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 ("dsk0",       (mode_t) (S_IFBLK | 0644),
4150028                     (dev_t) DEV_DSK0);
4150029     if (status) perror (NULL);
4150030     status = mknod ("dsk1",       (mode_t) (S_IFBLK | 0644),
4150031                     (dev_t) DEV_DSK1);
4150032     if (status) perror (NULL);
4150033     status = mknod ("dsk2",       (mode_t) (S_IFBLK | 0644),
4150034                     (dev_t) DEV_DSK2);
4150035     if (status) perror (NULL);
4150036     status = mknod ("dsk3",       (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);
```

```
}
//-----------------------------------------------------------------------
static void
cat_file_descriptor (int fd)
{
    ssize_t count;
    char buffer[BUFSIZ];

    for (;;)
      {
        count = read (fd, buffer, (size_t) BUFSIZ);
        if (count > 0)
          {
            write (STDOUT_FILENO, buffer, (size_t) count);
          }
        else
          {
            break;
          }
      }
}

```

applic/ccc.c

«

Si veda la sezione u0.1.

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
//-----------------------------------------------------------------------
int
main (void)
{
    pid_t        pid;
    //-----------------------------------------------------------------------
    pid = fork ();
    if (pid == 0)
      {
        setuid ((uid_t) 10);
        execve ("/bin/aaa", NULL, NULL);
        exit (0);
      }
    //-----------------------------------------------------------------------
    pid = fork ();
    if (pid == 0)
      {
        setuid ((uid_t) 11);
        execve ("/bin/bbb", NULL, NULL);
        exit (0);
      }
    //-----------------------------------------------------------------------
    while (1)
      {
        ; // Just loop, to consume CPU time: it must be killed manually.
      }
    return (0);
}
```

applic/chmod.c

«

Si veda la sezione u0.5.

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <sys/os16.h>
//-----------------------------------------------------------------------
static void usage (void);
//-----------------------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    int     status;
    mode_t  mode;
    char   *m;          // Pointer inside the octal mode string.
    int     digit;
    int     a;          // Argument index.
    //
    //
    //
    if (argc < 3)
      {
        usage ();
        return (1);
      }
    //
    // Get mode: must be the first argument.
    //
    for (m = argv[1]; *m != 0; m++)
      {
        digit = (*m - '0');
        if (digit < 0 || digit > 7)
          {
            usage ();
            return (2);
          }
        mode = mode * 8 + digit;
      }
    //
    // System call for all the remaining arguments.
    //
    for (a = 2; a < argc; a++)
      {
        status = chmod (argv[a], mode);
        if (status != 0)
          {
            perror (argv[a]);
            return (3);
          }
      }
    //
    // All done.
    //
    return (0);
}
//-----------------------------------------------------------------------
static void
usage (void)
{
    fprintf (stderr, "Usage:   chmod OCTAL_MODE FILE...\n");
    fprintf (stderr, "Example: chmod 0640 my_file\n");
}
```

applic/chown.c

«

Si veda la sezione u0.6.

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <ctype.h>
#include <pwd.h>
//-----------------------------------------------------------------------
static void usage (void);
//-----------------------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    char            *user;
    int              uid;
    struct passwd   *pws;
    struct stat      file_status;
    int              a;          // Argument index.
    int              status;
    //
    //
    //
    if (argc < 3)
      {
        usage ();
        return (1);
      }
    //
    // Get user id number.
    //
    user = argv[1];
    if (isdigit (*user))
      {
        uid = atoi (user);
      }
    else
      {
        pws = getpwnam (user);
        if (pws == NULL)
          {
            fprintf(stderr, "Unknown user \"%s\"!\n", user);
            return(2);
          }
        uid = pws->pw_uid;
      }
    //
    // Now we have the user id. Start scanning file names.
    //
    for (a = 2; a < argc; a++)
      {
        //
        // Verify if the file exists, through the return value of
        // 'stat()'. No other checks are made.
        //
        if (stat(argv[a], &file_status) == 0)
         {
            //
            // Try to change ownership.
            //
            status = chown (argv[a], uid, file_status.st_gid);
            if (status != 0)
              {
```

```
                perror (NULL);
                return (3);
              }
          }
        else
          {
            fprintf (stderr, "File \"%s\" does not exist!\n",
                     argv[a]);
            continue;
          }
      }
    //
    // All done.
    //
    return (0);
}
//-----------------------------------------------------------------------
static void
usage (void)
{
    fprintf (stderr, "Usage:   chown USER|UID FILE...\n");
    fprintf (stderr, "Example: chown user my_file\n");
}
```

applic/cp.c

«

Si veda la sezione u0.7.

```
#include <sys/os16.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <libgen.h>
//-----------------------------------------------------------------------
static void usage (void);
//-----------------------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    char       *source;
    char       *destination;
    char       *destination_full;
    struct stat file_status;
    int         dest_is_a_dir = 0;
    int         a;                      // Argument index.
    char        path[PATH_MAX];
    int         fd_source      = -1;
    int         fd_destination = -1;
    char        buffer_in[BUFSIZ];
    char       *buffer_out;
    ssize_t     count_in;             // Read counter.
    ssize_t     count_out;            // Write counter.
    //
    // There must be at least two arguments, plus the program name.
    //
    if (argc < 3)
      {
        usage ();
        return (1);
      }
    //
    // Select the last argument as the destination.
    //
    destination = argv[argc-1];
    //
    // Check if it is a directory and save it in a flag.
    //
    if (stat (destination, &file_status) == 0)
      {
        if (S_ISDIR (file_status.st_mode))
          {
            dest_is_a_dir = 1;
          }
      }
    //
    // If there are more than two arguments, verify that the last
    // one is a directory.
    //
    if (argc > 3)
      {
        if (!dest_is_a_dir)
          {
            usage ();
            fprintf (stderr, "The destination \"%s\" ",
                     destination);
            fprintf (stderr, "is not a directory!\n");
            return (1);
          }
      }
    //
    // Scan the arguments, excluded the last, that is the destination.
    //
    for (a = 1; a < (argc - 1); a++)
      {
```



```
        //
        // Source.
        //
        source = argv[a];
        //
        // Verify access permissions.
        //
        if (access (source, R_OK) < 0)
          {
            perror (source);
            continue;
          }
        //
        // Destination.
        //
        // If it is a directory, the destination path
        // must be corrected.
        //
        if (dest_is_a_dir)
          {
            path[0] = 0;
            strcat (path, destination);
            strcat (path, "/");
            strcat (path, basename (source));
            //
            // Update the destination path.
            //
            destination_full = path;
          }
        else
          {
            destination_full = destination;
          }
        //
        // Check if destination file exists.
        //
        if (stat (destination_full, &file_status) == 0)
          {
            fprintf (stderr, "The destination file, \"%s\", ",
                     destination_full);
            fprintf (stderr, "already exists!\n");
            continue;
          }
        //
        // Everything is ready for the copy.
        //
        fd_source = open (source, O_RDONLY);
        if (fd_source < 0)
          {
            perror (source);
            //
            // Continue with the next file.
            //
            continue;
          }
        //
        fd_destination = creat (destination_full, 0777);
        if (fd_destination < 0)
          {
            perror (destination);
            close (fd_source);
            //
            // Continue with the next file.
            //
            continue;
          }
        //
        // Copy the data.
        //
        while (1)
          {
            count_in = read (fd_source, buffer_in, (size_t) BUFSIZ);
            if (count_in > 0)
              {
                for (buffer_out = buffer_in; count_in > 0;)
                  {
                    count_out = write (fd_destination, buffer_out,
                                       (size_t) count_in);
                    if (count_out < 0)
                      {
                        perror (destination);
                        close (fd_source);
                        close (fd_destination);
                        return (3);
                      }
                    //
                    // If not all data is written, continue writing,
                    // but change the buffer start position and the
                    // amount to be written.
                    //
                    buffer_out += count_out;
                    count_in -= count_out;
                  }
              }
            else if (count_in < 0)
              {
                perror (source);
                close (fd_source);
                close (fd_destination);
              }
            else
```

```
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       ;
```



```
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:      ; Symbol '___mkargv' is used by Bcc inside the function
4230128       ret         ; 'main()' and must be present for a successful
4230129                   ; compilation.
4230130   ;---------------------------------------------------------------------
4230131   .align 2
4230132   .data
4230133   ;
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.
```

```
    //
    if (argc == 1)
      {
        timer = time (NULL);
        printf ("%s\n", ctime (&timer));
        return (0);
      }
    //
    // There is one argument and must be the date do set.
    //
    input = argv[1];
    //
    // First get current date, for default values.
    //
    timer   = time (NULL);
    timeptr = gmtime (&timer);
    //
    // Verify to have a correct input.
    //
    length = (int) strlen (input);
    if (length == 8 || length == 10 || length == 12)
      {
        for (i = 0; i < length; i++)
          {
            if (!isdigit (input[i]))
              {
                usage ();
                return (2);
              }
          }
      }
    else
      {
        printf ("input: \"%s\"; length: %i\n", input, length);
        usage ();
        return (3);
      }
    //
    // Select the month.
    //
    string[0] = input[0];
    string[1] = input[1];
    string[2] = '\0';
    timeptr->tm_mon = atoi (string);
    //
    // Select the day.
    //
    string[0] = input[2];
    string[1] = input[3];
    string[2] = '\0';
    timeptr->tm_mday = atoi (string);
    //
    // Select the hour.
    //
    string[0] = input[4];
    string[1] = input[5];
    string[2] = '\0';
    timeptr->tm_hour = atoi (string);
    //
    // Select the minute.
    //
    string[0] = input[6];
    string[1] = input[7];
    string[2] = '\0';
    timeptr->tm_min = atoi (string);
    //
    // Select the year: must verify if there is a century.
    //
    if (length == 12)
      {
        string[0] = input[8];
        string[1] = input[9];
        string[2] = input[10];
        string[3] = input[11];
        string[4] = '\0';
        timeptr->tm_year = atoi (string);
      }
    else if (length == 10)
      {
        sprintf (string, "%04i", timeptr->tm_year);
        string[2] = input[8];
        string[3] = input[9];
        string[4] = '\0';
        timeptr->tm_year = atoi (string);
      }
    //
    // Now convert to 'time_t'.
    //
    timer = mktime (timeptr);
    //
    // Save to the system.
    //
    stime (&timer);
    //
    return (0);
}
//---------------------------------------------------------------------
static void
usage (void)
{
    fprintf (stderr, "Usage: date [MMDDHHMM[[CC]YY]]\n");
}
```


