

os16: directory «applic/»	1850
applic/MAKEDEV.c	1850
applic/aaa.c	1850
applic/bbb.c	1851
applic/cat.c	1851
applic/ccc.c	1852
applic/chmod.c	1852
applic/chown.c	1853
applic/cp.c	1854
applic/crt0.s	1856
applic/date.c	1857
applic/ed.c	1859
applic/getty.c	1873
applic/init.c	1874
applic/kill.c	1876
applic/ln.c	1879
applic/login.c	1880
applic/ls.c	1882
applic/man.c	1885
applic/mkdir.c	1888
applic/more.c	1890
applic/mount.c	1892
applic/ps.c	1893
applic/rm.c	1894
applic/shell.c	1894
applic/touch.c	1897
applic/tty.c	1898
applic/umount.c	1898

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

os16: directory «applic/»	1850
applic/MAKEDEV.c	1850
applic/aaa.c	1850
applic/bbb.c	1851
applic/cat.c	1851
applic/ccc.c	1852
applic/chmod.c	1852
applic/chown.c	1853
applic/cp.c	1854
applic/crt0.s	1856
applic/date.c	1857
applic/ed.c	1859
applic/getty.c	1873
applic/init.c	1874
applic/kill.c	1876
applic/ln.c	1879
applic/login.c	1880
applic/ls.c	1882
applic/man.c	1885
applic/mkdir.c	1888
applic/more.c	1890

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

os16: directory «applic/»

applic/MAKEDEV.c

Si veda la sezione u0.3.

```

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

```

applic/aaa.c

Si veda la sezione u0.1.

```

4180001 #include <unistd.h>
4180002 #include <stdio.h>
4180003 //-----
4180004 int

```

```

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

```

applic/bbb.c

Si veda la sezione u0.1.

```

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

```

applic/cat.c

Si veda la sezione u0.3.

```

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

```

```

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

```

applic/ccc.c

« Si veda la sezione u0.1.

```

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

```

applic/chmod.c

« Si veda la sezione u0.5.

```

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

```

1852

```

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

```

applic/chown.c

« Si veda la sezione u0.6.

```

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

```

1853

```

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

```

applic/cp.c

<

Si veda la sezione u0.7.

```

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

```

1854

```

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

```

1855

```

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

```

applic/crt0.s

Si veda la sezione u0.2.

```

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

```

1856

```

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

```

applic/date.c

Si veda la sezione u0.8.

```

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

```

1857

```

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

```

1858

```

4240129 }

```

applic/ed.c

Si veda la sezione u0.9.

```

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

```

1859

```

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

```

1860

```

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

```

1861

```

4250298     break;
4250299     //
4250300 case 'q':
4250301     while (isblank(*cp))
4250302     {
4250303         cp++;
4250304     }
4250305     //
4250306     if (have1 || *cp)
4250307     {
4250308         fprintf(stderr, "Bad quit command\n");
4250309         break;
4250310     }
4250311     //
4250312     if (!dirty)
4250313     {
4250314         return;
4250315     }
4250316     //
4250317     printf ("Really quit? ");
4250318     fflush (stdout);
4250319     //
4250320     buf[0] = '\0';
4250321     fgets (buf, sizeof(buf), stdin);
4250322     cp = buf;
4250323     //
4250324     while (isblank (*cp))
4250325     {
4250326         cp++;
4250327     }
4250328     //
4250329     if ((*cp == 'y' || *cp == 'Y'))
4250330     {
4250331         return;
4250332     }
4250333     //
4250334     break;
4250335     //
4250336 case 'x':
4250337     if (*cp && !isblank(*cp))
4250338     {
4250339         fprintf (stderr, "Bad read command\n");
4250340         break;
4250341     }
4250342     //
4250343     while (isblank(*cp))
4250344     {
4250345         cp++;
4250346     }
4250347     //
4250348     if (*cp == '\0')
4250349     {
4250350         fprintf (stderr, "No filename\n");
4250351         break;
4250352     }
4250353     //
4250354     if (!have1)
4250355     {
4250356         num1 = lastnum;
4250357     }
4250358     //
4250359     // Open the file and add to the buffer
4250360     // at the next line.
4250361     //
4250362     if (readlines (cp, num1 + 1))
4250363     {
4250364         //
4250365         // If the file open fails, just
4250366         // break the command.
4250367         //
4250368         break;
4250369     }
4250370     //
4250371     // Set the default file name, if no
4250372     // previous name is available.
4250373     //
4250374     if (filename == NULL)
4250375     {
4250376         filename = strdup (cp);
4250377     }
4250378     //
4250379     break;
4250380
4250381 case 's':
4250382     subcommand (cp, num1, num2);
4250383     break;
4250384     //
4250385 case 'w':
4250386     if (*cp && !isblank(*cp))
4250387     {
4250388         fprintf(stderr, "Bad write command\n");
4250389         break;
4250390     }
4250391     //
4250392     while (isblank(*cp))
4250393     {
4250394         cp++;
4250395     }
4250396     //
4250397     if (!have1)
4250398     {

```

```

4250399         num1 = 1;
4250400         num2 = lastnum;
4250401     }
4250402     //
4250403     // If the file name is not specified, use the
4250404     // default one.
4250405     //
4250406     if (*cp == '\0')
4250407     {
4250408         cp = filename;
4250409     }
4250410     //
4250411     // If even the default file name is not specified,
4250412     // tell it.
4250413     //
4250414     if (cp == NULL)
4250415     {
4250416         fprintf (stderr, "No file name specified\n");
4250417         break;
4250418     }
4250419     //
4250420     // Write the file.
4250421     //
4250422     writelines (cp, num1, num2);
4250423     //
4250424     break;
4250425     //
4250426 case 'z':
4250427     switch (*cp)
4250428     {
4250429         case '-':
4250430             printlines (curnum-21, curnum, false);
4250431             break;
4250432         case '.':
4250433             printlines (curnum-11, curnum+10, false);
4250434             break;
4250435         default:
4250436             printlines (curnum, curnum+21, false);
4250437             break;
4250438     }
4250439     break;
4250440     //
4250441 case '.':
4250442     if (have1)
4250443     {
4250444         fprintf (stderr, "No arguments allowed\n");
4250445         break;
4250446     }
4250447     printlines (curnum, curnum, false);
4250448     break;
4250449     //
4250450 case '-':
4250451     if (setcurnum (curnum - 1))
4250452     {
4250453         printlines (curnum, curnum, false);
4250454     }
4250455     break;
4250456     //
4250457 case '=':
4250458     printf ("%d\n", num1);
4250459     break;
4250460     //
4250461 case '\0':
4250462     if (have1)
4250463     {
4250464         printlines (num2, num2, false);
4250465         break;
4250466     }
4250467     //
4250468     if (setcurnum (curnum + 1))
4250469     {
4250470         printlines (curnum, curnum, false);
4250471     }
4250472     break;
4250473     //
4250474     default:
4250475         fprintf (stderr, "Unimplemented command\n");
4250476         break;
4250477     }
4250478     }
4250479 }
4250480 //-----
4250481 // Do the substitute command.
4250482 // The current line is set to the last substitution done.
4250483 //-----
4250484 void
4250485 subcommand (char *cp, num_t num1, num_t num2)
4250486 {
4250487     int    delim;
4250488     char  *oldstr;
4250489     char  *newstr;
4250490     len_t  oldlen;
4250491     len_t  newlen;
4250492     len_t  deltalen;
4250493     len_t  offset;
4250494     line_t *lp;
4250495     line_t *nlp;
4250496     bool   globalflag;
4250497     bool   printflag;
4250498     bool   didsub;
4250499     bool   needprint;

```



```

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

```

1864

```

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

```

1865

```

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

```

```

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

```

```

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

```

1868

```

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

```

1869

```

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

```

1870

```

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

```

1871

```

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

```

1872

```

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

```

applic/getty.c

Si veda la sezione u0.1.

«

```

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

```

1873

```

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

```

applic/init.c

« [Si veda la sezione u0.2.](#)

```

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

