014 #include <sys/posix.h>
438015 //-----
438016 #define PROMPT_SIZE 16
438017 //-----
438018 static void sh_cd (int argc, char *argv[]);
438019 static void sh_pwd (int argc, char *argv[]);
438020 static void sh_umask (int argc, char *argv[]);
438021 //-----
438022 int
438023 main (int argc, char *argv[], char *envp[])
438024 {
438025     char buffer_cmd[ARG_MAX/2];
438026     char *argv_cmd[ARG_MAX/16];
438027     char prompt[PROMPT_SIZE];
438028     uid_t uid;
438029     int argc_cmd;
438030     pid_t pid_cmd;
438031     pid_t pid_dead;
438032     int status;
438033     //
438034     //
438035     //
438036     uid = getuid ();
438037     //
438038     // Load processes, reading the keyboard.
438039     //
438040     while (1)
438041     {
438042         if (uid == 0)
438043         {
438044             strncpy (prompt, "# ", PROMPT_SIZE);
438045         }
438046         else
438047         {
438048             strncpy (prompt, "$ ", PROMPT_SIZE);
438049         }
438050         //
438051         input_line (buffer_cmd, prompt, (ARG_MAX/2), INPUT_LINE_ECHO);
438052         //
438053         // Clear 'argv_cmd[]';
438054         //
438055         for (argc_cmd = 0; argc_cmd < (ARG_MAX/16); argc_cmd++)
438056         {
438057             argv_cmd[argc_cmd] = NULL;
438058         }
438059         //
438060         // Initialize the command scan.
438061         //
438062         argv_cmd[0] = strtok (buffer_cmd, " \t");
438063         //
438064         // Verify: if the input is not valid, loop again.
438065         //
438066         if (argv_cmd[0] == NULL)
438067         {
438068             continue;
438069         }
438070         //
438071         // Find the arguments.
438072         //
438073         for (argc_cmd = 1;
438074              argc_cmd < ((ARG_MAX/16)-1) && argv_cmd[argc_cmd-1] != NULL;
438075              argc_cmd++)
438076         {
438077             argv_cmd[argc_cmd] = strtok (NULL, " \t");
438078         }
438079         //
438080         // If there are too many arguments, show a message and continue.
438081         //
438082         if (argc_cmd[argc_cmd-1] != NULL)
438083         {
438084             errset (E2BIG); // Argument list too long.
438085             perror (NULL);
438086             continue;
438087         }
438088         //
438089         // Correct the value for 'argc_cmd', because actually
438090         // it counts also the NULL element.
438091         //
438092         argc_cmd--;
438093         //
438094         // Verify if it is an internal command.
438095         //
438096         if (strcmp (argv_cmd[0], "exit") == 0)
438097         {
438098             return (0);
438099         }
438100         else if (strcmp (argv_cmd[0], "cd") == 0)
438101         {
438102             sh_cd (argc_cmd, argv_cmd);
438103             continue;
438104         }
438105         else if (strcmp (argv_cmd[0], "pwd") == 0)
438106         {
438107             sh_pwd (argc_cmd, argv_cmd);
438108             continue;
438109         }

```

```

438010     else if (strcmp (argv_cmd[0], "umask") == 0)
438011     {
438012         sh_umask (argc_cmd, argv_cmd);
438013         continue;
438014     }
438015     //
438016     // It should be a program to run.
438017     //
438018     pid_cmd = fork ();
438019     if (pid_cmd == -1)
438020     {
438021         printf ("%s: cannot run command", argv[0]);
438022         perror (NULL);
438023     }
438024     else if (pid_cmd == 0)
438025     {
438026         execvp (argv_cmd[0], argv_cmd);
438027         perror (NULL);
438028         exit (0);
438029     }
438030     while (1)
438031     {
438032         pid_dead = wait (&status);
438033         if (pid_dead == pid_cmd)
438034         {
438035             break;
438036         }
438037     }
438038     printf ("pid %i terminated with status %i.\n",
438039            (int) pid_dead, status);
438040 }
438041 }
438042 //-----
438043 static void
438044 sh_cd (int argc, char *argv[])
438045 {
438046     int status;
438047     //
438048     if (argc != 2)
438049     {
438050         errset (EINVAL);           // Invalid argument.
438051         perror (NULL);
438052         return;
438053     }
438054     //
438055     status = chdir (argv[1]);
438056     if (status != 0)
438057     {
438058         perror (NULL);
438059     }
438060     return;
438061 }
438062 //-----
438063 static void
438064 sh_pwd (int argc, char *argv[])
438065 {
438066     char path[PATH_MAX];
438067     void *pstatus;
438068     //
438069     if (argc != 1)
438070     {
438071         errset (EINVAL);           // Invalid argument.
438072         perror (NULL);
438073         return;
438074     }
438075     //
438076     // Get the current directory.
438077     //
438078     pstatus = getcwd (path, (size_t) PATH_MAX);
438079     if (pstatus == NULL)
438080     {
438081         perror (NULL);
438082     }
438083     else
438084     {
438085         printf ("%s\n", path);
438086     }
438087     return;
438088 }
438089 //-----
438090 static void
438091 sh_umask (int argc, char *argv[])
438092 {
438093     sysmsg_uarea_t msg;
438094     char *m;           // Index inside the umask octal string.
438095     int mask;
438096     int digit;
438097     //
438098     if (argc > 2)
438099     {
438100         errset (EINVAL);           // Invalid argument.
438101         perror (NULL);
438102         return;
438103     }
438104     //
438105     // If no argument is available, the umask is shown, with a direct
438106     // system call.
438107     //
438108     if (argc == 1)
438109     {
438110         sys (SYS_UAREA, &msg, (sizeof msg));

```

1896

```

438011     printf ("%04o\n", msg.umask);
438012     return;
438013 }
438014 //
438015 // Get the mask: must be the first argument.
438016 //
438017 for (mask = 0, m = argv[1]; *m != 0; m++)
438018 {
438019     digit = (*m - '0');
438020     if (digit < 0 || digit > 7)
438021     {
438022         errset (EINVAL);           // Invalid argument.
438023         perror (NULL);
438024         return;
438025     }
438026     mask = mask * 8 + digit;
438027 }
438028 //
438029 // Set the umask and return.
438030 //
438031 umask (mask);
438032 return;
438033 }

```

applic/touch.c

Si veda la sezione u0.20.