applic/ed.c

«

Si veda la sezione u0.9.

```
//---------------------------------------------------------------------
// 2009.08.18
// Modified by Daniele Giacomini for 'os16', to harmonize with it,
// even, when possible, on coding style.
//
// The original was taken form ELKS sources: 'elkscmd/misc_utils/ed.c'.
//---------------------------------------------------------------------
//
// Copyright (c) 1993 by David I. Bell
// Permission is granted to use, distribute, or modify this source,
// provided that this copyright notice remains intact.
//
// The "ed" built-in command (much simplified)
//
//---------------------------------------------------------------------

#include <stdio.h>
#include <ctype.h>
#include <unistd.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
//---------------------------------------------------------------------
#define isoctal(ch)  (((ch) >= '0') && ((ch) <= '7'))
#define USERSIZE     1024       /* max line length typed in by user */
#define INITBUFSIZE  1024       /* initial buffer size */
//---------------------------------------------------------------------
typedef int num_t;
typedef int len_t;
//
// The following is the type definition of structure 'line_t', but the
// structure contains pointers to the same kind of type. With the
// compiler Bcc, it is the only way to declare it.
//
typedef struct line line_t;
//
struct line {
    line_t *next;
    line_t *prev;
    len_t   len;
    char    data[1];
};
//
static  line_t  lines;
static  line_t *curline;
static  num_t   curnum;
static  num_t   lastnum;
static  num_t   marks[26];
static  bool    dirty;
static  char    *filename;
static  char    searchstring[USERSIZE];
//
static  char   *bufbase;
static  char   *bufptr;
static  len_t   bufused;
static  len_t   bufsize;
//---------------------------------------------------------------------
static  void    docommands  (void);
static  void    subcommand  (char *cp, num_t num1, num_t num2);
static  bool    getnum      (char **retcp, bool *rethavenum,
                             num_t *retnum);
static  bool    setcurnum   (num_t num);
static  bool    initedit    (void);
static  void    termedit    (void);
static  void    addlines    (num_t num);
static  bool    insertline  (num_t num, char *data, len_t len);
static  bool    deletelines (num_t num1, num_t num2);
static  bool    printlines  (num_t num1, num_t num2, bool expandflag);
static  bool    writelines  (char *file, num_t num1, num_t num2);
static  bool    readlines   (char *file, num_t num);
static  num_t   searchlines (char *str, num_t num1, num_t num2);
static  len_t   findstring  (line_t *lp, char *str, len_t len,
                             len_t offset);
static  line_t *findline    (num_t num);
//---------------------------------------------------------------------
// Main.
//---------------------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    if (!initedit ()) return (2);
    //
    if (argc > 1)
      {
        filename = strdup (argv[1]);
        if (filename == NULL)
          {
            fprintf (stderr, "No memory\n");
            termedit ();
            return (1);
          }
        //
        if (!readlines (filename, 1))
          {
```

```
            termedit ();
            return (0);
          }
        //
        if (lastnum) setcurnum(1);
        //
        dirty = false;
      }
    //
    docommands ();
    //
    termedit ();
    return (0);
}
//---------------------------------------------------------------------
// Read commands until we are told to stop.
//---------------------------------------------------------------------
void
docommands (void)
{
    char   *cp;
    int     len;
    num_t   num1;
    num_t   num2;
    bool    have1;
    bool    have2;
    char    buf[USERSIZE];
    //
    while (true)
      {
        printf(": ");
        fflush (stdout);
        //
        if (fgets (buf, sizeof(buf), stdin) == NULL)
          {
            return;
          }
        //
        len = strlen (buf);
        if (len == 0)
          {
            return;
          }
        //
        cp = &buf[len - 1];
        if (*cp != '\n')
          {
            fprintf(stderr, "Command line too long\n");
            do
              {
                len = fgetc(stdin);
              }
            while ((len != EOF) && (len != '\n'));
            //
            continue;
          }
        //
        while ((cp > buf) && isblank (cp[-1]))
          {
            cp--;
          }
        //
        *cp = '\0';
        //
        cp = buf;
        //
        while (isblank (*cp))
          {
            //*cp++;
            cp++;
          }
        //
        have1 = false;
        have2 = false;
        //
        if ((curnum == 0) && (lastnum > 0))
          {
            curnum = 1;
            curline = lines.next;
          }
        //
        if (!getnum (&cp, &have1, &num1))
          {
            continue;
          }
        //
        while (isblank (*cp))
          {
            cp++;
          }
        //
        if (*cp == ',')
          {
            cp++;
            if (!getnum (&cp, &have2, &num2))
              {
                continue;
              }
            //
            if (!have1)
              {
```



```
                num1 = 1;
              }
            if (!have2)
              {
                num2 = lastnum;
              }
            have1 = true;
            have2 = true;
          }
        //
        if (!have1)
          {
            num1 = curnum;
          }
        if (!have2)
          {
            num2 = num1;
          }
        //
        // Command interpretation switch.
        //
        switch (*cp++)
          {
            case 'a':
                addlines (num1 + 1);
                break;
                //
            case 'c':
                deletelines (num1, num2);
                addlines (num1);
                break;
                //
            case 'd':
                deletelines (num1, num2);
                break;
                //
            case 'f':
                if (*cp && !isblank (*cp))
                  {
                    fprintf (stderr, "Bad file command\n");
                    break;
                  }
                //
                while (isblank (*cp))
                  {
                    cp++;
                  }
                if (*cp == '\0')
                  {
                    if (filename)
                      {
                        printf ("\"%s\"\n", filename);
                      }
                    else
                      {
                        printf ("No filename\n");
                      }
                    break;
                  }
                //
                cp = strdup (cp);
                //
                if (cp == NULL)
                  {
                    fprintf (stderr, "No memory for filename\n");
                    break;
                  }
                //
                if (filename)
                  {
                    free(filename);
                  }
                //
                filename = cp;
                break;
                //
            case 'i':
                addlines (num1);
                break;
                //
            case 'k':
                while (isblank(*cp))
                  {
                    cp++;
                  }
                //
                if ((*cp < 'a') || (*cp > 'a') || cp[1])
                  {
                    fprintf (stderr, "Bad mark name\n");
                    break;
                  }
                //
                marks[*cp - 'a'] = num2;
                break;
                //
            case 'l':
                printlines (num1, num2, true);
                break;
                //
            case 'p':
                printlines (num1, num2, false);
```

```
                break;
                //
            case 'q':
                while (isblank(*cp))
                  {
                    cp++;
                  }
                //
                if (have1 || *cp)
                  {
                    fprintf (stderr, "Bad quit command\n");
                    break;
                  }
                //
                if (!dirty)
                  {
                    return;
                  }
                //
                printf ("Really quit? ");
                fflush (stdout);
                //
                buf[0] = '\0';
                fgets (buf, sizeof(buf), stdin);
                cp = buf;
                //
                while (isblank (*cp))
                  {
                    cp++;
                  }
                //
                if ((*cp == 'y') || (*cp == 'Y'))
                  {
                    return;
                  }
                //
                break;
                //
            case 'r':
                if (*cp && !isblank(*cp))
                  {
                    fprintf (stderr, "Bad read command\n");
                    break;
                  }
                //
                while (isblank(*cp))
                  {
                    cp++;
                  }
                //
                if (*cp == '\0')
                  {
                    fprintf (stderr, "No filename\n");
                    break;
                  }
                //
                if (!have1)
                  {
                    num1 = lastnum;
                  }
                //
                // Open the file and add to the buffer
                // at the next line.
                //
                if (readlines (cp, num1 + 1))
                  {
                    //
                    // If the file open fails, just
                    // break the command.
                    //
                    break;
                  }
                //
                // Set the default file name, if no
                // previous name is available.
                //
                if (filename == NULL)
                  {
                    filename = strdup (cp);
                  }
                //
                break;

            case 's':
                subcommand (cp, num1, num2);
                break;
                //
            case 'w':
                if (*cp && !isblank(*cp))
                  {
                    fprintf(stderr, "Bad write command\n");
                    break;
                  }
                //
                while (isblank(*cp))
                  {
                    cp++;
                  }
                //
                if (!have1)
                  {
```



```
                    num1 = 1;
                    num2 = lastnum;
                  }
                //
                // If the file name is not specified, use the
                // default one.
                //
                if (*cp == '\0')
                  {
                    cp = filename;
                  }
                //
                // If even the default file name is not specified,
                // tell it.
                //
                if (cp == NULL)
                  {
                    fprintf (stderr, "No file name specified\n");
                    break;
                  }
                //
                // Write the file.
                //
                writelines (cp, num1, num2);
                //
                break;
                //
            case 'z':
                switch (*cp)
                  {
                    case '-':
                        printlines (curnum-21, curnum, false);
                        break;
                    case '.':
                        printlines (curnum-11, curnum+10, false);
                        break;
                    default:
                        printlines (curnum, curnum+21, false);
                        break;
                  }
                break;
                //
            case '.':
                if (have1)
                  {
                    fprintf (stderr, "No arguments allowed\n");
                    break;
                  }
                printlines (curnum, curnum, false);
                break;
                //
            case '-':
                if (setcurnum (curnum - 1))
                  {
