os16: directory «applic/»

«

applic/MAKEDEV.c

«

Si veda la sezione u0.3.

```
4150001 #include <unistd.h>
4150002 #include <stdlib.h>
4150003 #include <sys/stat.h>
4150004 #include <fcntl.h>
4150005 #include <kernel/devices.h>
4150006 #include <stdio.h>
4150007 //------------------------------------------------------------------------
4150008 int
4150009 main (void)
```

```
4150010 {
4150011     int status;
4150012     status = mknod ("mem",         (mode_t) (S_IFCHR | 0444),
4150013                     (dev_t) DEV_MEM);
4150014     if (status) perror (NULL);
4150015     status = mknod ("null",        (mode_t) (S_IFCHR | 0666),
4150016                     (dev_t) DEV_NULL);
4150017     if (status) perror (NULL);
4150018     status = mknod ("port",        (mode_t) (S_IFCHR | 0644),
4150019                     (dev_t) DEV_PORT);
4150020     if (status) perror (NULL);
4150021     status = mknod ("zero",        (mode_t) (S_IFCHR | 0666),
4150022                     (dev_t) DEV_ZERO);
4150023     if (status) perror (NULL);
4150024     status = mknod ("tty",         (mode_t) (S_IFCHR | 0666),
4150025                     (dev_t) DEV_TTY);
4150026     if (status) perror (NULL);
4150027     status = mknod ("dsk0",        (mode_t) (S_IFBLK | 0644),
4150028                     (dev_t) DEV_DSK0);
4150029     if (status) perror (NULL);
4150030     status = mknod ("dsk1",        (mode_t) (S_IFBLK | 0644),
4150031                     (dev_t) DEV_DSK1);
4150032     if (status) perror (NULL);
4150033     status = mknod ("dsk2",        (mode_t) (S_IFBLK | 0644),
4150034                     (dev_t) DEV_DSK2);
4150035     if (status) perror (NULL);
4150036     status = mknod ("dsk3",        (mode_t) (S_IFBLK | 0644),
4150037                     (dev_t) DEV_DSK3);
4150038     if (status) perror (NULL);
4150039     status = mknod ("kmem_ps",     (mode_t) (S_IFCHR | 0444),
4150040                     (dev_t) DEV_KMEM_PS);
4150041     if (status) perror (NULL);
4150042     status = mknod ("kmem_mmp",    (mode_t) (S_IFCHR | 0444),
4150043                     (dev_t) DEV_KMEM_MMP);
4150044     if (status) perror (NULL);
4150045     status = mknod ("kmem_sb",     (mode_t) (S_IFCHR | 0444),
4150046                     (dev_t) DEV_KMEM_SB);
4150047     if (status) perror (NULL);
4150048     status = mknod ("kmem_inode", (mode_t) (S_IFCHR | 0444),
4150049                     (dev_t) DEV_KMEM_INODE);
4150050     if (status) perror (NULL);
4150051     status = mknod ("kmem_file",  (mode_t) (S_IFCHR | 0444),
4150052                     (dev_t) DEV_KMEM_FILE);
```

```
4150053     if (status) perror (NULL);
4150054     status = mknod ("console",    (mode_t) (S_IFCHR | 0644),
4150055                     (dev_t) DEV_CONSOLE);
4150056     if (status) perror (NULL);
4150057     status = mknod ("console0",   (mode_t) (S_IFCHR | 0644),
4150058                     (dev_t) DEV_CONSOLE0);
4150059     if (status) perror (NULL);
4150060     status = mknod ("console1",   (mode_t) (S_IFCHR | 0644),
4150061                     (dev_t) DEV_CONSOLE1);
4150062     if (status) perror (NULL);
4150063     status = mknod ("console2",   (mode_t) (S_IFCHR | 0644),
4150064                     (dev_t) DEV_CONSOLE2);
4150065     if (status) perror (NULL);
4150066     status = mknod ("console3",   (mode_t) (S_IFCHR | 0644),
4150067                     (dev_t) DEV_CONSOLE3);
4150068     if (status) perror (NULL);
4150069
4150070     return (0);
4150071 }
```

applic/aaa.c

«

Si veda la sezione u0.1.

```
4160001 #include <unistd.h>
4160002 #include <stdio.h>
4160003 //--------------------------------------------------------------------
4160004 int
4160005 main (void)
4160006 {
4160007     unsigned int count;
4160008     for (count = 0; count < 60; count++)
4160009       {
4160010         printf ("a");
4160011         sleep (1);
4160012       }
4160013     return (8);
4160014 }
```

applic/bbb.c

«

Si veda la sezione u0.1.

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

applic/cat.c

«

Si veda la sezione u0.3.

```
4180001  #include <fcntl.h>
4180002  #include <sys/stat.h>
4180003  #include <stddef.h>
4180004  #include <unistd.h>
4180005  #include <stdio.h>
4180006  #include <stdlib.h>
4180007  #include <errno.h>
4180008  //------------------------------------------------------------------
4180009  static void cat_file_descriptor (int fd);
4180010  //------------------------------------------------------------------
4180011  int
4180012  main (int argc, char *argv[], char *envp[])
4180013  {
4180014      int          i;
4180015      int          fd;
4180016      struct stat file_status;
4180017      //
4180018      // Check if the input comes from standard input.
```

```
4180019       //
4180020       if (argc < 2)
4180021         {
4180022           cat_file_descriptor (STDIN_FILENO);
4180023           exit (0);
4180024         }
4180025       //
4180026       // There is at least an argument: scan them.
4180027       //
4180028       for(i = 1; i < argc; i++)
4180029         {
4180030           //
4180031           // Verify if the file exists.
4180032           //
4180033           if (stat(argv[i], &file_status) != 0)
4180034             {
4180035               fprintf (stderr, "File \"%s\" does not exist!\n",
4180036                        argv[i]);
4180037               continue;
4180038             }
4180039           //
4180040           // File exists: check the file type.
4180041           //
4180042           if (S_ISDIR (file_status.st_mode))
4180043             {
4180044               fprintf (stderr, "Cannot \"cat\" "
4180045                                "\"%s\": it is a directory!\n",
4180046                        argv[i]);
4180047               continue;
4180048             }
4180049           //
4180050           // File exists and can be "cat"ed.
4180051           //
4180052           fd = open (argv[i], O_RDONLY);
4180053           if (fd >= 0)
4180054             {
4180055               cat_file_descriptor (fd);
4180056               close (fd);
4180057             }
4180058           else
4180059             {
4180060               perror (NULL);
4180061               exit (1);
```

```
4180062             }
4180063           }
4180064       return (0);
4180065   }
4180066   //---------------------------------------------------------------------
4180067   static void
4180068   cat_file_descriptor (int fd)
4180069   {
4180070       ssize_t count;
4180071       char buffer[BUFSIZ];
4180072
4180073       for (;;)
4180074         {
4180075           count = read (fd, buffer, (size_t) BUFSIZ);
4180076           if (count > 0)
4180077             {
4180078               write (STDOUT_FILENO, buffer, (size_t) count);
4180079             }
4180080           else
4180081             {
4180082               break;
4180083             }
4180084         }
4180085   }
4180086
```

applic/ccc.c

«

Si veda la sezione u0.1.

```
4190001   #include <unistd.h>
4190002   #include <stdlib.h>
4190003   #include <signal.h>
4190004   //---------------------------------------------------------------------
4190005   int
4190006   main (void)
4190007   {
4190008       pid_t        pid;
4190009       //-----------------------------------------------------------------
4190010       pid = fork ();
4190011       if (pid == 0)
4190012         {
```

```
4190013           setuid ((uid_t) 10);
4190014           execve ("/bin/aaa", NULL, NULL);
4190015           exit (0);
4190016         }
4190017       //-------------------------------------------------------------------
4190018       pid = fork ();
4190019       if (pid == 0)
4190020         {
4190021           setuid ((uid_t) 11);
4190022           execve ("/bin/bbb", NULL, NULL);
4190023           exit (0);
4190024         }
4190025       //-------------------------------------------------------------------
4190026       while (1)
4190027         {
4190028           ; // Just loop, to consume CPU time: it must be killed manually.
4190029         }
4190030       return (0);
4190031   }
```

applic/chmod.c

«

Si veda la sezione u0.5.

```
4200001   #include <unistd.h>
4200002   #include <stdlib.h>
4200003   #include <sys/stat.h>
4200004   #include <sys/types.h>
4200005   #include <fcntl.h>
4200006   #include <errno.h>
4200007   #include <signal.h>
4200008   #include <stdio.h>
4200009   #include <sys/wait.h>
4200010   #include <stdio.h>
4200011   #include <string.h>
4200012   #include <limits.h>
4200013   #include <sys/os16.h>
4200014   //-------------------------------------------------------------------
4200015   static void usage (void);
4200016   //-------------------------------------------------------------------
4200017   int
4200018   main (int argc, char *argv[], char *envp[])
```

```
4200019 {
4200020     int     status;
4200021     mode_t  mode;
4200022     char   *m;          // Pointer inside the octal mode string.
4200023     int     digit;
4200024     int     a;          // Argument index.
4200025     //
4200026     //
4200027     //
4200028     if (argc < 3)
4200029       {
4200030         usage ();
4200031         return (1);
4200032       }
4200033     //
4200034     // Get mode: must be the first argument.
4200035     //
4200036     for (m = argv[1]; *m != 0; m++)
4200037       {
4200038         digit = (*m - '0');
4200039         if (digit < 0 || digit > 7)
4200040           {
4200041             usage ();
4200042             return (2);
4200043           }
4200044         mode = mode * 8 + digit;
4200045       }
4200046     //
4200047     // System call for all the remaining arguments.
4200048     //
4200049     for (a = 2; a < argc; a++)
4200050       {
4200051         status = chmod (argv[a], mode);
4200052         if (status != 0)
4200053           {
4200054             perror (argv[a]);
4200055             return (3);
4200056           }
4200057       }
4200058     //
4200059     // All done.
4200060     //
4200061     return (0);
```

```
4200062 }
4200063 //---------------------------------------------------------------------
4200064 static void
4200065 usage (void)
4200066 {
4200067     fprintf (stderr, "Usage:   chmod OCTAL_MODE FILE...\n");
4200068     fprintf (stderr, "Example: chmod 0640 my_file\n");
4200069 }
```

applic/chown.c

Si veda la sezione u0.6.

```
4210001 #include <unistd.h>
4210002 #include <stdlib.h>
4210003 #include <sys/stat.h>
4210004 #include <sys/types.h>
4210005 #include <fcntl.h>
4210006 #include <errno.h>
4210007 #include <stdio.h>
4210008 #include <ctype.h>
4210009 #include <pwd.h>
4210010 //---------------------------------------------------------------------
4210011 static void usage (void);
4210012 //---------------------------------------------------------------------
4210013 int
4210014 main (int argc, char *argv[], char *envp[])
4210015 {
4210016     char            *user;
4210017     int              uid;
4210018     struct passwd   *pws;
4210019     struct stat      file_status;
4210020     int              a;          // Argument index.
4210021     int              status;
4210022     //
4210023     //
4210024     //
4210025     if (argc < 3)
4210026       {
4210027         usage ();
4210028         return (1);
4210029       }
```

```
4210030       //
4210031       // Get user id number.
4210032       //
4210033       user = argv[1];
4210034       if (isdigit (*user))
4210035         {
4210036           uid = atoi (user);
4210037         }
4210038       else
4210039         {
4210040           pws = getpwnam (user);
4210041           if (pws == NULL)
4210042             {
4210043               fprintf(stderr, "Unknown user \"%s\"!\n", user);
4210044               return(2);
4210045             }
4210046           uid = pws->pw_uid;
4210047         }
4210048       //
4210049       // Now we have the user id. Start scanning file names.
4210050       //
4210051       for (a = 2; a < argc; a++)
4210052         {
4210053           //
4210054           // Verify if the file exists, through the return value of
4210055           // `stat()'. No other checks are made.
4210056           //
4210057           if (stat(argv[a], &file_status) == 0)
4210058            {
4210059               //
4210060               // Try to change ownership.
4210061               //
4210062               status = chown (argv[a], uid, file_status.st_gid);
4210063               if (status != 0)
4210064                 {
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/os16.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       {
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);
```

```
                      if (count_out < 0)
                        {
                          perror (destination);
                          close (fd_source);
                          close (fd_destination);
                          return (3);
                        }
                      //
                      // If not all data is written, continue writing,
                      // but change the buffer start position and the
                      // amount to be written.
                      //
                      buffer_out += count_out;
                      count_in -= count_out;
                    }
                }
              else if (count_in < 0)
                {
                  perror (source);
                  close (fd_source);
                  close (fd_destination);
                }
              else
                {
                  break;
                }
            }
          //
          if (close (fd_source))
            {
              perror (source);
            }
          if (close (fd_destination))
            {
              perror (destination);
              return (4);
            }
        }
      //
      // All done.
      //
      return (0);
  }
```

```
4220195 //---------------------------------------------------------------------
4220196 static void
4220197 usage (void)
4220198 {
4220199     fprintf (stderr, "Usage: cp OLD_NAME NEW_NAME\n");
4220200     fprintf (stderr, "       cp FILE... DIRECTORY\n");
4220201 }
```