```

1874

```

427031 char *inittab_runlevels;
427032 char *inittab_action;
427033 char *inittab_process;
427034 int eof;
427035 int fd;
427036 //
427037 // It follows a table for commands to be respawn.
427038 //
427039 struct {
427040 pid_t pid;
427041 char command[COMMAND_MAX];
427042 } respawn[RESPAWN_MAX];
427043 //-----
427044 signal (SIGHUP, SIG_IGN);
427045 signal (SIGINT, SIG_IGN);
427046 signal (SIGQUIT, SIG_IGN);
427047 signal (SIGILL, SIG_IGN);
427048 signal (SIGABRT, SIG_IGN);
427049 signal (SIGFPE, SIG_IGN);
427050 // signal (SIGKILL, SIG_IGN); Cannot ignore SIGKILL.
427051 signal (SIGSEGV, SIG_IGN);
427052 signal (SIGPIPE, SIG_IGN);
427053 signal (SIGALRM, SIG_IGN);
427054 signal (SIGTERM, SIG_IGN);
427055 // signal (SIGSTOP, SIG_IGN); Cannot ignore SIGSTOP.
427056 signal (SIGTSTP, SIG_IGN);
427057 signal (SIGCONT, SIG_IGN);
427058 signal (SIGTTIN, SIG_IGN);
427059 signal (SIGTTOU, SIG_IGN);
427060 signal (SIGUSR1, SIG_IGN);
427061 signal (SIGUSR2, SIG_IGN);
427062 //-----
427063 printf ("init\n");
427064 // heap_clear ();
427065 // process_info ();
427066 //-----
427067 //
427068 // Reset the 'respawn' table.
427069 //
427070 for (r = 0; r < RESPAWN_MAX; r++)
427071 {
427072 respawn[r].pid = 0;
427073 respawn[r].command[0] = 0;
427074 respawn[r].command[COMMAND_MAX-1] = 0;
427075 }
427076 //
427077 // Read the '/etc/inittab' file.
427078 //
427079 fd = open ("/etc/inittab", O_RDONLY);
427080 //
427081 if (fd < 0)
427082 {
427083 perror ("Cannot open file '/etc/inittab'");
427084 exit (-1);
427085 }
427086 //
427087 //
427088 //
427089 //
427090 for (eof = 0, r = 0; !eof && r < RESPAWN_MAX; r++)
427091 {
427092 for (b = 0; b < LINE_MAX; b++)
427093 {
427094 size_read = read (fd, &buffer[b], (size_t) 1);
427095 if (size_read <= 0)
427096 {
427097 buffer[b] = 0;
427098 eof = 1; // Close the read loop.
427099 break;
427100 }
427101 if (buffer[b] == '\n')
427102 {
427103 buffer[b] = 0;
427104 break;
427105 }
427106 }
427107 //
427108 // Remove comments: just replace '#' with '\0'.
427109 //
427110 for (b = 0; b < LINE_MAX; b++)
427111 {
427112 if (buffer[b] == '#')
427113 {
427114 buffer[b] = 0;
427115 break;
427116 }
427117 }
427118 //
427119 // If the buffer is an empty string, just loop to next
427120 // record.
427121 //
427122 if (strlen (buffer) == 0)
427123 {
427124 r--;
427125 continue;
427126 }
427127 //
427128 //
427129 //
427130 inittab_id = strtok (buffer, ":");
427131 inittab_runlevels = strtok (NULL, ":");

```

1875

```

4270132     inittab_action = strtok (NULL, "*");
4270133     inittab_process = strtok (NULL, "*");
4270134     //
4270135     // Only action 'respawn' is used.
4270136     //
4270137     if (strcmp (inittab_action, "respawn") == 0)
4270138     {
4270139         strncpy (respawn[r].command, inittab_process, COMMAND_MAX);
4270140     }
4270141     else
4270142     {
4270143         r--;
4270144     }
4270145     }
4270146     //
4270147     //
4270148     //
4270149 close (fd);
4270150 //
4270151 // Define common environment.
4270152 //
4270153 exec_envp[0] = "PATH=/bin:/usr/bin:/sbin:/usr/sbin";
4270154 exec_envp[1] = "CONSOLE=/dev/console";
4270155 exec_envp[2] = NULL;
4270156 //
4270157 // Start processes.
4270158 //
4270159 for (r = 0; r < RESPAWN_MAX; r++)
4270160 {
4270161     if (strlen (respawn[r].command) > 0)
4270162     {
4270163         respawn[r].pid = fork ();
4270164         if (respawn[r].pid == 0)
4270165         {
4270166             exec_argv[0] = strtok (respawn[r].command, " \\t");
4270167             exec_argv[1] = strtok (NULL, " \\t");
4270168             exec_argv[2] = NULL;
4270169             execve (exec_argv[0], exec_argv, exec_envp);
4270170             perror (NULL);
4270171             exit (0);
4270172         }
4270173     }
4270174 }
4270175 //
4270176 // Wait for the death of child.
4270177 //
4270178 while (1)
4270179 {
4270180     pid = wait (&status);
4270181     for (r = 0; r < RESPAWN_MAX; r++)
4270182     {
4270183         if (pid == respawn[r].pid)
4270184         {
4270185             //
4270186             // Run it again.
4270187             //
4270188             respawn[r].pid = fork ();
4270189             if (respawn[r].pid == 0)
4270190             {
4270191                 exec_argv[0] = strtok (respawn[r].command, " \\t");
4270192                 exec_argv[1] = strtok (NULL, " \\t");
4270193                 exec_argv[2] = NULL;
4270194                 execve (exec_argv[0], exec_argv, exec_envp);
4270195                 exit (0);
4270196             }
4270197             break;
4270198         }
4270199     }
4270200 }
4270201 }

```

```

4280027 //
4280028 // There must be at least an option, plus the program name.
4280029 //
4280030 if (argc < 2)
4280031 {
4280032     usage ();
4280033     return (1);
4280034 }
4280035 //
4280036 // Check for options.
4280037 //
4280038 while ((opt = getopt (argc, argv, "l:s:")) != -1)
4280039 {
4280040     switch (opt)
4280041     {
4280042     case 'l':
4280043         option_l = 1;
4280044         break;
4280045     case 's':
4280046         option_s = 1;
4280047         //
4280048         // In that case, there must be at least three arguments:
4280049         // the option, the signal and the process id.
4280050         //
4280051         if (argc < 4)
4280052         {
4280053             usage ();
4280054             return (1);
4280055         }
4280056         //
4280057         // Argument numbers are ok. Check the signal.
4280058         //
4280059         if (strcmp (optarg, "HUP") == 0)
4280060         {
4280061             signal = SIGHUP;
4280062         }
4280063         else if (strcmp (optarg, "INT") == 0)
4280064         {
4280065             signal = SIGINT;
4280066         }
4280067         else if (strcmp (optarg, "QUIT") == 0)
4280068         {
4280069             signal = SIGQUIT;
4280070         }
4280071         else if (strcmp (optarg, "ILL") == 0)
4280072         {
4280073             signal = SIGILL;
4280074         }
4280075         else if (strcmp (optarg, "ABRT") == 0)
4280076         {
4280077             signal = SIGABRT;
4280078         }
4280079         else if (strcmp (optarg, "FPE") == 0)
4280080         {
4280081             signal = SIGFPE;
4280082         }
4280083         else if (strcmp (optarg, "KILL") == 0)
4280084         {
4280085             signal = SIGKILL;
4280086         }
4280087         else if (strcmp (optarg, "SEGV") == 0)
4280088         {
4280089             signal = SIGSEGV;
4280090         }
4280091         else if (strcmp (optarg, "PIPE") == 0)
4280092         {
4280093             signal = SIGPIPE;
4280094         }
4280095         else if (strcmp (optarg, "ALRM") == 0)
4280096         {
4280097             signal = SIGALRM;
4280098         }
4280099         else if (strcmp (optarg, "TERM") == 0)
4280100         {
4280101             signal = SIGTERM;
4280102         }
4280103         else if (strcmp (optarg, "STOP") == 0)
4280104         {
4280105             signal = SIGSTOP;
4280106         }
4280107         else if (strcmp (optarg, "TSTP") == 0)
4280108         {
4280109             signal = SIGTSTP;
4280110         }
4280111         else if (strcmp (optarg, "CONT") == 0)
4280112         {
4280113             signal = SIGCONT;
4280114         }
4280115         else if (strcmp (optarg, "CHLD") == 0)
4280116         {
4280117             signal = SIGCHLD;
4280118         }
4280119         else if (strcmp (optarg, "TTIN") == 0)
4280120         {
4280121             signal = SIGTTIN;
4280122         }
4280123         else if (strcmp (optarg, "TTOU") == 0)
4280124         {
4280125             signal = SIGTTOU;
4280126         }
4280127         else if (strcmp (optarg, "USR1") == 0)

```

applic/kill.c

« Si veda la sezione u0.10.

```

4280001 #include <sys/os16.h>
4280002 #include <sys/stat.h>
4280003 #include <sys/types.h>
4280004 #include <unistd.h>
4280005 #include <stdlib.h>
4280006 #include <fcntl.h>
4280007 #include <errno.h>
4280008 #include <signal.h>
4280009 #include <stdio.h>
4280010 #include <string.h>
4280011 #include <limits.h>
4280012 #include <libgen.h>
4280013 //-----
4280014 static void usage (void);
4280015 //-----
4280016 int
4280017 main (int argc, char *argv[], char *envp[])
4280018 {
4280019     int     signal;
4280020     int     pid;
4280021     int     a; // Index inside arguments.
4280022     int     option_s = 0;
4280023     int     option_l = 0;
4280024     int     opt;
4280025     extern char *optarg;
4280026     extern int  optopt;

