Sorgenti delle applicazioni

96.1 os32: directory «applic/»

«

96.1.1 applic/MAKEDEV.c

«

Si veda la sezione 92.6.

```
5660001 #include <unistd.h>
5660002 #include <stdlib.h>
5660003 #include <sys/stat.h>
5660004 #include <fcntl.h>
5660005 #include <kernel/dev.h>
5660006 #include <stdio.h>
5660007 //-------------------------------------------------------
5660008 int
5660009 main (void)
5660010 {
5660011   int status;
5660012   //
5660013   status =
5660014     mknod ("mem", (mode_t) (S_IFCHR | 0444),
5660015            (dev_t) DEV_MEM);
5660016   if (status)
5660017     perror (NULL);
5660018   status =
5660019     mknod ("null", (mode_t) (S_IFCHR | 0666),
5660020            (dev_t) DEV_NULL);
```

```
5660021     if (status)
5660022       perror (NULL);
5660023     status =
5660024       mknod ("port", (mode_t) (S_IFCHR | 0644),
5660025              (dev_t) DEV_PORT);
5660026     if (status)
5660027       perror (NULL);
5660028     status =
5660029       mknod ("zero", (mode_t) (S_IFCHR | 0666),
5660030              (dev_t) DEV_ZERO);
5660031     if (status)
5660032       perror (NULL);
5660033     status =
5660034       mknod ("tty", (mode_t) (S_IFCHR | 0666),
5660035              (dev_t) DEV_TTY);
5660036     if (status)
5660037       perror (NULL);
5660038     status = mknod ("kmem_ps", (mode_t) (S_IFCHR | 0444),
5660039                     (dev_t) DEV_KMEM_PS);
5660040     if (status)
5660041       perror (NULL);
5660042     status =
5660043       mknod ("kmem_mmp", (mode_t) (S_IFCHR | 0444),
5660044              (dev_t) DEV_KMEM_MMP);
5660045     if (status)
5660046       perror (NULL);
5660047     status = mknod ("kmem_sb", (mode_t) (S_IFCHR | 0444),
5660048                     (dev_t) DEV_KMEM_SB);
5660049     if (status)
5660050       perror (NULL);
5660051     status =
5660052       mknod ("kmem_inode", (mode_t) (S_IFCHR | 0444),
5660053              (dev_t) DEV_KMEM_INODE);
5660054     if (status)
5660055       perror (NULL);
5660056     status =
5660057       mknod ("kmem_file", (mode_t) (S_IFCHR | 0444),
```

```
5660058             (dev_t) DEV_KMEM_FILE);
5660059     if (status)
5660060       perror (NULL);
5660061     status = mknod ("console", (mode_t) (S_IFCHR | 0644),
5660062                     (dev_t) DEV_CONSOLE);
5660063     if (status)
5660064       perror (NULL);
5660065     status =
5660066       mknod ("console0", (mode_t) (S_IFCHR | 0644),
5660067              (dev_t) DEV_CONSOLE0);
5660068     if (status)
5660069       perror (NULL);
5660070     status =
5660071       mknod ("console1", (mode_t) (S_IFCHR | 0644),
5660072              (dev_t) DEV_CONSOLE1);
5660073     if (status)
5660074       perror (NULL);
5660075     status =
5660076       mknod ("console2", (mode_t) (S_IFCHR | 0644),
5660077              (dev_t) DEV_CONSOLE2);
5660078     if (status)
5660079       perror (NULL);
5660080     status =
5660081       mknod ("console3", (mode_t) (S_IFCHR | 0644),
5660082              (dev_t) DEV_CONSOLE3);
5660083     if (status)
5660084       perror (NULL);
5660085
5660086     status =
5660087       mknod ("dm00", (mode_t) (S_IFBLK | 0644),
5660088              (dev_t) DEV_DM00);
5660089     if (status)
5660090       perror (NULL);
5660091     status =
5660092       mknod ("dm01", (mode_t) (S_IFBLK | 0644),
5660093              (dev_t) DEV_DM01);
5660094     if (status)
```

```
5660095       perror (NULL);
5660096     status =
5660097       mknod ("dm02", (mode_t) (S_IFBLK | 0644),
5660098              (dev_t) DEV_DM02);
5660099     if (status)
5660100       perror (NULL);
5660101     status =
5660102       mknod ("dm03", (mode_t) (S_IFBLK | 0644),
5660103              (dev_t) DEV_DM03);
5660104     if (status)
5660105       perror (NULL);
5660106     status =
5660107       mknod ("dm04", (mode_t) (S_IFBLK | 0644),
5660108              (dev_t) DEV_DM04);
5660109     if (status)
5660110       perror (NULL);
5660111     status =
5660112       mknod ("dm10", (mode_t) (S_IFBLK | 0644),
5660113              (dev_t) DEV_DM10);
5660114     if (status)
5660115       perror (NULL);
5660116     status =
5660117       mknod ("dm11", (mode_t) (S_IFBLK | 0644),
5660118              (dev_t) DEV_DM11);
5660119     if (status)
5660120       perror (NULL);
5660121     status =
5660122       mknod ("dm12", (mode_t) (S_IFBLK | 0644),
5660123              (dev_t) DEV_DM12);
5660124     if (status)
5660125       perror (NULL);
5660126     status =
5660127       mknod ("dm13", (mode_t) (S_IFBLK | 0644),
5660128              (dev_t) DEV_DM13);
5660129     if (status)
5660130       perror (NULL);
5660131     status =
```

```
5660132       mknod ("dm14", (mode_t) (S_IFBLK | 0644),
5660133              (dev_t) DEV_DM14);
5660134     if (status)
5660135       perror (NULL);
5660136     status =
5660137       mknod ("dm20", (mode_t) (S_IFBLK | 0644),
5660138              (dev_t) DEV_DM20);
5660139     if (status)
5660140       perror (NULL);
5660141     status =
5660142       mknod ("dm21", (mode_t) (S_IFBLK | 0644),
5660143              (dev_t) DEV_DM21);
5660144     if (status)
5660145       perror (NULL);
5660146     status =
5660147       mknod ("dm22", (mode_t) (S_IFBLK | 0644),
5660148              (dev_t) DEV_DM22);
5660149     if (status)
5660150       perror (NULL);
5660151     status =
5660152       mknod ("dm23", (mode_t) (S_IFBLK | 0644),
5660153              (dev_t) DEV_DM23);
5660154     if (status)
5660155       perror (NULL);
5660156     status =
5660157       mknod ("dm24", (mode_t) (S_IFBLK | 0644),
5660158              (dev_t) DEV_DM24);
5660159     if (status)
5660160       perror (NULL);
5660161     status =
5660162       mknod ("dm30", (mode_t) (S_IFBLK | 0644),
5660163              (dev_t) DEV_DM30);
5660164     if (status)
5660165       perror (NULL);
5660166     status =
5660167       mknod ("dm31", (mode_t) (S_IFBLK | 0644),
5660168              (dev_t) DEV_DM31);
```

```
5660169     if (status)
5660170       perror (NULL);
5660171     status =
5660172       mknod ("dm32", (mode_t) (S_IFBLK | 0644),
5660173              (dev_t) DEV_DM32);
5660174     if (status)
5660175       perror (NULL);
5660176     status =
5660177       mknod ("dm33", (mode_t) (S_IFBLK | 0644),
5660178              (dev_t) DEV_DM33);
5660179     if (status)
5660180       perror (NULL);
5660181     status =
5660182       mknod ("dm34", (mode_t) (S_IFBLK | 0644),
5660183              (dev_t) DEV_DM34);
5660184     if (status)
5660185       perror (NULL);
5660186     //
5660187     return (0);
5660188   }
```

96.1.2 applic/aaa.c

«

Si veda la sezione 86.1.

```
5670001   #include <unistd.h>
5670002   #include <stdio.h>
5670003   //-------------------------------------------------------
5670004   int
5670005   main (void)
5670006   {
5670007     unsigned int count;
5670008     for (count = 0; count < 60; count++)
5670009       {
5670010         printf ("a");
5670011         sleep (1);
5670012       }
```

```
5670013     return (8);
5670014   }
```

96.1.3 applic/allocated.c

Si veda la sezione 86.2.

```
5680001   #include <sys/os32.h>
5680002   #include <kernel/memory.h>
5680003   #include <unistd.h>
5680004   #include <stdio.h>
5680005   #include <fcntl.h>
5680006   #include <unistd.h>
5680007   #include <stdlib.h>
5680008   //-------------------------------------------------------
5680009   uint32_t mb_table[MEM_MAX_BLOCKS / 32]; // Memory
5680010                                           // blocks map.
5680011   unsigned int mb_max = MEM_MAX_BLOCKS;   // Memory
5680012                                           // blocks max.
5680013   //-------------------------------------------------------
5680014   int
5680015   main (int argc, char *argv[], char *envp[])
5680016   {
5680017     unsigned int block;
5680018     unsigned int blocks = MEM_MAX_BLOCKS;
5680019     int i;
5680020     int j;
5680021     uint32_t mask;
5680022     unsigned int start = 0;
5680023     unsigned int stop = 0;
5680024     unsigned int status = 0;
5680025     int fd;
5680026     ssize_t size_read;
5680027     char *buffer = (char *) mb_table;
5680028     //
5680029     fd = open ("/dev/kmem_map", O_RDONLY);
5680030     if (fd < 0)
```

```
5680031       {
5680032         printf ("[%s] Cannot open \"/dev/kmem_map\" ",
5680033                 argv[0]);
5680034         perror (NULL);
5680035         return (0);
5680036       }
5680037     //
5680038     lseek (fd, (off_t) 0, SEEK_SET);
5680039     for (i = 0; i < (MEM_MAX_BLOCKS / 8); i += size_read)
5680040       {
5680041         size_read = read (fd, &buffer[i], BUFSIZ);
5680042         if (size_read < 0)
5680043           {
5680044             printf
5680045               ("[%s] Cannot read "
5680046                "\"/dev/kmem_map\" %i %i  ",
5680047                argv[0], size_read, sizeof (mb_table));
5680048             perror (NULL);
5680049             return (0);
5680050           }
5680051       }
5680052     //
5680053     printf ("Hex mem map, blocks of %x:", MEM_BLOCK_SIZE);
5680054     //
5680055     for (block = 0; block < blocks; block++)
5680056       {
5680057         i = block / 32;
5680058         j = block % 32;
5680059         mask = 0x80000000 >> j;
5680060         if (mb_table[i] & mask)
5680061           {
5680062             //
5680063             // Allocated block
5680064             //
5680065             if (status == 0)
5680066               {
5680067                 status = 1;
```

```
5680068                 start = block;
5680069               }
5680070           }
5680071         else
5680072           {
5680073             //
5680074             // Not allocated block.
5680075             //
5680076             if (status == 1)
5680077               {
5680078                 status = 0;
5680079                 stop = block;
5680080               }
5680081           }
5680082         //
5680083         //
5680084         //
5680085         if (stop > 0)
5680086           {
5680087             printf (" %x-%x", start, stop);
5680088             start = 0;
5680089             stop = 0;
5680090           }
5680091       }
5680092     printf ("\n");
5680093
5680094     return (0);
5680095   }
```

96.1.4 applic/arp.c

Si veda la sezione 92.1. «

```
5690001   #include <sys/os32.h>
5690002   #include <kernel/net/arp.h>
5690003   #include <unistd.h>
5690004   #include <stdio.h>
```

```
5690005 #include <fcntl.h>
5690006 #include <unistd.h>
5690007 #include <stdlib.h>
5690008 #include <time.h>
5690009 //------------------------------------------------------------
5690010 int
5690011 main (int argc, char *argv[], char *envp[])
5690012 {
5690013   int fd;
5690014   ssize_t size_read;
5690015   char buffer[sizeof (arp_t)];
5690016   int a;
5690017   arp_t *arp_table_item;
5690018   //
5690019   // All options are ignored.
5690020   //
5690021   // Open '/dev/kmem_arp', to get the ARP table.
5690022   //
5690023   fd = open ("/dev/kmem_arp", O_RDONLY);
5690024   if (fd < 0)
5690025     {
5690026       printf ("[%s] Cannot open \"/dev/kmem_arp\" ",
5690027               argv[0]);
5690028       perror (NULL);
5690029       exit (0);
5690030     }
5690031   //
5690032   // Scan ARP items and then print body.
5690033   //
5690034   for (a = 0; a < ARP_MAX_ITEMS; a++)
5690035     {
5690036       lseek (fd, (off_t) a, SEEK_SET);
5690037       size_read = read (fd, buffer, sizeof (arp_t));
5690038       if (size_read < sizeof (arp_t))
5690039         {
5690040           printf
5690041             ("[%s] Cannot read "
```

```
5690042                 "\"/dev/kmem_arp\" item %i ", argv[0], a);
5690043             perror (NULL);
5690044             continue;
5690045           }
5690046         arp_table_item = (arp_t *) buffer;
5690047         if (arp_table_item->time > 0)
5690048           {
5690049             printf ("%i.%i.%i.%i  ",
5690050                     arp_table_item->ip >> 24 & 0x000000FF,
5690051                     arp_table_item->ip >> 16 & 0x000000FF,
5690052                     arp_table_item->ip >> 8 & 0x000000FF,
5690053                     arp_table_item->ip >> 0 & 0x000000FF);
5690054             //
5690055             printf ("%02x:%02x:%02x:%02x:%02x:%02x  ",
5690056                     arp_table_item->mac[0],
5690057                     arp_table_item->mac[1],
5690058                     arp_table_item->mac[2],
5690059                     arp_table_item->mac[3],
5690060                     arp_table_item->mac[4],
5690061                     arp_table_item->mac[5]);
5690062             //
5690063             printf ("%3us\n",
5690064                     (unsigned int) (time (NULL) -
5690065                                     arp_table_item->time));
5690066           }
5690067       }
5690068     close (fd);
5690069     return (0);
5690070   }
```

96.1.5 applic/bbb.c

Si veda la sezione 86.1.

```
5700001   #include <unistd.h>
5700002   #include <stdio.h>
5700003   #include <stdlib.h>
```

```
5700004 //-------------------------------------------------------
5700005 int
5700006 main (void)
5700007 {
5700008   unsigned int count;
5700009   for (count = 0; count < 30; count++)
5700010     {
5700011       printf ("b");
5700012       sleep (2);
5700013     }
5700014   exit (0);
5700015   return (0);
5700016 }
```

96.1.6 applic/cat.c

«

Si veda la sezione 86.4.

```
5710001 #include <fcntl.h>
5710002 #include <sys/stat.h>
5710003 #include <stddef.h>
5710004 #include <unistd.h>
5710005 #include <stdio.h>
5710006 #include <stdlib.h>
5710007 #include <errno.h>
5710008 //-------------------------------------------------------
5710009 static void cat_file_descriptor (int fd);
5710010 //-------------------------------------------------------
5710011 int
5710012 main (int argc, char *argv[], char *envp[])
5710013 {
5710014   int i;
5710015   int fd;
5710016   struct stat file_status;
5710017   //
5710018   // Check if the input comes from standard input.
5710019   //
```

```
5710020     if (argc < 2)
5710021       {
5710022         cat_file_descriptor (STDIN_FILENO);
5710023         exit (0);
5710024       }
5710025     //
5710026     // There is at least an argument: scan them.
5710027     //
5710028     for (i = 1; i < argc; i++)
5710029       {
5710030         //
5710031         // Verify if the file exists.
5710032         //
5710033         if (stat (argv[i], &file_status) != 0)
5710034           {
5710035             fprintf (stderr,
5710036                      "File \"%s\" does not exist!\n",
5710037                      argv[i]);
5710038             continue;
5710039           }
5710040         //
5710041         // File exists: check the file type.
5710042         //
5710043         if (S_ISDIR (file_status.st_mode))
5710044           {
5710045             fprintf (stderr, "Cannot \"cat\" "
5710046                      "\"%s\": it is a directory!\n", argv[i]);
5710047             continue;
5710048           }
5710049         //
5710050         // File exists and can be "cat"ed.
5710051         //
5710052         fd = open (argv[i], O_RDONLY);
5710053         if (fd >= 0)
5710054           {
5710055             cat_file_descriptor (fd);
5710056             close (fd);
```

```
5710057           }
5710058         else
5710059           {
5710060             perror (NULL);
5710061             exit (1);
5710062           }
5710063       }
5710064     return (0);
5710065   }
5710066
5710067   //-------------------------------------------------------
5710068   static void
5710069   cat_file_descriptor (int fd)
5710070   {
5710071     ssize_t count;
5710072     char buffer[BUFSIZ];
5710073
5710074     for (;;)
5710075       {
5710076         count = read (fd, buffer, (size_t) BUFSIZ);
5710077         if (count > 0)
5710078           {
5710079             write (STDOUT_FILENO, buffer, (size_t) count);
5710080           }
5710081         else
5710082           {
5710083             break;
5710084           }
5710085       }
5710086   }
```

96.1.7 applic/ccc.c

«

Si veda la sezione 86.1.

```
5720001   #include <unistd.h>
5720002   #include <stdlib.h>
```

```
5720003 #include <signal.h>
5720004 //-------------------------------------------------------
5720005 int
5720006 main (void)
5720007 {
5720008   pid_t pid;
5720009   // ------------------------------------------------
5720010   pid = fork ();
5720011   if (pid == 0)
5720012     {
5720013       setuid ((uid_t) 10);
5720014       execve ("/bin/aaa", NULL, NULL);
5720015       exit (0);
5720016     }
5720017   // ------------------------------------------------
5720018   pid = fork ();
5720019   if (pid == 0)
5720020     {
5720021       setuid ((uid_t) 11);
5720022       execve ("/bin/bbb", NULL, NULL);
5720023       exit (0);
5720024     }
5720025   // ------------------------------------------------
5720026   while (1)
5720027     {
5720028       ; // Just loop, to consume CPU time: it must be
5720029       // killed manually.
5720030     }
5720031   return (0);
5720032 }
```