                    printlines (curnum, curnum, false);
                  }
                break;
                //
            case '=':
                printf ("%d\n", num1);
                break;
                //
            case '\0':
                if (have1)
                  {
                    printlines (num2, num2, false);
                    break;
                  }
                //
                if (setcurnum (curnum + 1))
                  {
                    printlines (curnum, curnum, false);
                  }
                break;
                //
            default:
                fprintf (stderr, "Unimplemented command\n");
                break;
          }
      }
}
//----------------------------------------------------------------------
// Do the substitute command.
// The current line is set to the last substitution done.
//----------------------------------------------------------------------
void
subcommand (char *cp, num_t num1, num_t num2)
{
    int     delim;
    char   *oldstr;
    char   *newstr;
    len_t   oldlen;
    len_t   newlen;
    len_t   deltalen;
    len_t   offset;
    line_t *lp;
    line_t *nlp;
    bool    globalflag;
    bool    printflag;
    bool    didsub;
    bool    needprint;
```

```

    if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
      {
        fprintf (stderr, "Bad line range for substitute\n");
        return;
      }
    //
    globalflag = false;
    printflag = false;
    didsub = false;
    needprint = false;
    //
    if (isblank (*cp) || (*cp == '\0'))
      {
        fprintf (stderr, "Bad delimiter for substitute\n");
        return;
      }
    //
    delim = *cp++;
    oldstr = cp;
    //
    cp = strchr (cp, delim);
    //
    if (cp == NULL)
      {
        fprintf (stderr, "Missing 2nd delimiter for substitute\n");
        return;
      }
    //
    *cp++ = '\0';
    //
    newstr = cp;
    cp = strchr (cp, delim);
    //
    if (cp)
      {
        *cp++ = '\0';
      }
    else
      {
        cp = "";
      }
    while (*cp)
      {
        switch (*cp++)
          {
            case 'g':
                globalflag = true;
                break;
                //
            case 'p':
                printflag = true;
                break;
                //
            default:
                fprintf (stderr, "Unknown option for substitute\n");
                return;
          }
      }
    //
    if (*oldstr == '\0')
      {
        if (searchstring[0] == '\0')
          {
            fprintf (stderr, "No previous search string\n");
            return;
          }
        oldstr = searchstring;
      }
    //
    if (oldstr != searchstring)
      {
        strcpy (searchstring, oldstr);
      }
    //
    lp = findline (num1);
    if (lp == NULL)
      {
        return;
      }
    //
    oldlen = strlen(oldstr);
    newlen = strlen(newstr);
    deltalen = newlen - oldlen;
    offset = 0;
    //
    while (num1 <= num2)
      {
        offset = findstring (lp, oldstr, oldlen, offset);
        if (offset < 0)
          {
            if (needprint)
              {
                printlines (num1, num1, false);
                needprint = false;
              }
            //
            offset = 0;
            lp = lp->next;
            num1++;
            continue;
```



```
          }
        //
        needprint = printflag;
        didsub = true;
        dirty = true;

        //------------------------------------------------------------
        // If the replacement string is the same size or shorter
        // than the old string, then the substitution is easy.
        //------------------------------------------------------------

        if (deltalen <= 0)
          {
            memcpy (&lp->data[offset], newstr, newlen);
            //
            if (deltalen)
              {
                memcpy (&lp->data[offset + newlen],
                        &lp->data[offset + oldlen],
                        lp->len - offset - oldlen);
                //
                lp->len += deltalen;
              }
            //
            offset += newlen;
            //
            if (globalflag)
              {
                continue;
              }
            //
            if (needprint)
              {
                printlines(num1, num1, false);
                needprint = false;
              }
            //
            lp = nlp->next;
            num1++;
            continue;
          }

        //------------------------------------------------------------
        // The new string is larger, so allocate a new line
        // structure and use that.  Link it in in place of
        // the old line structure.
        //------------------------------------------------------------

        nlp = (line_t *) malloc (sizeof (line_t) + lp->len + deltalen);
        //
        if (nlp == NULL)
          {
            fprintf (stderr, "Cannot get memory for line\n");
            return;
          }
        //
        nlp->len = lp->len + deltalen;
        //
        memcpy (nlp->data, lp->data, offset);
        //
        memcpy (&nlp->data[offset], newstr, newlen);
        //
        memcpy (&nlp->data[offset + newlen],
                &lp->data[offset + oldlen],
                lp->len - offset - oldlen);
        //
        nlp->next = lp->next;
        nlp->prev = lp->prev;
        nlp->prev->next = nlp;
        nlp->next->prev = nlp;
        //
        if (curline == lp)
          {
            curline = nlp;
          }
        //
        free(lp);
        lp = nlp;
        //
        offset += newlen;
        //
        if (globalflag)
          {
            continue;
          }
        //
        if (needprint)
          {
            printlines (num1, num1, false);
            needprint = false;
          }
        //
        lp = lp->next;
        num1++;
      }
    //
    if (!didsub)
      {
        fprintf (stderr, "No substitutions found for \"%s\"\n", oldstr);
      }
}
```

```
4250702 //------------------------------------------------------------------------
4250703 // Search a line for the specified string starting at the specified
4250704 // offset in the line.  Returns the offset of the found string, or -1.
4250705 //------------------------------------------------------------------------
4250706 len_t
4250707 findstring (line_t *lp, char *str, len_t len, len_t offset)
4250708 {
4250709     len_t    left;
4250710     char    *cp;
4250711     char    *ncp;
4250712     //
4250713     cp = &lp->data[offset];
4250714     left = lp->len - offset;
4250715     //
4250716     while (left >= len)
4250717       {
4250718         ncp = memchr(cp, *str, left);
4250719         if (ncp == NULL)
4250720           {
4250721             return (len_t) -1;
4250722           }
4250723         //
4250724         left -= (ncp - cp);
4250725         if (left < len)
4250726           {
4250727             return (len_t) -1;
4250728           }
4250729         //
4250730         cp = ncp;
4250731         if (memcmp(cp, str, len) == 0)
4250732           {
4250733             return (len_t) (cp - lp->data);
4250734           }
4250735         //
4250736         cp++;
4250737         left--;
4250738       }
4250739     //
4250740     return (len_t) -1;
4250741 }
4250742 //------------------------------------------------------------------------
4250743 // Add lines which are typed in by the user.
4250744 // The lines are inserted just before the specified line number.
4250745 // The lines are terminated by a line containing a single dot (ugly!),
4250746 // or by an end of file.
4250747 //------------------------------------------------------------------------
4250748 void
4250749 addlines (num_t num)
4250750 {
4250751     int     len;
4250752     char    buf[USERSIZE + 1];
4250753     //
4250754     while (fgets (buf, sizeof (buf), stdin))
4250755       {
4250756         if ((buf[0] == '.') && (buf[1] == '\n') && (buf[2] == '\0'))
4250757           {
4250758             return;
4250759           }
4250760         //
4250761         len = strlen (buf);
4250762         //
4250763         if (len == 0)
4250764           {
4250765             return;
4250766           }
4250767         //
4250768         if (buf[len - 1] != '\n')
4250769          {
4250770             fprintf (stderr, "Line too long\n");
4250771             //
4250772             do
4250773               {
4250774                 len = fgetc(stdin);
4250775               }
4250776             while ((len != EOF) && (len != '\n'));
4250777             //
4250778             return;
4250779           }
4250780         //
4250781         if (!insertline (num++, buf, len))
4250782           {
4250783             return;
4250784           }
4250785       }
4250786 }
4250787 //------------------------------------------------------------------------
4250788 // Parse a line number argument if it is present.  This is a sum
4250789 // or difference of numbers, '.', '$', 'x, or a search string.
4250790 // Returns true if successful (whether or not there was a number).
4250791 // Returns false if there was a parsing error, with a message output.
4250792 // Whether there was a number is returned indirectly, as is the number.
4250793 // The character pointer which stopped the scan is also returned.
4250794 //------------------------------------------------------------------------
4250795 static bool
4250796 getnum (char **retcp, bool *rethavenum, num_t *retnum)
4250797 {
4250798     char   *cp;
4250799     char   *str;
4250800     bool    havenum;
4250801     num_t   value;
4250802     num_t   num;
```



```
4250803     num_t   sign;
4250804     //
4250805     cp = *retcp;
4250806     havenum = false;
4250807     value = 0;
4250808     sign = 1;
4250809     //
4250810     while (true)
4250811       {
4250812         while (isblank(*cp))
4250813           {
4250814             cp++;
4250815           }
4250816         //
4250817         switch (*cp)
4250818           {
4250819             case '.':
4250820                 havenum = true;
4250821                 num = curnum;
4250822                 cp++;
4250823                 break;
4250824                 //
4250825             case '$':
4250826                 havenum = true;
4250827                 num = lastnum;
4250828                 cp++;
4250829                 break;
4250830                 //
4250831             case '\'':
4250832                 cp++;
4250833                 if ((*cp < 'a') || (*cp > 'z'))
4250834                   {
4250835                     fprintf (stderr, "Bad mark name\n");
4250836                     return false;
4250837                   }
4250838                 //
4250839                 havenum = true;
4250840                 num = marks[*cp++ - 'a'];
4250841                 break;
4250842                 //
4250843             case '/':
4250844                 str = ++cp;
4250845                 cp = strchr (str, '/');
4250846                 if (cp)
4250847                   {
4250848                     *cp++ = '\0';
4250849                   }
4250850                 else
4250851                   {
4250852                     cp = "";
4250853                   }
4250854                 num = searchlines (str, curnum, lastnum);
4250855                 if (num == 0)
4250856                   {
4250857                     return false;
4250858                   }
4250859                 //
4250860                 havenum = true;
4250861                 break;
4250862                 //
4250863             default:
4250864                 if (!isdigit (*cp))
4250865                  {
4250866                     *retcp = cp;
4250867                     *rethavenum = havenum;
4250868                     *retnum = value;
4250869                     return true;
4250870                   }
4250871                 //
4250872                 num = 0;
4250873                 while (isdigit(*cp))
4250874                   {
4250875                     num = num * 10 + *cp++ - '0';
4250876                   }
4250877                 havenum = true;
4250878                 break;
4250879           }
4250880         //
4250881         value += num * sign;
4250882         //
4250883         while (isblank (*cp))
4250884           {
4250885             cp++;
4250886           }
4250887         //
4250888         switch (*cp)