```

```

4280128     {
4280129         signal = SIGUSR1;
4280130     }
4280131     else if (strcmp (optarg, "USR2") == 0)
4280132     {
4280133         signal = SIGUSR2;
4280134     }
4280135     else
4280136     {
4280137         fprintf (stderr, "Unknown signal %s.\n", optarg);
4280138         return (1);
4280139     }
4280140     break;
4280141     case '?':
4280142         fprintf (stderr, "Unknown option -%c.\n", optopt);
4280143         usage ();
4280144         return (1);
4280145     break;
4280146     case ':':
4280147         fprintf (stderr, "Missing argument for option -%c\n",
4280148                 optopt);
4280149         usage ();
4280150         return (1);
4280151     break;
4280152     default:
4280153         fprintf (stderr, "Getopt problem: unknown option %c\n",
4280154                 opt);
4280155         return (1);
4280156     }
4280157 }
4280158 //
4280159 //
4280160 //
4280161 if (option_l && option_s)
4280162 {
4280163     fprintf (stderr, "Options \"-l\" and \"-s\" together ");
4280164     fprintf (stderr, "are incompatible.\n");
4280165     usage ();
4280166     return (1);
4280167 }
4280168 //
4280169 // Option "-l".
4280170 //
4280171 if (option_l)
4280172 {
4280173     printf ("HUP ");
4280174     printf ("INT ");
4280175     printf ("QUIT ");
4280176     printf ("ILL ");
4280177     printf ("ABRT ");
4280178     printf ("FPE ");
4280179     printf ("KILL ");
4280180     printf ("SEGV ");
4280181     printf ("PIPE ");
4280182     printf ("ALRM ");
4280183     printf ("TERM ");
4280184     printf ("STOP ");
4280185     printf ("TSTP ");
4280186     printf ("CONT ");
4280187     printf ("CHLD ");
4280188     printf ("TTIN ");
4280189     printf ("TTOU ");
4280190     printf ("USR1 ");
4280191     printf ("USR2 ");
4280192     printf ("\n");
4280193 }
4280194 //
4280195 // Option "-s".
4280196 //
4280197 if (option_s)
4280198 {
4280199     //
4280200     // Scan arguments.
4280201     //
4280202     for (a = 3; a < argc; a++)
4280203     {
4280204         //
4280205         // Get PID.
4280206         //
4280207         pid = atoi (argv[a]);
4280208         if (pid > 0)
4280209         {
4280210             //
4280211             // Kill.
4280212             //
4280213             if (kill (pid, signal) < 0)
4280214             {
4280215                 perror (argv[a]);
4280216             }
4280217         }
4280218         else
4280219         {
4280220             fprintf (stderr, "Invalid PID %s.", argv[a]);
4280221         }
4280222     }
4280223 }
4280224 //
4280225 // All done.
4280226 //
4280227 return (0);
4280228 }

```

1878

```

4280229 //-----
4280230 static void
4280231 usage (void)
4280232 {
4280233     fprintf (stderr, "Usage: kill -s SIGNAL_NAME PID...\n");
4280234     fprintf (stderr, "        kill -l\n");
4280235 }

```

applic/ln.c

Si veda la sezione u0.11.

«

```

4280001 #include <sys/osal6.h>
4280002 #include <sys/stat.h>
4280003 #include <sys/types.h>
4280004 #include <unistd.h>
4280005 #include <stdlib.h>
4280006 #include <fcntl.h>
4280007 #include <errno.h>
4280008 #include <signal.h>
4280009 #include <stdio.h>
4280010 #include <string.h>
4280011 #include <limits.h>
4280012 #include <libgen.h>
4280013 //-----
4280014 static void usage (void);
4280015 //-----
4280016 int
4280017 main (int argc, char *argv[], char *envp[])
4280018 {
4280019     char *source;
4280020     char *destination;
4280021     char *new_destination;
4280022     struct stat file_status;
4280023     int dest_is_a_dir = 0;
4280024     int a; // Argument index.
4280025     char path[PATH_MAX];
4280026     //
4280027     // There must be at least two arguments, plus the program name.
4280028     //
4280029     if (argc < 3)
4280030     {
4280031         usage ();
4280032         return (1);
4280033     }
4280034     //
4280035     // Select the last argument as the destination.
4280036     //
4280037     destination = argv[argc-1];
4280038     //
4280039     // Check if it is a directory and save it in a flag.
4280040     //
4280041     if (stat (destination, &file_status) == 0)
4280042     {
4280043         if (S_ISDIR (file_status.st_mode))
4280044         {
4280045             dest_is_a_dir = 1;
4280046         }
4280047     }
4280048     //
4280049     // If there are more than two arguments, verify that the last
4280050     // one is a directory.
4280051     //
4280052     if (argc > 3)
4280053     {
4280054         if (!dest_is_a_dir)
4280055         {
4280056             usage ();
4280057             fprintf (stderr, "The destination \"%s\" ",
4280058                     destination);
4280059             fprintf (stderr, "is not a directory!\n");
4280060             return (1);
4280061         }
4280062     }
4280063     //
4280064     // Scan the arguments, excluded the last, that is the destination.
4280065     //
4280066     for (a = 1; a < (argc - 1); a++)
4280067     {
4280068         //
4280069         // Source.
4280070         //
4280071         source = argv[a];
4280072         //
4280073         // Verify access permissions.
4280074         //
4280075         if (access (source, R_OK) < 0)
4280076         {
4280077             perror (source);
4280078             continue;
4280079         }
4280080         //
4280081         // Destination.
4280082         //
4280083         // If it is a directory, the destination path
4280084         // must be corrected.
4280085         //
4280086         if (dest_is_a_dir)
4280087         {
4280088             path[0] = 0;
4280089             strcat (path, destination);

```

1879


```

428090     strcat (path, "/");
428091     strcat (path, basename (source));
428092     //
428093     // Update the destination path.
428094     //
428095     new_destination = path;
428096     }
428097     else
428098     {
428099         new_destination = destination;
428100     }
428101     //
428102     // Check if destination file exists.
428103     //
428104     if (stat (new_destination, &file_status) == 0)
428105     {
428106         fprintf (stderr, "The destination file, \"%s\", ",
428107                 new_destination);
428108         fprintf (stderr, "already exists!\n");
428109         continue;
428110     }
428111     //
428112     // Everything is ready for the link.
428113     //
428114     if (link (source, new_destination) < 0)
428115     {
428116         perror (new_destination);
428117         continue;
428118     }
428119     }
428120     //
428121     // All done.
428122     //
428123     return (0);
428124 }
428125 //-----
428126 static void
428127 usage (void)
428128 {
428129     fprintf (stderr, "Usage: ln OLD_NAME NEW_NAME\n");
428130     fprintf (stderr, "        ln FILE... DIRECTORY\n");
428131 }

```

applic/login.c

« Si veda la sezione u0.12.

```

430001 #include <unistd.h>
430002 #include <stdlib.h>
430003 #include <sys/stat.h>
430004 #include <sys/types.h>
430005 #include <fcntl.h>
430006 #include <errno.h>
430007 #include <unistd.h>
430008 #include <signal.h>
430009 #include <stdio.h>
430010 #include <sys/wait.h>
430011 #include <stdio.h>
430012 #include <string.h>
430013 #include <limits.h>
430014 #include <stdint.h>
430015 #include <sys/unistd.h>
430016 //-----
430017 #define LOGIN_MAX      64
430018 #define PASSWORD_MAX  64
430019 #define HOME_MAX      64
430020 #define LINE_MAX      1024
430021 //-----
430022 int
430023 main (int argc, char *argv[], char *envp[])
430024 {
430025     char login[LOGIN_MAX];
430026     char password[PASSWORD_MAX];
430027     char buffer[LINE_MAX];
430028     char *user_name;
430029     char *user_password;
430030     char *user_uid;
430031     char *user_gid;
430032     char *user_description;
430033     char *user_home;
430034     char *user_shell;
430035     uid_t uid;
430036     uid_t euid;
430037     int fd;
430038     ssize_t size_read;
430039     int b; // Index inside buffer.
430040     int loop;
430041     char *exec_argv[2];
430042     int status;
430043     char *tty_path;
430044     //
430045     // Check if login is running correctly.
430046     //
430047     euid = geteuid ();
430048     uid = geteuid ();
430049     // //
430050     // // Show process info.
430051     // //
430052     // heap_clear ();
430053     // process_info ();
430054     //

```

1880

```

430055 // Check privileges.
430056 //
430057 if (!(uid == 0 && euid == 0))
430058 {
430059     printf ("%s: can only run with root privileges!\n", argv[0]);
430060     exit (-1);
430061 }
430062 //
430063 // Prepare arguments for the shell call.
430064 //
430065 exec_argv[0] = "-";
430066 exec_argv[1] = NULL;
430067 //
430068 // Login.
430069 //
430070 while (1)
430071 {
430072     fd = open ("/etc/passwd", O_RDONLY);
430073     //
430074     if (fd < 0)
430075     {
430076         perror ("Cannot open file '/etc/passwd'");
430077         exit (-1);
430078     }
430079     //
430080     printf ("Log in as \"root\" or \"user\" "
430081            "with password \"ciao\" :-)\n");
430082     input_line (login, "login:", LOGIN_MAX, INPUT_LINE_ECHO);
430083     //
430084     //
430085     //
430086     loop = 1;
430087     while (loop)
430088     {
430089         for (b = 0; b < LINE_MAX; b++)
430090         {
430091             size_read = read (fd, &buffer[b], (size_t) 1);
430092             if (size_read <= 0)
430093             {
430094                 buffer[b] = 0;
430095                 loop = 0; // Close the middle loop.
430096                 break;
430097             }
430098             if (buffer[b] == '\n')
430099             {
430100                 buffer[b] = 0;
430101                 break;
430102             }
430103         }
430104         //
430105         user_name = strtok (buffer, ":");
430106         user_password = strtok (NULL, ":");
430107         user_uid = strtok (NULL, ":");
430108         user_gid = strtok (NULL, ":");
430109         user_description = strtok (NULL, ":");
430110         user_home = strtok (NULL, ":");
430111         user_shell = strtok (NULL, ":");
430112         //
430113         if (strcmp (user_name, login) == 0)
430114         {
430115             input_line (password, "password:", PASSWORD_MAX,
430116                       INPUT_LINE_STARS);
430117             //
430118             // Compare passwords: empty passwords are not allowed.
430119             //
430120             if (strcmp (user_password, password) == 0)
430121             {
430122                 uid = atoi (user_uid);
430123                 euid = uid;
430124                 //
430125                 // Find the controlling terminal and change
430126                 // property and access permissions.
430127                 //
430128                 tty_path = ttyname (STDIN_FILENO);
430129                 if (tty_path != NULL)
430130                 {
430131                     status = chown (tty_path, uid, 0);
430132                     if (status != 0)
430133                     {
430134                         perror (NULL);
430135                     }
430136                     status = chmod (tty_path, 0600);
430137                     if (status != 0)
430138                     {
430139                         perror (NULL);
430140                     }
430141                 }
430142                 //
430143                 // Cd to the home directory, if present.
430144                 //
430145                 status = chdir (user_home);
430146                 if (status != 0)
430147                 {
430148                     perror (NULL);
430149                 }
430150                 //
430151                 // Now change personality.
430152                 //
430153                 setuid (uid);
430154                 seteuid (euid);
430155                 //