applic/crt0.s

«

Si veda la sezione u0.2.

```
4230001 .extern _main
4230002 .extern __stdio_stream_setup
4230003 .extern __dirent_directory_stream_setup
4230004 .extern __atexit_setup
4230005 .extern __environment_setup
4230006 .global ___mkargv
4230007 ;---------------------------------------------------------------------
4230008 ; Please note that, all segments are already set from the scheduler,
4230009 ; and there is also data inside the stack, so that the call to `main()'
4230010 ; function will result as expected.
4230011 ;
4230012 ; This is a modified version of `crt0.s' with a smaller stack size.
4230013 ;---------------------------------------------------------------------
4230014 ; The following statement says that the code will start at "startup"
4230015 ; label.
4230016 ;---------------------------------------------------------------------
4230017 entry startup
4230018 ;---------------------------------------------------------------------
4230019 .text
4230020 ;---------------------------------------------------------------------
4230021 startup:
4230022     ;
4230023     ; Jump after initial data.
4230024     ;
4230025     jmp startup_code
4230026     ;
4230027 filler:
4230028     ;
4230029     ; After four bytes, from the start, there is the
4230030     ; magic number and other data.
```

```
4230031     ;
4230032     .space (0x0004 - (filler - startup))
4230033     ;
4230034 magic:
4230035     .data4 0x6F733136   ; os16
4230036     .data4 0x6170706C   ; appl
4230037     ;
4230038 segoff:                 ;
4230039     .data2 __segoff     ; Data segment offset.
4230040 etext:                  ;
4230041     .data2 __etext      ; End of code
4230042 edata:                  ;
4230043     .data2 __edata      ; End of initialized data.
4230044 ebss:                   ;
4230045     .data2 __end        ; End of not initialized data.
4230046 stack_size:             ;
4230047     .data2 0x2000       ; Requested stack size. Every single application
4230048                         ; might change this value.
4230049     ;
4230050     ; At the next label, the work begins.
4230051     ;
4230052 .align 2
4230053 startup_code:
4230054     ;
4230055     ; Before the call to the main function, it is necessary to extract
4230056     ; the value to assign to the global variable 'environ'. It is
4230057     ; described as 'char **environ' and should contain the same address
4230058     ; pointed by 'envp'. To get this value, the stack is popped and then
4230059     ; pushed again. Please recall that the stack was prepared from
4230060     ; the process management, at the 'exec()' system call.
4230061     ;
4230062     pop ax              ; argc
4230063     pop bx              ; argv
4230064     pop cx              ; envp
4230065     mov _environ, cx    ; Variable 'environ' comes from <unistd.h>.
4230066     push cx
4230067     push bx
4230068     push ax
4230069     ;
4230070     ; Could it be enough? Of course not! To be able to handle the
4230071     ; environment, it must be copied inside the table
4230072     ; '_environment_table[][]', that is defined inside <stdlib.h>.
4230073     ; To copy the environment it is used the function
```

```
4230074     ; '_environment_setup()', passing the 'envp' pointer.
4230075     ;
4230076     push cx
4230077     call __environment_setup
4230078     add  sp, #2
4230079     ;
4230080     ; After the environment copy is done, the value for the traditional
4230081     ; variable 'environ' is updated, to point to the new array of
4230082     ; pointer. The updated value comes from variable '_environment',
4230083     ; defined inside <stdlib.h>. Then, also the 'argv' contained inside
4230084     ; the stack is replaced with the new value.
4230085     ;
4230086     mov  ax, #__environment
4230087     mov  _environ, ax
4230088     ;
4230089     pop ax              ; argc
4230090     pop bx              ; argv[][]
4230091     pop cx              ; envp[][]
4230092     mov cx, #__environment
4230093     push cx
4230094     push bx
4230095     push ax
4230096     ;
4230097     ; Setup standard I/O streams and at-exit table.
4230098     ;
4230099     call __stdio_stream_setup
4230100     call __dirent_directory_stream_setup
4230101     call __atexit_setup
4230102     ;
4230103     ; Call the main function. The arguments are already pushed inside
4230104     ; the stack.
4230105     ;
4230106     call _main
4230107     ;
4230108     ; Save the return value at the symbol 'exit_value'.
4230109     ;
4230110     mov  exit_value, ax
4230111     ;
4230112 .align 2
4230113 halt:
4230114     ;
4230115     push #2             ; Size of message.
4230116     push #exit_value    ; Pointer to the message.
```

```
4230117     push #6             ; SYS_EXIT
4230118     call _sys
4230119     add  sp, #2
4230120     add  sp, #2
4230121     add  sp, #2
4230122     ;
4230123     jmp halt
4230124     ;
4230125 ;-------------------------------------------------------------------------
4230126 .align 2
4230127 ___mkargv:       ; Symbol '___mkargv' is used by Bcc inside the function
4230128     ret          ; 'main()' and must be present for a successful
4230129                  ; compilation.
4230130 ;-------------------------------------------------------------------------
4230131 .align 2
4230132 .data
4230133 ;
4230134 exit_value:
4230135     .data2 0x0000
4230136 ;-------------------------------------------------------------------------
4230137 .align 2
4230138 .bss
```