4250889           {
4250890             case '-':
4250891                 sign = -1;
4250892                 cp++;
4250893                 break;
4250894                 //
4250895             case '+':
4250896                 sign = 1;
4250897                 cp++;
4250898                 break;
4250899                 //
4250900             default:
4250901                 *retcp = cp;
4250902                 *rethavenum = havenum;
4250903                 *retnum = value;
```

```
                return true;
            }
        }
}
//---------------------------------------------------------------------
// Initialize everything for editing.
//---------------------------------------------------------------------
bool
initedit (void)
{
    int i;
    //
    bufsize = INITBUFSIZE;
    bufbase = malloc (bufsize);
    //
    if (bufbase == NULL)
      {
        fprintf (stderr, "No memory for buffer\n");
        return false;
      }
    //
    bufptr = bufbase;
    bufused = 0;
    //
    lines.next = &lines;
    lines.prev = &lines;
    //
    curline = NULL;
    curnum = 0;
    lastnum = 0;
    dirty = false;
    filename = NULL;
    searchstring[0] = '\0';
    //
    for (i = 0; i < 26; i++)
      {
        marks[i] = 0;
      }
    //
    return true;
}
//---------------------------------------------------------------------
// Finish editing.
//---------------------------------------------------------------------
void
termedit (void)
{
    if (bufbase) free(bufbase);
    bufbase = NULL;
    //
    bufptr = NULL;
    bufsize = 0;
    bufused = 0;
    //
    if (filename) free(filename);
    filename = NULL;
    //
    searchstring[0] = '\0';
    //
    if (lastnum) deletelines (1, lastnum);
    //
    lastnum = 0;
    curnum = 0;
    curline = NULL;
}
//---------------------------------------------------------------------
// Read lines from a file at the specified line number.
// Returns true if the file was successfully read.
//---------------------------------------------------------------------
bool
readlines (char *file, num_t num)
{
    int     fd;
    int     cc;
    len_t   len;
    len_t   linecount;
    len_t   charcount;
    char   *cp;
    //
    if ((num < 1) || (num > lastnum + 1))
      {
        fprintf (stderr, "Bad line for read\n");
        return false;
      }
    //
    fd = open (file, O_RDONLY);
    if (fd < 0)
      {
        perror (file);
        return false;
      }
    //
    bufptr = bufbase;
    bufused = 0;
    linecount = 0;
    charcount = 0;
    //
    printf ("\"%s\", ", file);
    fflush(stdout);
    //
    do
```



```
      {
        cp = memchr(bufptr, '\n', bufused);
        if (cp)
          {
            len = (cp - bufptr) + 1;
            //
            if (!insertline (num, bufptr, len))
              {
                close (fd);
                return false;
              }
            //
            bufptr += len;
            bufused -= len;
            charcount += len;
            linecount++;
            num++;
            continue;
          }
        //
        if (bufptr != bufbase)
          {
            memcpy (bufbase, bufptr, bufused);
            bufptr = bufbase + bufused;
          }
        //
        if (bufused >= bufsize)
          {
            len = (bufsize * 3) / 2;
            cp = realloc (bufbase, len);
            if (cp == NULL)
              {
                fprintf (stderr, "No memory for buffer\n");
                close (fd);
                return false;
              }
            //
            bufbase = cp;
            bufptr = bufbase + bufused;
            bufsize = len;
          }
        //
        cc = read (fd, bufptr, bufsize - bufused);
        bufused += cc;
        bufptr = bufbase;
      }
    while (cc > 0);
    //
    if (cc < 0)
      {
        perror (file);
        close (fd);
        return false;
      }
    //
    if (bufused)
      {
        if (!insertline (num, bufptr, bufused))
          {
            close (fd);
            return -1;
          }
        linecount++;
        charcount += bufused;
      }
    //
    close (fd);
    //
    printf ("%d lines%s, %d chars\n",
            linecount,
            (bufused ? " (incomplete)" : ""),
            charcount);
    //
    return true;
}
//---------------------------------------------------------------------
// Write the specified lines out to the specified file.
// Returns true if successful, or false on an error with a message
// output.
//---------------------------------------------------------------------
bool
writelines (char *file, num_t num1, num_t num2)
{
    int     fd;
    line_t *lp;
    len_t   linecount;
    len_t   charcount;
    //
    if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
      {
        fprintf (stderr, "Bad line range for write\n");
        return false;
      }
    //
    linecount = 0;
    charcount = 0;
    //
    fd = creat (file, 0666);
    if (fd < 0)
      {
        perror (file);
```

```
          return false;
        }
      //
      printf("\"%s\", ", file);
      fflush (stdout);
      //
      lp = findline (num1);
      if (lp == NULL)
        {
          close (fd);
          return false;
        }
      //
      while (num1++ <= num2)
        {
          if (write(fd, lp->data, lp->len) != lp->len)
            {
              perror(file);
              close(fd);
              return false;
            }
          //
          charcount += lp->len;
          linecount++;
          lp = lp->next;
        }
      //
      if (close(fd) < 0)
        {
          perror(file);
          return false;
        }
      //
      printf ("%d lines, %d chars\n", linecount, charcount);
      //
      return true;
}
//---------------------------------------------------------------------
// Print lines in a specified range.
// The last line printed becomes the current line.
// If expandflag is true, then the line is printed specially to
// show magic characters.
//---------------------------------------------------------------------
bool
printlines (num_t num1, num_t num2, bool expandflag)
{
    line_t        *lp;
    unsigned char *cp;
    int            ch;
    len_t          count;
    //
    if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
      {
        fprintf (stderr, "Bad line range for print\n");
        return false;
      }
    //
    lp = findline (num1);
    if (lp == NULL)
      {
        return false;
      }
    //
    while (num1 <= num2)
      {
        if (!expandflag)
          {
            write (STDOUT_FILENO, lp->data, lp->len);
            setcurnum (num1++);
            lp = lp->next;
            continue;
          }

        //-------------------------------------------------------------
        // Show control characters and characters with the
        // high bit set specially.
        //-------------------------------------------------------------

        cp = (unsigned char *) lp->data;
        count = lp->len;
        //
        if ((count > 0) && (cp[count - 1] == '\n'))
          {
            count--;
          }
        //
        while (count-- > 0)
          {
            ch = *cp++;
            if (ch & 0x80)
              {
                fputs ("M-", stdout);
                ch &= 0x7f;
              }
            if (ch < ' ')
              {
                fputc ('^', stdout);
                ch += '@';
              }
            if (ch == 0x7f)
              {
```



```
                fputc ('^', stdout);
                ch = '?';
              }
            fputc (ch, stdout);
          }
        //
        fputs ("$\n", stdout);
        //
        setcurnum (num1++);
        lp = lp->next;
      }
    //
    return true;
}
//---------------------------------------------------------------------
// Insert a new line with the specified text.
// The line is inserted so as to become the specified line,
// thus pushing any existing and further lines down one.
// The inserted line is also set to become the current line.
// Returns true if successful.
//---------------------------------------------------------------------
bool
insertline (num_t num, char *data, len_t len)
{
    line_t    *newlp;
    line_t    *lp;
    //
    if ((num < 1) || (num > lastnum + 1))
      {
        fprintf (stderr, "Inserting at bad line number\n");
        return false;
      }
    //
    newlp = (line_t *) malloc (sizeof (line_t) + len - 1);
    if (newlp == NULL)
      {
        fprintf (stderr, "Failed to allocate memory for line\n");
        return false;
      }
    //
    memcpy (newlp->data, data, len);
    newlp->len = len;
    //
    if (num > lastnum)
      {
        lp = &lines;
      }
    else
      {
        lp = findline (num);
        if (lp == NULL)
          {
            free ((char *) newlp);
            return false;
          }
      }
    //
    newlp->next = lp;
    newlp->prev = lp->prev;
    lp->prev->next = newlp;
    lp->prev = newlp;
    //
    lastnum++;
    dirty = true;
    //
    return setcurnum (num);
}
//---------------------------------------------------------------------
// Delete lines from the given range.
//---------------------------------------------------------------------
bool
deletelines (num_t num1, num_t num2)
{
    line_t   *lp;
    line_t   *nlp;
    line_t   *plp;
    num_t     count;
    //
    if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
      {
        fprintf (stderr, "Bad line numbers for delete\n");
        return false;
      }
    //
    lp = findline (num1);
    if (lp == NULL)
      {
        return false;
      }
    //
    if ((curnum >= num1) && (curnum <= num2))
      {
        if (num2 < lastnum)
          {
            setcurnum (num2 + 1);
          }
        else if (num1 > 1)
          {
            setcurnum (num1 - 1);
          }
        else
```

```
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;
```



```
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/os16.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     setpgrp ();
4260031     //
4260032     // Open the terminal, that should become the controlling terminal:
4260033     // close the standard input and open the new terminal (r/w).
```

```
4260034     //
4260035     close (0);
4260036     fdn = open (device_name, O_RDWR);
4260037     if (fdn < 0)
4260038       {
4260039         //
4260040         // Cannot open terminal. A message should appear, at least
4260041         // to the current console.
4260042         //
4260043         perror (NULL);
4260044         return (-1);
4260045       }
4260046     //
4260047     // Reset terminal device permissions and ownership.
4260048     //
4260049     status = fchown (fdn, (uid_t) 0, (gid_t) 0);
4260050     if (status != 0)
4260051       {
4260052         perror (NULL);
4260053       }
4260054     status = fchmod (fdn, 0644);
4260055     if (status != 0)
4260056       {
4260057         perror (NULL);
4260058       }
4260059     //
4260060     // The terminal is open and it should be already the controlling
4260061     // one: show '/etc/issue'. The same variable 'fdn' is used, because
4260062     // the controlling terminal will never be closed (the exit syscall
4260063     // will do it).
4260064     //
4260065     fdn = open ("/etc/issue", O_RDONLY);
4260066     if (fdn > 0)
4260067       {
4260068         //
4260069         // The file is present and is shown.
4260070         //
4260071         for (size_read = 1; size_read > 0;)
4260072           {
4260073             size_read = read (fdn, buffer, (size_t) (BUFSIZ - 1));
4260074             if (size_read < 0)
4260075               {
4260076                 break;
4260077               }
4260078             buffer[size_read] = '\0';
4260079             printf ("%s", buffer);
4260080           }
4260081         close (fdn);
4260082       }
4260083     //
4260084     // Show the terminal.
4260085     //
4260086     printf ("This is terminal %s\n", device_name);
4260087     //
4260088     // It is time to exec login: the environment is inherited directly
4260089     // from 'init'.
4260090     //
4260091     exec_argv[0] = "login";
4260092     exec_argv[1] = NULL;
4260093     exec_envp    = envp;
4260094     execve ("/bin/login", exec_argv, exec_envp);
4260095     //
4260096     // If 'execve()' returns, it is an error.
4260097     //
4260098     exit (-1);
4260099 }
```