```

1881

```

4300156 // Run the shell, replacing the login process; the
4300157 // environment is taken from 'init'.
4300158 //
4300159 execve (user_shell, exec_argv, envp);
4300160 exit (0);
4300161 }
4300162 //
4300163 // Login failed: will try again.
4300164 //
4300165 loop = 0; // Close the middle loop.
4300166 break;
4300167 }
4300168 }
4300169 close (fd);
4300170 }
4300171 }

```

applic/ls.c

« Si veda la sezione u0.13.

```

4310001 #include <sys/osi6.h>
4310002 #include <sys/stat.h>
4310003 #include <sys/types.h>
4310004 #include <unistd.h>
4310005 #include <stdlib.h>
4310006 #include <fcntl.h>
4310007 #include <errno.h>
4310008 #include <signal.h>
4310009 #include <stdio.h>
4310010 #include <string.h>
4310011 #include <limits.h>
4310012 #include <libgen.h>
4310013 #include <dirent.h>
4310014 #include <pwd.h>
4310015 #include <time.h>
4310016
4310017 #define BUFFER_SIZE 16384
4310018 #define LIST_SIZE 256
4310019
4310020 //-----
4310021 static void usage (void);
4310022 //-----
4310023 //-----
4310024 int compare (void *p1, void *p2);
4310025 //-----
4310026 int
4310027 main (int argc, char *argv[], char *envp[])
4310028 {
4310029     int option_a = 0;
4310030     int option_l = 0;
4310031     int opt;
4310032 // extern char *optarg; // not used.
4310033 extern int optind;
4310034 extern int optopt;
4310035 struct stat file_status;
4310036 DIR *dp;
4310037 struct dirent *dir;
4310038 char buffer[BUFFER_SIZE];
4310039 int b; // Buffer index.
4310040 char *list[LIST_SIZE];
4310041 int l; // List index.
4310042 int len; // Name length.
4310043 char *path = NULL;
4310044 char pathname[PATH_MAX];
4310045 struct passwd *pws;
4310046 struct tm *tms;
4310047 //
4310048 // Check for options.
4310049 //
4310050 while ((opt = getopt (argc, argv, "al*")) != -1)
4310051 {
4310052     switch (opt)
4310053     {
4310054     case 'l':
4310055         option_l = 1;
4310056         break;
4310057     case 'a':
4310058         option_a = 1;
4310059         break;
4310060     case '?':
4310061         fprintf (stderr, "Unknown option -%c.\n", optopt);
4310062         usage ();
4310063         return (1);
4310064         break;
4310065     case ':':
4310066         fprintf (stderr, "Missing argument for option -%c\n",
4310067                 optopt);
4310068         usage ();
4310069         return (1);
4310070         break;
4310071     default:
4310072         fprintf (stderr, "Getopt problem: unknown option %c\n",
4310073                 opt);
4310074         return (1);
4310075     }
4310076 }
4310077 //
4310078 // If no arguments are present, at least the current directory is
4310079 // read.
4310080 //

```

1882

```

4310081 if (optind == argc)
4310082 {
4310083     //
4310084     // There are no more arguments. Replace the program name,
4310085     // corresponding to 'argv[0]', with the current directory
4310086     // path string.
4310087     //
4310088     argv[0] = ".";
4310089     argc = 1;
4310090     optind = 0;
4310091 }
4310092 //
4310093 // This is a very simplified 'ls': if there is only a name
4310094 // and it is a directory, the directory content is taken as
4310095 // the new 'argv[]' array.
4310096 //
4310097 if (optind == (argc - 1))
4310098 {
4310099     //
4310100     // There is a request for a single name. Test if it exists
4310101     // and if it is a directory.
4310102     //
4310103     if (stat(argv[optind], &file_status) != 0)
4310104     {
4310105         fprintf (stderr, "File \"%s\" does not exist!\n",
4310106                 argv[optind]);
4310107         return (2);
4310108     }
4310109     //
4310110     if (S_ISDIR (file_status.st_mode))
4310111     {
4310112         //
4310113         // Save the directory inside the 'path' pointer.
4310114         //
4310115         path = argv[optind];
4310116         //
4310117         // Open the directory.
4310118         //
4310119         dp = opendir (argv[optind]);
4310120         if (dp == NULL)
4310121         {
4310122             perror (argv[optind]);
4310123             return (3);
4310124         }
4310125         //
4310126         // Read the directory and fill the buffer with names.
4310127         //
4310128         b = 0;
4310129         l = 0;
4310130         while ((dir = readdir (dp)) != NULL)
4310131         {
4310132             len = strlen (dir->d_name);
4310133             //
4310134             // Check if the buffer can hold it.
4310135             //
4310136             if ((b + len + 1) > BUFFER_SIZE)
4310137             {
4310138                 fprintf (stderr, "not enough memory\n");
4310139                 break;
4310140             }
4310141             //
4310142             // Consider the directory item only if there is
4310143             // a valid name. If it is empty, just ignore it.
4310144             //
4310145             if (len > 0)
4310146             {
4310147                 strcpy (&buffer[b], dir->d_name);
4310148                 list[l] = &buffer[b];
4310149                 b += len + 1;
4310150                 l++;
4310151             }
4310152         }
4310153         //
4310154         // Close the directory.
4310155         //
4310156         closedir (dp);
4310157         //
4310158         // Sort the list.
4310159         //
4310160         qsort (list, (size_t) l, sizeof (char *), compare);
4310161         //
4310162         //
4310163         // Convert the directory list into a new 'argv[]' array,
4310164         // with a valid 'argc'. The variable 'optind' must be
4310165         // reset to the first element index, because there is
4310166         // no program name inside the new 'argv[]' at index zero.
4310167         //
4310168         argv = list;
4310169         argc = l;
4310170         optind = 0;
4310171     }
4310172     //
4310173     // Scan arguments, or list converted into 'argv[]'.
4310174     //
4310175     for (; optind < argc; optind++)
4310176     {
4310177         if (argv[optind][0] == '.')
4310178         {
4310179             //
4310180             // Current name starts with '.'.
4310181         }

```

1883

```

4310182 //
4310183 if (!option_a)
4310184 {
4310185 //
4310186 // Do not show name starting with `.`.
4310187 //
4310188 continue;
4310189 }
4310190 }
4310191 //
4310192 // Build the pathname.
4310193 //
4310194 if (path == NULL)
4310195 {
4310196 strcpy (&pathname[0], argv[optind]);
4310197 }
4310198 else
4310199 {
4310200 strcpy (pathname, path);
4310201 strcat (pathname, "/");
4310202 strcat (pathname, argv[optind]);
4310203 }
4310204 //
4310205 // Check if file exists, reading status.
4310206 //
4310207 if (stat(pathname, &file_status) != 0)
4310208 {
4310209 fprintf (stderr, "File \"%s\" does not exist!\n",
4310210         pathname);
4310211 return (2);
4310212 }
4310213 //
4310214 // Show file name.
4310215 //
4310216 if (option_l)
4310217 {
4310218 //
4310219 // Print the file type.
4310220 //
4310221 if (S_ISBLK (file_status.st_mode)) printf ("b");
4310222 else if (S_ISCHR (file_status.st_mode)) printf ("c");
4310223 else if (S_ISFIFO (file_status.st_mode)) printf ("p");
4310224 else if (S_ISREG (file_status.st_mode)) printf ("-");
4310225 else if (S_ISDIR (file_status.st_mode)) printf ("d");
4310226 else if (S_ISLNK (file_status.st_mode)) printf ("l");
4310227 else if (S_ISOCK (file_status.st_mode)) printf ("s");
4310228 else
4310229 printf ("?");
4310230 //
4310231 // Print permissions.
4310232 //
4310233 if (S_IRUSR & file_status.st_mode) printf ("r");
4310234 else printf ("-");
4310235 if (S_IWUSR & file_status.st_mode) printf ("w");
4310236 else printf ("-");
4310237 if (S_IXUSR & file_status.st_mode) printf ("x");
4310238 else printf ("-");
4310239 if (S_IRGRP & file_status.st_mode) printf ("r");
4310240 else printf ("-");
4310241 if (S_IWGRP & file_status.st_mode) printf ("w");
4310242 else printf ("-");
4310243 if (S_IXGRP & file_status.st_mode) printf ("x");
4310244 else printf ("-");
4310245 if (S_IROTH & file_status.st_mode) printf ("r");
4310246 else printf ("-");
4310247 if (S_IWOTH & file_status.st_mode) printf ("w");
4310248 else printf ("-");
4310249 if (S_IXOTH & file_status.st_mode) printf ("x");
4310250 else printf ("-");
4310251 //
4310252 // Print links.
4310253 //
4310254 printf (" %3i", (int) file_status.st_nlink);
4310255 //
4310256 // Print owner.
4310257 //
4310258 pws = getpwuid (file_status.st_uid);
4310259 printf (" %s", pws->pw_name);
4310260 //
4310261 // Print group (no group available);
4310262 //
4310263 printf (" (no group)");
4310264 //
4310265 // Print file size or device major-minor.
4310266 //
4310267 if (S_ISBLK (file_status.st_mode)
4310268 || S_ISCHR (file_status.st_mode))
4310269 {
4310270 printf (" %3i", (int) major (file_status.st_rdev));
4310271 printf (" %3i", (int) minor (file_status.st_rdev));
4310272 }
4310273 else
4310274 {
4310275 printf (" %8i", (int) file_status.st_size);
4310276 }
4310277 //
4310278 // Print modification date and time.
4310279 //
4310280 tms = localtime (&(file_status.st_mtime));
4310281 printf (" %4u-%02u-%02u %02u:%02u",
4310282         tms->tm_year, tms->tm_mon, tms->tm_mday,