applic/date.c

«

Si veda la sezione u0.8.

```
4240001 #include <unistd.h>
4240002 #include <stdlib.h>
4240003 #include <errno.h>
4240004 #include <time.h>
4240005 #include <ctype.h>
4240006 //-------------------------------------------------------------------------
4240007 static void  usage          (void);
4240008 //-------------------------------------------------------------------------
4240009 int
4240010 main (int argc, char *argv[], char *envp[])
4240011 {
4240012     struct tm *timeptr;
4240013     char       string[5];
4240014     time_t     timer;
4240015     int        length;
```

```
4240016       char      *input;
4240017       int        i;
4240018       //
4240019       // There can be at most an argument.
4240020       //
4240021       if (argc > 2)
4240022         {
4240023           usage ();
4240024           return (1);
4240025         }
4240026       //
4240027       // Check if there is no argument: must show the date.
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 [MMDDHHMM[[CC]YY]]\n");
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
4250017 #include <stdio.h>
4250018 #include <ctype.h>
4250019 #include <unistd.h>
4250020 #include <stdbool.h>
4250021 #include <string.h>
4250022 #include <stdlib.h>
4250023 #include <fcntl.h>
4250024 //-----------------------------------------------------------------------
4250025 #define isoctal(ch)  (((ch) >= '0') && ((ch) <= '7'))
4250026 #define USERSIZE     1024       /* max line length typed in by user */
4250027 #define INITBUFSIZE  1024       /* initial buffer size */
4250028 //-----------------------------------------------------------------------
4250029 typedef int num_t;
4250030 typedef int len_t;
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           {
```

```
4250096             termedit ();
4250097             return (0);
4250098           }
4250099         //
4250100         if (lastnum) setcurnum(1);
4250101         //
4250102         dirty = false;
4250103       }
4250104     //
4250105     docommands ();
4250106     //
4250107     termedit ();
4250108     return (0);
4250109 }
4250110 //------------------------------------------------------------------------
4250111 // Read commands until we are told to stop.
4250112 //------------------------------------------------------------------------
4250113 void
4250114 docommands (void)
4250115 {
4250116     char   *cp;
4250117     int     len;
4250118     num_t   num1;
4250119     num_t   num2;
4250120     bool    have1;
4250121     bool    have2;
4250122     char    buf[USERSIZE];
4250123     //
4250124     while (true)
4250125       {
4250126         printf(": ");
4250127         fflush (stdout);
4250128         //
4250129         if (fgets (buf, sizeof(buf), stdin) == NULL)
4250130           {
4250131             return;
4250132           }
4250133         //
4250134         len = strlen (buf);
4250135         if (len == 0)
4250136           {
4250137             return;
4250138           }
```

```
4250139           //
4250140           cp = &buf[len - 1];
4250141           if (*cp != '\n')
4250142             {
4250143               fprintf(stderr, "Command line too long\n");
4250144               do
4250145                 {
4250146                   len = fgetc(stdin);
4250147                 }
4250148               while ((len != EOF) && (len != '\n'));
4250149               //
4250150               continue;
4250151             }
4250152           //
4250153           while ((cp > buf) && isblank (cp[-1]))
4250154             {
4250155               cp--;
4250156             }
4250157           //
4250158           *cp = '\0';
4250159           //
4250160           cp = buf;
4250161           //
4250162           while (isblank (*cp))
4250163             {
4250164               //*cp++;
4250165               cp++;
4250166             }
4250167           //
4250168           have1 = false;
4250169           have2 = false;
4250170           //
4250171           if ((curnum == 0) && (lastnum > 0))
4250172             {
4250173               curnum = 1;
4250174               curline = lines.next;
4250175             }
4250176           //
4250177           if (!getnum (&cp, &have1, &num1))
4250178             {
4250179               continue;
4250180             }
4250181           //
```

```
4250182           while (isblank (*cp))
4250183             {
4250184               cp++;
4250185             }
4250186           //
4250187           if (*cp == ',')
4250188             {
4250189               cp++;
4250190               if (!getnum (&cp, &have2, &num2))
4250191                 {
4250192                   continue;
4250193                 }
4250194               //
4250195               if (!have1)
4250196                 {
4250197                   num1 = 1;
4250198                 }
4250199               if (!have2)
4250200                 {
4250201                   num2 = lastnum;
4250202                 }
4250203               have1 = true;
4250204               have2 = true;
4250205             }
4250206           //
4250207           if (!have1)
4250208             {
4250209               num1 = curnum;
4250210             }
4250211           if (!have2)
4250212             {
4250213               num2 = num1;
4250214             }
4250215           //
4250216           // Command interpretation switch.
4250217           //
4250218           switch (*cp++)
4250219             {
4250220               case 'a':
4250221                   addlines (num1 + 1);
4250222                   break;
4250223                   //
4250224               case 'c':
```

```
4250225                     deletelines (num1, num2);
4250226                     addlines (num1);
4250227                     break;
4250228                     //
4250229                 case 'd':
4250230                     deletelines (num1, num2);
4250231                     break;
4250232                     //
4250233                 case 'f':
4250234                     if (*cp && !isblank (*cp))
4250235                       {
4250236                         fprintf (stderr, "Bad file command\n");
4250237                         break;
4250238                       }
4250239                     //
4250240                     while (isblank (*cp))
4250241                       {
4250242                         cp++;
4250243                       }
4250244                     if (*cp == '\0')
4250245                       {
4250246                         if (filename)
4250247                           {
4250248                             printf ("\"%s\"\n", filename);
4250249                           }
4250250                         else
4250251                           {
4250252                             printf ("No filename\n");
4250253                           }
4250254                         break;
4250255                       }
4250256                     //
4250257                     cp = strdup (cp);
4250258                     //
4250259                     if (cp == NULL)
4250260                       {
4250261                         fprintf (stderr, "No memory for filename\n");
4250262                         break;
4250263                       }
4250264                     //
4250265                     if (filename)
4250266                       {
4250267                         free(filename);
```

```
4250268                     }
4250269                 //
4250270                 filename = cp;
4250271                 break;
4250272                 //
4250273             case 'i':
4250274                 addlines (num1);
4250275                 break;
4250276                 //
4250277             case 'k':
4250278                 while (isblank(*cp))
4250279                   {
4250280                     cp++;
4250281                   }
4250282                 //
4250283                 if ((*cp < 'a') || (*cp > 'a') || cp[1])
4250284                   {
4250285                     fprintf (stderr, "Bad mark name\n");
4250286                     break;
4250287                   }
4250288                 //
4250289                 marks[*cp - 'a'] = num2;
4250290                 break;
4250291                 //
4250292             case 'l':
4250293                 printlines (num1, num2, true);
4250294                 break;
4250295                 //
4250296             case 'p':
4250297                 printlines (num1, num2, false);
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 'r':
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;
```

```
                    //
                case '.':
                    if (have1)
                      {
                        fprintf (stderr, "No arguments allowed\n");
                        break;
                      }
                    printlines (curnum, curnum, false);
                    break;
                    //
                case '-':
                    if (setcurnum (curnum - 1))
                      {
                        printlines (curnum, curnum, false);
                      }
                    break;
                    //
                case '=':
                    printf ("%d\n", num1);
                    break;
                    //
                case '\0':
                    if (have1)
                      {
                        printlines (num2, num2, false);
                        break;
                      }
                    //
                    if (setcurnum (curnum + 1))
                      {
                        printlines (curnum, curnum, false);
                      }
                    break;
                    //
                default:
                    fprintf (stderr, "Unimplemented command\n");
                    break;
              }
          }
    }
    //-----------------------------------------------------------------------
    // Do the substitute command.
    // The current line is set to the last substitution done.
```

```
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;
4250500 
4250501     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4250502       {
4250503         fprintf (stderr, "Bad line range for substitute\n");
4250504         return;
4250505       }
4250506     //
4250507     globalflag = false;
4250508     printflag = false;
4250509     didsub = false;
4250510     needprint = false;
4250511     //
4250512     if (isblank (*cp) || (*cp == '\0'))
4250513       {
4250514         fprintf (stderr, "Bad delimiter for substitute\n");
4250515         return;
4250516       }
4250517     //
4250518     delim = *cp++;
4250519     oldstr = cp;
4250520     //
4250521     cp = strchr (cp, delim);
4250522     //
4250523     if (cp == NULL)
4250524       {
4250525         fprintf (stderr, "Missing 2nd delimiter for substitute\n");
```

```
4250526           return;
4250527         }
4250528       //
4250529       *cp++ = '\0';
4250530       //
4250531       newstr = cp;
4250532       cp = strchr (cp, delim);
4250533       //
4250534       if (cp)
4250535         {
4250536           *cp++ = '\0';
4250537         }
4250538       else
4250539         {
4250540           cp = "";
4250541         }
4250542       while (*cp)
4250543         {
4250544           switch (*cp++)
4250545             {
4250546               case 'g':
4250547                   globalflag = true;
4250548                   break;
4250549                   //
4250550               case 'p':
4250551                   printflag = true;
4250552                   break;
4250553                   //
4250554               default:
4250555                   fprintf (stderr, "Unknown option for substitute\n");
4250556                   return;
4250557             }
4250558         }
4250559       //
4250560       if (*oldstr == '\0')
4250561         {
4250562           if (searchstring[0] == '\0')
4250563             {
4250564               fprintf (stderr, "No previous search string\n");
4250565               return;
4250566             }
4250567           oldstr = searchstring;
4250568         }
```

```
4250569       //
4250570       if (oldstr != searchstring)
4250571         {
4250572           strcpy (searchstring, oldstr);
4250573         }
4250574       //
4250575       lp = findline (num1);
4250576       if (lp == NULL)
4250577         {
4250578           return;
4250579         }
4250580       //
4250581       oldlen = strlen(oldstr);
4250582       newlen = strlen(newstr);
4250583       deltalen = newlen - oldlen;
4250584       offset = 0;
4250585       //
4250586       while (num1 <= num2)
4250587         {
4250588           offset = findstring (lp, oldstr, oldlen, offset);
4250589           if (offset < 0)
4250590             {
4250591               if (needprint)
4250592                 {
4250593                   printlines (num1, num1, false);
4250594                   needprint = false;
4250595                 }
4250596               //
4250597               offset = 0;
4250598               lp = lp->next;
4250599               num1++;
4250600               continue;
4250601             }
4250602           //
4250603           needprint = printflag;
4250604           didsub = true;
4250605           dirty = true;
4250606
4250607           //-------------------------------------------------------------
4250608           // If the replacement string is the same size or shorter
4250609           // than the old string, then the substitution is easy.
4250610           //-------------------------------------------------------------
4250611
```

```
4250612           if (deltalen <= 0)
4250613             {
4250614               memcpy (&lp->data[offset], newstr, newlen);
4250615               //
4250616               if (deltalen)
4250617                 {
4250618                   memcpy (&lp->data[offset + newlen],
4250619                           &lp->data[offset + oldlen],
4250620                           lp->len - offset - oldlen);
4250621                   //
4250622                   lp->len += deltalen;
4250623                 }
4250624               //
4250625               offset += newlen;
4250626               //
4250627               if (globalflag)
4250628                 {
4250629                   continue;
4250630                 }
4250631               //
4250632               if (needprint)
4250633                 {
4250634                   printlines(num1, num1, false);
4250635                   needprint = false;
4250636                 }
4250637               //
4250638               lp = nlp->next;
4250639               num1++;
4250640               continue;
4250641             }
4250642
4250643           //---------------------------------------------------------------
4250644           // The new string is larger, so allocate a new line
4250645           // structure and use that.  Link it in in place of
4250646           // the old line structure.
4250647           //---------------------------------------------------------------
4250648
4250649           nlp = (line_t *) malloc (sizeof (line_t) + lp->len + deltalen);
4250650           //
4250651           if (nlp == NULL)
4250652             {
4250653               fprintf (stderr, "Cannot get memory for line\n");
4250654               return;
```

```
4250655             }
4250656           //
4250657           nlp->len = lp->len + deltalen;
4250658           //
4250659           memcpy (nlp->data, lp->data, offset);
4250660           //
4250661           memcpy (&nlp->data[offset], newstr, newlen);
4250662           //
4250663           memcpy (&nlp->data[offset + newlen],
4250664                   &lp->data[offset + oldlen],
4250665                   lp->len - offset - oldlen);
4250666           //
4250667           nlp->next = lp->next;
4250668           nlp->prev = lp->prev;
4250669           nlp->prev->next = nlp;
4250670           nlp->next->prev = nlp;
4250671           //
4250672           if (curline == lp)
4250673             {
4250674               curline = nlp;
4250675             }
4250676           //
4250677           free(lp);
4250678           lp = nlp;
4250679           //
4250680           offset += newlen;
4250681           //
4250682           if (globalflag)
4250683             {
4250684               continue;
4250685             }
4250686           //
4250687           if (needprint)
4250688             {
4250689               printlines (num1, num1, false);
4250690               needprint = false;
4250691             }
4250692           //
4250693           lp = lp->next;
4250694           num1++;
4250695         }
4250696       //
4250697       if (!didsub)
```

```
4250698         {
4250699           fprintf (stderr, "No substitutions found for \"%s\"\n", oldstr);
4250700         }
4250701   }
4250702   //-----------------------------------------------------------------------
4250703   // Search a line for the specified string starting at the specified
4250704   // offset in the line.  Returns the offset of the found string, or -1.
4250705   //-----------------------------------------------------------------------
4250706   len_t
4250707   findstring (line_t *lp, char *str, len_t len, len_t offset)
4250708   {
4250709       len_t    left;
4250710       char    *cp;
4250711       char    *ncp;
4250712       //
4250713       cp = &lp->data[offset];
4250714       left = lp->len - offset;
4250715       //
4250716       while (left >= len)
4250717         {
4250718           ncp = memchr(cp, *str, left);
4250719           if (ncp == NULL)
4250720             {
4250721               return (len_t) -1;
4250722             }
4250723           //
4250724           left -= (ncp - cp);
4250725           if (left < len)
4250726             {
4250727               return (len_t) -1;
4250728             }
4250729           //
4250730           cp = ncp;
4250731           if (memcmp(cp, str, len) == 0)
4250732             {
4250733               return (len_t) (cp - lp->data);
4250734             }
4250735           //
4250736           cp++;
4250737           left--;
4250738         }
4250739       //
4250740       return (len_t) -1;
```

```
4250741 }
4250742 //---------------------------------------------------------------------
4250743 // Add lines which are typed in by the user.
4250744 // The lines are inserted just before the specified line number.
4250745 // The lines are terminated by a line containing a single dot (ugly!),
4250746 // or by an end of file.
4250747 //---------------------------------------------------------------------
4250748 void
4250749 addlines (num_t num)
4250750 {
4250751     int     len;
4250752     char    buf[USERSIZE + 1];
4250753     //
4250754     while (fgets (buf, sizeof (buf), stdin))
4250755       {
4250756         if ((buf[0] == '.') && (buf[1] == '\n') && (buf[2] == '\0'))
4250757           {
4250758             return;
4250759           }
4250760         //
4250761         len = strlen (buf);
4250762         //
4250763         if (len == 0)
4250764           {
4250765             return;
4250766           }
4250767         //
4250768         if (buf[len - 1] != '\n')
4250769          {
4250770             fprintf (stderr, "Line too long\n");
4250771             //
4250772             do
4250773               {
4250774                 len = fgetc(stdin);
4250775               }
4250776             while ((len != EOF) && (len != '\n'));
4250777             //
4250778             return;
4250779           }
4250780         //
4250781         if (!insertline (num++, buf, len))
4250782           {
4250783             return;
```

```
4250784             }
4250785         }
4250786 }
4250787 //-----------------------------------------------------------------------
4250788 // Parse a line number argument if it is present.  This is a sum
4250789 // or difference of numbers, '.', '$', 'x, or a search string.
4250790 // Returns true if successful (whether or not there was a number).
4250791 // Returns false if there was a parsing error, with a message output.
4250792 // Whether there was a number is returned indirectly, as is the number.
4250793 // The character pointer which stopped the scan is also returned.
4250794 //-----------------------------------------------------------------------
4250795 static bool
4250796 getnum (char **retcp, bool *rethavenum, num_t *retnum)
4250797 {
4250798     char   *cp;
4250799     char   *str;
4250800     bool    havenum;
4250801     num_t   value;
4250802     num_t   num;
4250803     num_t   sign;
4250804     //
4250805     cp = *retcp;
4250806     havenum = false;
4250807     value = 0;
4250808     sign = 1;
4250809     //
4250810     while (true)
4250811       {
4250812         while (isblank(*cp))
4250813           {
4250814             cp++;
4250815           }
4250816         //
4250817         switch (*cp)
4250818           {
4250819             case '.':
4250820                 havenum = true;
4250821                 num = curnum;
4250822                 cp++;
4250823                 break;
4250824                 //
4250825             case '$':
4250826                 havenum = true;
```

```
4250827                 num = lastnum;
4250828                 cp++;
4250829                 break;
4250830                 //
4250831             case '\'':
4250832                 cp++;
4250833                 if ((*cp < 'a') || (*cp > 'z'))
4250834                   {
4250835                     fprintf (stderr, "Bad mark name\n");
4250836                     return false;
4250837                   }
4250838                 //
4250839                 havenum = true;
4250840                 num = marks[*cp++ - 'a'];
4250841                 break;
4250842                 //
4250843             case '/':
4250844                 str = ++cp;
4250845                 cp = strchr (str, '/');
4250846                 if (cp)
4250847                   {
4250848                     *cp++ = '\0';
4250849                   }
4250850                 else
4250851                   {
4250852                     cp = "";
4250853                   }
4250854                 num = searchlines (str, curnum, lastnum);
4250855                 if (num == 0)
4250856                   {
4250857                     return false;
4250858                   }
4250859                 //
4250860                 havenum = true;
4250861                 break;
4250862                 //
4250863             default:
4250864                 if (!isdigit (*cp))
4250865                  {
4250866                     *retcp = cp;
4250867                     *rethavenum = havenum;
4250868                     *retnum = value;
4250869                     return true;
```

```
                    }
                  //
                  num = 0;
                  while (isdigit(*cp))
                    {
                      num = num * 10 + *cp++ - '0';
                    }
                  havenum = true;
                  break;
              }
            //
            value += num * sign;
            //
            while (isblank (*cp))
              {
                cp++;
              }
            //
            switch (*cp)
              {
                case '-':
                    sign = -1;
                    cp++;
                    break;
                    //
                case '+':
                    sign = 1;
                    cp++;
                    break;
                    //
                default:
                    *retcp = cp;
                    *rethavenum = havenum;
                    *retnum = value;
                    return true;
              }
        }
    }
//-----------------------------------------------------------------------
// Initialize everything for editing.
//-----------------------------------------------------------------------
bool
initedit (void)
```

```
{
    int i;
    //
    bufsize = INITBUFSIZE;
    bufbase = malloc (bufsize);
    //
    if (bufbase == NULL)
      {
        fprintf (stderr, "No memory for buffer\n");
        return false;
      }
    //
    bufptr = bufbase;
    bufused = 0;
    //
    lines.next = &lines;
    lines.prev = &lines;
    //
    curline = NULL;
    curnum = 0;
    lastnum = 0;
    dirty = false;
    filename = NULL;
    searchstring[0] = '\0';
    //
    for (i = 0; i < 26; i++)
      {
        marks[i] = 0;
      }
    //
    return true;
}
//-----------------------------------------------------------------------
// Finish editing.
//-----------------------------------------------------------------------
void
termedit (void)
{
    if (bufbase) free(bufbase);
    bufbase = NULL;
    //
    bufptr = NULL;
    bufsize = 0;
```

```
4250956     bufused = 0;
4250957     //
4250958     if (filename) free(filename);
4250959     filename = NULL;
4250960     //
4250961     searchstring[0] = '\0';
4250962     //
4250963     if (lastnum) deletelines (1, lastnum);
4250964     //
4250965     lastnum = 0;
4250966     curnum = 0;
4250967     curline = NULL;
4250968 }
4250969 //-----------------------------------------------------------------------
4250970 // Read lines from a file at the specified line number.
4250971 // Returns true if the file was successfully read.
4250972 //-----------------------------------------------------------------------
4250973 bool
4250974 readlines (char *file, num_t num)
4250975 {
4250976     int     fd;
4250977     int     cc;
4250978     len_t   len;
4250979     len_t   linecount;
4250980     len_t   charcount;
4250981     char   *cp;
4250982     //
4250983     if ((num < 1) || (num > lastnum + 1))
4250984       {
4250985         fprintf (stderr, "Bad line for read\n");
4250986         return false;
4250987       }
4250988     //
4250989     fd = open (file, O_RDONLY);
4250990     if (fd < 0)
4250991       {
4250992         perror (file);
4250993         return false;
4250994       }
4250995     //
4250996     bufptr = bufbase;
4250997     bufused = 0;
4250998     linecount = 0;
```

```
      charcount = 0;
      //
      printf ("\"%s\", ", file);
      fflush(stdout);
      //
      do
        {
          cp = memchr(bufptr, '\n', bufused);
          if (cp)
            {
              len = (cp - bufptr) + 1;
              //
              if (!insertline (num, bufptr, len))
                {
                  close (fd);
                  return false;
                }
              //
              bufptr += len;
              bufused -= len;
              charcount += len;
              linecount++;
              num++;
              continue;
            }
          //
          if (bufptr != bufbase)
            {
              memcpy (bufbase, bufptr, bufused);
              bufptr = bufbase + bufused;
            }
          //
          if (bufused >= bufsize)
            {
              len = (bufsize * 3) / 2;
              cp = realloc (bufbase, len);
              if (cp == NULL)
                {
                  fprintf (stderr, "No memory for buffer\n");
                  close (fd);
                  return false;
                }
              //
```

```
4251042               bufbase = cp;
4251043               bufptr = bufbase + bufused;
4251044               bufsize = len;
4251045             }
4251046           //
4251047           cc = read (fd, bufptr, bufsize - bufused);
4251048           bufused += cc;
4251049           bufptr = bufbase;
4251050         }
4251051       while (cc > 0);
4251052       //
4251053       if (cc < 0)
4251054         {
4251055           perror (file);
4251056           close (fd);
4251057           return false;
4251058         }
4251059       //
4251060       if (bufused)
4251061         {
4251062           if (!insertline (num, bufptr, bufused))
4251063             {
4251064               close (fd);
4251065               return -1;
4251066             }
4251067           linecount++;
4251068           charcount += bufused;
4251069         }
4251070       //
4251071       close (fd);
4251072       //
4251073       printf ("%d lines%s, %d chars\n",
4251074               linecount,
4251075               (bufused ? " (incomplete)" : ""),
4251076               charcount);
4251077       //
4251078       return true;
4251079   }
4251080   //---------------------------------------------------------------------
4251081   // Write the specified lines out to the specified file.
4251082   // Returns true if successful, or false on an error with a message
4251083   // output.
4251084   //---------------------------------------------------------------------
```

```
4251085 bool
4251086 writelines (char *file, num_t num1, num_t num2)
4251087 {
4251088     int     fd;
4251089     line_t *lp;
4251090     len_t   linecount;
4251091     len_t   charcount;
4251092     //
4251093     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4251094       {
4251095         fprintf (stderr, "Bad line range for write\n");
4251096         return false;
4251097       }
4251098     //
4251099     linecount = 0;
4251100     charcount = 0;
4251101     //
4251102     fd = creat (file, 0666);
4251103     if (fd < 0)
4251104       {
4251105         perror (file);
4251106         return false;
4251107       }
4251108     //
4251109     printf("\"%s\", ", file);
4251110     fflush (stdout);
4251111     //
4251112     lp = findline (num1);
4251113     if (lp == NULL)
4251114       {
4251115         close (fd);
4251116         return false;
4251117       }
4251118     //
4251119     while (num1++ <= num2)
4251120       {
4251121         if (write(fd, lp->data, lp->len) != lp->len)
4251122           {
4251123             perror(file);
4251124             close(fd);
4251125             return false;
4251126           }
4251127         //
```

```
4251128           charcount += lp->len;
4251129           linecount++;
4251130           lp = lp->next;
4251131         }
4251132       //
4251133       if (close(fd) < 0)
4251134         {
4251135           perror(file);
4251136           return false;
4251137         }
4251138       //
4251139       printf ("%d lines, %d chars\n", linecount, charcount);
4251140       //
4251141       return true;
4251142   }
4251143   //------------------------------------------------------------------------
4251144   // Print lines in a specified range.
4251145   // The last line printed becomes the current line.
4251146   // If expandflag is true, then the line is printed specially to
4251147   // show magic characters.
4251148   //------------------------------------------------------------------------
4251149   bool
4251150   printlines (num_t num1, num_t num2, bool expandflag)
4251151   {
4251152       line_t        *lp;
4251153       unsigned char *cp;
4251154       int            ch;
4251155       len_t          count;
4251156       //
4251157       if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4251158         {
4251159           fprintf (stderr, "Bad line range for print\n");
4251160           return false;
4251161         }
4251162       //
4251163       lp = findline (num1);
4251164       if (lp == NULL)
4251165         {
4251166           return false;
4251167         }
4251168       //
4251169       while (num1 <= num2)
4251170         {
```

```
4251171           if (!expandflag)
4251172             {
4251173               write (STDOUT_FILENO, lp->data, lp->len);
4251174               setcurnum (num1++);
4251175               lp = lp->next;
4251176               continue;
4251177             }
4251178
4251179           //-----------------------------------------------------------------
4251180           // Show control characters and characters with the
4251181           // high bit set specially.
4251182           //-----------------------------------------------------------------
4251183
4251184           cp = (unsigned char *) lp->data;
4251185           count = lp->len;
4251186           //
4251187           if ((count > 0) && (cp[count - 1] == '\n'))
4251188             {
4251189               count--;
4251190             }
4251191           //
4251192           while (count-- > 0)
4251193             {
4251194               ch = *cp++;
4251195               if (ch & 0x80)
4251196                 {
4251197                   fputs ("M-", stdout);
4251198                   ch &= 0x7f;
4251199                 }
4251200               if (ch < ' ')
4251201                 {
4251202                   fputc ('^', stdout);
4251203                   ch += '@';
4251204                 }
4251205               if (ch == 0x7f)
4251206                 {
4251207                   fputc ('^', stdout);
4251208                   ch = '?';
4251209                 }
4251210               fputc (ch, stdout);
4251211             }
4251212           //
4251213           fputs ("$\n", stdout);
```

```
4251214         //
4251215         setcurnum (num1++);
4251216         lp = lp->next;
4251217       }
4251218     //
4251219     return true;
4251220 }
4251221 //---------------------------------------------------------------------
4251222 // Insert a new line with the specified text.
4251223 // The line is inserted so as to become the specified line,
4251224 // thus pushing any existing and further lines down one.
4251225 // The inserted line is also set to become the current line.
4251226 // Returns true if successful.
4251227 //---------------------------------------------------------------------
4251228 bool
4251229 insertline (num_t num, char *data, len_t len)
4251230 {
4251231     line_t    *newlp;
4251232     line_t    *lp;
4251233     //
4251234     if ((num < 1) || (num > lastnum + 1))
4251235       {
4251236         fprintf (stderr, "Inserting at bad line number\n");
4251237         return false;
4251238       }
4251239     //
4251240     newlp = (line_t *) malloc (sizeof (line_t) + len - 1);
4251241     if (newlp == NULL)
4251242       {
4251243         fprintf (stderr, "Failed to allocate memory for line\n");
4251244         return false;
4251245       }
4251246     //
4251247     memcpy (newlp->data, data, len);
4251248     newlp->len = len;
4251249     //
4251250     if (num > lastnum)
4251251       {
4251252         lp = &lines;
4251253       }
4251254     else
4251255       {
4251256         lp = findline (num);
```

```
4251257           if (lp == NULL)
4251258             {
4251259               free ((char *) newlp);
4251260               return false;
4251261             }
4251262         }
4251263       //
4251264       newlp->next = lp;
4251265       newlp->prev = lp->prev;
4251266       lp->prev->next = newlp;
4251267       lp->prev = newlp;
4251268       //
4251269       lastnum++;
4251270       dirty = true;
4251271       //
4251272       return setcurnum (num);
4251273   }
4251274   //-----------------------------------------------------------------------
4251275   // Delete lines from the given range.
4251276   //-----------------------------------------------------------------------
4251277   bool
4251278   deletelines (num_t num1, num_t num2)
4251279   {
4251280       line_t   *lp;
4251281       line_t   *nlp;
4251282       line_t   *plp;
4251283       num_t     count;
4251284       //
4251285       if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4251286         {
4251287           fprintf (stderr, "Bad line numbers for delete\n");
4251288           return false;
4251289         }
4251290       //
4251291       lp = findline (num1);
4251292       if (lp == NULL)
4251293         {
4251294           return false;
4251295         }
4251296       //
4251297       if ((curnum >= num1) && (curnum <= num2))
4251298         {
4251299           if (num2 < lastnum)
```

```
4251300             {
4251301               setcurnum (num2 + 1);
4251302             }
4251303           else if (num1 > 1)
4251304             {
4251305               setcurnum (num1 - 1);
4251306             }
4251307           else
4251308             {
4251309               curnum = 0;
4251310             }
4251311         }
4251312       //
4251313       count = num2 - num1 + 1;
4251314       //
4251315       if (curnum > num2)
4251316         {
4251317           curnum -= count;
4251318         }
4251319       //
4251320       lastnum -= count;
4251321       //
4251322       while (count-- > 0)
4251323         {
4251324           nlp = lp->next;
4251325           plp = lp->prev;
4251326           plp->next = nlp;
4251327           nlp->prev = plp;
4251328           lp->next = NULL;
4251329           lp->prev = NULL;
4251330           lp->len = 0;
4251331           free(lp);
4251332           lp = nlp;
4251333         }
4251334       //
4251335       dirty = true;
4251336       //
4251337       return true;
4251338   }
4251339   //---------------------------------------------------------------------
4251340   // Search for a line which contains the specified string.
4251341   // If the string is NULL, then the previously searched for string
4251342   // is used.  The currently searched for string is saved for future use.
```

```
4251343 // Returns the line number which matches, or 0 if there was no match
4251344 // with an error printed.
4251345 //------------------------------------------------------------------------
4251346 num_t
4251347 searchlines (char *str, num_t num1, num_t num2)
4251348 {
4251349     line_t *lp;
4251350     int     len;
4251351     //
4251352     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
4251353       {
4251354         fprintf (stderr, "Bad line numbers for search\n");
4251355         return 0;
4251356       }
4251357     //
4251358     if (*str == '\0')
4251359       {
4251360         if (searchstring[0] == '\0')
4251361           {
4251362             fprintf(stderr, "No previous search string\n");
4251363             return 0;
4251364           }
4251365         str = searchstring;
4251366       }
4251367     //
4251368     if (str != searchstring)
4251369       {
4251370         strcpy(searchstring, str);
4251371       }
4251372     //
4251373     len = strlen(str);
4251374     //
4251375     lp = findline (num1);
4251376     if (lp == NULL)
4251377       {
4251378         return 0;
4251379       }
4251380     //
4251381     while (num1 <= num2)
4251382       {
4251383         if (findstring(lp, str, len, 0) >= 0)
4251384           {
4251385             return num1;
```

```
4251386             }
4251387           //
4251388           num1++;
4251389           lp = lp->next;
4251390         }
4251391       //
4251392       fprintf (stderr, "Cannot find string \"%s\"\n", str);
4251393       //
4251394       return 0;
4251395   }
4251396   //-----------------------------------------------------------------------
4251397   // Return a pointer to the specified line number.
4251398   //-----------------------------------------------------------------------
4251399   line_t *
4251400   findline (num_t num)
4251401   {
4251402       line_t   *lp;
4251403       num_t     lnum;
4251404       //
4251405       if ((num < 1) || (num > lastnum))
4251406         {
4251407           fprintf (stderr, "Line number %d does not exist\n", num);
4251408           return NULL;
4251409         }
4251410       //
4251411       if (curnum <= 0)
4251412         {
4251413           curnum = 1;
4251414           curline = lines.next;
4251415         }
4251416       //
4251417       if (num == curnum)
4251418         {
4251419           return curline;
4251420         }
4251421       //
4251422       lp = curline;
4251423       lnum = curnum;
4251424       //
4251425       if (num < (curnum / 2))
4251426         {
4251427           lp = lines.next;
4251428           lnum = 1;
```

```
4251429       }
4251430     else if (num > ((curnum + lastnum) / 2))
4251431       {
4251432         lp = lines.prev;
4251433         lnum = lastnum;
4251434       }
4251435     //
4251436     while (lnum < num)
4251437       {
4251438         lp = lp->next;
4251439         lnum++;
4251440       }
4251441     //
4251442     while (lnum > num)
4251443       {
4251444         lp = lp->prev;
4251445         lnum--;
4251446       }
4251447     //
4251448     return lp;
4251449 }
4251450 //---------------------------------------------------------------------
4251451 // Set the current line number.
4251452 // Returns true if successful.
4251453 //---------------------------------------------------------------------
4251454 bool
4251455 setcurnum (num_t num)
4251456 {
4251457     line_t    *lp;
4251458     //
4251459     lp = findline (num);
4251460     if (lp == NULL)
4251461       {
4251462         return false;
4251463       }
4251464     //
4251465     curnum = num;
4251466     curline = lp;
4251467     //
4251468     return true;
4251469 }
4251470
```

```
4251471  /* END CODE */
```