96.1.8 applic/chgrp.c

Si veda la sezione 86.6.

```
5730001 #include <unistd.h>
5730002 #include <stdlib.h>
```

```
5730003 #include <sys/stat.h>
5730004 #include <sys/types.h>
5730005 #include <fcntl.h>
5730006 #include <errno.h>
5730007 #include <stdio.h>
5730008 #include <ctype.h>
5730009 #include <grp.h>
5730010 //-------------------------------------------------------------
5730011 static void usage (void);
5730012 //-------------------------------------------------------------
5730013 int
5730014 main (int argc, char *argv[], char *envp[])
5730015 {
5730016   char *group;
5730017   int gid;
5730018   struct group *grs;
5730019   struct stat file_status;
5730020   int a;          // Argument index.
5730021   int status;
5730022   //
5730023   //
5730024   //
5730025   if (argc < 3)
5730026     {
5730027       usage ();
5730028       return (1);
5730029     }
5730030   //
5730031   // Get group id number.
5730032   //
5730033   group = argv[1];
5730034   if (isdigit (*group))
5730035     {
5730036       gid = atoi (group);
5730037     }
5730038   else
5730039     {
```

```
5730040         grs = getgrnam (group);
5730041         if (grs == NULL)
5730042           {
5730043             fprintf (stderr, "Unknown group \"%s\"!\n",
5730044                      group);
5730045             return (2);
5730046           }
5730047         gid = grs->gr_gid;
5730048       }
5730049     //
5730050     // Now we have the group id. Start scanning file
5730051     // names.
5730052     //
5730053     for (a = 2; a < argc; a++)
5730054       {
5730055         //
5730056         // Verify if the file exists, through the return
5730057         // value of
5730058         // 'stat()'. No other checks are made.
5730059         //
5730060         if (stat (argv[a], &file_status) == 0)
5730061           {
5730062             //
5730063             // Try to change ownership.
5730064             //
5730065             status = chown (argv[a], file_status.st_uid, gid);
5730066             if (status != 0)
5730067               {
5730068                 perror (NULL);
5730069                 return (3);
5730070               }
5730071           }
5730072         else
5730073           {
5730074             fprintf (stderr,
5730075                      "File \"%s\" does not exist!\n",
5730076                      argv[a]);
```

```
5730077             continue;
5730078           }
5730079       }
5730080     //
5730081     // All done.
5730082     //
5730083     return (0);
5730084   }
5730085
5730086   //-----------------------------------------------------
5730087   static void
5730088   usage (void)
5730089   {
5730090     fprintf (stderr, "Usage:   chown GROUP|UID FILE...\n");
5730091     fprintf (stderr, "Example: chown group my_file\n");
5730092   }
```

96.1.9 applic/chmod.c

«

Si veda la sezione 86.7.

```
5740001   #include <unistd.h>
5740002   #include <stdlib.h>
5740003   #include <sys/stat.h>
5740004   #include <sys/types.h>
5740005   #include <fcntl.h>
5740006   #include <errno.h>
5740007   #include <signal.h>
5740008   #include <stdio.h>
5740009   #include <sys/wait.h>
5740010   #include <stdio.h>
5740011   #include <string.h>
5740012   #include <limits.h>
5740013   #include <sys/os32.h>
5740014   //-----------------------------------------------------
5740015   static void usage (void);
5740016   //-----------------------------------------------------
```

```
5740017 int
5740018 main (int argc, char *argv[], char *envp[])
5740019 {
5740020   int status;
5740021   mode_t mode;
5740022   char *m;      // Pointer inside the octal mode
5740023   // string.
5740024   int digit;
5740025   int a;        // Argument index.
5740026   //
5740027   //
5740028   //
5740029   if (argc < 3)
5740030     {
5740031       usage ();
5740032       return (1);
5740033     }
5740034   //
5740035   // Get mode: must be the first argument.
5740036   //
5740037   for (m = argv[1]; *m != 0; m++)
5740038     {
5740039       digit = (*m - '0');
5740040       if (digit < 0 || digit > 7)
5740041         {
5740042           usage ();
5740043           return (2);
5740044         }
5740045       mode = mode * 8 + digit;
5740046     }
5740047   //
5740048   // System call for all the remaining arguments.
5740049   //
5740050   for (a = 2; a < argc; a++)
5740051     {
5740052       status = chmod (argv[a], mode);
5740053       if (status != 0)
```

```
5740054           {
5740055             perror (argv[a]);
5740056             return (3);
5740057           }
5740058       }
5740059     //
5740060     // All done.
5740061     //
5740062     return (0);
5740063   }
5740064
5740065   //-----------------------------------------------------
5740066   static void
5740067   usage (void)
5740068   {
5740069     fprintf (stderr, "Usage:   chmod OCTAL_MODE FILE...\n");
5740070     fprintf (stderr, "Example: chmod 0640 my_file\n");
5740071   }
```

96.1.10 applic/chown.c

«

Si veda la sezione 86.8.

```
5750001   #include <unistd.h>
5750002   #include <stdlib.h>
5750003   #include <sys/stat.h>
5750004   #include <sys/types.h>
5750005   #include <fcntl.h>
5750006   #include <errno.h>
5750007   #include <stdio.h>
5750008   #include <ctype.h>
5750009   #include <pwd.h>
5750010   //-----------------------------------------------------
5750011   static void usage (void);
5750012   //-----------------------------------------------------
5750013   int
5750014   main (int argc, char *argv[], char *envp[])
```

```
5750015 {
5750016   char *user;
5750017   int uid;
5750018   struct passwd *pws;
5750019   struct stat file_status;
5750020   int a;          // Argument index.
5750021   int status;
5750022   //
5750023   //
5750024   //
5750025   if (argc < 3)
5750026     {
5750027       usage ();
5750028       return (1);
5750029     }
5750030   //
5750031   // Get user id number.
5750032   //
5750033   user = argv[1];
5750034   if (isdigit (*user))
5750035     {
5750036       uid = atoi (user);
5750037     }
5750038   else
5750039     {
5750040       pws = getpwnam (user);
5750041       if (pws == NULL)
5750042         {
5750043           fprintf (stderr, "Unknown user \"%s\"!\n", user);
5750044           return (2);
5750045         }
5750046       uid = pws->pw_uid;
5750047     }
5750048   //
5750049   // Now we have the user id. Start scanning file
5750050   // names.
5750051   //
```

```
5750052     for (a = 2; a < argc; a++)
5750053       {
5750054         //
5750055         // Verify if the file exists, through the return
5750056         // value of
5750057         // 'stat()'. No other checks are made.
5750058         //
5750059         if (stat (argv[a], &file_status) == 0)
5750060           {
5750061             //
5750062             // Try to change ownership.
5750063             //
5750064             status = chown (argv[a], uid, file_status.st_gid);
5750065             if (status != 0)
5750066               {
5750067                 perror (NULL);
5750068                 return (3);
5750069               }
5750070           }
5750071         else
5750072           {
5750073             fprintf (stderr,
5750074                      "File \"%s\" does not exist!\n",
5750075                      argv[a]);
5750076             continue;
5750077           }
5750078       }
5750079     //
5750080     // All done.
5750081     //
5750082     return (0);
5750083   }
5750084
5750085   //---------------------------------------------------------
5750086   static void
5750087   usage (void)
5750088   {
```

```
5750089     fprintf (stderr, "Usage:   chown USER|UID FILE...\n");
5750090     fprintf (stderr, "Example: chown user my_file\n");
5750091   }
```

96.1.11 applic/cp.c

«

Si veda la sezione 86.9.

```
5760001   #include <sys/os32.h>
5760002   #include <sys/stat.h>
5760003   #include <sys/types.h>
5760004   #include <unistd.h>
5760005   #include <stdlib.h>
5760006   #include <fcntl.h>
5760007   #include <errno.h>
5760008   #include <signal.h>
5760009   #include <stdio.h>
5760010   #include <string.h>
5760011   #include <limits.h>
5760012   #include <libgen.h>
5760013   //-------------------------------------------------------
5760014   static void usage (void);
5760015   //-------------------------------------------------------
5760016   int
5760017   main (int argc, char *argv[], char *envp[])
5760018   {
5760019     char *source;
5760020     char *destination;
5760021     char *destination_full;
5760022     struct stat file_status;
5760023     int dest_is_a_dir = 0;
5760024     int a;          // Argument index.
5760025     char path[PATH_MAX];
5760026     int fd_source = -1;
5760027     int fd_destination = -1;
5760028     char buffer_in[BUFSIZ];
5760029     char *buffer_out;
```

```
5760030     ssize_t count_in;      // Read counter.
5760031     ssize_t count_out;     // Write counter.
5760032     //
5760033     // There must be at least two arguments, plus the
5760034     // program name.
5760035     //
5760036     if (argc < 3)
5760037       {
5760038         usage ();
5760039         return (1);
5760040       }
5760041     //
5760042     // Select the last argument as the destination.
5760043     //
5760044     destination = argv[argc - 1];
5760045     //
5760046     // Check if it is a directory and save it in a flag.
5760047     //
5760048     if (stat (destination, &file_status) == 0)
5760049       {
5760050         if (S_ISDIR (file_status.st_mode))
5760051           {
5760052             dest_is_a_dir = 1;
5760053           }
5760054       }
5760055     //
5760056     // If there are more than two arguments, verify that
5760057     // the last
5760058     // one is a directory.
5760059     //
5760060     if (argc > 3)
5760061       {
5760062         if (!dest_is_a_dir)
5760063           {
5760064             usage ();
5760065             fprintf (stderr, "The destination \"%s\" ",
5760066                      destination);
```

```
5760067             fprintf (stderr, "is not a directory!\n");
5760068             return (1);
5760069           }
5760070       }
5760071     //
5760072     // Scan the arguments, excluded the last, that is
5760073     // the destination.
5760074     //
5760075     for (a = 1; a < (argc - 1); a++)
5760076       {
5760077         //
5760078         // Source.
5760079         //
5760080         source = argv[a];
5760081         //
5760082         // Verify access permissions.
5760083         //
5760084         if (access (source, R_OK) < 0)
5760085           {
5760086             perror (source);
5760087             continue;
5760088           }
5760089         //
5760090         // Destination.
5760091         //
5760092         // If it is a directory, the destination path
5760093         // must be corrected.
5760094         //
5760095         if (dest_is_a_dir)
5760096           {
5760097             path[0] = 0;
5760098             strcat (path, destination);
5760099             strcat (path, "/");
5760100             strcat (path, basename (source));
5760101             //
5760102             // Update the destination path.
5760103             //
```

```
5760104             destination_full = path;
5760105           }
5760106         else
5760107           {
5760108             destination_full = destination;
5760109           }
5760110         //
5760111         // Check if destination file exists.
5760112         //
5760113         if (stat (destination_full, &file_status) == 0)
5760114           {
5760115             fprintf (stderr,
5760116                      "The destination file, \"%s\", ",
5760117                      destination_full);
5760118             fprintf (stderr, "already exists!\n");
5760119             continue;
5760120           }
5760121         //
5760122         // Everything is ready for the copy.
5760123         //
5760124         fd_source = open (source, O_RDONLY);
5760125         if (fd_source < 0)
5760126           {
5760127             perror (source);
5760128             //
5760129             // Continue with the next file.
5760130             //
5760131             continue;
5760132           }
5760133         //
5760134         fd_destination = creat (destination_full, 0777);
5760135         if (fd_destination < 0)
5760136           {
5760137             perror (destination);
5760138             close (fd_source);
5760139             //
5760140             // Continue with the next file.
```

```
5760141             //
5760142             continue;
5760143           }
5760144         //
5760145         // Copy the data.
5760146         //
5760147         while (1)
5760148           {
5760149             count_in =
5760150               read (fd_source, buffer_in, (size_t) BUFSIZ);
5760151             if (count_in > 0)
5760152               {
5760153                 for (buffer_out = buffer_in; count_in > 0;)
5760154                   {
5760155                     count_out =
5760156                       write (fd_destination, buffer_out,
5760157                              (size_t) count_in);
5760158                     if (count_out < 0)
5760159                       {
5760160                         perror (destination);
5760161                         close (fd_source);
5760162                         close (fd_destination);
5760163                         return (3);
5760164                       }
5760165                     //
5760166                     // If not all data is written,
5760167                     // continue writing,
5760168                     // but change the buffer start
5760169                     // position and the
5760170                     // amount to be written.
5760171                     //
5760172                     buffer_out += count_out;
5760173                     count_in -= count_out;
5760174                   }
5760175               }
5760176             else if (count_in < 0)
5760177               {
```

```
5760178                 perror (source);
5760179                 close (fd_source);
5760180                 close (fd_destination);
5760181               }
5760182             else
5760183               {
5760184                 break;
5760185               }
5760186           }
5760187         //
5760188         if (close (fd_source))
5760189           {
5760190             perror (source);
5760191           }
5760192         if (close (fd_destination))
5760193           {
5760194             perror (destination);
5760195             return (4);
5760196           }
5760197       }
5760198     //
5760199     // All done.
5760200     //
5760201     return (0);
5760202   }
5760203
5760204   //-----------------------------------------------------------
5760205   static void
5760206   usage (void)
5760207   {
5760208     fprintf (stderr, "Usage: cp OLD_NAME NEW_NAME\n");
5760209     fprintf (stderr, "       cp FILE... DIRECTORY\n");
5760210   }
```

96.1.12 applic/crt0.mer.s

«

Si veda la sezione 84.5.2.

```
5770001 .extern main
5770002 .extern _stdio_stream_setup
5770003 .extern _dirent_directory_stream_setup
5770004 .extern _atexit_setup
5770005 .extern _environment_setup
5770006 .global startup
5770007 .global _data_end
5770008 #-------------------------------------------------------
5770009 # Please note that, all segment descriptors are already
5770010 # set from the scheduler, and there is also data inside
5770011 # the stack, so that the call to 'main()' function will
5770012 # result as expected.
5770013 #-------------------------------------------------------
5770014 # The following statement says that the code will start
5770015 # at "startup" label.
5770016 #-------------------------------------------------------
5770017 .section .text
5770018 #-------------------------------------------------------
5770019 startup:
5770020     #
5770021     # Jump after initial data.
5770022     #
5770023     jmp startup_code
5770024     #
5770025 filler:
5770026     #
5770027     # After four bytes, from the start, there is the
5770028     # magic number and other data.
5770029     #
5770030     .space (0x0004 - (filler - startup))
5770031     #
5770032 magic:
5770033     .quad 0x6F7333326170706C   # os32appl
5770034     ;
```

```
5770035 doffset:                #
5770036     .int _text_start    # Data offset from start.
5770037 etext:                  #
5770038     .int _text_end      # End of code
5770039 edata:                  #
5770040     .int _data_end      # End of initialized data.
5770041 ebss:                   #
5770042     .int _bss_end       # End of not initialized data.
5770043 stack_size:             #
5770044     .int 0x8000         # Requested stack size. Every
5770045                         # single application
5770046                         # might change this value.
5770047     #
5770048     # At the next label, the work begins.
5770049     #
5770050 .align 4
5770051 startup_code:
5770052     #
5770053     # Before the call to the main function, it is
5770054     # necessary to extract the value to assign to the
5770055     # global variable 'environ'. It is described as
5770056     # 'char **environ' and should contain the same
5770057     # address pointed by 'envp'. To get this value,
5770058     # the stack is popped and then pushed again.
5770059     # Please recall that the stack was prepared from
5770060     # the process management, at the 'exec()' system
5770061     # call.
5770062     #
5770063     pop %eax            # argc
5770064     pop %ebx            # argv
5770065     pop %ecx            # envp
5770066     mov %ecx, environ   # Variable 'environ' comes from
5770067                         # <unistd.h>.
5770068     push %ecx
5770069     push %ebx
5770070     push %eax
5770071     #
```

```
5770072      # Could it be enough? Of course not!
5770073      # To be able to handle the
5770074      # environment, it must be copied inside the table
5770075      # '_environment_table[][]', that is defined inside
5770076      # <stdlib.h>.
5770077      # To copy the environment it is used the function
5770078      # '_environment_setup()', passing the 'envp'
5770079      # pointer.
5770080      #
5770081      push %ecx
5770082      call _environment_setup
5770083      add  $4, %esp
5770084      #
5770085      # After the environment copy is done, the value for
5770086      # the traditional variable 'environ' is updated, to
5770087      # point to the new array of pointer.
5770088      # The updated value comes from variable
5770089      # '_environment', defined inside <stdlib.h>.
5770090      # Then, also the 'argv' contained inside
5770091      # the stack is replaced with the new value.
5770092      #
5770093      mov  $_environment, %eax
5770094      mov  %eax, environ
5770095      #
5770096      pop %eax             # argc
5770097      pop %ebx             # argv[][]
5770098      pop %ecx             # envp[][]
5770099      mov $_environment, %ecx
5770100      push %ecx
5770101      push %ebx
5770102      push %eax
5770103      #
5770104      # Setup standard I/O streams and at-exit table.
5770105      #
5770106      call _stdio_stream_setup
5770107      call _dirent_directory_stream_setup
5770108      call _atexit_setup
```

```
5770109      #
5770110      # Call the main function. The arguments are
5770111      # already pushed inside the stack.
5770112      #
5770113      call main
5770114      #
5770115      # Save the return value at the symbol 'exit_value'.
5770116      #
5770117      mov  %eax, exit_value
5770118      #
5770119  .align 4
5770120  halt:
5770121      #
5770122      pushl $2             # Size of message.
5770123      pushl $exit_value    # Pointer to the message.
5770124      pushl $6             # SYS_EXIT
5770125      call sys
5770126      add  $4, %esp
5770127      add  $4, %esp
5770128      add  $4, %esp
5770129      #
5770130      jmp halt
5770131      #
5770132  #-------------------------------------------------------
5770133  .align 4
5770134  .section .rodata
5770135  #
5770136  data_magic:
5770137      .quad 0x6F73333264617461   # os32data [1]
5770138  #
5770139  _data_end:
5770140      .int _bss_end
5770141  #
5770142  # [1] This is placed here just to be the same as the
5770143  #     other file 'crt0.sep.s'.
5770144  #     See the other file for an explanation.
5770145  #-------------------------------------------------------
```

```
5770146 .align 4
5770147 .section .data
5770148 #
5770149 exit_value:
5770150     .int 0x00000000
5770151 #-------------------------------------------------------
5770152 .align 4
5770153 .section .bss
```