applic/init.c

«

Si veda la sezione u0.2.

```
4270001 #include <unistd.h>
4270002 #include <stdio.h>
4270003 #include <stdlib.h>
4270004 #include <signal.h>
4270005 #include <sys/wait.h>
4270006 #include <limits.h>
4270007 #include <sys/os16.h>
4270008 #include <fcntl.h>
4270009 #include <string.h>
4270010 //---------------------------------------------------------------------
4270011 #define RESPAWN_MAX        7
4270012 #define COMMAND_MAX      100
4270013 #define LINE_MAX        1024
4270014 //---------------------------------------------------------------------
4270015 int
4270016 main (int argc, char *argv[], char *envp[])
4270017 {
4270018     //
4270019     // 'init.c' has its own 'init.crt0.s' with a very small stack
4270020     // size. Remember to verify to have enough room for the stack.
4270021     //
4270022     pid_t  pid;
4270023     int    status;
4270024     char  *exec_argv[3];
4270025     char  *exec_envp[3];
4270026     char   buffer[LINE_MAX];
4270027     int    r;                   // Respawn table index.
4270028     int    b;                   // Buffer index.
4270029     size_t size_read;
4270030     char  *inittab_id;
```



```
4270031     char  *inittab_runlevels;
4270032     char  *inittab_action;
4270033     char  *inittab_process;
4270034     int    eof;
4270035     int    fd;
4270036     //
4270037     // It follows a table for commands to be respawn.
4270038     //
4270039     struct {
4270040         pid_t   pid;
4270041         char    command[COMMAND_MAX];
4270042     } respawn[RESPAWN_MAX];
4270043
4270044     //---------------------------------------------------------------
4270045     signal (SIGHUP,  SIG_IGN);
4270046     signal (SIGINT,  SIG_IGN);
4270047     signal (SIGQUIT, SIG_IGN);
4270048     signal (SIGILL,  SIG_IGN);
4270049     signal (SIGABRT, SIG_IGN);
4270050     signal (SIGFPE,  SIG_IGN);
4270051 //  signal (SIGKILL, SIG_IGN);  Cannot ignore SIGKILL.
4270052     signal (SIGSEGV, SIG_IGN);
4270053     signal (SIGPIPE, SIG_IGN);
4270054     signal (SIGALRM, SIG_IGN);
4270055     signal (SIGTERM, SIG_IGN);
4270056 //  signal (SIGSTOP, SIG_IGN);  Cannot ignore SIGSTOP.
4270057     signal (SIGTSTP, SIG_IGN);
4270058     signal (SIGCONT, SIG_IGN);
4270059     signal (SIGTTIN, SIG_IGN);
4270060     signal (SIGTTOU, SIG_IGN);
4270061     signal (SIGUSR1, SIG_IGN);
4270062     signal (SIGUSR2, SIG_IGN);
4270063     //---------------------------------------------------------------
4270064     printf ("init\n");
4270065     //  heap_clear ();
4270066     //  process_info ();
4270067     //---------------------------------------------------------------
4270068     //
4270069     // Reset the 'respawn' table.
4270070     //
4270071     for (r = 0; r < RESPAWN_MAX; r++)
4270072       {
4270073         respawn[r].pid = 0;
4270074         respawn[r].command[0]                  = 0;
4270075         respawn[r].command[COMMAND_MAX-1] = 0;
4270076       }
4270077     //
4270078     // Read the '/etc/inittab' file.
4270079     //
4270080     fd = open ("/etc/inittab", O_RDONLY);
4270081     //
4270082     if (fd < 0)
4270083       {
4270084         perror ("Cannot open file '/etc/inittab'");
4270085         exit (-1);
4270086       }
4270087     //
4270088     //
4270089     //
4270090     for (eof = 0, r = 0; !eof && r < RESPAWN_MAX; r++)
4270091       {
4270092         for (b = 0; b < LINE_MAX; b++)
4270093           {
4270094             size_read = read (fd, &buffer[b], (size_t) 1);
4270095             if (size_read <= 0)
4270096               {
4270097                 buffer[b] = 0;
4270098                 eof       = 1;           // Close the read loop.
4270099                 break;
4270100               }
4270101             if (buffer[b] == '\n')
4270102               {
4270103                 buffer[b]  = 0;
4270104                 break;
4270105               }
4270106           }
4270107         //
4270108         // Remove comments: just replace '#' with '\0'.
4270109         //
4270110         for (b = 0; b < LINE_MAX; b++)
4270111           {
4270112             if (buffer[b] == '#')
4270113               {
4270114                 buffer[b] = 0;
4270115                 break;
4270116               }
4270117           }
4270118         //
4270119         // If the buffer is an empty string, just loop to next
4270120         // record.
4270121         //
4270122         if (strlen (buffer) == 0)
4270123           {
4270124             r--;
4270125             continue;
4270126           }
4270127         //
4270128         //
4270129         //
4270130         inittab_id        = strtok (buffer, ":");
4270131         inittab_runlevels = strtok (NULL, ":");
```

```
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 }
```

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;
```



```
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, ":ls:")) != -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)
```

```
                  {
                    signal = SIGUSR1;
                  }
                else if (strcmp (optarg, "USR2") == 0)
                  {
                    signal = SIGUSR2;
                  }
                else
                  {
                    fprintf (stderr, "Unknown signal %s.\n", optarg);
                    return (1);
                  }
                break;
            case '?':
                fprintf (stderr, "Unknown option -%c.\n", optopt);
                usage ();
                return (1);
                break;
            case ':':
                fprintf (stderr, "Missing argument for option -%c\n",
                         optopt);
                usage ();
                return (1);
                break;
            default:
                fprintf (stderr, "Getopt problem: unknown option %c\n",
                         opt);
                return (1);
            }
        }
      //
      //
      //
      if (option_l && option_s)
        {
          fprintf (stderr, "Options \"-l\" and \"-s\" together ");
          fprintf (stderr, "are incompatible.\n");
          usage ();
          return (1);
        }
      //
      // Option "-l".
      //
      if (option_l)
        {
          printf ("HUP ");
          printf ("INT ");
          printf ("QUIT ");
          printf ("ILL ");
          printf ("ABRT ");
          printf ("FPE ");
          printf ("KILL ");
          printf ("SEGV ");
          printf ("PIPE ");
          printf ("ALRM ");
          printf ("TERM ");
          printf ("STOP ");
          printf ("TSTP ");
          printf ("CONT ");
          printf ("CHLD ");
          printf ("TTIN ");
          printf ("TTOU ");
          printf ("USR1 ");
          printf ("USR2 ");
          printf ("\n");
        }
      //
      // Option "-s".
      //
      if (option_s)
        {
          //
          // Scan arguments.
          //
          for (a = 3; a < argc; a++)
            {
              //
              // Get PID.
              //
              pid = atoi (argv[a]);
              if (pid > 0)
                {
                  //
                  // Kill.
                  //
                  if (kill (pid, signal) < 0)
                    {
                      perror (argv[a]);
                    }
                }
              else
                {
                  fprintf (stderr, "Invalid PID %s.", argv[a]);
                }
            }
        }
      //
      // All done.
      //
      return (0);
}
```



```
//------------------------------------------------------------------
static void
usage (void)
{
    fprintf (stderr, "Usage: kill -s SIGNAL_NAME PID...\n");
    fprintf (stderr, "       kill -l\n");
}
```

applic/ln.c

«

Si veda la sezione u0.11.

```
#include <sys/os16.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <libgen.h>
//------------------------------------------------------------------
static void usage (void);
//------------------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    char       *source;
    char       *destination;
    char       *new_destination;
    struct stat file_status;
    int         dest_is_a_dir = 0;
    int         a;                      // Argument index.
    char        path[PATH_MAX];
    //
    // There must be at least two arguments, plus the program name.
    //
    if (argc < 3)
      {
        usage ();
        return (1);
      }
    //
    // Select the last argument as the destination.
    //
    destination = argv[argc-1];
    //
    // Check if it is a directory and save it in a flag.
    //
    if (stat (destination, &file_status) == 0)
      {
        if (S_ISDIR (file_status.st_mode))
          {
            dest_is_a_dir = 1;
          }
      }
    //
    // If there are more than two arguments, verify that the last
    // one is a directory.
    //
    if (argc > 3)
      {
        if (!dest_is_a_dir)
          {
            usage ();
            fprintf (stderr, "The destination \"%s\" ",
                     destination);
            fprintf (stderr, "is not a directory!\n");
            return (1);
          }
      }
    //
    // Scan the arguments, excluded the last, that is the destination.
    //
    for (a = 1; a < (argc - 1); a++)
      {
        //
        // Source.
        //
        source = argv[a];
        //
        // Verify access permissions.
        //
        if (access (source, R_OK) < 0)
          {
            perror (source);
            continue;
          }
        //
        // Destination.
        //
        // If it is a directory, the destination path
        // must be corrected.
        //
        if (dest_is_a_dir)
          {
            path[0] = 0;
            strcat (path, destination);
```

```
4290090             strcat (path, "/");
4290091             strcat (path, basename (source));
4290092             //
4290093             // Update the destination path.
4290094             //
4290095             new_destination = path;
4290096           }
4290097         else
4290098           {
4290099             new_destination = destination;
4290100           }
4290101         //
4290102         // Check if destination file exists.
4290103         //
4290104         if (stat (new_destination, &file_status) == 0)
4290105           {
4290106             fprintf (stderr, "The destination file, \"%s\", ",
4290107                      new_destination);
4290108             fprintf (stderr, "already exists!\n");
4290109             continue;
4290110           }
4290111         //
4290112         // Everything is ready for the link.
4290113         //
4290114         if (link (source, new_destination) < 0)
4290115           {
4290116             perror (new_destination);
4290117             continue;
4290118           }
4290119       }
4290120     //
4290121     // All done.
4290122     //
4290123     return (0);
4290124 }
4290125 //---------------------------------------------------------------------------
4290126 static void
4290127 usage (void)
4290128 {
4290129     fprintf (stderr, "Usage: ln OLD_NAME NEW_NAME\n");
4290130     fprintf (stderr, "       ln FILE... DIRECTORY\n");
4290131 }
```

applic/login.c

«

Si veda la sezione u0.12.

```
4300001 #include <unistd.h>
4300002 #include <stdlib.h>
4300003 #include <sys/stat.h>
4300004 #include <sys/types.h>
4300005 #include <fcntl.h>
4300006 #include <errno.h>
4300007 #include <unistd.h>
4300008 #include <signal.h>
4300009 #include <stdio.h>
4300010 #include <sys/wait.h>
4300011 #include <stdio.h>
4300012 #include <string.h>
4300013 #include <limits.h>
4300014 #include <stdint.h>
4300015 #include <sys/os16.h>
4300016 //---------------------------------------------------------------------------
4300017 #define LOGIN_MAX       64
4300018 #define PASSWORD_MAX    64
4300019 #define HOME_MAX        64
4300020 #define LINE_MAX        1024
4300021 //---------------------------------------------------------------------------
4300022 int
4300023 main (int argc, char *argv[], char *envp[])
4300024 {
4300025     char    login[LOGIN_MAX];
4300026     char    password[PASSWORD_MAX];
4300027     char    buffer[LINE_MAX];
4300028     char   *user_name;
4300029     char   *user_password;
4300030     char   *user_uid;
4300031     char   *user_gid;
4300032     char   *user_description;
4300033     char   *user_home;
4300034     char   *user_shell;
4300035     uid_t   uid;
4300036     uid_t   euid;
4300037     int     fd;
4300038     ssize_t size_read;
4300039     int     b;          // Index inside buffer.
4300040     int     loop;
4300041     char   *exec_argv[2];
4300042     int     status;
4300043     char   *tty_path;
4300044     //
4300045     // Check if login is running correctly.
4300046     //
4300047     euid = geteuid ();
4300048     uid  = geteuid ();
4300049     //  //
4300050     //  // Show process info.
4300051     //  //
4300052     //  heap_clear ();
4300053     //  process_info ();
4300054     //
```