```

1884

```

4310283         tms->tm_hour, tms->tm_min);
4310284 //
4310285 // Print file name, but with no additional path.
4310286 //
4310287 printf (" %s\n", argv[optind]);
4310288 }
4310289 else
4310290 {
4310291 //
4310292 // Just show the file name and go to the next line.
4310293 //
4310294 printf ("%s\n", argv[optind]);
4310295 }
4310296 }
4310297 //
4310298 // All done.
4310299 //
4310300 return (0);
4310301 }
4310302 //-----
4310303 static void
4310304 usage (void)
4310305 {
4310306 fprintf (stderr, "Usage: ls [OPTION] [FILE]...\n");
4310307 }
4310308 //-----
4310309 int
4310310 compare (void *p1, void *p2)
4310311 {
4310312 char **pp1 = p1;
4310313 char **pp2 = p2;
4310314 //
4310315 return (strcmp (*pp1, *pp2));
4310316 }
4310317

```

applic/man.c

Si veda la sezione u0.14.

«

```

4320001 #include <unistd.h>
4320002 #include <stdlib.h>
4320003 #include <errno.h>
4320004 //-----
4320005 #define MAX_LINES 20
4320006 #define MAX_COLUMNS 80
4320007 //-----
4320008 static char *man_page_directory = "/usr/share/man";
4320009 //-----
4320010 static void usage (void);
4320011 static FILE *open_man_page (int section, char *name);
4320012 static void build_path_name (int section, char *name, char *path);
4320013 //-----
4320014 int
4320015 main (int argc, char *argv[], char *envp[])
4320016 {
4320017 FILE *fp;
4320018 char *name;
4320019 int section;
4320020 int c;
4320021 int line = 1; // Line internal counter.
4320022 int column = 1; // Column internal counter.
4320023 int loop;
4320024 //
4320025 // There must be minimum an argument, and maximum two.
4320026 //
4320027 if (argc < 2 || argc > 3)
4320028 {
4320029 usage ();
4320030 return (1);
4320031 }
4320032 //
4320033 // If there are two arguments, there must be the
4320034 // section number.
4320035 //
4320036 if (argc == 3)
4320037 {
4320038 section = atoi (argv[1]);
4320039 name = argv[2];
4320040 }
4320041 else
4320042 {
4320043 section = 0;
4320044 name = argv[1];
4320045 }
4320046 //
4320047 // Try to open the manual page.
4320048 //
4320049 fp = open_man_page (section, name);
4320050 //
4320051 if (fp == NULL)
4320052 {
4320053 //
4320054 // Error opening file.
4320055 //
4320056 return (1);
4320057 }
4320058 //
4320059 //
4320060 // The following loop continues while the file
4320061 // gives characters, or when a command to change

```

1885

```

432062 // file or to quit is given.
432063 //
432064 for (loop = 1; loop; )
432065 {
432066 //
432067 // Read a single character.
432068 //
432069 c = getc (fp);
432070 //
432071 if (c == EOF)
432072 {
432073     loop = 0;
432074     break;
432075 }
432076 //
432077 // If the character read is a special one,
432078 // the line/column calculation is modified,
432079 // so that it is known when to stop scrolling.
432080 //
432081 switch (c)
432082 {
432083     case '\r':
432084         //
432085         // Displaying this character, the cursor should go
432086         // back to the first column. So the column counter
432087         // is reset.
432088         //
432089         column = 1;
432090         break;
432091     case '\n':
432092         //
432093         // Displaying this character, the cursor should go
432094         // back to the next line, at the first column.
432095         // So the column counter is reset and the line
432096         // counter is incremented.
432097         //
432098         line++;
432099         column = 1;
432100         break;
432101     case '\b':
432102         //
432103         // Displaying this character, the cursor should go
432104         // back one position, unless it is already at the
432105         // beginning.
432106         //
432107         if (column > 1)
432108             {
432109                 column--;
432110             }
432111         break;
432112     default:
432113         //
432114         // Any other character must increase the column
432115         // counter.
432116         //
432117         column++;
432118     }
432119 //
432120 // Display the character, even if it is a special one:
432121 // it is responsibility of the screen device management
432122 // to do something good with special characters.
432123 //
432124 putchar (c);
432125 //
432126 // If the column counter is gone beyond the screen columns,
432127 // then adjust the column counter and increment the line
432128 // counter.
432129 //
432130 if (column > MAX_COLUMNS)
432131 {
432132     column -= MAX_COLUMNS;
432133     line++;
432134 }
432135 //
432136 // Check if there is space for scrolling.
432137 //
432138 if (line < MAX_LINES)
432139 {
432140     continue;
432141 }
432142 //
432143 // Here, displayed lines are MAX_LINES.
432144 //
432145 if (column > 1)
432146 {
432147     //
432148     // Something was printed at the current line: must
432149     // do a new line.
432150     //
432151     putchar ('\n');
432152 }
432153 //
432154 // Show the more prompt.
432155 //
432156 printf ("--More--");
432157 fflush (stdout);
432158 //
432159 // Read a character from standard input.
432160 //
432161 c = getchar ();
432162 //

```

1886

```

432063 // Consider command 'q', but any other character
432064 // can be introduced, to let show the next page.
432065 //
432066 switch (c)
432067 {
432068     case 'Q':
432069     case 'q':
432070         //
432071         // Quit. But must erase the '--More--' prompt.
432072         //
432073         printf ("\b \b\b \b\b \b\b \b\b \b");
432074         printf ("\b \b\b \b\b \b\b \b");
432075         fclose (fp);
432076         return (0);
432077     }
432078 //
432079 // Backspace to overwrite '--More--' and the character
432080 // pressed.
432081 //
432082 printf ("\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b");
432083 //
432084 // Reset line/column counters.
432085 //
432086 column = 1;
432087 line = 1;
432088 }
432089 //
432090 // Close the file pointer if it is still open.
432091 //
432092 if (fp != NULL)
432093 {
432094     fclose (fp);
432095 }
432096 //
432097 return (0);
432098 }
432099 -----
432100 static void
432101 usage (void)
432102 {
432103     fprintf (stderr, "Usage: man [SECTION] NAME\n");
432104 }
432105 -----
432106 FILE *
432107 open_man_page (int section, char *name)
432108 {
432109     FILE *fp;
432110     char path[PATH_MAX];
432111     struct stat file_status;
432112     //
432113     //
432114     //
432115     if (section > 0)
432116     {
432117         build_path_name (section, name, path);
432118         //
432119         // Check if file exists.
432120         //
432121         if (stat (path, &file_status) != 0)
432122             {
432123                 fprintf (stderr, "Man page %s(%i) does not exist!\n",
432124                     name, section);
432125                 return (NULL);
432126             }
432127     }
432128     else
432129     {
432130         //
432131         // Must try a section.
432132         //
432133         for (section = 1; section < 9; section++)
432134             {
432135                 build_path_name (section, name, path);
432136                 //
432137                 // Check if file exists.
432138                 //
432139                 if (stat (path, &file_status) == 0)
432140                     {
432141                         //
432142                         // Found.
432143                         //
432144                         break;
432145                     }
432146             }
432147     }
432148     //
432149     // Check if a file was found.
432150     //
432151     if (section < 9)
432152     {
432153         fp = fopen (path, "r");
432154         //
432155         if (fp == NULL)
432156             {
432157                 //
432158                 // Error opening file.
432159                 //
432160                 perror (path);
432161                 return (NULL);
432162             }
432163     }
432164     else