96.1.13 applic/crt0.sep.s

Si veda la sezione 84.5.2.

```
5780001 .extern main
5780002 .extern _stdio_stream_setup
5780003 .extern _dirent_directory_stream_setup
5780004 .extern _atexit_setup
5780005 .extern _environment_setup
5780006 .global startup
5780007 .global _data_end
5780008 #-------------------------------------------------------
5780009 # Please note that, all segment descriptors are already
5780010 # set from the scheduler, and there is also data inside
5780011 # the stack, so that the call to 'main()' function will
5780012 # result as expected.
5780013 #-------------------------------------------------------
5780014 # The following statement says that the code will start
5780015 # at "startup" label.
5780016 #-------------------------------------------------------
5780017 .section .text
5780018 #-------------------------------------------------------
5780019 startup:
5780020     #
5780021     # Jump after initial data.
5780022     #
5780023     jmp startup_code
5780024     #
```

```
5780025 filler:
5780026     #
5780027     # After four bytes, from the start, there is the
5780028     # magic number and other data.
5780029     #
5780030     .space (0x0004 - (filler - startup))
5780031     #
5780032 magic:
5780033     .quad 0x6F7333326170706C   # os32appl
5780034     ;
5780035 doffset:                #
5780036     .int _text_end       # Data offset from start: at
5780037                          # the end of TEXT.
5780038 etext:                  #
5780039     .int _text_end       # End of code
5780040 edata:                  #
5780041     .int _data_end       # End of initialized data.
5780042 ebss:                   #
5780043     .int _bss_end        # End of not initialized data.
5780044 stack_size:             #
5780045     .int 0x8000          # Requested stack size. Every
5780046                          # single application
5780047                          # might change this value.
5780048     #
5780049     # At the next label, the work begins.
5780050     #
5780051 .align 4
5780052 startup_code:
5780053     #
5780054     # Before the call to the main function, it is
5780055     # necessary to extract the value to assign to the
5780056     # global variable 'environ'. It is described as
5780057     # 'char **environ' and should contain the same
5780058     # address pointed by 'envp'. To get this value, the
5780059     # stack is popped and then pushed again.
5780060     # Please recall that the stack was prepared from
5780061     # the process management, at the 'exec()' system
```

```
5780062     # call.
5780063     #
5780064     pop %eax            # argc
5780065     pop %ebx            # argv
5780066     pop %ecx            # envp
5780067     mov %ecx, environ   # Variable 'environ' comes from
5780068                         # <unistd.h>.
5780069     push %ecx
5780070     push %ebx
5780071     push %eax
5780072     #
5780073     # Could it be enough? Of course not! To be able to
5780074     # handle the environment, it must be copied inside
5780075     # the table '_environment_table[][]', that is
5780076     # defined inside <stdlib.h>.
5780077     # To copy the environment it is used the function
5780078     # '_environment_setup()', passing the 'envp'
5780079     # pointer.
5780080     #
5780081     push %ecx
5780082     call _environment_setup
5780083     add  $4, %esp
5780084     #
5780085     # After the environment copy is done, the value for
5780086     # the traditional variable 'environ' is updated,
5780087     # to point to the new array of pointer.
5780088     # The updated value comes from variable
5780089     # '_environment', defined inside <stdlib.h>.
5780090     # Then, also the 'argv' contained inside
5780091     # the stack is replaced with the new value.
5780092     #
5780093     mov  $_environment, %eax
5780094     mov  %eax, environ
5780095     #
5780096     pop %eax            # argc
5780097     pop %ebx            # argv[][]
5780098     pop %ecx            # envp[][]
```

```
5780099      mov $_environment, %ecx
5780100      push %ecx
5780101      push %ebx
5780102      push %eax
5780103      #
5780104      # Setup standard I/O streams and at-exit table.
5780105      #
5780106      call _stdio_stream_setup
5780107      call _dirent_directory_stream_setup
5780108      call _atexit_setup
5780109      #
5780110      # Call the main function. The arguments are already
5780111      # pushed inside the stack.
5780112      #
5780113      call main
5780114      #
5780115      # Save the return value at the symbol 'exit_value'.
5780116      #
5780117      mov  %eax, exit_value
5780118      #
5780119  .align 4
5780120  halt:
5780121      #
5780122      pushl $2            # Size of message.
5780123      pushl $exit_value   # Pointer to the message.
5780124      pushl $6            # SYS_EXIT
5780125      call sys
5780126      add  $4, %esp
5780127      add  $4, %esp
5780128      add  $4, %esp
5780129      #
5780130      jmp halt
5780131      #
5780132  #-------------------------------------------------
5780133  .align 4
5780134  .section .rodata
5780135  #
```

```
5780136 data_magic:
5780137     .quad 0x6F73333264617461   # os32data [1]
5780138 #
5780139 _data_end:
5780140     .int _bss_end
5780141 #
5780142 # [1] This signature is just a place holder, at the
5780143 #     beginning of the data segment, which starts at
5780144 #     address 0x00000000. This is to avoid constant
5780145 #     strings to be placed exactly at the beginning
5780146 #     (and it happened so), where the address is
5780147 #     equal to 'NULL'.
5780148 #-------------------------------------------------------
5780149 .align 4
5780150 .section .data
5780151 #
5780152 exit_value:
5780153     .int 0x00000000
5780154 #-------------------------------------------------------
5780155 .align 4
5780156 .section .bss
```

96.1.14 applic/date.c

Si veda la sezione 86.10.

```
5790001 #include <unistd.h>
5790002 #include <stdlib.h>
5790003 #include <errno.h>
5790004 #include <time.h>
5790005 #include <ctype.h>
5790006 //-------------------------------------------------------
5790007 static void usage (void);
5790008 //-------------------------------------------------------
5790009 int
5790010 main (int argc, char *argv[], char *envp[])
5790011 {
```

```
5790012     struct tm *timeptr;
5790013     char string[5];
5790014     time_t timer;
5790015     int length;
5790016     char *input;
5790017     int i;
5790018     int status;
5790019     //
5790020     // There can be at most an argument.
5790021     //
5790022     if (argc > 2)
5790023       {
5790024         usage ();
5790025         return (1);
5790026       }
5790027     //
5790028     // Check if there is no argument: must show the
5790029     // date.
5790030     //
5790031     if (argc == 1)
5790032       {
5790033         timer = time (NULL);
5790034         printf ("%s\n", ctime (&timer));
5790035         return (0);
5790036       }
5790037     //
5790038     // There is one argument and must be the date do
5790039     // set.
5790040     //
5790041     input = argv[1];
5790042     //
5790043     // First get current date, for default values.
5790044     //
5790045     timer = time (NULL);
5790046     timeptr = gmtime (&timer);
5790047     //
5790048     // Verify to have a correct input.
```

```
5790049     //
5790050     length = (int) strlen (input);
5790051     if (length == 8 || length == 10 || length == 12)
5790052       {
5790053         for (i = 0; i < length; i++)
5790054           {
5790055             if (!isdigit (input[i]))
5790056               {
5790057                 usage ();
5790058                 return (2);
5790059               }
5790060           }
5790061       }
5790062     else
5790063       {
5790064         printf ("input: \"%s\"; length: %i\n", input, length);
5790065         usage ();
5790066         return (3);
5790067       }
5790068     //
5790069     // Select the month.
5790070     //
5790071     string[0] = input[0];
5790072     string[1] = input[1];
5790073     string[2] = '\0';
5790074     timeptr->tm_mon = atoi (string);
5790075     //
5790076     // Select the day.
5790077     //
5790078     string[0] = input[2];
5790079     string[1] = input[3];
5790080     string[2] = '\0';
5790081     timeptr->tm_mday = atoi (string);
5790082     //
5790083     // Select the hour.
5790084     //
5790085     string[0] = input[4];
```

```
5790086     string[1] = input[5];
5790087     string[2] = '\0';
5790088     timeptr->tm_hour = atoi (string);
5790089     //
5790090     // Select the minute.
5790091     //
5790092     string[0] = input[6];
5790093     string[1] = input[7];
5790094     string[2] = '\0';
5790095     timeptr->tm_min = atoi (string);
5790096     //
5790097     // Select the year: must verify if there is a
5790098     // century.
5790099     //
5790100     if (length == 12)
5790101       {
5790102         string[0] = input[8];
5790103         string[1] = input[9];
5790104         string[2] = input[10];
5790105         string[3] = input[11];
5790106         string[4] = '\0';
5790107         timeptr->tm_year = atoi (string);
5790108       }
5790109     else if (length == 10)
5790110       {
5790111         sprintf (string, "%04i", timeptr->tm_year);
5790112         string[2] = input[8];
5790113         string[3] = input[9];
5790114         string[4] = '\0';
5790115         timeptr->tm_year = atoi (string);
5790116       }
5790117     //
5790118     // Now convert to `time_t'.
5790119     //
5790120     timer = mktime (timeptr);
5790121     //
5790122     // Save to the system.
```

```
5790123     //
5790124     status = stime (&timer);
5790125     if (status != 0)
5790126       {
5790127         perror (NULL);
5790128       }
5790129     //
5790130     return (0);
5790131   }
5790132
5790133   //---------------------------------------------------------
5790134   static void
5790135   usage (void)
5790136   {
5790137     fprintf (stderr, "Usage: date [MMDDHHMM[[CC]YY]]\n");
5790138   }
```