applic/getty.c

Si veda la sezione u0.1.

```
4260001  #include <unistd.h>
4260002  #include <stdio.h>
4260003  #include <stdlib.h>
4260004  #include <signal.h>
4260005  #include <sys/wait.h>
4260006  #include <limits.h>
4260007  #include <sys/os16.h>
4260008  #include <fcntl.h>
4260009  #include <stdio.h>
4260010  //-------------------------------------------------------------------
4260011  int
4260012  main (int argc, char *argv[], char *envp[])
4260013  {
4260014      char   *device_name;
4260015      int     fdn;
4260016      char   *exec_argv[2];
4260017      char  **exec_envp;
4260018      char    buffer[BUFSIZ];
4260019      ssize_t size_read;
4260020      int     status;
4260021      //
4260022      // The first argument is mandatory and must be a console terminal.
4260023      //
4260024      device_name = argv[1];
4260025      //
4260026      // A console terminal is correctly selected (but it is not checked
4260027      // if it is a really available one).
4260028      // Set as a process group leader.
4260029      //
4260030      setpgrp ();
4260031      //
4260032      // Open the terminal, that should become the controlling terminal:
4260033      // close the standard input and open the new terminal (r/w).
4260034      //
4260035      close (0);
4260036      fdn = open (device_name, O_RDWR);
```

```
4260037       if (fdn < 0)
4260038         {
4260039           //
4260040           // Cannot open terminal. A message should appear, at least
4260041           // to the current console.
4260042           //
4260043           perror (NULL);
4260044           return (-1);
4260045         }
4260046       //
4260047       // Reset terminal device permissions and ownership.
4260048       //
4260049       status = fchown (fdn, (uid_t) 0, (gid_t) 0);
4260050       if (status != 0)
4260051         {
4260052           perror (NULL);
4260053         }
4260054       status = fchmod (fdn, 0644);
4260055       if (status != 0)
4260056         {
4260057           perror (NULL);
4260058         }
4260059       //
4260060       // The terminal is open and it should be already the controlling
4260061       // one: show '/etc/issue'. The same variable 'fdn' is used, because
4260062       // the controlling terminal will never be closed (the exit syscall
4260063       // will do it).
4260064       //
4260065       fdn = open ("/etc/issue", O_RDONLY);
4260066       if (fdn > 0)
4260067         {
4260068           //
4260069           // The file is present and is shown.
4260070           //
4260071           for (size_read = 1; size_read > 0;)
4260072             {
4260073               size_read = read (fdn, buffer, (size_t) (BUFSIZ - 1));
4260074               if (size_read < 0)
4260075                 {
4260076                   break;
4260077                 }
4260078               buffer[size_read] = '\0';
4260079               printf ("%s", buffer);
```

```
4260080             }
4260081           close (fdn);
4260082         }
4260083       //
4260084       // Show the terminal.
4260085       //
4260086       printf ("This is terminal %s\n", device_name);
4260087       //
4260088       // It is time to exec login: the environment is inherited directly
4260089       // from `init'.
4260090       //
4260091       exec_argv[0] = "login";
4260092       exec_argv[1] = NULL;
4260093       exec_envp    = envp;
4260094       execve ("/bin/login", exec_argv, exec_envp);
4260095       //
4260096       // If `execve()' returns, it is an error.
4260097       //
4260098       exit (-1);
4260099   }
```