```

1887

```

4320264     {
4320265         //
4320266         // Opened right.
4320267         //
4320268         return (fp);
4320269     }
4320270     }
4320271     else
4320272     {
4320273         fprintf (stderr, "Man page %s does not exist!\n",
4320274                 name);
4320275         return (NULL);
4320276     }
4320277 }
4320278 //-----
4320279 void
4320280 build_path_name (int section, char *name, char *path)
4320281 {
4320282     char string_section[10];
4320283     //
4320284     // Convert the section number into a string.
4320285     //
4320286     sprintf (string_section, "%i", section);
4320287     //
4320288     // Prepare the path to the man file.
4320289     //
4320290     path[0] = 0;
4320291     strcat (path, man_page_directory);
4320292     strcat (path, "/");
4320293     strcat (path, name);
4320294     strcat (path, ".");
4320295     strcat (path, string_section);
4320296 }

```

applic/mkdir.c

« Si veda la sezione u0.15.

```

4330001 #include <sys/osi16.h>
4330002 #include <sys/stat.h>
4330003 #include <sys/types.h>
4330004 #include <unistd.h>
4330005 #include <stdlib.h>
4330006 #include <fcntl.h>
4330007 #include <errno.h>
4330008 #include <signal.h>
4330009 #include <stdio.h>
4330010 #include <string.h>
4330011 #include <limits.h>
4330012 #include <libgen.h>
4330013 //-----
4330014 static int mkdir_parents (const char *path, mode_t mode);
4330015 static void usage (void);
4330016 //-----
4330017 int
4330018 main (int argc, char *argv[], char *envp[])
4330019 {
4330020     sysmsg_uarea_t msg;
4330021     int status;
4330022     mode_t mode = 0;
4330023     int m; // Index inside mode argument.
4330024     int digit;
4330025     char **dir;
4330026     int d; // Directory index.
4330027     int option_p = 0;
4330028     int option_m = 0;
4330029     int opt;
4330030     extern char *optarg;
4330031     extern int optind;
4330032     extern int optopt;
4330033     //
4330034     // There must be at least an argument, plus the program name.
4330035     //
4330036     if (argc < 2)
4330037     {
4330038         usage ();
4330039         return (1);
4330040     }
4330041     //
4330042     // Check for options, starting from 'p'. The 'dir' pointer is used
4330043     // to calculate the argument pointer to the first directory [1].
4330044     // The macro-instruction 'max()' is declared inside <sys/osi16.h>
4330045     // and does the expected thing.
4330046     //
4330047     while ((opt = getopt (argc, argv, "pm:")) != -1)
4330048     {
4330049         switch (opt)
4330050         {
4330051             case 'm':
4330052                 option_m = 1;
4330053                 for (m = 0; m < strlen (optarg); m++)
4330054                 {
4330055                     digit = (optarg[m] - '0');
4330056                     if (digit < 0 || digit > 7)
4330057                     {
4330058                         usage ();
4330059                         return (2);
4330060                     }
4330061                     mode = mode * 8 + digit;
4330062                 }
4330063                 break;

```

1888

```

4330064         case 'p':
4330065             option_p = 1;
4330066             break;
4330067         case '?':
4330068             printf ("Unknown option -%c.\n", optopt);
4330069             usage ();
4330070             return (1);
4330071             break;
4330072         case ':':
4330073             printf ("Missing argument for option -%c.\n", optopt);
4330074             usage ();
4330075             return (2);
4330076             break;
4330077         default:
4330078             printf ("Getopt problem: unknown option %c.\n", opt);
4330079             return (3);
4330080     }
4330081     }
4330082     //
4330083     dir = argv + optind;
4330084     //
4330085     // Check if the mode is to be set to a default value.
4330086     //
4330087     if (!option_m)
4330088     {
4330089         //
4330090         // Default mode.
4330091         //
4330092         sys (SYS_UAREA, &msg, (sizeof msg));
4330093         mode = 0777 && ~msg.umask;
4330094     }
4330095     //
4330096     // Directory creation.
4330097     //
4330098     for (d = 0; dir[d] != NULL; d++)
4330099     {
4330100         if (option_p)
4330101         {
4330102             status = mkdir_parents (dir[d], mode);
4330103             if (status != 0)
4330104             {
4330105                 perror (dir[d]);
4330106                 return (3);
4330107             }
4330108         }
4330109         else
4330110         {
4330111             status = mkdir (dir[d], mode);
4330112             if (status != 0)
4330113             {
4330114                 perror (dir[d]);
4330115                 return (4);
4330116             }
4330117         }
4330118     }
4330119     //
4330120     // All done.
4330121     //
4330122     return (0);
4330123 }
4330124 //-----
4330125 static int
4330126 mkdir_parents (const char *path, mode_t mode)
4330127 {
4330128     char path_copy[PATH_MAX];
4330129     char *path_parent;
4330130     struct stat fst;
4330131     int status;
4330132     //
4330133     // Check if the path is empty.
4330134     //
4330135     if (path == NULL || strlen (path) == 0)
4330136     {
4330137         //
4330138         // Recursion ends here.
4330139         //
4330140         return (0);
4330141     }
4330142     //
4330143     // Check if it does already exist.
4330144     //
4330145     status = stat (path, &fst);
4330146     if (status == 0 && fst.st_mode & S_IFDIR)
4330147     {
4330148         //
4330149         // The path exists and is a directory.
4330150         //
4330151         return (0);
4330152     }
4330153     else if (status == 0 && !(fst.st_mode & S_IFDIR))
4330154     {
4330155         //
4330156         // The path exists but is not a directory.
4330157         //
4330158         errno = ENOTDIR; // Not a directory.
4330159         return (-1);
4330160     }
4330161     //
4330162     // Get the directory path.
4330163     //
4330164     strncpy (path_copy, path, PATH_MAX);

```

1889

```

4330165 path_parent = dirname (path_copy);
4330166 //
4330167 // If it is '.', or '/', the recursion is terminated.
4330168 //
4330169 if (strncmp (path_parent, ".", PATH_MAX) == 0 ||
4330170     strncmp (path_parent, "/", PATH_MAX) == 0)
4330171     {
4330172         return (0);
4330173     }
4330174 //
4330175 // Otherwise, continue the recursion.
4330176 //
4330177 status = mkdir_parents (path_parent, mode);
4330178 if (status != 0)
4330179     {
4330180         return (-1);
4330181     }
4330182 //
4330183 // Previous directories are there: create the current one.
4330184 //
4330185 status = mkdir (path, mode);
4330186 if (status)
4330187     {
4330188         perror (path);
4330189         return (-1);
4330190     }
4330191
4330192 return (0);
4330193 }
4330194 //-----
4330195 static void
4330196 usage (void)
4330197 {
4330198     fprintf (stderr, "Usage: mkdir [-p] [-m OCTAL_MODE] DIR...\n");
4330199 }

```

applic/more.c

« Si veda la sezione u0.16.

```

4340001 #include <unistd.h>
4340002 #include <errno.h>
4340003 //-----
4340004 #define MAX_LINES 20
4340005 #define MAX_COLUMNS 80
4340006 //-----
4340007 static void usage (void);
4340008 //-----
4340009 int
4340010 main (int argc, char *argv[], char *envp[])
4340011 {
4340012     FILE *fp;
4340013     char *name;
4340014     int c;
4340015     int line = 1; // Line internal counter.
4340016     int column = 1; // Column internal counter.
4340017     int a; // Index inside arguments.
4340018     int loop;
4340019 //
4340020 // There must be at least an argument, plus the program name.
4340021 //
4340022 if (argc < 2)
4340023     {
4340024         usage ();
4340025         return (1);
4340026     }
4340027 //
4340028 // No options are allowed.
4340029 //
4340030 for (a = 1; a < argc; a++)
4340031     {
4340032         //
4340033         // Get next name from arguments.
4340034         //
4340035         name = argv[a];
4340036         //
4340037         // Try to open the file, read only.
4340038         //
4340039         fp = fopen (name, "r");
4340040         //
4340041         if (fp == NULL)
4340042             {
4340043                 //
4340044                 // Error opening file.
4340045                 //
4340046                 perror (name);
4340047                 return (1);
4340048             }
4340049         //
4340050         // Print the file name to be displayed.
4340051         //
4340052         printf ("== %s ==\n", name);
4340053         line++;
4340054         //
4340055         // The following loop continues while the file
4340056         // gives characters, or when a command to change
4340057         // file or to quit is given.
4340058         //
4340059         for (loop = 1; loop; )
4340060             {
4340061                 //