96.1.15 applic/ed.c

«

Si veda la sezione 86.11.

```
5800001   //---------------------------------------------------------
5800002   // 2009.08.18
5800003   // Modified by Daniele Giacomini for 'os16', to
5800004   // harmonize with it, even, when possible, on coding
5800005   // style.
5800006   //
5800007   // The original was taken form ELKS sources:
5800008   // 'elkscmd/misc_utils/ed.c'.
5800009   //---------------------------------------------------------
5800010   //
5800011   // Copyright (c) 1993 by David I. Bell
5800012   // Permission is granted to use, distribute, or modify
5800013   // this source, provided that this copyright notice
5800014   // remains intact.
5800015   //
5800016   // The "ed" built-in command (much simplified)
```

```
5800017 //
5800018 //-------------------------------------------------------
5800019
5800020 #include <stdio.h>
5800021 #include <ctype.h>
5800022 #include <unistd.h>
5800023 #include <stdbool.h>
5800024 #include <string.h>
5800025 #include <stdlib.h>
5800026 #include <fcntl.h>
5800027 //-------------------------------------------------------
5800028 #define isoctal(ch)  (((ch) >= '0') && ((ch) <= '7'))
5800029 #define USERSIZE     1024       /* max line length typed in
5800030                                    by user */
5800031 #define INITBUFSIZE  1024       /* initial buffer size */
5800032 //-------------------------------------------------------
5800033 typedef int num_t;
5800034 typedef int len_t;
5800035 //
5800036 // The following is the type definition of structure
5800037 // 'line_t', but the structure contains pointers to the
5800038 // same kind of type. With the compiler Bcc, it is the
5800039 // only way to declare it.
5800040 //
5800041 typedef struct line line_t;
5800042 //
5800043 struct line
5800044 {
5800045   line_t *next;
5800046   line_t *prev;
5800047   len_t len;
5800048   char data[1];
5800049 };
5800050 //
5800051 static line_t lines;
5800052 static line_t *curline;
5800053 static num_t curnum;
```

```
5800054 static num_t lastnum;
5800055 static num_t marks[26];
5800056 static bool dirty;
5800057 static char *filename;
5800058 static char searchstring[USERSIZE];
5800059 //
5800060 static char *bufbase;
5800061 static char *bufptr;
5800062 static len_t bufused;
5800063 static len_t bufsize;
5800064 //------------------------------------------------------------
5800065 static void docommands (void);
5800066 static void subcommand (char *cp, num_t num1, num_t num2);
5800067 static bool getnum (char **retcp, bool * rethavenum,
5800068                     num_t * retnum);
5800069 static bool setcurnum (num_t num);
5800070 static bool initedit (void);
5800071 static void termedit (void);
5800072 static void addlines (num_t num);
5800073 static bool insertline (num_t num, char *data, len_t len);
5800074 static bool deletelines (num_t num1, num_t num2);
5800075 static bool printlines (num_t num1, num_t num2,
5800076                         bool expandflag);
5800077 static bool writelines (char *file, num_t num1, num_t num2);
5800078 static bool readlines (char *file, num_t num);
5800079 static num_t searchlines (char *str, num_t num1,
5800080                           num_t num2);
5800081 static len_t findstring (line_t * lp, char *str,
5800082                          len_t len, len_t offset);
5800083 static line_t *findline (num_t num);
5800084 //------------------------------------------------------------
5800085 // Main.
5800086 //------------------------------------------------------------
5800087 int
5800088 main (int argc, char *argv[], char *envp[])
5800089 {
5800090   if (!initedit ())
```

```
5800091       return (2);
5800092     //
5800093     if (argc > 1)
5800094       {
5800095         filename = strdup (argv[1]);
5800096         if (filename == NULL)
5800097           {
5800098             fprintf (stderr, "No memory\n");
5800099             termedit ();
5800100             return (1);
5800101           }
5800102         //
5800103         if (!readlines (filename, 1))
5800104           {
5800105             termedit ();
5800106             return (0);
5800107           }
5800108         //
5800109         if (lastnum)
5800110           setcurnum (1);
5800111         //
5800112         dirty = false;
5800113       }
5800114     //
5800115     docommands ();
5800116     //
5800117     termedit ();
5800118     return (0);
5800119   }
5800120
5800121   //-------------------------------------------------------
5800122   // Read commands until we are told to stop.
5800123   //-------------------------------------------------------
5800124   void
5800125   docommands (void)
5800126   {
5800127     char *cp;
```

```
5800128     int len;
5800129     num_t num1;
5800130     num_t num2;
5800131     bool have1;
5800132     bool have2;
5800133     char buf[USERSIZE];
5800134     //
5800135     while (true)
5800136       {
5800137         printf (": ");
5800138         fflush (stdout);
5800139         //
5800140         if (fgets (buf, sizeof (buf), stdin) == NULL)
5800141           {
5800142             return;
5800143           }
5800144         //
5800145         len = strlen (buf);
5800146         if (len == 0)
5800147           {
5800148             return;
5800149           }
5800150         //
5800151         cp = &buf[len - 1];
5800152         if (*cp != '\n')
5800153           {
5800154             fprintf (stderr, "Command line too long\n");
5800155             do
5800156               {
5800157                 len = fgetc (stdin);
5800158               }
5800159             while ((len != EOF) && (len != '\n'));
5800160             //
5800161             continue;
5800162           }
5800163         //
5800164         while ((cp > buf) && isblank (cp[-1]))
```

```
5800165           {
5800166             cp--;
5800167           }
5800168         //
5800169         *cp = '\0';
5800170         //
5800171         cp = buf;
5800172         //
5800173         while (isblank (*cp))
5800174           {
5800175             // *cp++;
5800176             cp++;
5800177           }
5800178         //
5800179         have1 = false;
5800180         have2 = false;
5800181         //
5800182         if ((curnum == 0) && (lastnum > 0))
5800183           {
5800184             curnum = 1;
5800185             curline = lines.next;
5800186           }
5800187         //
5800188         if (!getnum (&cp, &have1, &num1))
5800189           {
5800190             continue;
5800191           }
5800192         //
5800193         while (isblank (*cp))
5800194           {
5800195             cp++;
5800196           }
5800197         //
5800198         if (*cp == ',')
5800199           {
5800200             cp++;
5800201             if (!getnum (&cp, &have2, &num2))
```

```
5800202                 {
5800203                   continue;
5800204                 }
5800205               //
5800206               if (!have1)
5800207                 {
5800208                   num1 = 1;
5800209                 }
5800210               if (!have2)
5800211                 {
5800212                   num2 = lastnum;
5800213                 }
5800214               have1 = true;
5800215               have2 = true;
5800216             }
5800217           //
5800218           if (!have1)
5800219             {
5800220               num1 = curnum;
5800221             }
5800222           if (!have2)
5800223             {
5800224               num2 = num1;
5800225             }
5800226           //
5800227           // Command interpretation switch.
5800228           //
5800229           switch (*cp++)
5800230             {
5800231             case 'a':
5800232               addlines (num1 + 1);
5800233               break;
5800234               //
5800235             case 'c':
5800236               deletelines (num1, num2);
5800237               addlines (num1);
5800238               break;
```

```
5800239             //
5800240           case 'd':
5800241             deletelines (num1, num2);
5800242             break;
5800243             //
5800244           case 'f':
5800245             if (*cp && !isblank (*cp))
5800246               {
5800247                 fprintf (stderr, "Bad file command\n");
5800248                 break;
5800249               }
5800250             //
5800251             while (isblank (*cp))
5800252               {
5800253                 cp++;
5800254               }
5800255             if (*cp == '\0')
5800256               {
5800257                 if (filename)
5800258                   {
5800259                     printf ("\"%s\"\n", filename);
5800260                   }
5800261                 else
5800262                   {
5800263                     printf ("No filename\n");
5800264                   }
5800265                 break;
5800266               }
5800267             //
5800268             cp = strdup (cp);
5800269             //
5800270             if (cp == NULL)
5800271               {
5800272                 fprintf (stderr, "No memory for filename\n");
5800273                 break;
5800274               }
5800275             //
```

```
5800276             if (filename)
5800277               {
5800278                 free (filename);
5800279               }
5800280             //
5800281             filename = cp;
5800282             break;
5800283             //
5800284           case 'i':
5800285             addlines (num1);
5800286             break;
5800287             //
5800288           case 'k':
5800289             while (isblank (*cp))
5800290               {
5800291                 cp++;
5800292               }
5800293             //
5800294             if ((*cp < 'a') || (*cp > 'a') || cp[1])
5800295               {
5800296                 fprintf (stderr, "Bad mark name\n");
5800297                 break;
5800298               }
5800299             //
5800300             marks[*cp - 'a'] = num2;
5800301             break;
5800302             //
5800303           case 'l':
5800304             printlines (num1, num2, true);
5800305             break;
5800306             //
5800307           case 'p':
5800308             printlines (num1, num2, false);
5800309             break;
5800310             //
5800311           case 'q':
5800312             while (isblank (*cp))
```

```
5800313                 {
5800314                   cp++;
5800315                 }
5800316               //
5800317               if (have1 || *cp)
5800318                 {
5800319                   fprintf (stderr, "Bad quit command\n");
5800320                   break;
5800321                 }
5800322               //
5800323               if (!dirty)
5800324                 {
5800325                   return;
5800326                 }
5800327               //
5800328               printf ("Really quit? ");
5800329               fflush (stdout);
5800330               //
5800331               buf[0] = '\0';
5800332               fgets (buf, sizeof (buf), stdin);
5800333               cp = buf;
5800334               //
5800335               while (isblank (*cp))
5800336                 {
5800337                   cp++;
5800338                 }
5800339               //
5800340               if ((*cp == 'y') || (*cp == 'Y'))
5800341                 {
5800342                   return;
5800343                 }
5800344               //
5800345               break;
5800346               //
5800347             case 'r':
5800348               if (*cp && !isblank (*cp))
5800349                 {
```

```
5800350                 fprintf (stderr, "Bad read command\n");
5800351                 break;
5800352               }
5800353             //
5800354             while (isblank (*cp))
5800355               {
5800356                 cp++;
5800357               }
5800358             //
5800359             if (*cp == '\0')
5800360               {
5800361                 fprintf (stderr, "No filename\n");
5800362                 break;
5800363               }
5800364             //
5800365             if (!have1)
5800366               {
5800367                 num1 = lastnum;
5800368               }
5800369             //
5800370             // Open the file and add to the buffer
5800371             // at the next line.
5800372             //
5800373             if (readlines (cp, num1 + 1))
5800374               {
5800375                 //
5800376                 // If the file open fails, just
5800377                 // break the command.
5800378                 //
5800379                 break;
5800380               }
5800381             //
5800382             // Set the default file name, if no
5800383             // previous name is available.
5800384             //
5800385             if (filename == NULL)
5800386               {
```

```
5800387                 filename = strdup (cp);
5800388               }
5800389             //
5800390             break;
5800391
5800392           case 's':
5800393             subcommand (cp, num1, num2);
5800394             break;
5800395             //
5800396           case 'w':
5800397             if (*cp && !isblank (*cp))
5800398               {
5800399                 fprintf (stderr, "Bad write command\n");
5800400                 break;
5800401               }
5800402             //
5800403             while (isblank (*cp))
5800404               {
5800405                 cp++;
5800406               }
5800407             //
5800408             if (!have1)
5800409               {
5800410                 num1 = 1;
5800411                 num2 = lastnum;
5800412               }
5800413             //
5800414             // If the file name is not specified, use
5800415             // the
5800416             // default one.
5800417             //
5800418             if (*cp == '\0')
5800419               {
5800420                 cp = filename;
5800421               }
5800422             //
5800423             // If even the default file name is not
```

```
5800424             // specified,
5800425             // tell it.
5800426             //
5800427             if (cp == NULL)
5800428               {
5800429                 fprintf (stderr, "No file name specified\n");
5800430                 break;
5800431               }
5800432             //
5800433             // Write the file.
5800434             //
5800435             writelines (cp, num1, num2);
5800436             //
5800437             break;
5800438             //
5800439           case 'z':
5800440             switch (*cp)
5800441               {
5800442               case '-':
5800443                 printlines (curnum - 21, curnum, false);
5800444                 break;
5800445               case '.':
5800446                 printlines (curnum - 11, curnum + 10, false);
5800447                 break;
5800448               default:
5800449                 printlines (curnum, curnum + 21, false);
5800450                 break;
5800451               }
5800452             break;
5800453             //
5800454           case '.':
5800455             if (have1)
5800456               {
5800457                 fprintf (stderr, "No arguments allowed\n");
5800458                 break;
5800459               }
5800460             printlines (curnum, curnum, false);
```

```
5800461             break;
5800462             //
5800463           case '-':
5800464             if (setcurnum (curnum - 1))
5800465               {
5800466                 printlines (curnum, curnum, false);
5800467               }
5800468             break;
5800469             //
5800470           case '=':
5800471             printf ("%d\n", num1);
5800472             break;
5800473             //
5800474           case '\0':
5800475             if (have1)
5800476               {
5800477                 printlines (num2, num2, false);
5800478                 break;
5800479               }
5800480             //
5800481             if (setcurnum (curnum + 1))
5800482               {
5800483                 printlines (curnum, curnum, false);
5800484               }
5800485             break;
5800486             //
5800487           default:
5800488             fprintf (stderr, "Unimplemented command\n");
5800489             break;
5800490           }
5800491       }
5800492   }
5800493
5800494   //-------------------------------------------------------
5800495   // Do the substitute command.
5800496   // The current line is set to the last substitution
5800497   // done.
```

```
5800498 //-------------------------------------------------------
5800499 void
5800500 subcommand (char *cp, num_t num1, num_t num2)
5800501 {
5800502   int delim;
5800503   char *oldstr;
5800504   char *newstr;
5800505   len_t oldlen;
5800506   len_t newlen;
5800507   len_t deltalen;
5800508   len_t offset;
5800509   line_t *lp;
5800510   line_t *nlp;
5800511   bool globalflag;
5800512   bool printflag;
5800513   bool didsub;
5800514   bool needprint;
5800515
5800516   if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5800517     {
5800518       fprintf (stderr, "Bad line range for substitute\n");
5800519       return;
5800520     }
5800521   //
5800522   globalflag = false;
5800523   printflag = false;
5800524   didsub = false;
5800525   needprint = false;
5800526   //
5800527   if (isblank (*cp) || (*cp == '\0'))
5800528     {
5800529       fprintf (stderr, "Bad delimiter for substitute\n");
5800530       return;
5800531     }
5800532   //
5800533   delim = *cp++;
5800534   oldstr = cp;
```

```
5800535     //
5800536     cp = strchr (cp, delim);
5800537     //
5800538     if (cp == NULL)
5800539       {
5800540         fprintf (stderr,
5800541                  "Missing 2nd delimiter for " "substitute\n");
5800542         return;
5800543       }
5800544     //
5800545     *cp++ = '\0';
5800546     //
5800547     newstr = cp;
5800548     cp = strchr (cp, delim);
5800549     //
5800550     if (cp)
5800551       {
5800552         *cp++ = '\0';
5800553       }
5800554     else
5800555       {
5800556         cp = "";
5800557       }
5800558     while (*cp)
5800559       {
5800560         switch (*cp++)
5800561           {
5800562           case 'g':
5800563             globalflag = true;
5800564             break;
5800565             //
5800566           case 'p':
5800567             printflag = true;
5800568             break;
5800569             //
5800570           default:
5800571             fprintf (stderr,
```

```
5800572                     "Unknown option for substitute\n");
5800573           return;
5800574         }
5800575     }
5800576   //
5800577   if (*oldstr == '\0')
5800578     {
5800579       if (searchstring[0] == '\0')
5800580         {
5800581           fprintf (stderr, "No previous search string\n");
5800582           return;
5800583         }
5800584       oldstr = searchstring;
5800585     }
5800586   //
5800587   if (oldstr != searchstring)
5800588     {
5800589       strcpy (searchstring, oldstr);
5800590     }
5800591   //
5800592   lp = findline (num1);
5800593   if (lp == NULL)
5800594     {
5800595       return;
5800596     }
5800597   //
5800598   oldlen = strlen (oldstr);
5800599   newlen = strlen (newstr);
5800600   deltalen = newlen - oldlen;
5800601   offset = 0;
5800602   //
5800603   while (num1 <= num2)
5800604     {
5800605       offset = findstring (lp, oldstr, oldlen, offset);
5800606       if (offset < 0)
5800607         {
5800608           if (needprint)
```

```
5800609                 {
5800610                   printlines (num1, num1, false);
5800611                   needprint = false;
5800612                 }
5800613               //
5800614               offset = 0;
5800615               lp = lp->next;
5800616               num1++;
5800617               continue;
5800618             }
5800619           //
5800620           needprint = printflag;
5800621           didsub = true;
5800622           dirty = true;
5800623
5800624           // -----------------------------------------
5800625           // If the replacement string is the same size or
5800626           // shorter
5800627           // than the old string, then the substitution is
5800628           // easy.
5800629           // -----------------------------------------
5800630
5800631           if (deltalen <= 0)
5800632             {
5800633               memcpy (&lp->data[offset], newstr, newlen);
5800634               //
5800635               if (deltalen)
5800636                 {
5800637                   memcpy (&lp->data[offset + newlen],
5800638                           &lp->data[offset + oldlen],
5800639                           lp->len - offset - oldlen);
5800640                   //
5800641                   lp->len += deltalen;
5800642                 }
5800643               //
5800644               offset += newlen;
5800645               //
```

```
5800646             if (globalflag)
5800647               {
5800648                 continue;
5800649               }
5800650             //
5800651             if (needprint)
5800652               {
5800653                 printlines (num1, num1, false);
5800654                 needprint = false;
5800655               }
5800656             //
5800657             lp = nlp->next;
5800658             num1++;
5800659             continue;
5800660           }
5800661
5800662         // ------------------------------------------
5800663         // The new string is larger, so allocate a new
5800664         // line structure and use that.
5800665         // Link it in place of the old line structure.
5800666         // ------------------------------------------
5800667
5800668         nlp =
5800669           (line_t *) malloc (sizeof (line_t) + lp->len +
5800670                              deltalen);
5800671         //
5800672         if (nlp == NULL)
5800673           {
5800674             fprintf (stderr, "Cannot get memory for line\n");
5800675             return;
5800676           }
5800677         //
5800678         nlp->len = lp->len + deltalen;
5800679         //
5800680         memcpy (nlp->data, lp->data, offset);
5800681         //
5800682         memcpy (&nlp->data[offset], newstr, newlen);
```

```
5800683         //
5800684         memcpy (&nlp->data[offset + newlen],
5800685                 &lp->data[offset + oldlen],
5800686                 lp->len - offset - oldlen);
5800687         //
5800688         nlp->next = lp->next;
5800689         nlp->prev = lp->prev;
5800690         nlp->prev->next = nlp;
5800691         nlp->next->prev = nlp;
5800692         //
5800693         if (curline == lp)
5800694           {
5800695             curline = nlp;
5800696           }
5800697         //
5800698         free (lp);
5800699         lp = nlp;
5800700         //
5800701         offset += newlen;
5800702         //
5800703         if (globalflag)
5800704           {
5800705             continue;
5800706           }
5800707         //
5800708         if (needprint)
5800709           {
5800710             printlines (num1, num1, false);
5800711             needprint = false;
5800712           }
5800713         //
5800714         lp = lp->next;
5800715         num1++;
5800716       }
5800717     //
5800718     if (!didsub)
5800719       {
```

```
5800720         fprintf (stderr,
5800721                  "No substitutions found for \"%s\"\n",
5800722                  oldstr);
5800723       }
5800724   }
5800725
5800726   //-------------------------------------------------------
5800727   // Search a line for the specified string starting at
5800728   // the specified offset in the line.  Returns the
5800729   // offset of the found string, or -1.
5800730   //-------------------------------------------------------
5800731   len_t
5800732   findstring (line_t * lp, char *str, len_t len, len_t offset)
5800733   {
5800734     len_t left;
5800735     char *cp;
5800736     char *ncp;
5800737     //
5800738     cp = &lp->data[offset];
5800739     left = lp->len - offset;
5800740     //
5800741     while (left >= len)
5800742       {
5800743         ncp = memchr (cp, *str, left);
5800744         if (ncp == NULL)
5800745           {
5800746             return (len_t) - 1;
5800747           }
5800748         //
5800749         left -= (ncp - cp);
5800750         if (left < len)
5800751           {
5800752             return (len_t) - 1;
5800753           }
5800754         //
5800755         cp = ncp;
5800756         if (memcmp (cp, str, len) == 0)
```

```
5800757           {
5800758             return (len_t) (cp - lp->data);
5800759           }
5800760         //
5800761         cp++;
5800762         left--;
5800763       }
5800764     //
5800765     return (len_t) - 1;
5800766   }
5800767
5800768   //-------------------------------------------------------
5800769   // Add lines which are typed in by the user.
5800770   // The lines are inserted just before the specified
5800771   // line number.
5800772   // The lines are terminated by a line containing a
5800773   // single dot (ugly!), or by an end of file.
5800774   //-------------------------------------------------------
5800775   void
5800776   addlines (num_t num)
5800777   {
5800778     int len;
5800779     char buf[USERSIZE + 1];
5800780     //
5800781     while (fgets (buf, sizeof (buf), stdin))
5800782       {
5800783         if ((buf[0] == '.') && (buf[1] == '\n')
5800784             && (buf[2] == '\0'))
5800785           {
5800786             return;
5800787           }
5800788         //
5800789         len = strlen (buf);
5800790         //
5800791         if (len == 0)
5800792           {
5800793             return;
```

```
5800794           }
5800795         //
5800796         if (buf[len - 1] != '\n')
5800797           {
5800798             fprintf (stderr, "Line too long\n");
5800799             //
5800800             do
5800801               {
5800802                 len = fgetc (stdin);
5800803               }
5800804             while ((len != EOF) && (len != '\n'));
5800805             //
5800806             return;
5800807           }
5800808         //
5800809         if (!insertline (num++, buf, len))
5800810           {
5800811             return;
5800812           }
5800813       }
5800814   }
5800815
5800816   //-----------------------------------------------------
5800817   // Parse a line number argument if it is present.  This
5800818   // is a sum or difference of numbers, '.', '$', 'x, or
5800819   // a search string.
5800820   // Returns true if successful (whether or not there was
5800821   // a number).
5800822   // Returns false if there was a parsing error, with a
5800823   // message output.
5800824   // Whether there was a number is returned indirectly,
5800825   // as is the number.
5800826   // The character pointer which stopped the scan is also
5800827   // returned.
5800828   //-----------------------------------------------------
5800829   static bool
5800830   getnum (char **retcp, bool * rethavenum, num_t * retnum)
```

```
5800831 {
5800832   char *cp;
5800833   char *str;
5800834   bool havenum;
5800835   num_t value;
5800836   num_t num;
5800837   num_t sign;
5800838   //
5800839   cp = *retcp;
5800840   havenum = false;
5800841   value = 0;
5800842   sign = 1;
5800843   //
5800844   while (true)
5800845     {
5800846       while (isblank (*cp))
5800847         {
5800848           cp++;
5800849         }
5800850       //
5800851       switch (*cp)
5800852         {
5800853         case '.':
5800854           havenum = true;
5800855           num = curnum;
5800856           cp++;
5800857           break;
5800858           //
5800859         case '$':
5800860           havenum = true;
5800861           num = lastnum;
5800862           cp++;
5800863           break;
5800864           //
5800865         case '\'':
5800866           cp++;
5800867           if ((*cp < 'a') || (*cp > 'z'))
```

```
5800868                 {
5800869                   fprintf (stderr, "Bad mark name\n");
5800870                   return false;
5800871                 }
5800872               //
5800873               havenum = true;
5800874               num = marks[*cp++ - 'a'];
5800875               break;
5800876               //
5800877             case '/':
5800878               str = ++cp;
5800879               cp = strchr (str, '/');
5800880               if (cp)
5800881                 {
5800882                   *cp++ = '\0';
5800883                 }
5800884               else
5800885                 {
5800886                   cp = "";
5800887                 }
5800888               num = searchlines (str, curnum, lastnum);
5800889               if (num == 0)
5800890                 {
5800891                   return false;
5800892                 }
5800893               //
5800894               havenum = true;
5800895               break;
5800896               //
5800897             default:
5800898               if (!isdigit (*cp))
5800899                 {
5800900                   *retcp = cp;
5800901                   *rethavenum = havenum;
5800902                   *retnum = value;
5800903                   return true;
5800904                 }
```

```
5800905             //
5800906             num = 0;
5800907             while (isdigit (*cp))
5800908               {
5800909                 num = num * 10 + *cp++ - '0';
5800910               }
5800911             havenum = true;
5800912             break;
5800913           }
5800914         //
5800915         value += num * sign;
5800916         //
5800917         while (isblank (*cp))
5800918           {
5800919             cp++;
5800920           }
5800921         //
5800922         switch (*cp)
5800923           {
5800924           case '-':
5800925             sign = -1;
5800926             cp++;
5800927             break;
5800928             //
5800929           case '+':
5800930             sign = 1;
5800931             cp++;
5800932             break;
5800933             //
5800934           default:
5800935             *retcp = cp;
5800936             *rethavenum = havenum;
5800937             *retnum = value;
5800938             return true;
5800939           }
5800940       }
5800941   }
```

```
5800942
5800943 //-----------------------------------------------------------
5800944 // Initialize everything for editing.
5800945 //-----------------------------------------------------------
5800946 bool
5800947 initedit (void)
5800948 {
5800949   int i;
5800950   //
5800951   bufsize = INITBUFSIZE;
5800952   bufbase = malloc (bufsize);
5800953   //
5800954   if (bufbase == NULL)
5800955     {
5800956       fprintf (stderr, "No memory for buffer\n");
5800957       return false;
5800958     }
5800959   //
5800960   bufptr = bufbase;
5800961   bufused = 0;
5800962   //
5800963   lines.next = &lines;
5800964   lines.prev = &lines;
5800965   //
5800966   curline = NULL;
5800967   curnum = 0;
5800968   lastnum = 0;
5800969   dirty = false;
5800970   filename = NULL;
5800971   searchstring[0] = '\0';
5800972   //
5800973   for (i = 0; i < 26; i++)
5800974     {
5800975       marks[i] = 0;
5800976     }
5800977   //
5800978   return true;
```

```
5800979 }
5800980
5800981 //-----------------------------------------------------------
5800982 // Finish editing.
5800983 //-----------------------------------------------------------
5800984 void
5800985 termedit (void)
5800986 {
5800987   if (bufbase)
5800988     free (bufbase);
5800989   bufbase = NULL;
5800990   //
5800991   bufptr = NULL;
5800992   bufsize = 0;
5800993   bufused = 0;
5800994   //
5800995   if (filename)
5800996     free (filename);
5800997   filename = NULL;
5800998   //
5800999   searchstring[0] = '\0';
5801000   //
5801001   if (lastnum)
5801002     deletelines (1, lastnum);
5801003   //
5801004   lastnum = 0;
5801005   curnum = 0;
5801006   curline = NULL;
5801007 }
5801008
5801009 //-----------------------------------------------------------
5801010 // Read lines from a file at the specified line number.
5801011 // Returns true if the file was successfully read.
5801012 //-----------------------------------------------------------
5801013 bool
5801014 readlines (char *file, num_t num)
5801015 {
```

```
5801016     int fd;
5801017     int cc;
5801018     len_t len;
5801019     len_t linecount;
5801020     len_t charcount;
5801021     char *cp;
5801022     //
5801023     if ((num < 1) || (num > lastnum + 1))
5801024       {
5801025         fprintf (stderr, "Bad line for read\n");
5801026         return false;
5801027       }
5801028     //
5801029     fd = open (file, O_RDONLY);
5801030     if (fd < 0)
5801031       {
5801032         perror (file);
5801033         return false;
5801034       }
5801035     //
5801036     bufptr = bufbase;
5801037     bufused = 0;
5801038     linecount = 0;
5801039     charcount = 0;
5801040     //
5801041     printf ("\"%s\", ", file);
5801042     fflush (stdout);
5801043     //
5801044     do
5801045       {
5801046         cp = memchr (bufptr, '\n', bufused);
5801047         if (cp)
5801048           {
5801049             len = (cp - bufptr) + 1;
5801050             //
5801051             if (!insertline (num, bufptr, len))
5801052               {
```

```
5801053                 close (fd);
5801054                 return false;
5801055               }
5801056             //
5801057             bufptr += len;
5801058             bufused -= len;
5801059             charcount += len;
5801060             linecount++;
5801061             num++;
5801062             continue;
5801063           }
5801064         //
5801065         if (bufptr != bufbase)
5801066           {
5801067             memcpy (bufbase, bufptr, bufused);
5801068             bufptr = bufbase + bufused;
5801069           }
5801070         //
5801071         if (bufused >= bufsize)
5801072           {
5801073             len = (bufsize * 3) / 2;
5801074             cp = realloc (bufbase, len);
5801075             if (cp == NULL)
5801076               {
5801077                 fprintf (stderr, "No memory for buffer\n");
5801078                 close (fd);
5801079                 return false;
5801080               }
5801081             //
5801082             bufbase = cp;
5801083             bufptr = bufbase + bufused;
5801084             bufsize = len;
5801085           }
5801086         //
5801087         cc = read (fd, bufptr, bufsize - bufused);
5801088         bufused += cc;
5801089         bufptr = bufbase;
```

```
5801090       }
5801091     while (cc > 0);
5801092     //
5801093     if (cc < 0)
5801094       {
5801095         perror (file);
5801096         close (fd);
5801097         return false;
5801098       }
5801099     //
5801100     if (bufused)
5801101       {
5801102         if (!insertline (num, bufptr, bufused))
5801103           {
5801104             close (fd);
5801105             return -1;
5801106           }
5801107         linecount++;
5801108         charcount += bufused;
5801109       }
5801110     //
5801111     close (fd);
5801112     //
5801113     printf ("%d lines%s, %d chars\n",
5801114             linecount, (bufused ? " (incomplete)" : ""),
5801115             charcount);
5801116     //
5801117     return true;
5801118   }
5801119
5801120   //---------------------------------------------------------
5801121   // Write the specified lines out to the specified file.
5801122   // Returns true if successful, or false on an error
5801123   // with a message output.
5801124   //---------------------------------------------------------
5801125   bool
5801126   writelines (char *file, num_t num1, num_t num2)
```

```
5801127 {
5801128   int fd;
5801129   line_t *lp;
5801130   len_t linecount;
5801131   len_t charcount;
5801132   //
5801133   if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5801134     {
5801135       fprintf (stderr, "Bad line range for write\n");
5801136       return false;
5801137     }
5801138   //
5801139   linecount = 0;
5801140   charcount = 0;
5801141   //
5801142   fd = creat (file, 0666);
5801143   if (fd < 0)
5801144     {
5801145       perror (file);
5801146       return false;
5801147     }
5801148   //
5801149   printf ("\"%s\", ", file);
5801150   fflush (stdout);
5801151   //
5801152   lp = findline (num1);
5801153   if (lp == NULL)
5801154     {
5801155       close (fd);
5801156       return false;
5801157     }
5801158   //
5801159   while (num1++ <= num2)
5801160     {
5801161       if (write (fd, lp->data, lp->len) != lp->len)
5801162         {
5801163           perror (file);
```

```
5801164             close (fd);
5801165             return false;
5801166           }
5801167         //
5801168         charcount += lp->len;
5801169         linecount++;
5801170         lp = lp->next;
5801171       }
5801172     //
5801173     if (close (fd) < 0)
5801174       {
5801175         perror (file);
5801176         return false;
5801177       }
5801178     //
5801179     printf ("%d lines, %d chars\n", linecount, charcount);
5801180     //
5801181     return true;
5801182   }
5801183
5801184   //-------------------------------------------------------
5801185   // Print lines in a specified range.
5801186   // The last line printed becomes the current line.
5801187   // If expandflag is true, then the line is printed
5801188   // specially to show magic characters.
5801189   //-------------------------------------------------------
5801190   bool
5801191   printlines (num_t num1, num_t num2, bool expandflag)
5801192   {
5801193     line_t *lp;
5801194     unsigned char *cp;
5801195     int ch;
5801196     len_t count;
5801197     //
5801198     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5801199       {
5801200         fprintf (stderr, "Bad line range for print\n");
```

```
5801201         return false;
5801202       }
5801203     //
5801204     lp = findline (num1);
5801205     if (lp == NULL)
5801206       {
5801207         return false;
5801208       }
5801209     //
5801210     while (num1 <= num2)
5801211       {
5801212         if (!expandflag)
5801213           {
5801214             write (STDOUT_FILENO, lp->data, lp->len);
5801215             setcurnum (num1++);
5801216             lp = lp->next;
5801217             continue;
5801218           }
5801219
5801220         // ---------------------------------------------
5801221         // Show control characters and characters with
5801222         // the high bit set specially.
5801223         // ---------------------------------------------
5801224
5801225         cp = (unsigned char *) lp->data;
5801226         count = lp->len;
5801227         //
5801228         if ((count > 0) && (cp[count - 1] == '\n'))
5801229           {
5801230             count--;
5801231           }
5801232         //
5801233         while (count-- > 0)
5801234           {
5801235             ch = *cp++;
5801236             if (ch & 0x80)
5801237               {
```

```
5801238                 fputs ("M-", stdout);
5801239                 ch &= 0x7f;
5801240               }
5801241             if (ch < ' ')
5801242               {
5801243                 fputc ('^', stdout);
5801244                 ch += '@';
5801245               }
5801246             if (ch == 0x7f)
5801247               {
5801248                 fputc ('^', stdout);
5801249                 ch = '?';
5801250               }
5801251             fputc (ch, stdout);
5801252           }
5801253         //
5801254         fputs ("$\n", stdout);
5801255         //
5801256         setcurnum (num1++);
5801257         lp = lp->next;
5801258       }
5801259     //
5801260     return true;
5801261   }
5801262
5801263   //-------------------------------------------------------
5801264   // Insert a new line with the specified text.
5801265   // The line is inserted so as to become the specified
5801266   // line, thus pushing any existing and further lines
5801267   // down one.
5801268   // The inserted line is also set to become the current
5801269   // line.
5801270   // Returns true if successful.
5801271   //-------------------------------------------------------
5801272   bool
5801273   insertline (num_t num, char *data, len_t len)
5801274   {
```

```
5801275     line_t *newlp;
5801276     line_t *lp;
5801277     //
5801278     if ((num < 1) || (num > lastnum + 1))
5801279       {
5801280         fprintf (stderr, "Inserting at bad line number\n");
5801281         return false;
5801282       }
5801283     //
5801284     newlp = (line_t *) malloc (sizeof (line_t) + len - 1);
5801285     if (newlp == NULL)
5801286       {
5801287         fprintf (stderr,
5801288                  "Failed to allocate memory for line\n");
5801289         return false;
5801290       }
5801291     //
5801292     memcpy (newlp->data, data, len);
5801293     newlp->len = len;
5801294     //
5801295     if (num > lastnum)
5801296       {
5801297         lp = &lines;
5801298       }
5801299     else
5801300       {
5801301         lp = findline (num);
5801302         if (lp == NULL)
5801303           {
5801304             free ((char *) newlp);
5801305             return false;
5801306           }
5801307       }
5801308     //
5801309     newlp->next = lp;
5801310     newlp->prev = lp->prev;
5801311     lp->prev->next = newlp;
```

```
5801312   lp->prev = newlp;
5801313   //
5801314   lastnum++;
5801315   dirty = true;
5801316   //
5801317   return setcurnum (num);
5801318 }
5801319
5801320 //-------------------------------------------------------
5801321 // Delete lines from the given range.
5801322 //-------------------------------------------------------
5801323 bool
5801324 deletelines (num_t num1, num_t num2)
5801325 {
5801326   line_t *lp;
5801327   line_t *nlp;
5801328   line_t *plp;
5801329   num_t count;
5801330   //
5801331   if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5801332     {
5801333       fprintf (stderr, "Bad line numbers for delete\n");
5801334       return false;
5801335     }
5801336   //
5801337   lp = findline (num1);
5801338   if (lp == NULL)
5801339     {
5801340       return false;
5801341     }
5801342   //
5801343   if ((curnum >= num1) && (curnum <= num2))
5801344     {
5801345       if (num2 < lastnum)
5801346         {
5801347           setcurnum (num2 + 1);
5801348         }
```

```
5801349         else if (num1 > 1)
5801350           {
5801351             setcurnum (num1 - 1);
5801352           }
5801353         else
5801354           {
5801355             curnum = 0;
5801356           }
5801357       }
5801358     //
5801359     count = num2 - num1 + 1;
5801360     //
5801361     if (curnum > num2)
5801362       {
5801363         curnum -= count;
5801364       }
5801365     //
5801366     lastnum -= count;
5801367     //
5801368     while (count-- > 0)
5801369       {
5801370         nlp = lp->next;
5801371         plp = lp->prev;
5801372         plp->next = nlp;
5801373         nlp->prev = plp;
5801374         lp->next = NULL;
5801375         lp->prev = NULL;
5801376         lp->len = 0;
5801377         free (lp);
5801378         lp = nlp;
5801379       }
5801380     //
5801381     dirty = true;
5801382     //
5801383     return true;
5801384   }
5801385
```

```
5801386 //-----------------------------------------------------------
5801387 // Search for a line which contains the specified
5801388 // string.
5801389 // If the string is NULL, then the previously searched
5801390 // for string is used.  The currently searched for
5801391 // string is saved for future use.
5801392 // Returns the line number which matches, or 0 if there
5801393 // was no match with an error printed.
5801394 //-----------------------------------------------------------
5801395 num_t
5801396 searchlines (char *str, num_t num1, num_t num2)
5801397 {
5801398   line_t *lp;
5801399   int len;
5801400   //
5801401   if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5801402     {
5801403       fprintf (stderr, "Bad line numbers for search\n");
5801404       return 0;
5801405     }
5801406   //
5801407   if (*str == '\0')
5801408     {
5801409       if (searchstring[0] == '\0')
5801410         {
5801411           fprintf (stderr, "No previous search string\n");
5801412           return 0;
5801413         }
5801414       str = searchstring;
5801415     }
5801416   //
5801417   if (str != searchstring)
5801418     {
5801419       strcpy (searchstring, str);
5801420     }
5801421   //
5801422   len = strlen (str);
```

```
5801423     //
5801424     lp = findline (num1);
5801425     if (lp == NULL)
5801426       {
5801427         return 0;
5801428       }
5801429     //
5801430     while (num1 <= num2)
5801431       {
5801432         if (findstring (lp, str, len, 0) >= 0)
5801433           {
5801434             return num1;
5801435           }
5801436         //
5801437         num1++;
5801438         lp = lp->next;
5801439       }
5801440     //
5801441     fprintf (stderr, "Cannot find string \"%s\"\n", str);
5801442     //
5801443     return 0;
5801444   }
5801445
5801446   //-------------------------------------------------------
5801447   // Return a pointer to the specified line number.
5801448   //-------------------------------------------------------
5801449   line_t *
5801450   findline (num_t num)
5801451   {
5801452     line_t *lp;
5801453     num_t lnum;
5801454     //
5801455     if ((num < 1) || (num > lastnum))
5801456       {
5801457         fprintf (stderr,
5801458                  "Line number %d does not exist\n", num);
5801459         return NULL;
```

```
5801460       }
5801461     //
5801462     if (curnum <= 0)
5801463       {
5801464         curnum = 1;
5801465         curline = lines.next;
5801466       }
5801467     //
5801468     if (num == curnum)
5801469       {
5801470         return curline;
5801471       }
5801472     //
5801473     lp = curline;
5801474     lnum = curnum;
5801475     //
5801476     if (num < (curnum / 2))
5801477       {
5801478         lp = lines.next;
5801479         lnum = 1;
5801480       }
5801481     else if (num > ((curnum + lastnum) / 2))
5801482       {
5801483         lp = lines.prev;
5801484         lnum = lastnum;
5801485       }
5801486     //
5801487     while (lnum < num)
5801488       {
5801489         lp = lp->next;
5801490         lnum++;
5801491       }
5801492     //
5801493     while (lnum > num)
5801494       {
5801495         lp = lp->prev;
5801496         lnum--;
```

```
5801497       }
5801498     //
5801499     return lp;
5801500   }
5801501
5801502   //-------------------------------------------------------
5801503   // Set the current line number.
5801504   // Returns true if successful.
5801505   //-------------------------------------------------------
5801506   bool
5801507   setcurnum (num_t num)
5801508   {
5801509     line_t *lp;
5801510     //
5801511     lp = findline (num);
5801512     if (lp == NULL)
5801513       {
5801514         return false;
5801515       }
5801516     //
5801517     curnum = num;
5801518     curline = lp;
5801519     //
5801520     return true;
5801521   }
5801522
5801523   /* END CODE */
```