applic/init.c

«

Si veda la sezione u0.2.

```
4270001   #include <unistd.h>
4270002   #include <stdio.h>
4270003   #include <stdlib.h>
4270004   #include <signal.h>
4270005   #include <sys/wait.h>
4270006   #include <limits.h>
4270007   #include <sys/os16.h>
4270008   #include <fcntl.h>
4270009   #include <string.h>
4270010   //---------------------------------------------------------------------
4270011   #define RESPAWN_MAX        7
4270012   #define COMMAND_MAX      100
4270013   #define LINE_MAX        1024
4270014   //---------------------------------------------------------------------
4270015   int
4270016   main (int argc, char *argv[], char *envp[])
4270017   {
```

```
4270018      //
4270019      // 'init.c' has its own 'init.crt0.s' with a very small stack
4270020      // size. Remember to verify to have enough room for the stack.
4270021      //
4270022      pid_t  pid;
4270023      int    status;
4270024      char  *exec_argv[3];
4270025      char  *exec_envp[3];
4270026      char   buffer[LINE_MAX];
4270027      int    r;                    // Respawn table index.
4270028      int    b;                    // Buffer index.
4270029      size_t size_read;
4270030      char  *inittab_id;
4270031      char  *inittab_runlevels;
4270032      char  *inittab_action;
4270033      char  *inittab_process;
4270034      int    eof;
4270035      int    fd;
4270036      //
4270037      // It follows a table for commands to be respawn.
4270038      //
4270039      struct {
4270040          pid_t   pid;
4270041          char    command[COMMAND_MAX];
4270042      } respawn[RESPAWN_MAX];
4270043
4270044      //-----------------------------------------------------------------
4270045      signal (SIGHUP,  SIG_IGN);
4270046      signal (SIGINT,  SIG_IGN);
4270047      signal (SIGQUIT, SIG_IGN);
4270048      signal (SIGILL,  SIG_IGN);
4270049      signal (SIGABRT, SIG_IGN);
4270050      signal (SIGFPE,  SIG_IGN);
4270051  //  signal (SIGKILL, SIG_IGN);  Cannot ignore SIGKILL.
4270052      signal (SIGSEGV, SIG_IGN);
4270053      signal (SIGPIPE, SIG_IGN);
4270054      signal (SIGALRM, SIG_IGN);
4270055      signal (SIGTERM, SIG_IGN);
4270056  //  signal (SIGSTOP, SIG_IGN);  Cannot ignore SIGSTOP.
4270057      signal (SIGTSTP, SIG_IGN);
4270058      signal (SIGCONT, SIG_IGN);
4270059      signal (SIGTTIN, SIG_IGN);
4270060      signal (SIGTTOU, SIG_IGN);
```

```
4270061       signal (SIGUSR1, SIG_IGN);
4270062       signal (SIGUSR2, SIG_IGN);
4270063       //-----------------------------------------------------------------
4270064       printf ("init\n");
4270065       //  heap_clear ();
4270066       //  process_info ();
4270067       //-----------------------------------------------------------------
4270068       //
4270069       // Reset the 'respawn' table.
4270070       //
4270071       for (r = 0; r < RESPAWN_MAX; r++)
4270072         {
4270073           respawn[r].pid = 0;
4270074           respawn[r].command[0]                   = 0;
4270075           respawn[r].command[COMMAND_MAX-1] = 0;
4270076         }
4270077       //
4270078       // Read the '/etc/inittab' file.
4270079       //
4270080       fd = open ("/etc/inittab", O_RDONLY);
4270081       //
4270082       if (fd < 0)
4270083         {
4270084           perror ("Cannot open file '/etc/inittab'");
4270085           exit (-1);
4270086         }
4270087       //
4270088       //
4270089       //
4270090       for (eof = 0, r = 0; !eof && r < RESPAWN_MAX; r++)
4270091         {
4270092           for (b = 0; b < LINE_MAX; b++)
4270093             {
4270094               size_read = read (fd, &buffer[b], (size_t) 1);
4270095               if (size_read <= 0)
4270096                 {
4270097                   buffer[b] = 0;
4270098                   eof       = 1;           // Close the read loop.
4270099                   break;
4270100                 }
4270101               if (buffer[b] == '\n')
4270102                 {
4270103                   buffer[b]  = 0;
```

```
4270104                   break;
4270105                 }
4270106             }
4270107           //
4270108           // Remove comments: just replace '#' with '\0'.
4270109           //
4270110           for (b = 0; b < LINE_MAX; b++)
4270111             {
4270112               if (buffer[b] == '#')
4270113                 {
4270114                   buffer[b] = 0;
4270115                   break;
4270116                 }
4270117             }
4270118           //
4270119           // If the buffer is an empty string, just loop to next
4270120           // record.
4270121           //
4270122           if (strlen (buffer) == 0)
4270123             {
4270124               r--;
4270125               continue;
4270126             }
4270127           //
4270128           //
4270129           //
4270130           inittab_id        = strtok (buffer, ":");
4270131           inittab_runlevels = strtok (NULL, ":");
4270132           inittab_action    = strtok (NULL, ":");
4270133           inittab_process   = strtok (NULL, ":");
4270134           //
4270135           // Only action 'respawn' is used.
4270136           //
4270137           if (strcmp (inittab_action, "respawn") == 0)
4270138             {
4270139               strncpy (respawn[r].command, inittab_process, COMMAND_MAX);
4270140             }
4270141           else
4270142             {
4270143               r--;
4270144             }
4270145         }
4270146       //
```

```
4270147       //
4270148       //
4270149       close (fd);
4270150       //
4270151       // Define common environment.
4270152       //
4270153       exec_envp[0] = "PATH=/bin:/usr/bin:/sbin:/usr/sbin";
4270154       exec_envp[1] = "CONSOLE=/dev/console";
4270155       exec_envp[2] = NULL;
4270156       //
4270157       // Start processes.
4270158       //
4270159       for (r = 0; r < RESPAWN_MAX; r++)
4270160         {
4270161           if (strlen (respawn[r].command) > 0)
4270162             {
4270163               respawn[r].pid = fork ();
4270164               if (respawn[r].pid == 0)
4270165                 {
4270166                   exec_argv[0] = strtok (respawn[r].command, " \t");
4270167                   exec_argv[1] = strtok (NULL, " \t");
4270168                   exec_argv[2] = NULL;
4270169                   execve (exec_argv[0], exec_argv, exec_envp);
4270170                   perror (NULL);
4270171                   exit (0);
4270172                 }
4270173             }
4270174         }
4270175       //
4270176       // Wait for the death of child.
4270177       //
4270178       while (1)
4270179         {
4270180           pid = wait (&status);
4270181           for (r = 0; r < RESPAWN_MAX; r++)
4270182             {
4270183               if (pid == respawn[r].pid)
4270184                 {
4270185                   //
4270186                   // Run it again.
4270187                   //
4270188                   respawn[r].pid = fork ();
4270189                   if (respawn[r].pid == 0)
```

```
4270190                     {
4270191                       exec_argv[0] = strtok (respawn[r].command, " \t");
4270192                       exec_argv[1] = strtok (NULL, " \t");
4270193                       exec_argv[2] = NULL;
4270194                       execve (exec_argv[0], exec_argv, exec_envp);
4270195                       exit (0);
4270196                     }
4270197                   break;
4270198                 }
4270199             }
4270200         }
4270201 }
```

applic/kill.c

«

Si veda la sezione u0.10.

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

```
4280026       extern int   optopt;
4280027       //
4280028       // There must be at least an option, plus the program name.
4280029       //
4280030       if (argc < 2)
4280031         {
4280032           usage ();
4280033           return (1);
4280034         }
4280035       //
4280036       // Check for options.
4280037       //
4280038       while ((opt = getopt (argc, argv, ":ls:")) != -1)
4280039         {
4280040           switch (opt)
4280041             {
4280042               case 'l':
4280043                   option_l = 1;
4280044                   break;
4280045               case 's':
4280046                   option_s = 1;
4280047                   //
4280048                   // In that case, there must be at least three arguments:
4280049                   // the option, the signal and the process id.
4280050                   //
4280051                   if (argc < 4)
4280052                     {
4280053                       usage ();
4280054                       return (1);
4280055                     }
4280056                   //
4280057                   // Argument numbers are ok. Check the signal.
4280058                   //
4280059                   if (strcmp (optarg, "HUP") == 0)
4280060                     {
4280061                       signal = SIGHUP;
4280062                     }
4280063                   else if (strcmp (optarg, "INT") == 0)
4280064                     {
4280065                       signal = SIGINT;
4280066                     }
4280067                   else if (strcmp (optarg, "QUIT") == 0)
4280068                     {
```

```
4280069                         signal = SIGQUIT;
4280070                       }
4280071                     else if (strcmp (optarg, "ILL") == 0)
4280072                       {
4280073                         signal = SIGILL;
4280074                       }
4280075                     else if (strcmp (optarg, "ABRT") == 0)
4280076                       {
4280077                         signal = SIGABRT;
4280078                       }
4280079                     else if (strcmp (optarg, "FPE") == 0)
4280080                       {
4280081                         signal = SIGFPE;
4280082                       }
4280083                     else if (strcmp (optarg, "KILL") == 0)
4280084                       {
4280085                         signal = SIGKILL;
4280086                       }
4280087                     else if (strcmp (optarg, "SEGV") == 0)
4280088                       {
4280089                         signal = SIGSEGV;
4280090                       }
4280091                     else if (strcmp (optarg, "PIPE") == 0)
4280092                       {
4280093                         signal = SIGPIPE;
4280094                       }
4280095                     else if (strcmp (optarg, "ALRM") == 0)
4280096                       {
4280097                         signal = SIGALRM;
4280098                       }
4280099                     else if (strcmp (optarg, "TERM") == 0)
4280100                       {
4280101                         signal = SIGTERM;
4280102                       }
4280103                     else if (strcmp (optarg, "STOP") == 0)
4280104                       {
4280105                         signal = SIGSTOP;
4280106                       }
4280107                     else if (strcmp (optarg, "TSTP") == 0)
4280108                       {
4280109                         signal = SIGTSTP;
4280110                       }
4280111                     else if (strcmp (optarg, "CONT") == 0)
```

```
4280112                     {
4280113                       signal = SIGCONT;
4280114                     }
4280115                   else if (strcmp (optarg, "CHLD") == 0)
4280116                     {
4280117                       signal = SIGCHLD;
4280118                     }
4280119                   else if (strcmp (optarg, "TTIN") == 0)
4280120                     {
4280121                       signal = SIGTTIN;
4280122                     }
4280123                   else if (strcmp (optarg, "TTOU") == 0)
4280124                     {
4280125                       signal = SIGTTOU;
4280126                     }
4280127                   else if (strcmp (optarg, "USR1") == 0)
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 }
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 }
```

«

Si veda la sezione u0.11.

```
4290001  #include <sys/os16.h>
4290002  #include <sys/stat.h>
4290003  #include <sys/types.h>
4290004  #include <unistd.h>
4290005  #include <stdlib.h>
4290006  #include <fcntl.h>
4290007  #include <errno.h>
4290008  #include <signal.h>
4290009  #include <stdio.h>
4290010  #include <string.h>
4290011  #include <limits.h>
4290012  #include <libgen.h>
4290013  //------------------------------------------------------------------------
4290014  static void usage (void);
4290015  //------------------------------------------------------------------------
4290016  int
4290017  main (int argc, char *argv[], char *envp[])
4290018  {
4290019      char        *source;
4290020      char        *destination;
4290021      char        *new_destination;
4290022      struct stat file_status;
4290023      int          dest_is_a_dir = 0;
4290024      int          a;                         // Argument index.
4290025      char         path[PATH_MAX];
4290026      //
4290027      // There must be at least two arguments, plus the program name.
4290028      //
4290029      if (argc < 3)
4290030        {
4290031          usage ();
4290032          return (1);
4290033        }
4290034      //
4290035      // Select the last argument as the destination.
4290036      //
4290037      destination = argv[argc-1];
4290038      //
4290039      // Check if it is a directory and save it in a flag.
4290040      //
```

```
4290041       if (stat (destination, &file_status) == 0)
4290042         {
4290043           if (S_ISDIR (file_status.st_mode))
4290044             {
4290045               dest_is_a_dir = 1;
4290046             }
4290047         }
4290048       //
4290049       // If there are more than two arguments, verify that the last
4290050       // one is a directory.
4290051       //
4290052       if (argc > 3)
4290053         {
4290054           if (!dest_is_a_dir)
4290055             {
4290056               usage ();
4290057               fprintf (stderr, "The destination \"%s\" ",
4290058                        destination);
4290059               fprintf (stderr, "is not a directory!\n");
4290060               return (1);
4290061             }
4290062         }
4290063       //
4290064       // Scan the arguments, excluded the last, that is the destination.
4290065       //
4290066       for (a = 1; a < (argc - 1); a++)
4290067         {
4290068           //
4290069           // Source.
4290070           //
4290071           source = argv[a];
4290072           //
4290073           // Verify access permissions.
4290074           //
4290075           if (access (source, R_OK) < 0)
4290076             {
4290077               perror (source);
4290078               continue;
4290079             }
4290080           //
4290081           // Destination.
4290082           //
4290083           // If it is a directory, the destination path
```

```
4290084           // must be corrected.
4290085           //
4290086           if (dest_is_a_dir)
4290087             {
4290088               path[0] = 0;
4290089               strcat (path, destination);
4290090               strcat (path, "/");
4290091               strcat (path, basename (source));
4290092               //
4290093               // Update the destination path.
4290094               //
4290095               new_destination = path;
4290096             }
4290097           else
4290098             {
4290099               new_destination = destination;
4290100             }
4290101           //
4290102           // Check if destination file exists.
4290103           //
4290104           if (stat (new_destination, &file_status) == 0)
4290105             {
4290106               fprintf (stderr, "The destination file, \"%s\", ",
4290107                        new_destination);
4290108               fprintf (stderr, "already exists!\n");
4290109               continue;
4290110             }
4290111           //
4290112           // Everything is ready for the link.
4290113           //
4290114           if (link (source, new_destination) < 0)
4290115             {
4290116               perror (new_destination);
4290117               continue;
4290118             }
4290119         }
4290120       //
4290121       // All done.
4290122       //
4290123       return (0);
4290124   }
4290125   //------------------------------------------------------------------------
4290126   static void
```

```
4290127 usage (void)
4290128 {
4290129     fprintf (stderr, "Usage: ln OLD_NAME NEW_NAME\n");
4290130     fprintf (stderr, "       ln FILE... DIRECTORY\n");
4290131 }
```