```

490001 #include <fcntl.h>
490002 #include <sys/stat.h>
490003 #include <time.h>
490004 #include <stddef.h>
490005 #include <unistd.h>
490006 #include <errno.h>
490007 //-----
490008 static void usage (void);
490009 //-----
490010 int
490011 main (int argc, char *argv[], char *envp[])
490012 {
490013     int a;           // Argument index.
490014     int status;
490015     struct stat file_status;
490016     //
490017     // No options are known, but at least an argument must be given.
490018     //
490019     if (argc < 2)
490020     {
490021         usage ();
490022         return (1);
490023     }
490024     //
490025     // Scan arguments.
490026     //
490027     for(a = 1; a < argc; a++)
490028     {
490029         //
490030         // Verify if the file exists, through the return value of
490031         // 'stat()'. No other checks are made.
490032         //
490033         if (stat(argv[a], &file_status) == 0)
490034         {
490035             //
490036             // File exists: should be updated the times.
490037             //
490038             status = utime (argv[a], NULL);
490039             if (status != 0)
490040             {
490041                 perror (NULL);
490042                 return (2);
490043             }
490044         }
490045         else
490046         {
490047             //
490048             // File does not exist: should be created.
490049             //
490050             status = open (argv[a], O_WRONLY|O_CREAT|O_TRUNC, 0666);
490051             //
490052             if (status >= 0)
490053             {
490054                 //
490055                 // Here, the variable 'status' is the file
490056                 // descriptor to be closed.
490057                 //
490058                 status = close (status);
490059                 if (status != 0)
490060                 {
490061                     perror (NULL);
490062                     return (3);
490063                 }
490064             }
490065             else
490066             {
490067                 perror (NULL);
490068                 return (4);
490069             }
490070         }
490071     }
490072     return (0);
490073 }

```

1897

```

4390074 //-----
4390075 static void
4390076 usage (void)
4390077 {
4390078     fprintf (stderr, "Usage: touch FILE...\n");
4390079 }

```

applic/tty.c

<

Si veda la sezione [u0.21](#).

```

4400001 #include <fcntl.h>
4400002 #include <sys/stat.h>
4400003 #include <utime.h>
4400004 #include <stddef.h>
4400005 #include <unistd.h>
4400006 #include <errno.h>
4400007 #include <sys/osl6.h>
4400008 #include <sys/types.h>
4400009 //-----
4400010 static void usage (void);
4400011 //-----
4400012 int
4400013 main (int argc, char *argv[], char *envp[])
4400014 {
4400015     int dev_minor;
4400016     struct stat file_status;
4400017     //
4400018     // No options and no arguments.
4400019     //
4400020     if (argc > 1)
4400021     {
4400022         usage ();
4400023         return (1);
4400024     }
4400025     //
4400026     // Verify the standard input.
4400027     //
4400028     if (fstat (STDIN_FILENO, &file_status) == 0)
4400029     {
4400030         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
4400031         {
4400032             dev_minor = minor (file_status.st_rdev);
4400033             //
4400034             // If minor is equal to 0xFF, it is '/dev/console'
4400035             // that is not a controlling terminal, but just
4400036             // a reference for the current virtual console.
4400037             //
4400038             if (dev_minor < 0xFF)
4400039             {
4400040                 printf ("/dev/console%i\n", dev_minor);
4400041             }
4400042         }
4400043     }
4400044     else
4400045     {
4400046         perror ("Cannot get standard input file status");
4400047         return (2);
4400048     }
4400049     //
4400050     return (0);
4400051 }
4400052 //-----
4400053 static void
4400054 usage (void)
4400055 {
4400056     fprintf (stderr, "Usage: tty\n");
4400057 }
4400058 }

```

```

4410027     return (1);
4410028 }
4410029 //
4410030 // System call.
4410031 //
4410032 status = umount (argv[1]);
4410033 if (status != 0)
4410034 {
4410035     perror (argv[1]);
4410036     return (2);
4410037 }
4410038 //
4410039 return (0);
4410040 }
4410041 //-----
4410042 static void
4410043 usage (void)
4410044 {
4410045     fprintf (stderr, "Usage: umount MOUNT_POINT\n");
4410046 }

```

applic/umount.c

<

Si veda la sezione [u0.4](#).

```

4410001 #include <unistd.h>
4410002 #include <stdlib.h>
4410003 #include <sys/stat.h>
4410004 #include <sys/types.h>
4410005 #include <fcntl.h>
4410006 #include <errno.h>
4410007 #include <signal.h>
4410008 #include <stdio.h>
4410009 #include <sys/wait.h>
4410010 #include <stdio.h>
4410011 #include <string.h>
4410012 #include <limits.h>
4410013 #include <sys/osl6.h>
4410014 //-----
4410015 static void usage (void);
4410016 //-----
4410017 int
4410018 main (int argc, char *argv[], char *envp[])
4410019 {
4410020     int status;
4410021     //
4410022     // One argument is mandatory.
4410023     //
4410024     if (argc != 2)
4410025     {
4410026         usage ();

```