96.1.16 applic/getty.c

«

Si veda la sezione 92.2.

```
5810001   #include <unistd.h>
5810002   #include <stdio.h>
5810003   #include <stdlib.h>
5810004   #include <signal.h>
5810005   #include <sys/wait.h>
```

```
5810006 #include <limits.h>
5810007 #include <sys/os32.h>
5810008 #include <fcntl.h>
5810009 #include <stdio.h>
5810010 //------------------------------------------------------------
5810011 int
5810012 main (int argc, char *argv[], char *envp[])
5810013 {
5810014   char *device_name;
5810015   int fdn;
5810016   char *exec_argv[2];
5810017   char **exec_envp;
5810018   char buffer[BUFSIZ];
5810019   ssize_t size_read;
5810020   int status;
5810021   //
5810022   // The first argument is mandatory and must be a
5810023   // console terminal.
5810024   //
5810025   device_name = argv[1];
5810026   //
5810027   // A console terminal is correctly selected (but it
5810028   // is not checked
5810029   // if it is a really available one).
5810030   // Set as a process group leader.
5810031   //
5810032   setpgrp ();
5810033   //
5810034   // Open the terminal, that should become the
5810035   // controlling terminal:
5810036   // close the standard input and open the new
5810037   // terminal (r/w).
5810038   //
5810039   close (0);
5810040   fdn = open (device_name, O_RDWR);
5810041   if (fdn < 0)
5810042     {
```

```
5810043         //
5810044         // Cannot open terminal. A message should
5810045         // appear, at least
5810046         // to the current console.
5810047         //
5810048         perror (NULL);
5810049         return (-1);
5810050       }
5810051     //
5810052     // Reset terminal device permissions and ownership.
5810053     //
5810054     status = fchown (fdn, (uid_t) 0, (gid_t) 0);
5810055     if (status != 0)
5810056       {
5810057         perror (NULL);
5810058       }
5810059     status = fchmod (fdn, 0644);
5810060     if (status != 0)
5810061       {
5810062         perror (NULL);
5810063       }
5810064     //
5810065     // The terminal is open and it should be already the
5810066     // controlling
5810067     // one: show '/etc/issue'. The same variable 'fdn'
5810068     // is used, because
5810069     // the controlling terminal will never be closed
5810070     // (the exit syscall
5810071     // will do it).
5810072     //
5810073     fdn = open ("/etc/issue", O_RDONLY);
5810074     if (fdn > 0)
5810075       {
5810076         //
5810077         // The file is present and is shown.
5810078         //
5810079         for (size_read = 1; size_read > 0;)
```

```
5810080           {
5810081             size_read =
5810082               read (fdn, buffer, (size_t) (BUFSIZ - 1));
5810083             if (size_read < 0)
5810084               {
5810085                 break;
5810086               }
5810087             buffer[size_read] = '\0';
5810088             printf ("%s", buffer);
5810089           }
5810090         close (fdn);
5810091       }
5810092     //
5810093     // Show the terminal.
5810094     //
5810095     printf ("This is terminal %s\n", device_name);
5810096     //
5810097     // It is time to exec login: the environment is
5810098     // inherited directly
5810099     // from 'init'.
5810100     //
5810101     exec_argv[0] = "login";
5810102     exec_argv[1] = NULL;
5810103     exec_envp = envp;
5810104     execve ("/bin/login", exec_argv, exec_envp);
5810105     //
5810106     // If 'execve()' returns, it is an error.
5810107     //
5810108     exit (-1);
5810109   }
```