applic/login.c

«

Si veda la sezione u0.12.

```
4300001 #include <unistd.h>
4300002 #include <stdlib.h>
4300003 #include <sys/stat.h>
4300004 #include <sys/types.h>
4300005 #include <fcntl.h>
4300006 #include <errno.h>
4300007 #include <unistd.h>
4300008 #include <signal.h>
4300009 #include <stdio.h>
4300010 #include <sys/wait.h>
4300011 #include <stdio.h>
4300012 #include <string.h>
4300013 #include <limits.h>
4300014 #include <stdint.h>
4300015 #include <sys/os16.h>
4300016 //---------------------------------------------------------------------
4300017 #define LOGIN_MAX       64
4300018 #define PASSWORD_MAX    64
4300019 #define HOME_MAX        64
4300020 #define LINE_MAX        1024
4300021 //---------------------------------------------------------------------
4300022 int
4300023 main (int argc, char *argv[], char *envp[])
4300024 {
4300025     char    login[LOGIN_MAX];
4300026     char    password[PASSWORD_MAX];
4300027     char    buffer[LINE_MAX];
4300028     char   *user_name;
4300029     char   *user_password;
4300030     char   *user_uid;
4300031     char   *user_gid;
4300032     char   *user_description;
```

```
4300033       char   *user_home;
4300034       char   *user_shell;
4300035       uid_t   uid;
4300036       uid_t   euid;
4300037       int     fd;
4300038       ssize_t size_read;
4300039       int     b;          // Index inside buffer.
4300040       int     loop;
4300041       char   *exec_argv[2];
4300042       int     status;
4300043       char   *tty_path;
4300044       //
4300045       // Check if login is running correctly.
4300046       //
4300047       euid = geteuid ();
4300048       uid  = geteuid ();
4300049       //  //
4300050       //  // Show process info.
4300051       //  //
4300052       //  heap_clear ();
4300053       //  process_info ();
4300054       //
4300055       // Check privileges.
4300056       //
4300057       if (!(uid == 0 && euid == 0))
4300058         {
4300059           printf ("%s: can only run with root privileges!\n", argv[0]);
4300060           exit (-1);
4300061         }
4300062       //
4300063       // Prepare arguments for the shell call.
4300064       //
4300065       exec_argv[0] = "-";
4300066       exec_argv[1] = NULL;
4300067       //
4300068       // Login.
4300069       //
4300070       while (1)
4300071         {
4300072           fd = open ("/etc/passwd", O_RDONLY);
4300073           //
4300074           if (fd < 0)
4300075             {
```

```
4300076             perror ("Cannot open file `/etc/passwd'");
4300077             exit (-1);
4300078           }
4300079         //
4300080         printf ("Log in as \"root\" or \"user\" "
4300081                 "with password \"ciao\" :-)\n");
4300082         input_line (login, "login:", LOGIN_MAX, INPUT_LINE_ECHO);
4300083         //
4300084         //
4300085         //
4300086         loop = 1;
4300087         while (loop)
4300088           {
4300089             for (b = 0; b < LINE_MAX; b++)
4300090               {
4300091                 size_read = read (fd, &buffer[b], (size_t) 1);
4300092                 if (size_read <= 0)
4300093                   {
4300094                     buffer[b] = 0;
4300095                     loop = 0;           // Close the middle loop.
4300096                     break;
4300097                   }
4300098                 if (buffer[b] == '\n')
4300099                   {
4300100                     buffer[b] = 0;
4300101                     break;
4300102                   }
4300103               }
4300104             //
4300105             user_name        = strtok (buffer, ":");
4300106             user_password    = strtok (NULL, ":");
4300107             user_uid         = strtok (NULL, ":");
4300108             user_gid         = strtok (NULL, ":");
4300109             user_description = strtok (NULL, ":");
4300110             user_home        = strtok (NULL, ":");
4300111             user_shell       = strtok (NULL, ":");
4300112             //
4300113             if (strcmp (user_name, login) == 0)
4300114               {
4300115                 input_line (password, "password:", PASSWORD_MAX,
4300116                             INPUT_LINE_STARS);
4300117                 //
4300118                 // Compare passwords: empty passwords are not allowed.
```

```
                    //
                    if (strcmp (user_password, password) == 0)
                      {
                        uid  = atoi (user_uid);
                        euid = uid;
                        //
                        // Find the controlling terminal and change
                        // property and access permissions.
                        //
                        tty_path = ttyname (STDIN_FILENO);
                        if (tty_path != NULL)
                          {
                            status = chown (tty_path, uid, 0);
                            if (status != 0)
                              {
                                perror (NULL);
                              }
                            status = chmod (tty_path, 0600);
                            if (status != 0)
                              {
                                perror (NULL);
                              }
                          }
                        //
                        // Cd to the home directory, if present.
                        //
                        status = chdir (user_home);
                        if (status != 0)
                          {
                            perror (NULL);
                          }
                        //
                        // Now change personality.
                        //
                        setuid (uid);
                        seteuid (euid);
                        //
                        // Run the shell, replacing the login process; the
                        // environment is taken from 'init'.
                        //
                        execve (user_shell, exec_argv, envp);
                        exit (0);
                      }
```

```
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/os16.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     //
4310081     if (optind == argc)
4310082       {
4310083         //
4310084         // There are no more arguments. Replace the program name,
4310085         // corresponding to 'argv[0]', with the current directory
4310086         // path string.
4310087         //
4310088         argv[0] = ".";
4310089         argc    = 1;
4310090         optind  = 0;
4310091       }
4310092     //
4310093     // This is a very simplified 'ls': if there is only a name
4310094     // and it is a directory, the directory content is taken as
4310095     // the new 'argv[]' array.
4310096     //
4310097     if (optind == (argc -1))
4310098       {
4310099         //
4310100         // There is a request for a single name. Test if it exists
4310101         // and if it is a directory.
4310102         //
4310103         if (stat(argv[optind], &file_status) != 0)
4310104           {
4310105             fprintf (stderr, "File \"%s\" does not exist!\n",
4310106                      argv[optind]);
4310107             return (2);
4310108           }
4310109         //
4310110         if (S_ISDIR (file_status.st_mode))
4310111           {
4310112             //
4310113             // Save the directory inside the 'path' pointer.
```

```
4310114                 //
4310115                 path = argv[optind];
4310116                 //
4310117                 // Open the directory.
4310118                 //
4310119                 dp = opendir (argv[optind]);
4310120                 if (dp == NULL)
4310121                   {
4310122                     perror (argv[optind]);
4310123                     return (3);
4310124                   }
4310125                 //
4310126                 // Read the directory and fill the buffer with names.
4310127                 //
4310128                 b = 0;
4310129                 l = 0;
4310130                 while ((dir = readdir (dp)) != NULL)
4310131                   {
4310132                     len = strlen (dir->d_name);
4310133                     //
4310134                     // Check if the buffer can hold it.
4310135                     //
4310136                     if ((b + len + 1 ) > BUFFER_SIZE)
4310137                       {
4310138                         fprintf (stderr, "not enough memory\n");
4310139                         break;
4310140                       }
4310141                     //
4310142                     // Consider the directory item only if there is
4310143                     // a valid name. If it is empty, just ignore it.
4310144                     //
4310145                     if (len > 0)
4310146                       {
4310147                         strcpy (&buffer[b], dir->d_name);
4310148                         list[l] = &buffer[b];
4310149                         b += len + 1;
4310150                         l++;
4310151                       }
4310152                   }
4310153                 //
4310154                 // Close the directory.
4310155                 //
4310156                 closedir (dp);
```

```
4310157               //
4310158               // Sort the list.
4310159               //
4310160               qsort (list, (size_t) l, sizeof (char *), compare);
4310161
4310162               //
4310163               // Convert the directory list into a new 'argv[]' array,
4310164               // with a valid 'argc'. The variable 'optind' must be
4310165               // reset to the first element index, because there is
4310166               // no program name inside the new 'argv[]' at index zero.
4310167               //
4310168               argv   = list;
4310169               argc   = l;
4310170               optind = 0;
4310171             }
4310172         }
4310173       //
4310174       // Scan arguments, or list converted into 'argv[]'.
4310175       //
4310176       for (; optind < argc; optind++)
4310177         {
4310178           if (argv[optind][0] == '.')
4310179             {
4310180               //
4310181               // Current name starts with '.'.
4310182               //
4310183               if (!option_a)
4310184                 {
4310185                   //
4310186                   // Do not show name starting with '.'.
4310187                   //
4310188                   continue;
4310189                 }
4310190             }
4310191           //
4310192           // Build the pathname.
4310193           //
4310194           if (path == NULL)
4310195             {
4310196               strcpy (&pathname[0], argv[optind]);
4310197             }
4310198           else
4310199             {
```

```
4310200               strcpy (pathname, path);
4310201               strcat (pathname, "/");
4310202               strcat (pathname, argv[optind]);
4310203             }
4310204           //
4310205           // Check if file exists, reading status.
4310206           //
4310207           if (stat(pathname, &file_status) != 0)
4310208             {
4310209               fprintf (stderr, "File \"%s\" does not exist!\n",
4310210                        pathname);
4310211               return (2);
4310212             }
4310213           //
4310214           // Show file name.
4310215           //
4310216           if (option_l)
4310217             {
4310218               //
4310219               // Print the file type.
4310220               //
4310221               if      (S_ISBLK  (file_status.st_mode)) printf ("b");
4310222               else if (S_ISCHR  (file_status.st_mode)) printf ("c");
4310223               else if (S_ISFIFO (file_status.st_mode)) printf ("p");
4310224               else if (S_ISREG  (file_status.st_mode)) printf ("-");
4310225               else if (S_ISDIR  (file_status.st_mode)) printf ("d");
4310226               else if (S_ISLNK  (file_status.st_mode)) printf ("l");
4310227               else if (S_ISSOCK (file_status.st_mode)) printf ("s");
4310228               else                                     printf ("?");
4310229               //
4310230               // Print permissions.
4310231               //
4310232               if (S_IRUSR & file_status.st_mode) printf ("r");
4310233               else printf ("-");
4310234               if (S_IWUSR & file_status.st_mode) printf ("w");
4310235               else printf ("-");
4310236               if (S_IXUSR & file_status.st_mode) printf ("x");
4310237               else printf ("-");
4310238               if (S_IRGRP & file_status.st_mode) printf ("r");
4310239               else printf ("-");
4310240               if (S_IWGRP & file_status.st_mode) printf ("w");
4310241               else printf ("-");
4310242               if (S_IXGRP & file_status.st_mode) printf ("x");
```

```
4310243               else printf ("-");
4310244               if (S_IROTH & file_status.st_mode) printf ("r");
4310245               else printf ("-");
4310246               if (S_IWOTH & file_status.st_mode) printf ("w");
4310247               else printf ("-");
4310248               if (S_IXOTH & file_status.st_mode) printf ("x");
4310249               else printf ("-");
4310250               //
4310251               // Print links.
4310252               //
4310253               printf (" %3i", (int) file_status.st_nlink);
4310254               //
4310255               // Print owner.
4310256               //
4310257               pws = getpwuid (file_status.st_uid);
4310258               //
4310259               printf (" %s", pws->pw_name);
4310260               //
4310261               // Print group (no group available);
4310262               //
4310263               printf (" (no group)");
4310264               //
4310265               // Print file size or device major-minor.
4310266               //
4310267               if (S_ISBLK (file_status.st_mode)
4310268                   || S_ISCHR (file_status.st_mode))
4310269                 {
4310270                   printf (" %3i,", (int) major (file_status.st_rdev));
4310271                   printf (" %3i",  (int) minor (file_status.st_rdev));
4310272                 }
4310273               else
4310274                 {
4310275                   printf (" %8i", (int) file_status.st_size);
4310276                 }
4310277               //
4310278               // Print modification date and time.
4310279               //
4310280               tms = localtime (&(file_status.st_mtime));
4310281               printf (" %4u-%02u-%02u %02u:%02u",
4310282                       tms->tm_year, tms->tm_mon, tms->tm_mday,
4310283                       tms->tm_hour, tms->tm_min);
4310284               //
4310285               // Print file name, but with no additional path.
```

```
4310286               //
4310287               printf (" %s\n", argv[optind]);
4310288             }
4310289           else
4310290             {
4310291               //
4310292               // Just show the file name and go to the next line.
4310293               //
4310294               printf ("%s\n", argv[optind]);
4310295             }
4310296         }
4310297       //
4310298       // All done.
4310299       //
4310300       return (0);
4310301   }
4310302   //---------------------------------------------------------------------------
4310303   static void
4310304   usage (void)
4310305   {
4310306       fprintf (stderr, "Usage: ls [OPTION] [FILE]...\n");
4310307   }
4310308   //---------------------------------------------------------------------------
4310309   int
4310310   compare (void *p1, void *p2)
4310311   {
4310312       char **pp1 = p1;
4310313       char **pp2 = p2;
4310314       //
4310315       return (strcmp (*pp1, *pp2));
4310316   }
4310317
```