```

1890

```

4340062 // Read a single character.
4340063 //
4340064 c =getc (fp);
4340065 //
4340066 if (c == EOF)
4340067     {
4340068         loop = 0;
4340069         break;
4340070     }
4340071 //
4340072 // If the character read is a special one,
4340073 // the line/column calculation is modified,
4340074 // so that it is known when to stop scrolling.
4340075 //
4340076 switch (c)
4340077     {
4340078     case '\r':
4340079         //
4340080         // Displaying this character, the cursor should go
4340081         // back to the first column. So the column counter
4340082         // is reset.
4340083         //
4340084         column = 1;
4340085         break;
4340086     case '\n':
4340087         //
4340088         // Displaying this character, the cursor should go
4340089         // back to the next line, at the first column.
4340090         // So the column counter is reset and the line
4340091         // counter is incremented.
4340092         //
4340093         line++;
4340094         column = 1;
4340095         break;
4340096     case '\b':
4340097         //
4340098         // Displaying this character, the cursor should go
4340099         // back one position, unless it is already at the
4340100         // beginning.
4340101         //
4340102         if (column > 1)
4340103             {
4340104                 column--;
4340105             }
4340106         break;
4340107     default:
4340108         //
4340109         // Any other character must increase the column
4340110         // counter.
4340111         //
4340112         column++;
4340113     }
4340114 //
4340115 // Display the character, even if it is a special one:
4340116 // it is responsibility of the screen device management
4340117 // to do something good with special characters.
4340118 //
4340119 putchar (c);
4340120 //
4340121 // If the column counter is gone beyond the screen columns,
4340122 // then adjust the column counter and increment the line
4340123 // counter.
4340124 //
4340125 if (column > MAX_COLUMNS)
4340126     {
4340127         column = MAX_COLUMNS;
4340128         line++;
4340129     }
4340130 //
4340131 // Check if there is space for scrolling.
4340132 //
4340133 if (line < MAX_LINES)
4340134     {
4340135         continue;
4340136     }
4340137 //
4340138 // Here, displayed lines are MAX_LINES.
4340139 //
4340140 if (column > 1)
4340141     {
4340142         //
4340143         // Something was printed at the current line: must
4340144         // do a new line.
4340145         //
4340146         putchar ('\n');
4340147     }
4340148 //
4340149 // Show the more prompt.
4340150 //
4340151 printf ("--More--");
4340152 fflush (stdout);
4340153 //
4340154 // Read a character from standard input.
4340155 //
4340156 c = getchar ();
4340157 //
4340158 // Consider commands 'n' and 'q', but any other character
4340159 // can be introduced, to let show the next page.
4340160 //
4340161 switch (c)
4340162     {

```

1891

```

4340163         case 'N':
4340164         case 'n':
4340165             //
4340166             // Go to the next file, if any.
4340167             //
4340168             fclose (fp);
4340169             fp = NULL;
4340170             loop = 0;
4340171             break;
4340172         case 'Q':
4340173         case 'q':
4340174             //
4340175             // Quit. But must erase the "--More--" prompt.
4340176             //
4340177             printf ("\b \b\b \b\b \b\b \b\b \b");
4340178             printf ("\b \b\b \b\b \b\b \b\b \b");
4340179             fclose (fp);
4340180             return (0);
4340181         }
4340182         //
4340183         // Backspace to overwrite "--More--" and the character
4340184         // pressed.
4340185         //
4340186         printf ("\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b");
4340187         //
4340188         // Reset line/column counters.
4340189         //
4340190         column = 1;
4340191         line = 1;
4340192     }
4340193     //
4340194     // Close the file pointer if it is still open.
4340195     //
4340196     if (fp != NULL)
4340197     {
4340198         fclose (fp);
4340199     }
4340200 }
4340201 //
4340202 return (0);
4340203 }
4340204 //-----
4340205 static void
4340206 usage (void)
4340207 {
4340208     fprintf (stderr, "Usage: more FILE...\n");
4340209 }

```

applic/mount.c

Si veda la sezione u0.4.

```

4330001 #include <unistd.h>
4330002 #include <stdlib.h>
4330003 #include <sys/stat.h>
4330004 #include <sys/types.h>
4330005 #include <fcntl.h>
4330006 #include <errno.h>
4330007 #include <signal.h>
4330008 #include <stdio.h>
4330009 #include <sys/wait.h>
4330010 #include <stdio.h>
4330011 #include <string.h>
4330012 #include <limits.h>
4330013 #include <sys/osi6.h>
4330014 //-----
4330015 static void usage (void);
4330016 //-----
4330017 int
4330018 main (int argc, char *argv[], char *envp[])
4330019 {
4330020     int options;
4330021     int status;
4330022     //
4330023     //
4330024     //
4330025     if (argc < 3 || argc > 4)
4330026     {
4330027         usage ();
4330028         return (1);
4330029     }
4330030     //
4330031     // Set options.
4330032     //
4330033     if (argc == 4)
4330034     {
4330035         if (strcmp (argv[3], "rw") == 0)
4330036         {
4330037             options = MOUNT_DEFAULT;
4330038         }
4330039         else if (strcmp (argv[3], "ro") == 0)
4330040         {
4330041             options = MOUNT_RO;
4330042         }
4330043         else
4330044         {
4330045             printf ("Invalid mount option: only \"ro\" or \"rw\" "
4330046                 "are allowed\n");
4330047             return (2);
4330048         }
4330049     }

```

1892

```

4330050     else
4330051     {
4330052         options = MOUNT_DEFAULT;
4330053     }
4330054     //
4330055     // System call.
4330056     //
4330057     status = mount (argv[1], argv[2], options);
4330058     if (status != 0)
4330059     {
4330060         perror (NULL);
4330061         return (2);
4330062     }
4330063     //
4330064     return (0);
4330065 }
4330066 //-----
4330067 static void
4330068 usage (void)
4330069 {
4330070     fprintf (stderr, "Usage: mount DEVICE MOUNT_POINT "
4330071         " [MOUNT_OPTIONS]\n");
4330072 }

```

applic/ps.c

Si veda la sezione u0.17.

```

4360001 #include <kernel/proc.h>
4360002 #include <unistd.h>
4360003 #include <stdio.h>
4360004 #include <fcntl.h>
4360005 #include <unistd.h>
4360006 #include <stdlib.h>
4360007 //-----
4360008 void
4360009 print_proc_head (void)
4360010 {
4360011     printf (
4360012 "pp p ps
4360013 "id id rp tty uid euid suid usage s iaddr isiz daddr dsiz sp name\n"
4360014 );
4360015 }
4360016 //-----
4360017 void
4360018 print_proc_pid (proc_t *ps, pid_t pid)
4360019 {
4360020     char stat;
4360021     switch (ps->status)
4360022     {
4360023         case PROC_EMPTY : stat = '-'; break;
4360024         case PROC_CREATED : stat = 'c'; break;
4360025         case PROC_READY : stat = 'r'; break;
4360026         case PROC_RUNNING : stat = 'R'; break;
4360027         case PROC_SLEEPING : stat = 's'; break;
4360028         case PROC_ZOMBIE : stat = 'z'; break;
4360029         default : stat = '?'; break;
4360030     }
4360031 }
4360032 printf ("%2i %2i %2i %04x %4i %4i %4i %02i.%02i %c %05lx %04x ",
4360033 (unsigned int) ps->ppid,
4360034 (unsigned int) pid,
4360035 (unsigned int) ps->pgrp,
4360036 (unsigned int) ps->xdevice_tty,
4360037 (unsigned int) ps->uid,
4360038 (unsigned int) ps->euid,
4360039 (unsigned int) ps->suid,
4360040 (unsigned int) ((ps->usage / CLOCKS_PER_SEC) / 60),
4360041 (unsigned int) ((ps->usage / CLOCKS_PER_SEC) % 60),
4360042 stat,
4360043 (unsigned long int) ps->address_i,
4360044 (unsigned int) ps->size_i);
4360045 }
4360046 printf ("%05lx %04x %04x %s",
4360047 (unsigned long int) ps->address_d,
4360048 (unsigned int) ps->size_d,
4360049 (unsigned int) ps->sp,
4360050 ps->xname);
4360051 }
4360052 printf ("\n");
4360053 }
4360054 //-----
4360055 int
4360056 main (void)
4360057 {
4360058     pid_t pid;
4360059     proc_t *ps;
4360060     int fd;
4360061     ssize_t size_read;
4360062     char buffer[sizeof (proc_t)];
4360063 }
4360064 fd = open ("/dev/kmem_ps", O_RDONLY);
4360065 if (fd < 0)
4360066 {
4360067     perror ("ps: cannot open \"/dev/kmem_ps\"");
4360068     exit (0);
4360069 }
4360070 print_proc_head ();
4360071 for (pid = 0; pid < PROCESS_MAX; pid++)
4360072 {

```

1893

```

436074         lseek (fd, (off_t) pid, SEEK_SET);
436075         size_read = read (fd, buffer, sizeof (proc_t));
436076         if (size_read < sizeof (proc_t))
436077         {
436078             printf ("ps: cannot read \"/dev/kmem_ps\" pid %i", pid);
436079             perror (NULL);
436080             continue;
436081         }
436082         ps = (proc_t *) buffer;
436083         if (ps->status > 0)
436084         {
436085             ps->name[PATH_MAX-1] = 0; // Terminated string.
436086             print_proc_pid (ps, pid);
436087         }
436088     }
436089 }
436090
436091     close (fd);
436092     return (0);
}

```

applic/rm.c

<

Si veda la sezione u0.18.

```

437001 #include <fcntl.h>
437002 #include <sys/stat.h>
437003 #include <stddef.h>
437004 #include <unistd.h>
437005 #include <errno.h>
437006 //-----
437007 static void usage (void);
437008 //-----
437009 int
437010 main (int argc, char *argv[], char *envp[])
437011 {
437012     int a; // Argument index.
437013     int status;
437014     struct stat file_status;
437015     //
437016     // No options are known, but at least an argument must be given.
437017     //
437018     if (argc < 2)
437019     {
437020         usage ();
437021         return (1);
437022     }
437023     //
437024     // Scan arguments.
437025     //
437026     for(a = 1; a < argc; a++)
437027     {
437028         //
437029         // Verify if the file exists.
437030         //
437031         if (stat(argv[a], &file_status) != 0)
437032         {
437033             fprintf (stderr, "File \"%s\" does not exist!\n",
437034                     argv[a]);
437035             continue;
437036         }
437037         //
437038         // File exists: check the file type.
437039         //
437040         if (S_ISDIR (file_status.st_mode))
437041         {
437042             fprintf (stderr, "Cannot remove directory \"%s\"!\n",
437043                     argv[a]);
437044             continue;
437045         }
437046         //
437047         // Can remove it.
437048         //
437049         status = unlink (argv[a]);
437050         if (status != 0)
437051         {
437052             perror (NULL);
437053             return (2);
437054         }
437055     }
437056     return (0);
}
437057 //-----
437058 static void
437059 usage (void)
437060 {
437061     fprintf (stderr, "Usage: rm FILE...\n");
437062 }
437063 }