96.1.17 applic/http.c

«

Si veda la sezione 92.3.

```
5820001 #include <sys/stat.h>
5820002 #include <sys/types.h>
```

```
5820003 #include <unistd.h>
5820004 #include <stdlib.h>
5820005 #include <fcntl.h>
5820006 #include <errno.h>
5820007 #include <signal.h>
5820008 #include <stdio.h>
5820009 #include <string.h>
5820010 #include <limits.h>
5820011 #include <libgen.h>
5820012 #include <arpa/inet.h>
5820013 #include <sys/socket.h>
5820014 #include <stdint.h>
5820015 #include <stdbool.h>
5820016 //------------------------------------------------------------
5820017 #define DEBUG 0
5820018 static void usage (void);
5820019 static int send_file (int sfdn2, const char *path);
5820020 static int send_line (int sfdn2, const char *line);
5820021 char buffer[BUFSIZ];
5820022 char path_absolute[PATH_MAX];
5820023 //------------------------------------------------------------
5820024 int
5820025 main (int argc, char *argv[], char *envp[])
5820026 {
5820027   int opt;
5820028 //extern char *optarg;                // not used.
5820029   extern int optind;
5820030   extern int optopt;
5820031   //
5820032   int status;
5820033   int sfdn;
5820034   int sfdn2;
5820035   struct sockaddr_in sa_local;
5820036   struct sockaddr_in sa_remote;
5820037   socklen_t sa_remote_size = sizeof (struct sockaddr_in);
5820038   ssize_t recv_size;
5820039   char *addr = "0.0.0.0";
```

```
5820040     char *www = NULL;
5820041     char *path = NULL;
5820042     int port;
5820043     bool request_read;
5820044     int b;          // index inside the buffer string
5820045     // buffer
5820046     char *string = NULL;
5820047     struct stat file_status;
5820048     //
5820049     // Check for options: no options at the moment.
5820050     //
5820051     while ((opt = getopt (argc, argv, ":")) != -1)
5820052       {
5820053         switch (opt)
5820054           {
5820055           case '?':
5820056             fprintf (stderr, "Unknown option -%c.\n", optopt);
5820057             usage ();
5820058             return (1);
5820059             break;
5820060           case ':':
5820061             fprintf (stderr,
5820062                      "Missing argument for option -%c\n",
5820063                      optopt);
5820064             usage ();
5820065             return (1);
5820066             break;
5820067           default:
5820068             fprintf (stderr,
5820069                      "Getopt problem: "
5820070                      "unknown option %c\n", opt);
5820071             usage ();
5820072             return (1);
5820073           }
5820074       }
5820075     //
5820076     // Arguments.
```

```
5820077     //
5820078     if (optind == (argc - 2))
5820079       {
5820080         //
5820081         // There are exactly two arguments: the port and
5820082         // the www root path.
5820083         //
5820084         port = atoi (argv[argc - 2]);
5820085         www = argv[argc - 1];
5820086       }
5820087     else
5820088       {
5820089         //
5820090         // Arguments wrong!
5820091         //
5820092         printf ("optind = %i = %s, argc = %i\n", optind,
5820093                 argv[optind], argc);
5820094         usage ();
5820095         return (2);
5820096       }
5820097     //
5820098     // Set the local address.
5820099     //
5820100     sa_local.sin_family = AF_INET;
5820101     sa_local.sin_port = htons (port);
5820102     inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
5820103     //
5820104     // Open the socket.
5820105     //
5820106     sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
5820107     if (sfdn < 0)
5820108       {
5820109         perror (NULL);
5820110         return (3);
5820111       }
5820112     if (DEBUG)
5820113       {
```

```
5820114         printf ("HTTP: listening socket number "
5820115                 "is %i.\n", sfdn);
5820116       }
5820117     //
5820118     // Set it listening: bind the local 'sa' location.
5820119     //
5820120     status = bind (sfdn, (struct sockaddr *) &sa_local,
5820121                    sizeof (sa_local));
5820122     if (status < 0)
5820123       {
5820124         perror (NULL);
5820125         close (sfdn);
5820126         return (4);
5820127       }
5820128     //
5820129     // Listen (TCP).
5820130     //
5820131     status = listen (sfdn, 1);
5820132     if (status < 0)
5820133       {
5820134         perror (NULL);
5820135         close (sfdn);
5820136         return (5);
5820137       }
5820138     //
5820139     // Accept connections, inside a loop.
5820140     //
5820141     while (1)
5820142       {
5820143         //
5820144         // Accept.
5820145         //
5820146         if (DEBUG)
5820147           {
5820148             printf
5820149               ("HTTP: listening socket number is %i.\n",
5820150                sfdn);
```

```
5820151           }
5820152         //
5820153         sfdn2 =
5820154           accept (sfdn, (struct sockaddr *) &sa_remote,
5820155                   &sa_remote_size);
5820156         //
5820157         if (sfdn2 < 0)
5820158           {
5820159             perror (NULL);
5820160             close (sfdn2);
5820161             continue;
5820162           }
5820163         if (DEBUG)
5820164           {
5820165             printf
5820166               ("HTTP: new connection with socket "
5820167                "number %i.\n", sfdn2);
5820168           }
5820169         //
5820170         // Define the socket non blocking.
5820171         //
5820172         status = fcntl (sfdn2, F_SETFL, O_NONBLOCK);
5820173         if (status < 0)
5820174           {
5820175             perror (NULL);
5820176             return (9);
5820177           }
5820178         //
5820179         // Will read from the remote.
5820180         //
5820181         path_absolute[0] = 0;
5820182         request_read = 1;
5820183         while (request_read)
5820184           {
5820185             //
5820186             // Read a line from the remote side.
5820187             //
```

```
5820188             for (b = 0; b < (BUFSIZ - 2); b++, buffer[b] = 0)
5820189               {
5820190                 //
5820191                 // Read a single character from the
5820192                 // remote side.
5820193                 //
5820194                 recv_size =
5820195                   recv (sfdn2, &buffer[b], (size_t) 1, 0);
5820196                 //
5820197                 if (recv_size < 0)
5820198                   {
5820199                     if (errno == EAGAIN
5820200                         || errno == EWOULDBLOCK)
5820201                       {
5820202                         b--;
5820203                         continue;
5820204                       }
5820205                     else
5820206                       {
5820207                         perror (NULL);
5820208                         close (sfdn2);
5820209                         continue;
5820210                       }
5820211                   }
5820212                 if (recv_size == 0)
5820213                   {
5820214                     //
5820215                     // It is the end of stream, but
5820216                     // should not happen.
5820217                     //
5820218                     buffer[b] = 0;
5820219                     request_read = 0;
5820220                     if (DEBUG)
5820221                       {
5820222                         printf ("HTTP: end of stream, "
5820223                                 "but should not "
5820224                                 "happen here! "
```

```
5820225                                 "\"%s\"\n", buffer);
5820226                           }
5820227                         break;
5820228                       }
5820229                     //
5820230                     if (buffer[b] == '\r')
5820231                       {
5820232                         //
5820233                         // Ignore CR.
5820234                         //
5820235                         b--;
5820236                         continue;
5820237                       }
5820238                     //
5820239                     if (buffer[b] == '\n')
5820240                       {
5820241                         //
5820242                         // End of line.
5820243                         //
5820244                         buffer[b] = 0;
5820245                         if (DEBUG)
5820246                           {
5820247                             printf ("HTTP: %s\n", buffer);
5820248                           }
5820249                         break;
5820250                       }
5820251                   }
5820252                 //
5820253                 // Was it the end of the header?
5820254                 //
5820255                 if (strlen (buffer) == 0)
5820256                   {
5820257                     //
5820258                     // End of header.
5820259                     //
5820260                     request_read = 0;
5820261                     break;
```

```
5820262                 }
5820263               //
5820264               // We are reading the header: was it the GET
5820265               // command?
5820266               //
5820267               string = strtok (buffer, " ");
5820268               if (strncmp (string, "GET", 4) == 0)
5820269                 {
5820270                   //
5820271                   // It is a GET: find the path.
5820272                   //
5820273                   path = strtok (NULL, " ");
5820274                   strncat (path_absolute, www, PATH_MAX - 1);
5820275                   strncat (path_absolute, path,
5820276                            (PATH_MAX -
5820277                             strlen (path_absolute) - 1));
5820278                 }
5820279             }
5820280           //
5820281           // Verify to have received a 'GET' request.
5820282           //
5820283           if (strlen (path_absolute) == 0)
5820284             {
5820285               //
5820286               // There is no path inside the GET command;
5820287               // maybe there is
5820288               // no GET command either: 400
5820289               //
5820290               if (DEBUG)
5820291                 {
5820292                   printf ("HTTP: 400 Bad Request: "
5820293                           "no path inside the GET "
5820294                           "command.\n");
5820295                 }
5820296               send_line (sfdn2, "HTTP/1.0 400 Bad Request\r\n");
5820297               send_line (sfdn2, "Content-Type: text/html\r\n");
5820298               send_line (sfdn2, "Content-Length: 26\r\n");
```

```
5820299             send_line (sfdn2, "\r\n");
5820300             send_line (sfdn2, "<H1>400 Bad Request</H1>\r\n");
5820301           }
5820302         //
5820303         // Verify the path.
5820304         //
5820305         if (stat (path_absolute, &file_status) != 0)
5820306           {
5820307             //
5820308             // The path inside the GET command does not
5820309             // exists: 404
5820310             //
5820311             if (DEBUG)
5820312               {
5820313                 printf ("HTTP: 404 Not Found: "
5820314                         "the path \"%s\" does not "
5820315                         "exists.\n", path_absolute);
5820316               }
5820317             send_line (sfdn2, "HTTP/1.0 404 Not Found\r\n");
5820318             send_line (sfdn2, "Content-Type: text/html\r\n");
5820319             send_line (sfdn2, "Content-Length: 24\r\n");
5820320             send_line (sfdn2, "\r\n");
5820321             send_line (sfdn2, "<H1>404 Not Found</H1>\r\n");
5820322           }
5820323         else
5820324           {
5820325             //
5820326             // File exists: check the file type.
5820327             //
5820328             if (S_ISDIR (file_status.st_mode))
5820329               {
5820330                 //
5820331                 // Test to find 'index.html'.
5820332                 //
5820333                 strncat (path_absolute, "index.html",
5820334                          (PATH_MAX -
5820335                           strlen (path_absolute) - 1));
```

```
5820336                 //
5820337                 if (stat (path_absolute, &file_status) != 0)
5820338                   {
5820339                     //
5820340                     // The index file inside the path
5820341                     // requested
5820342                     // does not exists: 404
5820343                     //
5820344                     if (DEBUG)
5820345                       {
5820346                         printf ("HTTP: 404 Not Found: "
5820347                                 "the path \"%s\" does "
5820348                                 "not exists.\n",
5820349                                 path_absolute);
5820350                       }
5820351                     send_line (sfdn2,
5820352                                "HTTP/1.0 404 Not "
5820353                                "Found\r\n");
5820354                     send_line (sfdn2,
5820355                                "Content-Type: "
5820356                                "text/html\r\n");
5820357                     send_line (sfdn2,
5820358                                "Content-Length: 24\r\n");
5820359                     send_line (sfdn2, "\r\n");
5820360                     send_line (sfdn2,
5820361                                "<H1>404 Not Found"
5820362                                "</H1>\r\n");
5820363                   }
5820364               }
5820365             //
5820366             // There is a file to send.
5820367             //
5820368             send_file (sfdn2, path_absolute);
5820369           }
5820370         //
5820371         // The socket `sfdn2' might be already closed;
5820372         // if so, the variable was reset to zero.
```

```
5820373         //
5820374         if (sfdn2 != 0)
5820375           close (sfdn2);
5820376         buffer[0] = 0;
5820377         if (DEBUG)
5820378           {
5820379             printf
5820380               ("HTTP: connection closed: continue "
5820381                "listening.\n");
5820382           }
5820383         continue;
5820384       }
5820385     //
5820386     // All done.
5820387     //
5820388     close (sfdn);
5820389     return (0);
5820390   }
5820391
5820392   //-------------------------------------------------------
5820393   static int
5820394   send_file (int sfdn2, const char *path)
5820395   {
5820396   //  size_t sent_size;
5820397     size_t file_size;
5820398     struct stat file_status;
5820399     char ascii_size[32];
5820400     int fdn;
5820401     char *buffer_in = buffer;
5820402     char *buffer_out;
5820403     ssize_t count_in;      // Read counter.
5820404     ssize_t count_out;     // Write counter.
5820405     //
5820406     if (sfdn2 == 0)
5820407       return (-1);
5820408     //
5820409     if (stat (path, &file_status) != 0)
```

```
5820410       {
5820411         perror (NULL);
5820412         close (sfdn2);
5820413         sfdn2 = 0;
5820414         return (-1);
5820415       }
5820416     //
5820417     file_size = file_status.st_size;
5820418     sprintf (ascii_size, "%i", file_size);
5820419     //
5820420     fdn = open (path, O_RDONLY);
5820421     if (fdn < 0)
5820422       {
5820423         if (DEBUG)
5820424           {
5820425             printf ("HTTP: 403 Forbidden: %s ", path);
5820426           }
5820427         perror (path);
5820428         send_line (sfdn2, "HTTP/1.0 403 Forbidden\r\n");
5820429         send_line (sfdn2, "Content-Type: text/html\r\n");
5820430         send_line (sfdn2, "Content-Length: 24\r\n");
5820431         send_line (sfdn2, "\r\n");
5820432         send_line (sfdn2, "<H1>404 Forbidden</H1>\r\n");
5820433         close (sfdn2);
5820434         sfdn2 = 0;
5820435         return (-1);
5820436       }
5820437     //
5820438     send_line (sfdn2, "HTTP/1.0 200 OK\r\n");
5820439     send_line (sfdn2, "Content-Type: text/html\r\n");
5820440     send_line (sfdn2, "Content-Length: ");
5820441     send_line (sfdn2, ascii_size);
5820442     send_line (sfdn2, "\r\n");
5820443     send_line (sfdn2, "\r\n");
5820444     //
5820445     // Copy the data.
5820446     //
```

```
5820447     while (1)
5820448       {
5820449         count_in = read (fdn, buffer_in, (size_t) BUFSIZ);
5820450         if (count_in > 0)
5820451           {
5820452             for (buffer_out = buffer_in; count_in > 0;)
5820453               {
5820454                 count_out = send (sfdn2, buffer_out,
5820455                                   (size_t) count_in, 0);
5820456                 if (count_out < 0)
5820457                   {
5820458                     if (errno == EAGAIN
5820459                         || errno == EWOULDBLOCK)
5820460                       {
5820461                         continue;
5820462                       }
5820463                     else
5820464                       {
5820465                         fprintf (stderr,
5820466                                  "[HTTP] cannot "
5820467                                  "send 1!\n");
5820468                         perror (NULL);
5820469                         close (fdn);
5820470                         close (sfdn2);
5820471                         sfdn2 = 0;
5820472                         return (-1);
5820473                       }
5820474                   }
5820475                 //
5820476                 // If not all data is written, continue
5820477                 // writing, but change the buffer start
5820478                 // position and the
5820479                 // amount to be written.
5820480                 //
5820481                 buffer_out += count_out;
5820482                 count_in -= count_out;
5820483               }
```

```
5820484           }
5820485         else if (count_in < 0)
5820486           {
5820487             perror (path);
5820488             close (fdn);
5820489             close (sfdn2);
5820490             sfdn2 = 0;
5820491             return (-1);
5820492           }
5820493         else
5820494           {
5820495             break;
5820496           }
5820497       }
5820498     //
5820499     return (0);
5820500   }
5820501
5820502   //-------------------------------------------------------
5820503   static int
5820504   send_line (int sfdn2, const char *line)
5820505   {
5820506     size_t sent_size;
5820507     size_t line_size = strlen (line);
5820508     const char *start = line;
5820509     //
5820510     if (sfdn2 == 0)
5820511       return (-1);
5820512     //
5820513     while (1)
5820514       {
5820515         errno = 0;
5820516         sent_size =
5820517           send (sfdn2, start, (size_t) line_size, 0);
5820518         //
5820519         //
5820520         //
```

```
5820521         if (DEBUG)
5820522           {
5820523             printf
5820524               ("[HTTP] line_size=%i, sent_size=%i, "
5820525                "error=%i\n",
5820526                (int) line_size, (int) sent_size, errno);
5820527           }
5820528         if (sent_size < 0)
5820529           {
5820530             if (errno == EAGAIN || errno == EWOULDBLOCK)
5820531               {
5820532                 continue;
5820533               }
5820534             else
5820535               {
5820536                 fprintf (stderr, "[HTTP] cannot send 2!\n");
5820537                 perror (NULL);
5820538                 close (sfdn2);
5820539                 sfdn2 = 0;
5820540                 return (-1);
5820541               }
5820542           }
5820543         //
5820544         //
5820545         //
5820546         if (sent_size < line_size)
5820547           {
5820548             start = &start[sent_size];
5820549             line_size -= sent_size;
5820550             //
5820551             continue;
5820552           }
5820553         return (0);
5820554       }
5820555   }
5820556
5820557   //------------------------------------------------------
```

```
5820558 static void
5820559 usage (void)
5820560 {
5820561   fprintf (stderr,
5820562            "os32 http usage:\n"
5820563            "\n"
5820564            "http PORT WWW_ROOT_PATH\n"
5820565            "\n"
5820566            "PORT port number listening for "
5820567            "connections"
5820568            "\n"
5820569            "WWW_ROOT_PATH root for the published "
5820570            "documents." "\n");
5820571 }
```