applic/man.c

«

Si veda la sezione u0.14.

```
4320001   #include <unistd.h>
4320002   #include <stdlib.h>
4320003   #include <errno.h>
4320004   //---------------------------------------------------------------------------
4320005   #define MAX_LINES   20
```

```
4320006 #define MAX_COLUMNS 80
4320007 //---------------------------------------------------------------------
4320008 static char *man_page_directory = "/usr/share/man";
4320009 //---------------------------------------------------------------------
4320010 static void  usage          (void);
4320011 static FILE *open_man_page  (int section, char *name);
4320012 static void build_path_name (int section, char *name, char *path);
4320013 //---------------------------------------------------------------------
4320014 int
4320015 main (int argc, char *argv[], char *envp[])
4320016 {
4320017     FILE       *fp;
4320018     char       *name;
4320019     int         section;
4320020     int         c;
4320021     int         line   = 1; // Line internal counter.
4320022     int         column = 1; // Column internal counter.
4320023     int         loop;
4320024     //
4320025     // There must be minimum an argument, and maximum two.
4320026     //
4320027     if (argc < 2 || argc > 3)
4320028       {
4320029         usage ();
4320030         return (1);
4320031       }
4320032     //
4320033     // If there are two arguments, there must be the
4320034     // section number.
4320035     //
4320036     if (argc == 3)
4320037       {
4320038         section = atoi (argv[1]);
4320039         name = argv[2];
4320040       }
4320041     else
4320042       {
4320043         section = 0;
4320044         name = argv[1];
4320045       }
4320046     //
4320047     // Try to open the manual page.
4320048     //
```

```
4320049       fp = open_man_page (section, name);
4320050       //
4320051       if (fp == NULL)
4320052         {
4320053           //
4320054           // Error opening file.
4320055           //
4320056           return (1);
4320057         }
4320058
4320059       //
4320060       // The following loop continues while the file
4320061       // gives characters, or when a command to change
4320062       // file or to quit is given.
4320063       //
4320064       for (loop = 1; loop; )
4320065         {
4320066           //
4320067           // Read a single character.
4320068           //
4320069           c = getc (fp);
4320070           //
4320071           if (c == EOF)
4320072             {
4320073               loop = 0;
4320074               break;
4320075             }
4320076           //
4320077           // If the character read is a special one,
4320078           // the line/column calculation is modified,
4320079           // so that it is known when to stop scrolling.
4320080           //
4320081           switch (c)
4320082             {
4320083               case '\r':
4320084                   //
4320085                   // Displaying this character, the cursor should go
4320086                   // back to the first column. So the column counter
4320087                   // is reset.
4320088                   //
4320089                   column = 1;
4320090                   break;
4320091               case '\n':
```

```
4320092                     //
4320093                     // Displaying this character, the cursor should go
4320094                     // back to the next line, at the first column.
4320095                     // So the column counter is reset and the line
4320096                     // counter is incremented.
4320097                     //
4320098                     line++;
4320099                     column = 1;
4320100                     break;
4320101                 case '\b':
4320102                     //
4320103                     // Displaying this character, the cursor should go
4320104                     // back one position, unless it is already at the
4320105                     // beginning.
4320106                     //
4320107                     if (column > 1)
4320108                       {
4320109                         column--;
4320110                       }
4320111                     break;
4320112                 default:
4320113                     //
4320114                     // Any other character must increase the column
4320115                     // counter.
4320116                     //
4320117                     column++;
4320118               }
4320119             //
4320120             // Display the character, even if it is a special one:
4320121             // it is responsibility of the screen device management
4320122             // to do something good with special characters.
4320123             //
4320124             putchar (c);
4320125             //
4320126             // If the column counter is gone beyond the screen columns,
4320127             // then adjust the column counter and increment the line
4320128             // counter.
4320129             //
4320130             if (column > MAX_COLUMNS)
4320131               {
4320132                 column -= MAX_COLUMNS;
4320133                 line++;
4320134               }
```

```
4320135           //
4320136           // Check if there is space for scrolling.
4320137           //
4320138           if (line < MAX_LINES)
4320139             {
4320140               continue;
4320141             }
4320142           //
4320143           // Here, displayed lines are MAX_LINES.
4320144           //
4320145           if (column > 1)
4320146             {
4320147               //
4320148               // Something was printed at the current line: must
4320149               // do a new line.
4320150               //
4320151               putchar ('\n');
4320152             }
4320153           //
4320154           // Show the more prompt.
4320155           //
4320156           printf ("--More--");
4320157           fflush (stdout);
4320158           //
4320159           // Read a character from standard input.
4320160           //
4320161           c = getchar ();
4320162           //
4320163           // Consider command 'q', but any other character
4320164           // can be introduced, to let show the next page.
4320165           //
4320166           switch (c)
4320167             {
4320168               case 'Q':
4320169               case 'q':
4320170                   //
4320171                   // Quit. But must erase the '--More--' prompt.
4320172                   //
4320173                   printf ("\b \b\b \b\b \b\b \b\b \b");
4320174                   printf ("\b \b\b \b\b \b\b \b");
4320175                   fclose (fp);
4320176                   return (0);
4320177             }
```

```
4320178           //
4320179           // Backspace to overwrite `--More--' and the character
4320180           // pressed.
4320181           //
4320182           printf ("\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b");
4320183           //
4320184           // Reset line/column counters.
4320185           //
4320186           column = 1;
4320187           line = 1;
4320188         }
4320189       //
4320190       // Close the file pointer if it is still open.
4320191       //
4320192       if (fp != NULL)
4320193         {
4320194           fclose (fp);
4320195         }
4320196       //
4320197       return (0);
4320198   }
4320199   //---------------------------------------------------------------------
4320200   static void
4320201   usage (void)
4320202   {
4320203       fprintf (stderr, "Usage: man [SECTION] NAME\n");
4320204   }
4320205   //---------------------------------------------------------------------
4320206   FILE *
4320207   open_man_page (int section, char *name)
4320208   {
4320209       FILE       *fp;
4320210       char        path[PATH_MAX];
4320211       struct stat file_status;
4320212       //
4320213       //
4320214       //
4320215       if (section > 0)
4320216         {
4320217           build_path_name (section, name, path);
4320218           //
4320219           // Check if file exists.
4320220           //
```

```
4320221           if (stat (path, &file_status) != 0)
4320222             {
4320223               fprintf (stderr, "Man page %s(%i) does not exist!\n",
4320224                        name, section);
4320225               return (NULL);
4320226             }
4320227         }
4320228       else
4320229         {
4320230           //
4320231           // Must try a section.
4320232           //
4320233           for (section = 1; section < 9; section++)
4320234             {
4320235               build_path_name (section, name, path);
4320236               //
4320237               // Check if file exists.
4320238               //
4320239               if (stat (path, &file_status) == 0)
4320240                 {
4320241                   //
4320242                   // Found.
4320243                   //
4320244                   break;
4320245                 }
4320246             }
4320247         }
4320248       //
4320249       // Check if a file was found.
4320250       //
4320251       if (section < 9)
4320252         {
4320253           fp = fopen (path, "r");
4320254           //
4320255           if (fp == NULL)
4320256             {
4320257               //
4320258               // Error opening file.
4320259               //
4320260               perror (path);
4320261               return (NULL);
4320262             }
4320263           else
```

```
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/os16.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/os16.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;
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 exists.
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);
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               //
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                   {
4340163                     case 'N':
4340164                     case 'n':
4340165                         //
4340166                         // Go to the next file, if any.
4340167                         //
4340168                         fclose (fp);
4340169                         fp = NULL;
4340170                         loop = 0;
4340171                         break;
4340172                     case 'Q':
4340173                     case 'q':
4340174                         //
4340175                         // Quit. But must erase the `--More--' prompt.
4340176                         //
4340177                         printf ("\b \b\b \b\b \b\b \b\b \b");
4340178                         printf ("\b \b\b \b\b \b\b \b");
4340179                         fclose (fp);
4340180                         return (0);
4340181                   }
4340182                 //
4340183                 // Backspace to overwrite `--More--' and the character
4340184                 // pressed.
4340185                 //
4340186                 printf ("\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b\b \b");
```

```
4340187               //
4340188               // Reset line/column counters.
4340189               //
4340190               column = 1;
4340191               line = 1;
4340192             }
4340193           //
4340194           // Close the file pointer if it is still open.
4340195           //
4340196           if (fp != NULL)
4340197             {
4340198               fclose (fp);
4340199             }
4340200         }
4340201       //
4340202       return (0);
4340203   }
4340204   //---------------------------------------------------------------------
4340205   static void
4340206   usage (void)
4340207   {
4340208       fprintf (stderr, "Usage: more FILE...\n");
4340209   }
```

applic/mount.c

«

Si veda la sezione u0.4.

```
4350001   #include <unistd.h>
4350002   #include <stdlib.h>
4350003   #include <sys/stat.h>
4350004   #include <sys/types.h>
4350005   #include <fcntl.h>
4350006   #include <errno.h>
4350007   #include <signal.h>
4350008   #include <stdio.h>
4350009   #include <sys/wait.h>
4350010   #include <stdio.h>
4350011   #include <string.h>
4350012   #include <limits.h>
4350013   #include <sys/os16.h>
4350014   //---------------------------------------------------------------------
```

```
4350015 static void usage (void);
4350016 //---------------------------------------------------------------------
4350017 int
4350018 main (int argc, char *argv[], char *envp[])
4350019 {
4350020     int     options;
4350021     int     status;
4350022     //
4350023     //
4350024     //
4350025     if (argc < 3 || argc > 4)
4350026       {
4350027         usage ();
4350028         return (1);
4350029       }
4350030     //
4350031     // Set options.
4350032     //
4350033     if (argc == 4)
4350034       {
4350035         if      (strcmp (argv[3], "rw") == 0)
4350036           {
4350037             options = MOUNT_DEFAULT;
4350038           }
4350039         else if (strcmp (argv[3], "ro") == 0)
4350040           {
4350041             options = MOUNT_RO;
4350042           }
4350043         else
4350044           {
4350045             printf ("Invalid mount option: only \"ro\" or \"rw\" "
4350046                     "are allowed\n");
4350047             return (2);
4350048           }
4350049       }
4350050     else
4350051       {
4350052         options = MOUNT_DEFAULT;
4350053       }
4350054     //
4350055     // System call.
4350056     //
4350057     status = mount (argv[1], argv[2], options);
```

```
4350058     if (status != 0)
4350059       {
4350060         perror (NULL);
4350061         return (2);
4350062       }
4350063     //
4350064     return (0);
4350065 }
4350066 //---------------------------------------------------------------------
4350067 static void
4350068 usage (void)
4350069 {
4350070     fprintf (stderr, "Usage: mount DEVICE MOUNT_POINT "
4350071                      "[MOUNT_OPTIONS]\n");
4350072 }
```

applic/ps.c

«

Si veda la sezione u0.17.

```
4360001 #include <kernel/proc.h>
4360002 #include <unistd.h>
4360003 #include <stdio.h>
4360004 #include <fcntl.h>
4360005 #include <unistd.h>
4360006 #include <stdlib.h>
4360007 //---------------------------------------------------------------------
4360008 void
4360009 print_proc_head (void)
4360010 {
4360011     printf (
4360012 "pp  p pg                                                    \n"
4360013 "id id rp  tty  uid euid suid usage s iaddr isiz daddr dsiz sp   name\n"
4360014            );
4360015 }
4360016 //---------------------------------------------------------------------
4360017 void
4360018 print_proc_pid (proc_t *ps, pid_t pid)
4360019 {
4360020     char  stat;
4360021     switch (ps->status)
4360022       {
```

```
4360023           case PROC_EMPTY   : stat = '-'; break;
4360024           case PROC_CREATED : stat = 'c'; break;
4360025           case PROC_READY   : stat = 'r'; break;
4360026           case PROC_RUNNING : stat = 'R'; break;
4360027           case PROC_SLEEPING: stat = 's'; break;
4360028           case PROC_ZOMBIE  : stat = 'z'; break;
4360029           default           : stat = '?'; break;
4360030         }
4360031
4360032       printf ("%2i %2i %2i %04x %4i %4i %4i %02i.%02i %c %05lx %04x ",
4360033                 (unsigned int) ps->ppid,
4360034                 (unsigned int) pid,
4360035                 (unsigned int) ps->pgrp,
4360036                 (unsigned int) ps->device_tty,
4360037                 (unsigned int) ps->uid,
4360038                 (unsigned int) ps->euid,
4360039                 (unsigned int) ps->suid,
4360040                 (unsigned int) ((ps->usage / CLOCKS_PER_SEC) / 60),
4360041                 (unsigned int) ((ps->usage / CLOCKS_PER_SEC) % 60),
4360042                 stat,
4360043                 (unsigned long int) ps->address_i,
4360044                 (unsigned int) ps->size_i);
4360045
4360046       printf ("%05lx %04x %04x %s",
4360047                 (unsigned long int) ps->address_d,
4360048                 (unsigned int) ps->size_d,
4360049                 (unsigned int) ps->sp,
4360050                                ps->name);
4360051
4360052       printf ("\n");
4360053   }
4360054   //-------------------------------------------------------------------------
4360055   int
4360056   main (void)
4360057   {
4360058       pid_t   pid;
4360059       proc_t *ps;
4360060       int     fd;
4360061       ssize_t size_read;
4360062       char    buffer[sizeof (proc_t)];
4360063
4360064       fd = open ("/dev/kmem_ps", O_RDONLY);
4360065       if (fd < 0)
```

```
4360066         {
4360067           perror ("ps: cannot open \"/dev/kmem_ps\"");
4360068           exit (0);
4360069         }
4360070
4360071       print_proc_head ();
4360072       for (pid = 0; pid < PROCESS_MAX; pid++)
4360073         {
4360074           lseek (fd, (off_t) pid, SEEK_SET);
4360075           size_read = read (fd, buffer, sizeof (proc_t));
4360076           if (size_read < sizeof (proc_t))
4360077             {
4360078               printf ("ps: cannot read \"/dev/kmem_ps\" pid %i", pid);
4360079               perror (NULL);
4360080               continue;
4360081             }
4360082           ps = (proc_t *) buffer;
4360083           if (ps->status > 0)
4360084             {
4360085               ps->name[PATH_MAX-1] = 0; // Terminated string.
4360086               print_proc_pid (ps, pid);
4360087             }
4360088         }
4360089
4360090       close (fd);
4360091       return (0);
4360092   }
```