```
4300055     // Check privileges.
4300056     //
4300057     if (!(uid == 0 && euid == 0))
4300058       {
4300059         printf ("%s: can only run with root privileges!\n", argv[0]);
4300060         exit (-1);
4300061       }
4300062     //
4300063     // Prepare arguments for the shell call.
4300064     //
4300065     exec_argv[0] = "-";
4300066     exec_argv[1] = NULL;
4300067     //
4300068     // Login.
4300069     //
4300070     while (1)
4300071       {
4300072         fd = open ("/etc/passwd", O_RDONLY);
4300073         //
4300074         if (fd < 0)
4300075           {
4300076             perror ("Cannot open file '/etc/passwd'");
4300077             exit (-1);
4300078           }
4300079         //
4300080         printf ("Log in as \"root\" or \"user\" "
4300081                 "with password \"ciao\" :-)\n");
4300082         input_line (login, "login:", LOGIN_MAX, INPUT_LINE_ECHO);
4300083         //
4300084         //
4300085         //
4300086         loop = 1;
4300087         while (loop)
4300088           {
4300089             for (b = 0; b < LINE_MAX; b++)
4300090               {
4300091                 size_read = read (fd, &buffer[b], (size_t) 1);
4300092                 if (size_read <= 0)
4300093                   {
4300094                     buffer[b] = 0;
4300095                     loop = 0;           // Close the middle loop.
4300096                     break;
4300097                   }
4300098                 if (buffer[b] == '\n')
4300099                   {
4300100                     buffer[b] = 0;
4300101                     break;
4300102                   }
4300103               }
4300104             //
4300105             user_name        = strtok (buffer, ":");
4300106             user_password    = strtok (NULL, ":");
4300107             user_uid         = strtok (NULL, ":");
4300108             user_gid         = strtok (NULL, ":");
4300109             user_description = strtok (NULL, ":");
4300110             user_home        = strtok (NULL, ":");
4300111             user_shell       = strtok (NULL, ":");
4300112             //
4300113             if (strcmp (user_name, login) == 0)
4300114               {
4300115                 input_line (password, "password:", PASSWORD_MAX,
4300116                             INPUT_LINE_STARS);
4300117                 //
4300118                 // Compare passwords: empty passwords are not allowed.
4300119                 //
4300120                 if (strcmp (user_password, password) == 0)
4300121                   {
4300122                     uid  = atoi (user_uid);
4300123                     euid = uid;
4300124                     //
4300125                     // Find the controlling terminal and change
4300126                     // property and access permissions.
4300127                     //
4300128                     tty_path = ttyname (STDIN_FILENO);
4300129                     if (tty_path != NULL)
4300130                       {
4300131                         status = chown (tty_path, uid, 0);
4300132                         if (status != 0)
4300133                           {
4300134                             perror (NULL);
4300135                           }
4300136                         status = chmod (tty_path, 0600);
4300137                         if (status != 0)
4300138                           {
4300139                             perror (NULL);
4300140                           }
4300141                       }
4300142                     //
4300143                     // Cd to the home directory, if present.
4300144                     //
4300145                     status = chdir (user_home);
4300146                     if (status != 0)
4300147                       {
4300148                         perror (NULL);
4300149                       }
4300150                     //
4300151                     // Now change personality.
4300152                     //
4300153                     setuid (uid);
4300154                     seteuid (euid);
4300155                     //
```

```
                    // Run the shell, replacing the login process; the
                    // environment is taken from 'init'.
                    //
                    execve (user_shell, exec_argv, envp);
                    exit (0);
                  }
                //
                // Login failed: will try again.
                //
                loop = 0;               // Close the middle loop.
                break;
              }
          }
        close (fd);
      }
}
```

applic/ls.c

«

Si veda la sezione u0.13.

```
#include <sys/os16.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <libgen.h>
#include <dirent.h>
#include <pwd.h>
#include <time.h>

#define BUFFER_SIZE     16384
#define LIST_SIZE       256

//------------------------------------------------------------------------
static void usage         (void);
//------------------------------------------------------------------------
//------------------------------------------------------------------------
int compare (void *p1, void *p2);
//------------------------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    int            option_a = 0;
    int            option_l = 0;
    int            opt;
//  extern char   *optarg;              // not used.
    extern int     optind;
    extern int     optopt;
    struct stat    file_status;
    DIR           *dp;
    struct dirent *dir;
    char           buffer[BUFFER_SIZE];
    int            b;                   // Buffer index.
    char          *list[LIST_SIZE];
    int            l;                   // List index.
    int            len;                 // Name length.
    char          *path = NULL;
    char           pathname[PATH_MAX];
    struct passwd *pws;
    struct tm     *tms;
    //
    // Check for options.
    //
    while ((opt = getopt (argc, argv, ":al")) != -1)
      {
        switch (opt)
          {
            case 'l':
                option_l = 1;
                break;
            case 'a':
                option_a = 1;
                break;
            case '?':
                fprintf (stderr, "Unknown option -%c.\n", optopt);
                usage ();
                return (1);
                break;
            case ':':
                fprintf (stderr, "Missing argument for option -%c\n",
                         optopt);
                usage ();
                return (1);
                break;
            default:
                fprintf (stderr, "Getopt problem: unknown option %c\n",
                         opt);
                return (1);
          }
      }
    //
    // If no arguments are present, at least the current directory is
    // read.
    //
```



```
    if (optind == argc)
      {
        //
        // There are no more arguments. Replace the program name,
        // corresponding to 'argv[0]', with the current directory
        // path string.
        //
        argv[0] = ".";
        argc    = 1;
        optind  = 0;
      }
    //
    // This is a very simplified 'ls': if there is only a name
    // and it is a directory, the directory content is taken as
    // the new 'argv[]' array.
    //
    if (optind == (argc -1))
      {
        //
        // There is a request for a single name. Test if it exists
        // and if it is a directory.
        //
        if (stat(argv[optind], &file_status) != 0)
          {
            fprintf (stderr, "File \"%s\" does not exist!\n",
                     argv[optind]);
            return (2);
          }
        //
        if (S_ISDIR (file_status.st_mode))
          {
            //
            // Save the directory inside the 'path' pointer.
            //
            path = argv[optind];
            //
            // Open the directory.
            //
            dp = opendir (argv[optind]);
            if (dp == NULL)
              {
                perror (argv[optind]);
                return (3);
              }
            //
            // Read the directory and fill the buffer with names.
            //
            b = 0;
            l = 0;
            while ((dir = readdir (dp)) != NULL)
              {
                len = strlen (dir->d_name);
                //
                // Check if the buffer can hold it.
                //
                if ((b + len + 1 ) > BUFFER_SIZE)
                  {
                    fprintf (stderr, "not enough memory\n");
                    break;
                  }
                //
                // Consider the directory item only if there is
                // a valid name. If it is empty, just ignore it.
                //
                if (len > 0)
                  {
                    strcpy (&buffer[b], dir->d_name);
                    list[l] = &buffer[b];
                    b += len + 1;
                    l++;
                  }
              }
            //
            // Close the directory.
            //
            closedir (dp);
            //
            // Sort the list.
            //
            qsort (list, (size_t) l, sizeof (char *), compare);

            //
            // Convert the directory list into a new 'argv[]' array,
            // with a valid 'argc'. The variable 'optind' must be
            // reset to the first element index, because there is
            // no program name inside the new 'argv[]' at index zero.
            //
            argv   = list;
            argc   = l;
            optind = 0;
          }
      }
    //
    // Scan arguments, or list converted into 'argv[]'.
    //
    for (; optind < argc; optind++)
      {
        if (argv[optind][0] == '.')
          {
            //
            // Current name starts with '.'.
```

```
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_ISSOCK (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,
```



```
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
```

```
    // file or to quit is given.
    //
    for (loop = 1; loop; )
      {
        //
        // Read a single character.
        //
        c = getc (fp);
        //
        if (c == EOF)
          {
            loop = 0;
            break;
          }
        //
        // If the character read is a special one,
        // the line/column calculation is modified,
        // so that it is known when to stop scrolling.
        //
        switch (c)
          {
            case '\r':
                //
                // Displaying this character, the cursor should go
                // back to the first column. So the column counter
                // is reset.
                //
                column = 1;
                break;
            case '\n':
                //
                // Displaying this character, the cursor should go
                // back to the next line, at the first column.
                // So the column counter is reset and the line
                // counter is incremented.
                //
                line++;
                column = 1;
                break;
            case '\b':
                //
                // Displaying this character, the cursor should go
                // back one position, unless it is already at the
                // beginning.
                //
                if (column > 1)
                  {
                    column--;
                  }
                break;
            default:
                //
                // Any other character must increase the column
                // counter.
                //
                column++;
          }
        //
        // Display the character, even if it is a special one:
        // it is responsibility of the screen device management
        // to do something good with special characters.
        //
        putchar (c);
        //
        // If the column counter is gone beyond the screen columns,
        // then adjust the column counter and increment the line
        // counter.
        //
        if (column > MAX_COLUMNS)
          {
            column -= MAX_COLUMNS;
            line++;
          }
        //
        // Check if there is space for scrolling.
        //
        if (line < MAX_LINES)
          {
            continue;
          }
        //
        // Here, displayed lines are MAX_LINES.
        //
        if (column > 1)
          {
            //
            // Something was printed at the current line: must
            // do a new line.
            //
            putchar ('\n');
          }
        //
        // Show the more prompt.
        //
        printf ("--More--");
        fflush (stdout);
        //
        // Read a character from standard input.
        //
        c = getchar ();
        //
```



```
        // Consider command 'q', but any other character
        // can be introduced, to let show the next page.
        //
        switch (c)
          {
            case 'Q':
            case 'q':
                //
                // Quit. But must erase the '--More--' prompt.
                //
                printf ("\b \b\b \b\b \b\b \b\b \b");
                printf ("\b \b\b \b\b \b\b \b");
                fclose (fp);
                return (0);
          }
        //
        // Backspace to overwrite '--More--' and the character
        // pressed.
        //
        printf ("\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b");
        //
        // Reset line/column counters.
        //
        column = 1;
        line = 1;
      }
    //
    // Close the file pointer if it is still open.
    //
    if (fp != NULL)
      {
        fclose (fp);
      }
    //
    return (0);
}
//-----------------------------------------------------------------------
static void
usage (void)
{
    fprintf (stderr, "Usage: man [SECTION] NAME\n");
}
//-----------------------------------------------------------------------
FILE *
open_man_page (int section, char *name)
{
    FILE       *fp;
    char        path[PATH_MAX];
    struct stat file_status;
    //
    //
    //
    if (section > 0)
      {
        build_path_name (section, name, path);
        //
        // Check if file exists.
        //
        if (stat (path, &file_status) != 0)
          {
            fprintf (stderr, "Man page %s(%i) does not exist!\n",
                     name, section);
            return (NULL);
          }
      }
    else
      {
        //
        // Must try a section.
        //
        for (section = 1; section < 9; section++)
          {
            build_path_name (section, name, path);
            //
            // Check if file exists.
            //
            if (stat (path, &file_status) == 0)
              {
                //
                // Found.
                //
                break;
              }
          }
      }
    //
    // Check if a file was found.
    //
    if (section < 9)
      {
        fp = fopen (path, "r");
        //
        if (fp == NULL)
          {
            //
            // Error opening file.
            //
            perror (path);
            return (NULL);
          }
        else
```

```
          {
            //
            // Opened right.
            //
            return (fp);
          }
      }
    else
      {
        fprintf (stderr, "Man page %s does not exist!\n",
                 name);
        return (NULL);
      }
}
//---------------------------------------------------------------------
void
build_path_name (int section, char *name, char *path)
{
    char  string_section[10];
    //
    // Convert the section number into a string.
    //
    sprintf (string_section, "%i", section);
    //
    // Prepare the path to the man file.
    //
    path[0] = 0;
    strcat (path, man_page_directory);
    strcat (path, "/");
    strcat (path, name);
    strcat (path, ".");
    strcat (path, string_section);
}
```

applic/mkdir.c

«

Si veda la sezione u0.15.