96.1.18 applic/init.c

« Si veda la sezione 92.4.

```
5830001 #include <unistd.h>
5830002 #include <stdio.h>
5830003 #include <stdlib.h>
5830004 #include <signal.h>
5830005 #include <sys/wait.h>
5830006 #include <limits.h>
5830007 #include <sys/os32.h>
5830008 #include <fcntl.h>
5830009 #include <string.h>
5830010 //-------------------------------------------------------
5830011 #define RESPAWN_MAX        7
5830012 #define COMMAND_MAX      100
5830013 #define ARGUMENTS_MAX     32
5830014 #define LINE_MAX        1024
5830015 //-------------------------------------------------------
5830016 int
5830017 main (int argc, char *argv[], char *envp[])
5830018 {
```

```
5830019     //
5830020     // 'init.c' has its own 'init.crt0.s' with a very
5830021     // small stack
5830022     // size. Remember to verify to have enough room for
5830023     // the stack.
5830024     //
5830025     pid_t pid;
5830026     int status;
5830027     char *exec_argv[ARGUMENTS_MAX];
5830028     int count;
5830029     char *exec_envp[3];
5830030     char buffer[LINE_MAX];
5830031     int r;          // Respawn table index.
5830032     int b;          // Buffer index.
5830033     size_t size_read;
5830034     char *inittab_id;
5830035     char *inittab_runlevels;
5830036     char *inittab_action;
5830037     char *inittab_process;
5830038     int eof;
5830039     int fd;
5830040     //
5830041     // It follows a table for commands to be respawn.
5830042     //
5830043     struct
5830044     {
5830045       pid_t pid;
5830046       char command[COMMAND_MAX];
5830047     } respawn[RESPAWN_MAX];
5830048
5830049     // ------------------------------------------------
5830050     signal (SIGHUP, SIG_IGN);
5830051     signal (SIGINT, SIG_IGN);
5830052     signal (SIGQUIT, SIG_IGN);
5830053     signal (SIGILL, SIG_IGN);
5830054     signal (SIGABRT, SIG_IGN);
5830055     signal (SIGFPE, SIG_IGN);
```

```
5830056   //  signal (SIGKILL, SIG_IGN);  Cannot ignore SIGKILL.
5830057     signal (SIGSEGV, SIG_IGN);
5830058     signal (SIGPIPE, SIG_IGN);
5830059     signal (SIGALRM, SIG_IGN);
5830060     signal (SIGTERM, SIG_IGN);
5830061   //  signal (SIGSTOP, SIG_IGN);  Cannot ignore SIGSTOP.
5830062     signal (SIGTSTP, SIG_IGN);
5830063     signal (SIGCONT, SIG_IGN);
5830064     signal (SIGTTIN, SIG_IGN);
5830065     signal (SIGTTOU, SIG_IGN);
5830066     signal (SIGUSR1, SIG_IGN);
5830067     signal (SIGUSR2, SIG_IGN);
5830068     // ------------------------------------------------
5830069     printf ("init\n");
5830070     // heap_clear ();
5830071     // process_info ();
5830072     // ------------------------------------------------
5830073     //
5830074     // Reset the 'respawn' table.
5830075     //
5830076     for (r = 0; r < RESPAWN_MAX; r++)
5830077       {
5830078         respawn[r].pid = 0;
5830079         respawn[r].command[0] = 0;
5830080         respawn[r].command[COMMAND_MAX - 1] = 0;
5830081       }
5830082     //
5830083     // Read the '/etc/inittab' file.
5830084     //
5830085     fd = open ("/etc/inittab", O_RDONLY);
5830086     //
5830087     if (fd < 0)
5830088       {
5830089         perror ("Cannot open file '/etc/inittab'");
5830090         exit (-1);
5830091       }
5830092     //
```

```
5830093     //
5830094     //
5830095     for (eof = 0, r = 0; !eof && r < RESPAWN_MAX; r++)
5830096       {
5830097         for (b = 0; b < LINE_MAX; b++)
5830098           {
5830099             size_read = read (fd, &buffer[b], (size_t) 1);
5830100             if (size_read <= 0)
5830101               {
5830102                 buffer[b] = 0;
5830103                 eof = 1;  // Close the read loop.
5830104                 break;
5830105               }
5830106             if (buffer[b] == '\n')
5830107               {
5830108                 buffer[b] = 0;
5830109                 break;
5830110               }
5830111           }
5830112         //
5830113         // Remove comments: just replace '#' with '\0'.
5830114         //
5830115         for (b = 0; b < LINE_MAX; b++)
5830116           {
5830117             if (buffer[b] == '#')
5830118               {
5830119                 buffer[b] = 0;
5830120                 break;
5830121               }
5830122           }
5830123         //
5830124         // If the buffer is an empty string, just loop
5830125         // to next
5830126         // record.
5830127         //
5830128         if (strlen (buffer) == 0)
5830129           {
```

```
5830130             r--;
5830131             continue;
5830132           }
5830133         //
5830134         //
5830135         //
5830136         inittab_id = strtok (buffer, ":");
5830137         inittab_runlevels = strtok (NULL, ":");
5830138         inittab_action = strtok (NULL, ":");
5830139         inittab_process = strtok (NULL, ":");
5830140         //
5830141         // Only action 'respawn' is used.
5830142         //
5830143         if (strcmp (inittab_action, "respawn") == 0)
5830144           {
5830145             strncpy (respawn[r].command, inittab_process,
5830146                      COMMAND_MAX);
5830147           }
5830148         else
5830149           {
5830150             r--;
5830151           }
5830152       }
5830153     //
5830154     //
5830155     //
5830156     close (fd);
5830157     //
5830158     // Define common environment.
5830159     //
5830160     exec_envp[0] = "PATH=/bin:/usr/bin:/sbin:/usr/sbin";
5830161     exec_envp[1] = "CONSOLE=/dev/console";
5830162     exec_envp[2] = NULL;
5830163     //
5830164     // Start processes.
5830165     //
5830166     for (r = 0; r < RESPAWN_MAX; r++)
```

```
5830167     {
5830168       if (strlen (respawn[r].command) > 0)
5830169         {
5830170           respawn[r].pid = fork ();
5830171           if (respawn[r].pid == 0)
5830172             {
5830173               exec_argv[0] =
5830174                 strtok (respawn[r].command, " \t");
5830175               for (count = 1;
5830176                    count < (ARGUMENTS_MAX - 2); count++)
5830177                 {
5830178                   exec_argv[count] = strtok (NULL, " \t");
5830179                   if (exec_argv[count] == NULL)
5830180                     {
5830181                       break;
5830182                     }
5830183                 }
5830184               //
5830185               // Last element must be NULL, even if
5830186               // there are more
5830187               // arguments than allowed.
5830188               //
5830189               exec_argv[count] = NULL;
5830190               //
5830191               // Run!
5830192               //
5830193               execve (exec_argv[0], exec_argv, exec_envp);
5830194               perror (NULL);
5830195               exit (0);
5830196             }
5830197         }
5830198     }
5830199   //
5830200   // Wait for the death of child.
5830201   //
5830202   while (1)
5830203     {
```

```
5830204         pid = wait (&status);
5830205         for (r = 0; r < RESPAWN_MAX; r++)
5830206           {
5830207             if (pid == respawn[r].pid)
5830208               {
5830209                 //
5830210                 // Run it again.
5830211                 //
5830212                 respawn[r].pid = fork ();
5830213                 if (respawn[r].pid == 0)
5830214                   {
5830215                     exec_argv[0] =
5830216                       strtok (respawn[r].command, " \t");
5830217                     for (count = 1;
5830218                          count < (ARGUMENTS_MAX - 2); count++)
5830219                       {
5830220                         exec_argv[count] =
5830221                           strtok (NULL, " \t");
5830222                         if (exec_argv[count] == NULL)
5830223                           {
5830224                             break;
5830225                           }
5830226                       }
5830227                     //
5830228                     // Last element must be NULL, even
5830229                     // if there are more
5830230                     // arguments than allowed.
5830231                     //
5830232                     exec_argv[count] = NULL;
5830233                     //
5830234                     // Run!
5830235                     //
5830236                     execve (exec_argv[0], exec_argv,
5830237                             exec_envp);
5830238                     exit (0);
5830239                   }
5830240                 break;
```

```
5830241                 }
5830242             }
5830243         }
5830244 }
```

96.1.19 applic/ipconfig.c

«

Si veda la sezione 92.5.

```
5840001 #include <sys/os32.h>
5840002 #include <kernel/net.h>
5840003 #include <unistd.h>
5840004 #include <stdio.h>
5840005 #include <fcntl.h>
5840006 #include <unistd.h>
5840007 #include <stdlib.h>
5840008 //-------------------------------------------------------
5840009 #define NET_BUFFER_MAX 1024     // [1]
5840010 //
5840011 // [1] Enough to be able to read important data from
5840012 //     the 'net_table[]', without stack overflow.
5840013 //     In fact, the table 'net_table[]' contains
5840014 //     also the interface frames, and there is no sense
5840015 //     to read a full item.
5840016 //
5840017 //-------------------------------------------------------
5840018 int
5840019 main (int argc, char *argv[], char *envp[])
5840020 {
5840021   int fd;
5840022   ssize_t size_read;
5840023   char buffer[NET_BUFFER_MAX];
5840024   int n;
5840025   net_t *net_table_item;
5840026   char string[80];
5840027
5840028   //
```

```
5840029     // All options are ignored, at the moment.
5840030     //
5840031
5840032     //
5840033     // Open `/dev/kmem_net', to get the network
5840034     // interface table.
5840035     //
5840036     fd = open ("/dev/kmem_net", O_RDONLY);
5840037     if (fd < 0)
5840038       {
5840039         printf ("[%s] Cannot open \"/dev/kmem_net\" ",
5840040                 argv[0]);
5840041         perror (NULL);
5840042         exit (0);
5840043       }
5840044     //
5840045     // Print header.
5840046     //
5840047     printf ("dev   "
5840048             "address/mask         "
5840049             "mac                 " "io      irq\n");
5840050     //
5840051     // Scan NET items and then print body.
5840052     //
5840053     for (n = 0; n < NET_MAX_DEVICES; n++)
5840054       {
5840055         lseek (fd, (off_t) n, SEEK_SET);
5840056         size_read = read (fd, buffer, NET_BUFFER_MAX);
5840057         if (size_read < NET_BUFFER_MAX)
5840058           {
5840059             printf
5840060               ("[%s] Cannot read \"/dev/kmem_net\" "
5840061                "item %i ", argv[0], n);
5840062             perror (NULL);
5840063             continue;
5840064           }
5840065         net_table_item = (net_t *) buffer;
```

```
5840066         if (net_table_item->type != NET_DEV_NULL)
5840067           {
5840068             sprintf (string, "net%i    ", n);
5840069             string[6] = '\0';
5840070             printf ("%s", string);
5840071             //
5840072             sprintf (string, "%i.%i.%i.%i/%i "
5840073                      "                ",
5840074                      net_table_item->ip >> 24 & 0x000000FF,
5840075                      net_table_item->ip >> 16 & 0x000000FF,
5840076                      net_table_item->ip >> 8 & 0x000000FF,
5840077                      net_table_item->ip >> 0 & 0x000000FF,
5840078                      net_table_item->m);
5840079             string[20] = '\0';
5840080             printf ("%s", string);
5840081             //
5840082             if (net_table_item->type & NET_DEV_ETH)
5840083               {
5840084                 printf
5840085                   ("%02x:%02x:%02x:%02x:%02x:%02x  "
5840086                    "0x%04x  %i",
5840087                    net_table_item->ethernet.mac[0],
5840088                    net_table_item->ethernet.mac[1],
5840089                    net_table_item->ethernet.mac[2],
5840090                    net_table_item->ethernet.mac[3],
5840091                    net_table_item->ethernet.mac[4],
5840092                    net_table_item->ethernet.mac[5],
5840093                    net_table_item->ethernet.base_io,
5840094                    net_table_item->ethernet.irq);
5840095               }
5840096             printf ("\n");
5840097           }
5840098       }
5840099     close (fd);
5840100     return (0);
5840101   }
```

96.1.20 applic/kill.c

Si veda la sezione 86.12.

```
5850001 #include <sys/os32.h>
5850002 #include <sys/stat.h>
5850003 #include <sys/types.h>
5850004 #include <unistd.h>
5850005 #include <stdlib.h>
5850006 #include <fcntl.h>
5850007 #include <errno.h>
5850008 #include <signal.h>
5850009 #include <stdio.h>
5850010 #include <string.h>
5850011 #include <limits.h>
5850012 #include <libgen.h>
5850013 //-------------------------------------------------------
5850014 static void usage (void);
5850015 //-------------------------------------------------------
5850016 int
5850017 main (int argc, char *argv[], char *envp[])
5850018 {
5850019   int signal;
5850020   int pid;
5850021   int a;          // Index inside arguments.
5850022   int option_s = 0;
5850023   int option_l = 0;
5850024   int opt;
5850025   extern char *optarg;
5850026   extern int optopt;
5850027   //
5850028   // There must be at least an option, plus the
5850029   // program name.
5850030   //
5850031   if (argc < 2)
5850032     {
5850033       usage ();
5850034       return (1);
```

```
5850035       }
5850036     //
5850037     // Check for options.
5850038     //
5850039     while ((opt = getopt (argc, argv, ":ls:")) != -1)
5850040       {
5850041         switch (opt)
5850042           {
5850043           case 'l':
5850044             option_l = 1;
5850045             break;
5850046           case 's':
5850047             option_s = 1;
5850048             //
5850049             // In that case, there must be at least
5850050             // three arguments:
5850051             // the option, the signal and the process
5850052             // id.
5850053             //
5850054             if (argc < 4)
5850055               {
5850056                 usage ();
5850057                 return (1);
5850058               }
5850059             //
5850060             // Argument numbers are ok. Check the
5850061             // signal.
5850062             //
5850063             if (strcmp (optarg, "HUP") == 0)
5850064               {
5850065                 signal = SIGHUP;
5850066               }
5850067             else if (strcmp (optarg, "INT") == 0)
5850068               {
5850069                 signal = SIGINT;
5850070               }
5850071             else if (strcmp (optarg, "QUIT") == 0)
```

```
5850072                 {
5850073                   signal = SIGQUIT;
5850074                 }
5850075               else if (strcmp (optarg, "ILL") == 0)
5850076                 {
5850077                   signal = SIGILL;
5850078                 }
5850079               else if (strcmp (optarg, "ABRT") == 0)
5850080                 {
5850081                   signal = SIGABRT;
5850082                 }
5850083               else if (strcmp (optarg, "FPE") == 0)
5850084                 {
5850085                   signal = SIGFPE;
5850086                 }
5850087               else if (strcmp (optarg, "KILL") == 0)
5850088                 {
5850089                   signal = SIGKILL;
5850090                 }
5850091               else if (strcmp (optarg, "SEGV") == 0)
5850092                 {
5850093                   signal = SIGSEGV;
5850094                 }
5850095               else if (strcmp (optarg, "PIPE") == 0)
5850096                 {
5850097                   signal = SIGPIPE;
5850098                 }
5850099               else if (strcmp (optarg, "ALRM") == 0)
5850100                 {
5850101                   signal = SIGALRM;
5850102                 }
5850103               else if (strcmp (optarg, "TERM") == 0)
5850104                 {
5850105                   signal = SIGTERM;
5850106                 }
5850107               else if (strcmp (optarg, "STOP") == 0)
5850108                 {
```

```
5850109                 signal = SIGSTOP;
5850110               }
5850111             else if (strcmp (optarg, "TSTP") == 0)
5850112               {
5850113                 signal = SIGTSTP;
5850114               }
5850115             else if (strcmp (optarg, "CONT") == 0)
5850116               {
5850117                 signal = SIGCONT;
5850118               }
5850119             else if (strcmp (optarg, "CHLD") == 0)
5850120               {
5850121                 signal = SIGCHLD;
5850122               }
5850123             else if (strcmp (optarg, "TTIN") == 0)
5850124               {
5850125                 signal = SIGTTIN;
5850126               }
5850127             else if (strcmp (optarg, "TTOU") == 0)
5850128               {
5850129                 signal = SIGTTOU;
5850130               }
5850131             else if (strcmp (optarg, "USR1") == 0)
5850132               {
5850133                 signal = SIGUSR1;
5850134               }
5850135             else if (strcmp (optarg, "USR2") == 0)
5850136               {
5850137                 signal = SIGUSR2;
5850138               }
5850139             else
5850140               {
5850141                 fprintf (stderr, "Unknown signal %s.\n",
5850142                          optarg);
5850143                 return (1);
5850144               }
5850145             break;
```

```
5850146           case '?':
5850147             fprintf (stderr, "Unknown option -%c.\n", optopt);
5850148             usage ();
5850149             return (1);
5850150             break;
5850151           case ':':
5850152             fprintf (stderr,
5850153                      "Missing argument for option -%c\n",
5850154                      optopt);
5850155             usage ();
5850156             return (1);
5850157             break;
5850158           default:
5850159             fprintf (stderr,
5850160                      "Getopt problem: unknown option "
5850161                      "%c\n", opt);
5850162             return (1);
5850163           }
5850164       }
5850165     //
5850166     //
5850167     //
5850168     if (option_l && option_s)
5850169       {
5850170         fprintf (stderr,
5850171                  "Options \"-l\" and \"-s\" together ");
5850172         fprintf (stderr, "are incompatible.\n");
5850173         usage ();
5850174         return (1);
5850175       }
5850176     //
5850177     // Option "-l".
5850178     //
5850179     if (option_l)
5850180       {
5850181         printf ("HUP ");
5850182         printf ("INT ");
```

```
5850183         printf ("QUIT ");
5850184         printf ("ILL ");
5850185         printf ("ABRT ");
5850186         printf ("FPE ");
5850187         printf ("KILL ");
5850188         printf ("SEGV ");
5850189         printf ("PIPE ");
5850190         printf ("ALRM ");
5850191         printf ("TERM ");
5850192         printf ("STOP ");
5850193         printf ("TSTP ");
5850194         printf ("CONT ");
5850195         printf ("CHLD ");
5850196         printf ("TTIN ");
5850197         printf ("TTOU ");
5850198         printf ("USR1 ");
5850199         printf ("USR2 ");
5850200         printf ("\n");
5850201       }
5850202     //
5850203     // Option "-s".
5850204     //
5850205     if (option_s)
5850206       {
5850207         //
5850208         // Scan arguments.
5850209         //
5850210         for (a = 3; a < argc; a++)
5850211           {
5850212             //
5850213             // Get PID.
5850214             //
5850215             pid = atoi (argv[a]);
5850216             if (pid > 0)
5850217               {
5850218                 //
5850219                 // Kill.
```

```
5850220                 //
5850221                 if (kill (pid, signal) < 0)
5850222                   {
5850223                     perror (argv[a]);
5850224                   }
5850225               }
5850226             else
5850227               {
5850228                 fprintf (stderr, "Invalid PID %s.", argv[a]);
5850229               }
5850230           }
5850231       }
5850232     //
5850233     // All done.
5850234     //
5850235     return (0);
5850236   }
5850237
5850238   //---------------------------------------------------
5850239   static void
5850240   usage (void)
5850241   {
5850242     fprintf (stderr, "Usage: kill -s SIGNAL_NAME PID...\n");
5850243     fprintf (stderr, "       kill -l\n");
5850244   }
```