applic/rm.c

«

Si veda la sezione u0.18.

```
4370001   #include <fcntl.h>
4370002   #include <sys/stat.h>
4370003   #include <stddef.h>
4370004   #include <unistd.h>
4370005   #include <errno.h>
4370006   //---------------------------------------------------------------------
4370007   static void usage (void);
4370008   //---------------------------------------------------------------------
4370009   int
4370010   main (int argc, char *argv[], char *envp[])
```

```
4370011 {
4370012     int         a;              // Argument index.
4370013     int         status;
4370014     struct stat file_status;
4370015     //
4370016     // No options are known, but at least an argument must be given.
4370017     //
4370018     if (argc < 2)
4370019       {
4370020         usage ();
4370021         return (1);
4370022       }
4370023     //
4370024     // Scan arguments.
4370025     //
4370026     for(a = 1; a < argc; a++)
4370027       {
4370028         //
4370029         // Verify if the file exists.
4370030         //
4370031         if (stat(argv[a], &file_status) != 0)
4370032           {
4370033             fprintf (stderr, "File \"%s\" does not exist!\n",
4370034                      argv[a]);
4370035             continue;
4370036           }
4370037         //
4370038         // File exists: check the file type.
4370039         //
4370040         if (S_ISDIR (file_status.st_mode))
4370041           {
4370042             fprintf (stderr, "Cannot remove directory \"%s\"!\n",
4370043                      argv[a]);
4370044             continue;
4370045           }
4370046         //
4370047         // Can remove it.
4370048         //
4370049         status = unlink (argv[a]);
4370050         if (status != 0)
4370051           {
4370052             perror (NULL);
4370053             return (2);
```

```
4370054             }
4370055           }
4370056         return (0);
4370057     }
4370058     //---------------------------------------------------------------------
4370059     static void
4370060     usage (void)
4370061     {
4370062         fprintf (stderr, "Usage: rm FILE...\n");
4370063     }
```

applic/shell.c

«

Si veda la sezione u0.19.

```
4380001     #include <unistd.h>
4380002     #include <stdlib.h>
4380003     #include <sys/stat.h>
4380004     #include <sys/types.h>
4380005     #include <fcntl.h>
4380006     #include <errno.h>
4380007     #include <unistd.h>
4380008     #include <signal.h>
4380009     #include <stdio.h>
4380010     #include <sys/wait.h>
4380011     #include <stdio.h>
4380012     #include <string.h>
4380013     #include <limits.h>
4380014     #include <sys/os16.h>
4380015     //---------------------------------------------------------------------
4380016     #define PROMPT_SIZE     16
4380017     //---------------------------------------------------------------------
4380018     static void sh_cd       (int argc, char *argv[]);
4380019     static void sh_pwd      (int argc, char *argv[]);
4380020     static void sh_umask    (int argc, char *argv[]);
4380021     //---------------------------------------------------------------------
4380022     int
4380023     main (int argc, char *argv[], char *envp[])
4380024     {
4380025       char    buffer_cmd[ARG_MAX/2];
4380026       char   *argv_cmd[ARG_MAX/16];
4380027       char    prompt[PROMPT_SIZE];
```

```
4380028     uid_t   uid;
4380029     int     argc_cmd;
4380030     pid_t   pid_cmd;
4380031     pid_t   pid_dead;
4380032     int     status;
4380033     //
4380034     //
4380035     //
4380036     uid = geteuid ();
4380037     //
4380038     // Load processes, reading the keyboard.
4380039     //
4380040     while (1)
4380041       {
4380042         if (uid == 0)
4380043           {
4380044             strncpy (prompt, "# ", PROMPT_SIZE);
4380045           }
4380046         else
4380047           {
4380048             strncpy (prompt, "$ ", PROMPT_SIZE);
4380049           }
4380050         //
4380051         input_line (buffer_cmd, prompt, (ARG_MAX/2), INPUT_LINE_ECHO);
4380052         //
4380053         // Clear 'argv_cmd[]';
4380054         //
4380055         for (argc_cmd = 0; argc_cmd < (ARG_MAX/16); argc_cmd++)
4380056           {
4380057             argv_cmd[argc_cmd] = NULL;
4380058           }
4380059         //
4380060         // Initialize the command scan.
4380061         //
4380062         argv_cmd[0] = strtok (buffer_cmd, " \t");
4380063         //
4380064         // Verify: if the input is not valid, loop again.
4380065         //
4380066         if (argv_cmd[0] == NULL)
4380067           {
4380068             continue;
4380069           }
4380070         //
```

```
4380071         // Find the arguments.
4380072         //
4380073         for (argc_cmd = 1;
4380074              argc_cmd < ((ARG_MAX/16)-1) && argv_cmd[argc_cmd-1] != NULL;
4380075              argc_cmd++)
4380076           {
4380077             argv_cmd[argc_cmd] = strtok (NULL, " \t");
4380078           }
4380079         //
4380080         // If there are too many arguments, show a message and continue.
4380081         //
4380082         if (argv_cmd[argc_cmd-1] != NULL)
4380083           {
4380084             errset (E2BIG);             // Argument list too long.
4380085             perror (NULL);
4380086             continue;
4380087           }
4380088         //
4380089         // Correct the value for 'argc_cmd', because actually
4380090         // it counts also the NULL element.
4380091         //
4380092         argc_cmd--;
4380093         //
4380094         // Verify if it is an internal command.
4380095         //
4380096         if      (strcmp (argv_cmd[0], "exit") == 0)
4380097           {
4380098             return (0);
4380099           }
4380100         else if (strcmp (argv_cmd[0], "cd") == 0)
4380101           {
4380102             sh_cd (argc_cmd, argv_cmd);
4380103             continue;
4380104           }
4380105         else if (strcmp (argv_cmd[0], "pwd") == 0)
4380106           {
4380107             sh_pwd (argc_cmd, argv_cmd);
4380108             continue;
4380109           }
4380110         else if (strcmp (argv_cmd[0], "umask") == 0)
4380111           {
4380112             sh_umask (argc_cmd, argv_cmd);
4380113             continue;
```

```
4380114           }
4380115         //
4380116         // It should be a program to run.
4380117         //
4380118         pid_cmd = fork ();
4380119         if (pid_cmd == -1)
4380120           {
4380121             printf ("%s: cannot run command", argv[0]);
4380122             perror (NULL);
4380123           }
4380124         else if (pid_cmd == 0)
4380125           {
4380126             execvp (argv_cmd[0], argv_cmd);
4380127             perror (NULL);
4380128             exit (0);
4380129           }
4380130         while (1)
4380131           {
4380132             pid_dead = wait (&status);
4380133             if (pid_dead == pid_cmd)
4380134               {
4380135                 break;
4380136               }
4380137           }
4380138         printf ("pid %i terminated with status %i.\n",
4380139                 (int) pid_dead, status);
4380140       }
4380141   }
4380142   //---------------------------------------------------------------------
4380143   static void
4380144   sh_cd (int argc, char *argv[])
4380145   {
4380146     int status;
4380147     //
4380148     if (argc != 2)
4380149       {
4380150         errset (EINVAL);                // Invalid argument.
4380151         perror (NULL);
4380152         return;
4380153       }
4380154     //
4380155     status = chdir (argv[1]);
4380156     if (status != 0)
```

```
4380157     {
4380158       perror (NULL);
4380159     }
4380160   return;
4380161 }
4380162 //-----------------------------------------------------------------------
4380163 static void
4380164 sh_pwd (int argc, char *argv[])
4380165 {
4380166   char  path[PATH_MAX];
4380167   void *pstatus;
4380168   //
4380169   if (argc != 1)
4380170     {
4380171       errset (EINVAL);                // Invalid argument.
4380172       perror (NULL);
4380173       return;
4380174     }
4380175   //
4380176   // Get the current directory.
4380177   //
4380178   pstatus = getcwd (path, (size_t) PATH_MAX);
4380179   if (pstatus == NULL)
4380180     {
4380181       perror (NULL);
4380182     }
4380183   else
4380184     {
4380185       printf ("%s\n", path);
4380186     }
4380187   return;
4380188 }
4380189 //-----------------------------------------------------------------------
4380190 static void
4380191 sh_umask (int argc, char *argv[])
4380192 {
4380193   sysmsg_uarea_t msg;
4380194   char          *m;           // Index inside the umask octal string.
4380195   int            mask;
4380196   int            digit;
4380197   //
4380198   if (argc > 2)
4380199     {
```

```
4380200         errset (EINVAL);                // Invalid argument.
4380201         perror (NULL);
4380202         return;
4380203       }
4380204     //
4380205     // If no argument is available, the umask is shown, with a direct
4380206     // system call.
4380207     //
4380208     if (argc == 1)
4380209       {
4380210         sys (SYS_UAREA, &msg, (sizeof msg));
4380211         printf ("%04o\n", msg.umask);
4380212         return;
4380213       }
4380214     //
4380215     // Get the mask: must be the first argument.
4380216     //
4380217     for (mask = 0, m = argv[1]; *m != 0; m++)
4380218       {
4380219         digit = (*m - '0');
4380220         if (digit < 0 || digit > 7)
4380221           {
4380222             errset (EINVAL);            // Invalid argument.
4380223             perror (NULL);
4380224             return;
4380225           }
4380226         mask = mask * 8 + digit;
4380227       }
4380228     //
4380229     // Set the umask and return.
4380230     //
4380231     umask (mask);
4380232     return;
4380233   }
```

applic/touch.c

«

Si veda la sezione u0.20.

```
4390001   #include <fcntl.h>
4390002   #include <sys/stat.h>
4390003   #include <utime.h>
```

```
#include <stddef.h>
#include <unistd.h>
#include <errno.h>
//---------------------------------------------------------------------
static void usage (void);
//---------------------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    int         a;              // Argument index.
    int         status;
    struct stat file_status;
    //
    // No options are known, but at least an argument must be given.
    //
    if (argc < 2)
      {
        usage ();
        return (1);
      }
    //
    // Scan arguments.
    //
    for(a = 1; a < argc; a++)
      {
        //
        // Verify if the file exists, through the return value of
        // 'stat()'. No other checks are made.
        //
        if (stat(argv[a], &file_status) == 0)
         {
            //
            // File exists: should be updated the times.
            //
            status = utime (argv[a], NULL);
            if (status != 0)
              {
                perror (NULL);
                return (2);
              }
         }
        else
          {
```

```
4390047                 //
4390048                 // File does not exist: should be created.
4390049                 //
4390050                 status = open (argv[a], O_WRONLY|O_CREAT|O_TRUNC, 0666);
4390051                 //
4390052                 if (status >= 0)
4390053                   {
4390054                     //
4390055                     // Here, the variable 'status' is the file
4390056                     // descriptor to be closed.
4390057                     //
4390058                     status = close (status);
4390059                     if (status != 0)
4390060                       {
4390061                         perror (NULL);
4390062                         return (3);
4390063                       }
4390064                   }
4390065                 else
4390066                   {
4390067                     perror (NULL);
4390068                     return (4);
4390069                   }
4390070               }
4390071           }
4390072       return (0);
4390073   }
4390074   //------------------------------------------------------------------------
4390075   static void
4390076   usage (void)
4390077   {
4390078       fprintf (stderr, "Usage: touch FILE...\n");
4390079   }
```

applic/tty.c

«

Si veda la sezione u0.21.

```
4400001   #include <fcntl.h>
4400002   #include <sys/stat.h>
4400003   #include <utime.h>
4400004   #include <stddef.h>
```

```
4400005 #include <unistd.h>
4400006 #include <errno.h>
4400007 #include <sys/os16.h>
4400008 #include <sys/types.h>
4400009 //-----------------------------------------------------------------------
4400010 static void     usage (void);
4400011 //-----------------------------------------------------------------------
4400012 int
4400013 main (int argc, char *argv[], char *envp[])
4400014 {
4400015   int             dev_minor;
4400016   struct stat     file_status;
4400017   //
4400018   // No options and no arguments.
4400019   //
4400020   if (argc > 1)
4400021     {
4400022       usage ();
4400023       return (1);
4400024     }
4400025   //
4400026   // Verify the standard input.
4400027   //
4400028   if (fstat (STDIN_FILENO, &file_status) == 0)
4400029     {
4400030       if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
4400031         {
4400032           dev_minor = minor (file_status.st_rdev);
4400033           //
4400034           // If minor is equal to 0xFF, it is '/dev/console'
4400035           // that is not a controlling terminal, but just
4400036           // a reference for the current virtual console.
4400037           //
4400038           if (dev_minor < 0xFF)
4400039             {
4400040               printf ("/dev/console%i\n", dev_minor);
4400041             }
4400042         }
4400043     }
4400044   else
4400045     {
4400046       perror ("Cannot get standard input file status");
4400047       return (2);
```

```
4400048       }
4400049     //
4400050     return (0);
4400051   }
4400052
4400053   //---------------------------------------------------------------------
4400054   static void
4400055   usage (void)
4400056   {
4400057     fprintf (stderr, "Usage: tty\n");
4400058   }
```

applic/umount.c

«

Si veda la sezione u0.4.

```
4410001   #include <unistd.h>
4410002   #include <stdlib.h>
4410003   #include <sys/stat.h>
4410004   #include <sys/types.h>
4410005   #include <fcntl.h>
4410006   #include <errno.h>
4410007   #include <signal.h>
4410008   #include <stdio.h>
4410009   #include <sys/wait.h>
4410010   #include <stdio.h>
4410011   #include <string.h>
4410012   #include <limits.h>
4410013   #include <sys/os16.h>
4410014   //---------------------------------------------------------------------
4410015   static void usage (void);
4410016   //---------------------------------------------------------------------
4410017   int
4410018   main (int argc, char *argv[], char *envp[])
4410019   {
4410020       int     status;
4410021       //
4410022       // One argument is mandatory.
4410023       //
4410024       if (argc != 2)
4410025         {
4410026           usage ();
```

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