```
#include <sys/os16.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <libgen.h>
//---------------------------------------------------------------------
static int  mkdir_parents (const char *path, mode_t mode);
static void usage         (void);
//---------------------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    sysmsg_uarea_t msg;
    int            status;
    mode_t         mode     = 0;
    int            m;           // Index inside mode argument.
    int            digit;
    char         **dir;
    int            d;           // Directory index.
    int            option_p = 0;
    int            option_m = 0;
    int            opt;
    extern char   *optarg;
    extern int     optind;
    extern int     optopt;
    //
    // There must be at least an argument, plus the program name.
    //
    if (argc < 2)
      {
        usage ();
        return (1);
      }
    //
    // Check for options, starting from 'p'. The 'dir' pointer is used
    // to calculate the argument pointer to the first directory [1].
    // The macro-instruction 'max()' is declared inside <sys/os16.h>
    // and does the expected thing.
    //
    while ((opt = getopt (argc, argv, ":pm:")) != -1)
      {
        switch (opt)
          {
            case 'm':
                option_m = 1;
                for (m = 0; m < strlen (optarg); m++)
                  {
                    digit = (optarg[m] - '0');
                    if (digit < 0 || digit > 7)
                      {
                        usage ();
                        return (2);
                      }
                    mode = mode * 8 + digit;
                  }
                break;
```



```
            case 'p':
                option_p = 1;
                break;
            case '?':
                printf ("Unknown option -%c.\n", optopt);
                usage ();
                return (1);
                break;
            case ':':
                printf ("Missing argument for option -%c\n", optopt);
                usage ();
                return (2);
                break;
            default:
                printf ("Getopt problem: unknown option %c\n", opt);
                return (3);
          }
      }
    //
    dir = argv + optind;
    //
    // Check if the mode is to be set to a default value.
    //
    if (!option_m)
      {
        //
        // Default mode.
        //
        sys (SYS_UAREA, &msg, (sizeof msg));
        mode = 0777 & ~msg.umask;
      }
    //
    // Directory creation.
    //
    for (d = 0; dir[d] != NULL; d++)
      {
        if (option_p)
          {
            status = mkdir_parents (dir[d], mode);
            if (status != 0)
             {
                perror (dir[d]);
                return (3);
             }
          }
        else
          {
            status = mkdir (dir[d], mode);
            if (status != 0)
              {
                perror (dir[d]);
                return (4);
              }
          }
      }
    //
    // All done.
    //
    return (0);
}
//---------------------------------------------------------------------
static int
mkdir_parents (const char *path, mode_t mode)
{
    char        path_copy[PATH_MAX];
    char       *path_parent;
    struct stat fst;
    int         status;
    //
    // Check if the path is empty.
    //
    if (path == NULL || strlen (path) == 0)
      {
        //
        // Recursion ends here.
        //
        return (0);
      }
    //
    // Check if it does already exists.
    //
    status = stat (path, &fst);
    if (status == 0 && fst.st_mode & S_IFDIR)
      {
        //
        // The path exists and is a directory.
        //
        return (0);
      }
    else if (status == 0 && !(fst.st_mode & S_IFDIR))
      {
        //
        // The path exists but is not a directory.
        //
        errno = ENOTDIR;        // Not a directory.
        return (-1);
      }
    //
    // Get the directory path.
    //
    strncpy (path_copy, path, PATH_MAX);
```

```
    path_parent = dirname (path_copy);
    //
    // If it is '.', or '/', the recursion is terminated.
    //
    if (strncmp (path_parent, ".", PATH_MAX) == 0  ||
        strncmp (path_parent, "/", PATH_MAX) == 0)
      {
        return (0);
      }
    //
    // Otherwise, continue the recursion.
    //
    status = mkdir_parents (path_parent, mode);
    if (status != 0)
      {
        return (-1);
      }
    //
    // Previous directories are there: create the current one.
    //
    status = mkdir (path, mode);
    if (status)
      {
        perror (path);
        return (-1);
      }

    return (0);
}
//-----------------------------------------------------------------------
static void
usage (void)
{
    fprintf (stderr, "Usage: mkdir [-p] [-m OCTAL_MODE] DIR...\n");
}
```

applic/more.c

« Si veda la sezione u0.16.

```
#include <unistd.h>
#include <errno.h>
//-----------------------------------------------------------------------
#define MAX_LINES   20
#define MAX_COLUMNS 80
//-----------------------------------------------------------------------
static void usage (void);
//-----------------------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    FILE   *fp;
    char   *name;
    int     c;
    int     line   = 1; // Line internal counter.
    int     column = 1; // Column internal counter.
    int     a;          // Index inside arguments.
    int     loop;
    //
    // There must be at least an argument, plus the program name.
    //
    if (argc < 2)
      {
        usage ();
        return (1);
      }
    //
    // No options are allowed.
    //
    for (a = 1; a < argc ; a++)
      {
        //
        // Get next name from arguments.
        //
        name = argv[a];
        //
        // Try to open the file, read only.
        //
        fp = fopen (name, "r");
        //
        if (fp == NULL)
          {
            //
            // Error opening file.
            //
            perror (name);
            return (1);
          }
        //
        // Print the file name to be displayed.
        //
        printf ("== %s ==\n", name);
        line++;
        //
        // The following loop continues while the file
        // gives characters, or when a command to change
        // file or to quit is given.
        //
        for (loop = 1; loop; )
          {
            //
```



```
            // Read a single character.
            //
            c = getc (fp);
            //
            if (c == EOF)
              {
                loop = 0;
                break;
              }
            //
            // If the character read is a special one,
            // the line/column calculation is modified,
            // so that it is known when to stop scrolling.
            //
            switch (c)
              {
                case '\r':
                    //
                    // Displaying this character, the cursor should go
                    // back to the first column. So the column counter
                    // is reset.
                    //
                    column = 1;
                    break;
                case '\n':
                    //
                    // Displaying this character, the cursor should go
                    // back to the next line, at the first column.
                    // So the column counter is reset and the line
                    // counter is incremented.
                    //
                    line++;
                    column = 1;
                    break;
                case '\b':
                    //
                    // Displaying this character, the cursor should go
                    // back one position, unless it is already at the
                    // beginning.
                    //
                    if (column > 1)
                      {
                        column--;
                      }
                    break;
                default:
                    //
                    // Any other character must increase the column
                    // counter.
                    //
                    column++;
              }
            //
            // Display the character, even if it is a special one:
            // it is responsibility of the screen device management
            // to do something good with special characters.
            //
            putchar (c);
            //
            // If the column counter is gone beyond the screen columns,
            // then adjust the column counter and increment the line
            // counter.
            //
            if (column > MAX_COLUMNS)
              {
                column -= MAX_COLUMNS;
                line++;
              }
            //
            // Check if there is space for scrolling.
            //
            if (line < MAX_LINES)
              {
                continue;
              }
            //
            // Here, displayed lines are MAX_LINES.
            //
            if (column > 1)
              {
                //
                // Something was printed at the current line: must
                // do a new line.
                //
                putchar ('\n');
              }
            //
            // Show the more prompt.
            //
            printf ("--More--");
            fflush (stdout);
            //
            // Read a character from standard input.
            //
            c = getchar ();
            //
            // Consider commands 'n' and 'q', but any other character
            // can be introduced, to let show the next page.
            //
            switch (c)
              {
```

```
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");
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/os16.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 pg                                                        \n"
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->device_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->name);
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
4360071     print_proc_head ();
4360072     for (pid = 0; pid < PROCESS_MAX; pid++)
4360073       {
```

```
4360074         lseek (fd, (off_t) pid, SEEK_SET);
4360075         size_read = read (fd, buffer, sizeof (proc_t));
4360076         if (size_read < sizeof (proc_t))
4360077           {
4360078             printf ("ps: cannot read \"/dev/kmem_ps\" pid %i", pid);
4360079             perror (NULL);
4360080             continue;
4360081           }
4360082         ps = (proc_t *) buffer;
4360083         if (ps->status > 0)
4360084           {
4360085             ps->name[PATH_MAX-1] = 0; // Terminated string.
4360086             print_proc_pid (ps, pid);
4360087           }
4360088       }
4360089
4360090     close (fd);
4360091     return (0);
4360092 }
```

applic/rm.c

«

Si veda la sezione u0.18.

```
4370001 #include <fcntl.h>
4370002 #include <sys/stat.h>
4370003 #include <stddef.h>
4370004 #include <unistd.h>
4370005 #include <errno.h>
4370006 //-------------------------------------------------------------------------
4370007 static void usage (void);
4370008 //-------------------------------------------------------------------------
4370009 int
4370010 main (int argc, char *argv[], char *envp[])
4370011 {
4370012     int         a;              // Argument index.
4370013     int         status;
4370014     struct stat file_status;
4370015     //
4370016     // No options are known, but at least an argument must be given.
4370017     //
4370018     if (argc < 2)
4370019       {
4370020         usage ();
4370021         return (1);
4370022       }
4370023     //
4370024     // Scan arguments.
4370025     //
4370026     for(a = 1; a < argc; a++)
4370027       {
4370028         //
4370029         // Verify if the file exists.
4370030         //
4370031         if (stat(argv[a], &file_status) != 0)
4370032           {
4370033             fprintf (stderr, "File \"%s\" does not exist!\n",
4370034                      argv[a]);
4370035             continue;
4370036           }
4370037         //
4370038         // File exists: check the file type.
4370039         //
4370040         if (S_ISDIR (file_status.st_mode))
4370041           {
4370042             fprintf (stderr, "Cannot remove directory \"%s\"!\n",
4370043                      argv[a]);
4370044             continue;
4370045           }
4370046         //
4370047         // Can remove it.
4370048         //
4370049         status = unlink (argv[a]);
4370050         if (status != 0)
4370051           {
4370052             perror (NULL);
4370053             return (2);
4370054           }
4370055       }
4370056     return (0);
4370057 }
4370058 //-------------------------------------------------------------------------
4370059 static void
4370060 usage (void)
4370061 {
4370062     fprintf (stderr, "Usage: rm FILE...\n");
4370063 }
```

applic/shell.c

«

Si veda la sezione u0.19.