```

applic/shell.c

<

Si veda la sezione u0.19.

```

438001 #include <unistd.h>
438002 #include <stdlib.h>
438003 #include <sys/stat.h>
438004 #include <sys/types.h>
438005 #include <fcntl.h>
438006 #include <errno.h>
438007 #include <unistd.h>
438008 #include <signal.h>

```

```

438009 #include <stdio.h>
438010 #include <sys/wait.h>
438011 #include <stdio.h>
438012 #include <string.h>
438013 #include <limits.h>
438014 #include <sys/posix.h>
438015 //-----
438016 #define PROMPT_SIZE 16
438017 //-----
438018 static void sh_cd (int argc, char *argv[]);
438019 static void sh_pwd (int argc, char *argv[]);
438020 static void sh_umask (int argc, char *argv[]);
438021 //-----
438022 int
438023 main (int argc, char *argv[], char *envp[])
438024 {
438025     char buffer_cmd[ARG_MAX/2];
438026     char *argv_cmd[ARG_MAX/16];
438027     char prompt[PROMPT_SIZE];
438028     uid_t uid;
438029     int argc_cmd;
438030     pid_t pid_cmd;
438031     pid_t pid_dead;
438032     int status;
438033     //
438034     //
438035     //
438036     uid = getuid ();
438037     //
438038     // Load processes, reading the keyboard.
438039     //
438040     while (1)
438041     {
438042         if (uid == 0)
438043         {
438044             strncpy (prompt, "# ", PROMPT_SIZE);
438045         }
438046         else
438047         {
438048             strncpy (prompt, "$ ", PROMPT_SIZE);
438049         }
438050         //
438051         input_line (buffer_cmd, prompt, (ARG_MAX/2), INPUT_LINE_ECHO);
438052         //
438053         // Clear 'argv_cmd[]';
438054         //
438055         for (argc_cmd = 0; argc_cmd < (ARG_MAX/16); argc_cmd++)
438056         {
438057             argv_cmd[argc_cmd] = NULL;
438058         }
438059         //
438060         // Initialize the command scan.
438061         //
438062         argv_cmd[0] = strtok (buffer_cmd, " \t");
438063         //
438064         // Verify: if the input is not valid, loop again.
438065         //
438066         if (argv_cmd[0] == NULL)
438067         {
438068             continue;
438069         }
438070         //
438071         // Find the arguments.
438072         //
438073         for (argc_cmd = 1;
438074              argc_cmd < ((ARG_MAX/16)-1) && argv_cmd[argc_cmd-1] != NULL;
438075              argc_cmd++)
438076         {
438077             argv_cmd[argc_cmd] = strtok (NULL, " \t");
438078         }
438079         //
438080         // If there are too many arguments, show a message and continue.
438081         //
438082         if (argc_cmd[argc_cmd-1] != NULL)
438083         {
438084             errset (E2BIG); // Argument list too long.
438085             perror (NULL);
438086             continue;
438087         }
438088         //
438089         // Correct the value for 'argc_cmd', because actually
438090         // it counts also the NULL element.
438091         //
438092         argc_cmd--;
438093         //
438094         // Verify if it is an internal command.
438095         //
438096         if (strcmp (argv_cmd[0], "exit") == 0)
438097         {
438098             return (0);
438099         }
438100         else if (strcmp (argv_cmd[0], "cd") == 0)
438101         {
438102             sh_cd (argc_cmd, argv_cmd);
438103             continue;
438104         }
438105         else if (strcmp (argv_cmd[0], "pwd") == 0)
438106         {
438107             sh_pwd (argc_cmd, argv_cmd);
438108             continue;
438109         }

```



```

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));

```

1896

```

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.

```

490001 #include <fcntl.h>
490002 #include <sys/stat.h>
490003 #include <time.h>
490004 #include <stddef.h>
490005 #include <unistd.h>
490006 #include <errno.h>
490007 //-----
490008 static void usage (void);
490009 //-----
490010 int
490011 main (int argc, char *argv[], char *envp[])
490012 {
490013     int a;           // Argument index.
490014     int status;
490015     struct stat file_status;
490016     //
490017     // No options are known, but at least an argument must be given.
490018     //
490019     if (argc < 2)
490020     {
490021         usage ();
490022         return (1);
490023     }
490024     //
490025     // Scan arguments.
490026     //
490027     for(a = 1; a < argc; a++)
490028     {
490029         //
490030         // Verify if the file exists, through the return value of
490031         // 'stat()'. No other checks are made.
490032         //
490033         if (stat(argv[a], &file_status) == 0)
490034         {
490035             //
490036             // File exists: should be updated the times.
490037             //
490038             status = utime (argv[a], NULL);
490039             if (status != 0)
490040             {
490041                 perror (NULL);
490042                 return (2);
490043             }
490044         }
490045         else
490046         {
490047             //
490048             // File does not exist: should be created.
490049             //
490050             status = open (argv[a], O_WRONLY|O_CREAT|O_TRUNC, 0666);
490051             //
490052             if (status >= 0)
490053             {
490054                 //
490055                 // Here, the variable 'status' is the file
490056                 // descriptor to be closed.
490057                 //
490058                 status = close (status);
490059                 if (status != 0)
490060                 {
490061                     perror (NULL);
490062                     return (3);
490063                 }
490064             }
490065             else
490066             {
490067                 perror (NULL);
490068                 return (4);
490069             }
490070         }
490071     }
490072     return (0);
490073 }

```

1897

```

4390074 //-----
4390075 static void
4390076 usage (void)
4390077 {
4390078     fprintf (stderr, "Usage: touch FILE...\n");
4390079 }

```

applic/tty.c

<

Si veda la sezione u0.21.

```

4400001 #include <fcntl.h>
4400002 #include <sys/stat.h>
4400003 #include <utime.h>
4400004 #include <stddef.h>
4400005 #include <unistd.h>
4400006 #include <errno.h>
4400007 #include <sys/osl6.h>
4400008 #include <sys/types.h>
4400009 //-----
4400010 static void usage (void);
4400011 //-----
4400012 int
4400013 main (int argc, char *argv[], char *envp[])
4400014 {
4400015     int dev_minor;
4400016     struct stat file_status;
4400017     //
4400018     // No options and no arguments.
4400019     //
4400020     if (argc > 1)
4400021     {
4400022         usage ();
4400023         return (1);
4400024     }
4400025     //
4400026     // Verify the standard input.
4400027     //
4400028     if (fstat (STDIN_FILENO, &file_status) == 0)
4400029     {
4400030         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
4400031         {
4400032             dev_minor = minor (file_status.st_rdev);
4400033             //
4400034             // If minor is equal to 0xFF, it is '/dev/console'
4400035             // that is not a controlling terminal, but just
4400036             // a reference for the current virtual console.
4400037             //
4400038             if (dev_minor < 0xFF)
4400039             {
4400040                 printf ("/dev/console%i\n", dev_minor);
4400041             }
4400042         }
4400043     }
4400044     else
4400045     {
4400046         perror ("Cannot get standard input file status");
4400047         return (2);
4400048     }
4400049     //
4400050     return (0);
4400051 }
4400052 //-----
4400053 static void
4400054 usage (void)
4400055 {
4400056     fprintf (stderr, "Usage: tty\n");
4400057 }
4400058 //-----

```

```

4410027     return (1);
4410028 }
4410029 //
4410030 // System call.
4410031 //
4410032 status = umount (argv[1]);
4410033 if (status != 0)
4410034 {
4410035     perror (argv[1]);
4410036     return (2);
4410037 }
4410038 //
4410039 return (0);
4410040 }
4410041 //-----
4410042 static void
4410043 usage (void)
4410044 {
4410045     fprintf (stderr, "Usage: umount MOUNT_POINT\n");
4410046 }

```

applic/umount.c

<

Si veda la sezione u0.4.

```

4410001 #include <unistd.h>
4410002 #include <stdlib.h>
4410003 #include <sys/stat.h>
4410004 #include <sys/types.h>
4410005 #include <fcntl.h>
4410006 #include <errno.h>
4410007 #include <signal.h>
4410008 #include <stdio.h>
4410009 #include <sys/wait.h>
4410010 #include <stdio.h>
4410011 #include <string.h>
4410012 #include <limits.h>
4410013 #include <sys/osl6.h>
4410014 //-----
4410015 static void usage (void);
4410016 //-----
4410017 int
4410018 main (int argc, char *argv[], char *envp[])
4410019 {
4410020     int status;
4410021     //
4410022     // One argument is mandatory.
4410023     //
4410024     if (argc != 2)
4410025     {
4410026         usage ();

```