```
4380001 #include <unistd.h>
4380002 #include <stdlib.h>
4380003 #include <sys/stat.h>
4380004 #include <sys/types.h>
4380005 #include <fcntl.h>
4380006 #include <errno.h>
4380007 #include <unistd.h>
4380008 #include <signal.h>
```



```
4380009 #include <stdio.h>
4380010 #include <sys/wait.h>
4380011 #include <stdio.h>
4380012 #include <string.h>
4380013 #include <limits.h>
4380014 #include <sys/os16.h>
4380015 //-------------------------------------------------------------------------
4380016 #define PROMPT_SIZE     16
4380017 //-------------------------------------------------------------------------
4380018 static void sh_cd       (int argc, char *argv[]);
4380019 static void sh_pwd      (int argc, char *argv[]);
4380020 static void sh_umask    (int argc, char *argv[]);
4380021 //-------------------------------------------------------------------------
4380022 int
4380023 main (int argc, char *argv[], char *envp[])
4380024 {
4380025   char    buffer_cmd[ARG_MAX/2];
4380026   char   *argv_cmd[ARG_MAX/16];
4380027   char    prompt[PROMPT_SIZE];
4380028   uid_t   uid;
4380029   int     argc_cmd;
4380030   pid_t   pid_cmd;
4380031   pid_t   pid_dead;
4380032   int     status;
4380033   //
4380034   //
4380035   //
4380036   uid = geteuid ();
4380037   //
4380038   // Load processes, reading the keyboard.
4380039   //
4380040   while (1)
4380041     {
4380042       if (uid == 0)
4380043         {
4380044           strncpy (prompt, "# ", PROMPT_SIZE);
4380045         }
4380046       else
4380047         {
4380048           strncpy (prompt, "$ ", PROMPT_SIZE);
4380049         }
4380050       //
4380051       input_line (buffer_cmd, prompt, (ARG_MAX/2), INPUT_LINE_ECHO);
4380052       //
4380053       // Clear 'argv_cmd[]';
4380054       //
4380055       for (argc_cmd = 0; argc_cmd < (ARG_MAX/16); argc_cmd++)
4380056         {
4380057           argv_cmd[argc_cmd] = NULL;
4380058         }
4380059       //
4380060       // Initialize the command scan.
4380061       //
4380062       argv_cmd[0] = strtok (buffer_cmd, " \t");
4380063       //
4380064       // Verify: if the input is not valid, loop again.
4380065       //
4380066       if (argv_cmd[0] == NULL)
4380067         {
4380068           continue;
4380069         }
4380070       //
4380071       // Find the arguments.
4380072       //
4380073       for (argc_cmd = 1;
4380074            argc_cmd < ((ARG_MAX/16)-1) && argv_cmd[argc_cmd-1] != NULL;
4380075            argc_cmd++)
4380076         {
4380077           argv_cmd[argc_cmd] = strtok (NULL, " \t");
4380078         }
4380079       //
4380080       // If there are too many arguments, show a message and continue.
4380081       //
4380082       if (argv_cmd[argc_cmd-1] != NULL)
4380083         {
4380084           errset (E2BIG);             // Argument list too long.
4380085           perror (NULL);
4380086           continue;
4380087         }
4380088       //
4380089       // Correct the value for 'argc_cmd', because actually
4380090       // it counts also the NULL element.
4380091       //
4380092       argc_cmd--;
4380093       //
4380094       // Verify if it is an internal command.
4380095       //
4380096       if      (strcmp (argv_cmd[0], "exit") == 0)
4380097         {
4380098           return (0);
4380099         }
4380100       else if (strcmp (argv_cmd[0], "cd") == 0)
4380101         {
4380102           sh_cd (argc_cmd, argv_cmd);
4380103           continue;
4380104         }
4380105       else if (strcmp (argv_cmd[0], "pwd") == 0)
4380106         {
4380107           sh_pwd (argc_cmd, argv_cmd);
4380108           continue;
4380109         }
```

```
4380110       else if (strcmp (argv_cmd[0], "umask") == 0)
4380111         {
4380112           sh_umask (argc_cmd, argv_cmd);
4380113           continue;
4380114         }
4380115       //
4380116       // It should be a program to run.
4380117       //
4380118       pid_cmd = fork ();
4380119       if (pid_cmd == -1)
4380120         {
4380121           printf ("%s: cannot run command", argv[0]);
4380122           perror (NULL);
4380123         }
4380124       else if (pid_cmd == 0)
4380125         {
4380126           execvp (argv_cmd[0], argv_cmd);
4380127           perror (NULL);
4380128           exit (0);
4380129         }
4380130       while (1)
4380131         {
4380132           pid_dead = wait (&status);
4380133           if (pid_dead == pid_cmd)
4380134             {
4380135               break;
4380136             }
4380137         }
4380138       printf ("pid %i terminated with status %i.\n",
4380139               (int) pid_dead, status);
4380140     }
4380141 }
4380142 //---------------------------------------------------------------------
4380143 static void
4380144 sh_cd (int argc, char *argv[])
4380145 {
4380146   int status;
4380147   //
4380148   if (argc != 2)
4380149     {
4380150       errset (EINVAL);                // Invalid argument.
4380151       perror (NULL);
4380152       return;
4380153     }
4380154   //
4380155   status = chdir (argv[1]);
4380156   if (status != 0)
4380157     {
4380158       perror (NULL);
4380159     }
4380160   return;
4380161 }
4380162 //---------------------------------------------------------------------
4380163 static void
4380164 sh_pwd (int argc, char *argv[])
4380165 {
4380166   char  path[PATH_MAX];
4380167   void *pstatus;
4380168   //
4380169   if (argc != 1)
4380170     {
4380171       errset (EINVAL);                // Invalid argument.
4380172       perror (NULL);
4380173       return;
4380174     }
4380175   //
4380176   // Get the current directory.
4380177   //
4380178   pstatus = getcwd (path, (size_t) PATH_MAX);
4380179   if (pstatus == NULL)
4380180     {
4380181       perror (NULL);
4380182     }
4380183   else
4380184     {
4380185       printf ("%s\n", path);
4380186     }
4380187   return;
4380188 }
4380189 //---------------------------------------------------------------------
4380190 static void
4380191 sh_umask (int argc, char *argv[])
4380192 {
4380193   sysmsg_uarea_t msg;
4380194   char          *m;           // Index inside the umask octal string.
4380195   int            mask;
4380196   int            digit;
4380197   //
4380198   if (argc > 2)
4380199     {
4380200       errset (EINVAL);                // Invalid argument.
4380201       perror (NULL);
4380202       return;
4380203     }
4380204   //
4380205   // If no argument is available, the umask is shown, with a direct
4380206   // system call.
4380207   //
4380208   if (argc == 1)
4380209     {
4380210       sys (SYS_UAREA, &msg, (sizeof msg));
```



```
4380211       printf ("%04o\n", msg.umask);
4380212       return;
4380213     }
4380214   //
4380215   // Get the mask: must be the first argument.
4380216   //
4380217   for (mask = 0, m = argv[1]; *m != 0; m++)
4380218     {
4380219       digit = (*m - '0');
4380220       if (digit < 0 || digit > 7)
4380221         {
4380222           errset (EINVAL);            // Invalid argument.
4380223           perror (NULL);
4380224           return;
4380225         }
4380226       mask = mask * 8 + digit;
4380227     }
4380228   //
4380229   // Set the umask and return.
4380230   //
4380231   umask (mask);
4380232   return;
4380233 }
```

applic/touch.c

«

Si veda la sezione u0.20.

```
4390001 #include <fcntl.h>
4390002 #include <sys/stat.h>
4390003 #include <utime.h>
4390004 #include <stddef.h>
4390005 #include <unistd.h>
4390006 #include <errno.h>
4390007 //---------------------------------------------------------------------
4390008 static void usage (void);
4390009 //---------------------------------------------------------------------
4390010 int
4390011 main (int argc, char *argv[], char *envp[])
4390012 {
4390013     int         a;              // Argument index.
4390014     int         status;
4390015     struct stat file_status;
4390016     //
4390017     // No options are known, but at least an argument must be given.
4390018     //
4390019     if (argc < 2)
4390020       {
4390021         usage ();
4390022         return (1);
4390023       }
4390024     //
4390025     // Scan arguments.
4390026     //
4390027     for(a = 1; a < argc; a++)
4390028       {
4390029         //
4390030         // Verify if the file exists, through the return value of
4390031         // 'stat()'. No other checks are made.
4390032         //
4390033         if (stat(argv[a], &file_status) == 0)
4390034          {
4390035             //
4390036             // File exists: should be updated the times.
4390037             //
4390038             status = utime (argv[a], NULL);
4390039             if (status != 0)
4390040               {
4390041                 perror (NULL);
4390042                 return (2);
4390043               }
4390044          }
4390045         else
4390046           {
4390047             //
4390048             // File does not exist: should be created.
4390049             //
4390050             status = open (argv[a], O_WRONLY|O_CREAT|O_TRUNC, 0666);
4390051             //
4390052             if (status >= 0)
4390053               {
4390054                 //
4390055                 // Here, the variable 'status' is the file
4390056                 // descriptor to be closed.
4390057                 //
4390058                 status = close (status);
4390059                 if (status != 0)
4390060                   {
4390061                     perror (NULL);
4390062                     return (3);
4390063                   }
4390064               }
4390065             else
4390066               {
4390067                 perror (NULL);
4390068                 return (4);
4390069               }
4390070           }
4390071       }
4390072     return (0);
4390073 }
```

```
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/os16.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 //---------------------------------------------------------------------
4400054 static void
4400055 usage (void)
4400056 {
4400057   fprintf (stderr, "Usage: tty\n");
4400058 }
```

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/os16.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 ();
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 }
```

1900