96.1.21 applic/ln.c

Si veda la sezione 86.13.

```
5860001   #include <sys/os32.h>
5860002   #include <sys/stat.h>
5860003   #include <sys/types.h>
5860004   #include <unistd.h>
5860005   #include <stdlib.h>
5860006   #include <fcntl.h>
5860007   #include <errno.h>
```

```
5860008 #include <signal.h>
5860009 #include <stdio.h>
5860010 #include <string.h>
5860011 #include <limits.h>
5860012 #include <libgen.h>
5860013 //-------------------------------------------------------
5860014 static void usage (void);
5860015 //-------------------------------------------------------
5860016 int
5860017 main (int argc, char *argv[], char *envp[])
5860018 {
5860019   char *source;
5860020   char *destination;
5860021   char *new_destination;
5860022   struct stat file_status;
5860023   int dest_is_a_dir = 0;
5860024   int a;         // Argument index.
5860025   char path[PATH_MAX];
5860026   //
5860027   // There must be at least two arguments, plus the
5860028   // program name.
5860029   //
5860030   if (argc < 3)
5860031     {
5860032       usage ();
5860033       return (1);
5860034     }
5860035   //
5860036   // Select the last argument as the destination.
5860037   //
5860038   destination = argv[argc - 1];
5860039   //
5860040   // Check if it is a directory and save it in a flag.
5860041   //
5860042   if (stat (destination, &file_status) == 0)
5860043     {
5860044       if (S_ISDIR (file_status.st_mode))
```

```
5860045           {
5860046             dest_is_a_dir = 1;
5860047           }
5860048       }
5860049     //
5860050     // If there are more than two arguments, verify that
5860051     // the last
5860052     // one is a directory.
5860053     //
5860054     if (argc > 3)
5860055       {
5860056         if (!dest_is_a_dir)
5860057           {
5860058             usage ();
5860059             fprintf (stderr, "The destination \"%s\" ",
5860060                      destination);
5860061             fprintf (stderr, "is not a directory!\n");
5860062             return (1);
5860063           }
5860064       }
5860065     //
5860066     // Scan the arguments, excluded the last, that is
5860067     // the destination.
5860068     //
5860069     for (a = 1; a < (argc - 1); a++)
5860070       {
5860071         //
5860072         // Source.
5860073         //
5860074         source = argv[a];
5860075         //
5860076         // Verify access permissions.
5860077         //
5860078         if (access (source, R_OK) < 0)
5860079           {
5860080             perror (source);
5860081             continue;
```

```
5860082           }
5860083         //
5860084         // Destination.
5860085         //
5860086         // If it is a directory, the destination path
5860087         // must be corrected.
5860088         //
5860089         if (dest_is_a_dir)
5860090           {
5860091             path[0] = 0;
5860092             strcat (path, destination);
5860093             strcat (path, "/");
5860094             strcat (path, basename (source));
5860095             //
5860096             // Update the destination path.
5860097             //
5860098             new_destination = path;
5860099           }
5860100         else
5860101           {
5860102             new_destination = destination;
5860103           }
5860104         //
5860105         // Check if destination file exists.
5860106         //
5860107         if (stat (new_destination, &file_status) == 0)
5860108           {
5860109             fprintf (stderr,
5860110                      "The destination file, \"%s\", ",
5860111                      new_destination);
5860112             fprintf (stderr, "already exists!\n");
5860113             continue;
5860114           }
5860115         //
5860116         // Everything is ready for the link.
5860117         //
5860118         if (link (source, new_destination) < 0)
```

```
5860119             {
5860120               perror (new_destination);
5860121               continue;
5860122             }
5860123         }
5860124       //
5860125       // All done.
5860126       //
5860127       return (0);
5860128     }
5860129
5860130     //-----------------------------------------------------------
5860131     static void
5860132     usage (void)
5860133     {
5860134       fprintf (stderr, "Usage: ln OLD_NAME NEW_NAME\n");
5860135       fprintf (stderr, "       ln FILE... DIRECTORY\n");
5860136     }
```

96.1.22 applic/login.c

«

Si veda la sezione 86.14.

```
5870001 #include <unistd.h>
5870002 #include <stdlib.h>
5870003 #include <sys/stat.h>
5870004 #include <sys/types.h>
5870005 #include <fcntl.h>
5870006 #include <errno.h>
5870007 #include <unistd.h>
5870008 #include <signal.h>
5870009 #include <stdio.h>
5870010 #include <sys/wait.h>
5870011 #include <stdio.h>
5870012 #include <string.h>
5870013 #include <limits.h>
5870014 #include <stdint.h>
```

```
5870015 #include <sys/os32.h>
5870016 //-------------------------------------------------------
5870017 #define LOGIN_MAX       64
5870018 #define PASSWORD_MAX    64
5870019 #define HOME_MAX        64
5870020 #define LINE_MAX        1024
5870021 //-------------------------------------------------------
5870022 int
5870023 main (int argc, char *argv[], char *envp[])
5870024 {
5870025   char login[LOGIN_MAX];
5870026   char password[PASSWORD_MAX];
5870027   char buffer[LINE_MAX];
5870028   char *user_name;
5870029   char *user_password;
5870030   char *user_uid;
5870031   char *user_gid;
5870032   char *user_description;
5870033   char *user_home;
5870034   char *user_shell;
5870035   uid_t uid;
5870036   uid_t euid;
5870037   gid_t gid;
5870038   gid_t egid;
5870039   int fd;
5870040   ssize_t size_read;
5870041   int b;          // Index inside buffer.
5870042   int loop;
5870043   char *exec_argv[2];
5870044   int status;
5870045   char *tty_path;
5870046   //
5870047   // Check if login is running correctly.
5870048   //
5870049   euid = geteuid ();
5870050   uid = getuid ();
5870051   //
```

```
5870052     // Check privileges.
5870053     //
5870054     if (!(uid == 0 && euid == 0))
5870055       {
5870056         printf
5870057           ("%s: can only run with root privileges!\n",
5870058            argv[0]);
5870059         exit (-1);
5870060       }
5870061     //
5870062     // Prepare arguments for the shell call.
5870063     //
5870064     exec_argv[0] = "-";
5870065     exec_argv[1] = NULL;
5870066     //
5870067     // Login.
5870068     //
5870069     while (1)
5870070       {
5870071         fd = open ("/etc/passwd", O_RDONLY);
5870072         //
5870073         if (fd < 0)
5870074           {
5870075             perror ("Cannot open file `/etc/passwd'");
5870076             exit (-1);
5870077           }
5870078         //
5870079         printf ("Log in as \"root\" or \"user\" "
5870080                 "with password \"ciao\" :-)\n");
5870081         input_line (login, "login: ", LOGIN_MAX,
5870082                     INPUT_LINE_ECHO);
5870083         //
5870084         //
5870085         //
5870086         loop = 1;
5870087         while (loop)
5870088           {
```

```
5870089             for (b = 0; b < LINE_MAX; b++)
5870090               {
5870091                 size_read = read (fd, &buffer[b], (size_t) 1);
5870092                 if (size_read <= 0)
5870093                   {
5870094                     buffer[b] = 0;
5870095                     loop = 0;     // Close the middle
5870096                     // loop.
5870097                     break;
5870098                   }
5870099                 if (buffer[b] == '\n')
5870100                   {
5870101                     buffer[b] = 0;
5870102                     break;
5870103                   }
5870104               }
5870105             //
5870106             // Please notice that `strtok()' does not
5870107             // allow to have empty fields! If it finds
5870108             // a `::', it will treat it as a single `:'.
5870109             //
5870110             user_name = strtok (buffer, ":");
5870111             user_password = strtok (NULL, ":");
5870112             user_uid = strtok (NULL, ":");
5870113             user_gid = strtok (NULL, ":");
5870114             user_description = strtok (NULL, ":");
5870115             user_home = strtok (NULL, ":");
5870116             user_shell = strtok (NULL, ":");
5870117             //
5870118             if (strcmp (user_name, login) == 0)
5870119               {
5870120                 input_line (password, "password: ",
5870121                             PASSWORD_MAX, INPUT_LINE_HIDDEN);
5870122                 //
5870123                 // Compare passwords: empty passwords
5870124                 // are not allowed.
5870125                 //
```

```
5870126                 if (strcmp (user_password, password) == 0)
5870127                   {
5870128                     uid = atoi (user_uid);
5870129                     euid = uid;
5870130                     gid = atoi (user_gid);
5870131                     egid = gid;
5870132                     //
5870133                     // Find the controlling terminal and
5870134                     // change
5870135                     // property and access permissions.
5870136                     //
5870137                     tty_path = ttyname (STDIN_FILENO);
5870138                     if (tty_path != NULL)
5870139                       {
5870140                         status = chown (tty_path, uid, 0);
5870141                         if (status != 0)
5870142                           {
5870143                             perror (NULL);
5870144                           }
5870145                         status = chmod (tty_path, 0600);
5870146                         if (status != 0)
5870147                           {
5870148                             perror (NULL);
5870149                           }
5870150                       }
5870151                     //
5870152                     // Cd to the home directory, if
5870153                     // present.
5870154                     //
5870155                     status = chdir (user_home);
5870156                     if (status != 0)
5870157                       {
5870158                         perror (NULL);
5870159                       }
5870160                     //
5870161                     // Now change personality: first the
5870162                     // group,
```

```
5870163                         // otherwise, it would be not
5870164                         // possible to do
5870165                         // after changing the UID to an
5870166                         // unprivileged
5870167                         // one.
5870168                         //
5870169                         setgid (gid);
5870170                         setegid (egid);
5870171                         //
5870172                         setuid (uid);
5870173                         seteuid (euid);
5870174                         //
5870175                         // Run the shell, replacing the
5870176                         // login process; the
5870177                         // environment is taken from 'init'.
5870178                         //
5870179                         execve (user_shell, exec_argv, envp);
5870180                         exit (0);
5870181                       }
5870182                     //
5870183                     // Login failed: will try again.
5870184                     //
5870185                     loop = 0; // Close the middle loop.
5870186                     break;
5870187                   }
5870188               }
5870189           close (fd);
5870190       }
5870191   }
```

96.1.23 applic/ls.c

«

Si veda la sezione 86.15.

```
5880001 #include <sys/stat.h>
5880002 #include <sys/types.h>
5880003 #include <unistd.h>
```

```
5880004 #include <stdlib.h>
5880005 #include <fcntl.h>
5880006 #include <errno.h>
5880007 #include <signal.h>
5880008 #include <stdio.h>
5880009 #include <string.h>
5880010 #include <limits.h>
5880011 #include <libgen.h>
5880012 #include <dirent.h>
5880013 #include <pwd.h>
5880014 #include <grp.h>
5880015 #include <time.h>
5880016 //-------------------------------------------------------
5880017 #define BUFFER_SIZE     131072
5880018 #define LIST_SIZE       8000
5880019 //-------------------------------------------------------
5880020 static void usage (void);
5880021 static int compare (const void *p1, const void *p2);
5880022 //-------------------------------------------------------
5880023 //
5880024 // Static variables to avoid stack overflow.
5880025 //
5880026 static char buffer[BUFFER_SIZE];
5880027 static char *list[LIST_SIZE];
5880028 //-------------------------------------------------------
5880029 int
5880030 main (int argc, char *argv[], char *envp[])
5880031 {
5880032   int option_a = 0;
5880033   int option_l = 0;
5880034   int opt;
5880035 //  extern char   *optarg;              // not used.
5880036   extern int optind;
5880037   extern int optopt;
5880038   struct stat file_status;
5880039   DIR *dp;
5880040   struct dirent *dir;
```

```
5880041     int b;          // Buffer index.
5880042     int l;          // List index.
5880043     int len;        // Name length.
5880044     char *path = NULL;
5880045     char pathname[PATH_MAX];
5880046     struct passwd *pws;
5880047     struct group *grs;
5880048     struct tm *tms;
5880049     //
5880050     // Check for options.
5880051     //
5880052     while ((opt = getopt (argc, argv, ":al")) != -1)
5880053       {
5880054         switch (opt)
5880055           {
5880056           case 'l':
5880057             option_l = 1;
5880058             break;
5880059           case 'a':
5880060             option_a = 1;
5880061             break;
5880062           case '?':
5880063             fprintf (stderr, "Unknown option -%c.\n", optopt);
5880064             usage ();
5880065             return (1);
5880066             break;
5880067           case ':':
5880068             fprintf (stderr,
5880069                      "Missing argument for option -%c\n",
5880070                      optopt);
5880071             usage ();
5880072             return (1);
5880073             break;
5880074           default:
5880075             fprintf (stderr,
5880076                      "Getopt problem: unknown option "
5880077                      "%c\n", opt);
```

```
5880078             return (1);
5880079           }
5880080       }
5880081     //
5880082     // If no arguments are present, at least the current
5880083     // directory is
5880084     // read.
5880085     //
5880086     if (optind == argc)
5880087       {
5880088         //
5880089         // There are no more arguments. Replace the
5880090         // program name,
5880091         // corresponding to 'argv[0]', with the current
5880092         // directory
5880093         // path string.
5880094         //
5880095         argv[0] = ".";
5880096         argc = 1;
5880097         optind = 0;
5880098       }
5880099     //
5880100     // This is a very simplified 'ls': if there is only
5880101     // a name
5880102     // and it is a directory, the directory content is
5880103     // taken as
5880104     // the new 'argv[]' array.
5880105     //
5880106     if (optind == (argc - 1))
5880107       {
5880108         //
5880109         // There is a request for a single name. Test if
5880110         // it exists
5880111         // and if it is a directory.
5880112         //
5880113         if (stat (argv[optind], &file_status) != 0)
5880114           {
```

```
5880115             fprintf (stderr,
5880116                      "File \"%s\" does not exist!\n",
5880117                      argv[optind]);
5880118             return (2);
5880119           }
5880120         //
5880121         if (S_ISDIR (file_status.st_mode))
5880122           {
5880123             //
5880124             // Save the directory inside the 'path'
5880125             // pointer.
5880126             //
5880127             path = argv[optind];
5880128             //
5880129             // Open the directory.
5880130             //
5880131             dp = opendir (argv[optind]);
5880132             if (dp == NULL)
5880133               {
5880134                 perror (argv[optind]);
5880135                 return (3);
5880136               }
5880137             //
5880138             // Read the directory and fill the buffer
5880139             // with names.
5880140             //
5880141             b = 0;
5880142             l = 0;
5880143             while ((dir = readdir (dp)) != NULL)
5880144               {
5880145                 len = strlen (dir->d_name);
5880146                 //
5880147                 // Check if the buffer can hold it.
5880148                 //
5880149                 if ((b + len + 1) > BUFFER_SIZE)
5880150                   {
5880151                     fprintf (stderr, "not enough memory\n");
```

```
5880152                     break;
5880153                   }
5880154                 //
5880155                 // Consider the directory item only if
5880156                 // there is
5880157                 // a valid name. If it is empty, just
5880158                 // ignore it.
5880159                 //
5880160                 if (len > 0)
5880161                   {
5880162                     strcpy (&buffer[b], dir->d_name);
5880163                     list[l] = &buffer[b];
5880164                     b += len + 1;
5880165                     l++;
5880166                   }
5880167               }
5880168             //
5880169             // Close the directory.
5880170             //
5880171             closedir (dp);
5880172             //
5880173             // Sort the list.
5880174             //
5880175             qsort (list, (size_t) l, sizeof (char *),
5880176                    compare);
5880177             //
5880178             // Convert the directory list into a new
5880179             // 'argv[]' array,
5880180             // with a valid 'argc'. The variable
5880181             // 'optind' must be
5880182             // reset to the first element index, because
5880183             // there is
5880184             // no program name inside the new 'argv[]'
5880185             // at index zero.
5880186             //
5880187             argv = list;
5880188             argc = l;
```

```
5880189             optind = 0;
5880190           }
5880191       }
5880192     //
5880193     // Scan arguments, or list converted into `argv[]'.
5880194     //
5880195     for (; optind < argc; optind++)
5880196       {
5880197         if (argv[optind][0] == '.')
5880198           {
5880199             //
5880200             // Current name starts with `.'.
5880201             //
5880202             if (!option_a)
5880203               {
5880204                 //
5880205                 // Do not show name starting with `.'.
5880206                 //
5880207                 continue;
5880208               }
5880209           }
5880210         //
5880211         // Build the pathname.
5880212         //
5880213         if (path == NULL)
5880214           {
5880215             strcpy (&pathname[0], argv[optind]);
5880216           }
5880217         else
5880218           {
5880219             strcpy (pathname, path);
5880220             strcat (pathname, "/");
5880221             strcat (pathname, argv[optind]);
5880222           }
5880223         //
5880224         // Check if file exists, reading status.
5880225         //
```

```
5880226         if (stat (pathname, &file_status) != 0)
5880227           {
5880228             fprintf (stderr,
5880229                      "File \"%s\" does not exist!\n",
5880230                      pathname);
5880231             return (2);
5880232           }
5880233         //
5880234         // Show file name.
5880235         //
5880236         if (option_l)
5880237           {
5880238             //
5880239             // Print the file type.
5880240             //
5880241             if (S_ISBLK (file_status.st_mode))
5880242               printf ("b");
5880243             else if (S_ISCHR (file_status.st_mode))
5880244               printf ("c");
5880245             else if (S_ISFIFO (file_status.st_mode))
5880246               printf ("p");
5880247             else if (S_ISREG (file_status.st_mode))
5880248               printf ("-");
5880249             else if (S_ISDIR (file_status.st_mode))
5880250               printf ("d");
5880251             else if (S_ISLNK (file_status.st_mode))
5880252               printf ("l");
5880253             else if (S_ISSOCK (file_status.st_mode))
5880254               printf ("s");
5880255             else
5880256               printf ("?");
5880257             //
5880258             // Print permissions.
5880259             //
5880260             if (S_IRUSR & file_status.st_mode)
5880261               printf ("r");
5880262             else
```

```
5880263             printf ("-");
5880264           if (S_IWUSR & file_status.st_mode)
5880265             printf ("w");
5880266           else
5880267             printf ("-");
5880268           if (S_IXUSR & file_status.st_mode)
5880269             printf ("x");
5880270           else
5880271             printf ("-");
5880272           if (S_IRGRP & file_status.st_mode)
5880273             printf ("r");
5880274           else
5880275             printf ("-");
5880276           if (S_IWGRP & file_status.st_mode)
5880277             printf ("w");
5880278           else
5880279             printf ("-");
5880280           if (S_IXGRP & file_status.st_mode)
5880281             printf ("x");
5880282           else
5880283             printf ("-");
5880284           if (S_IROTH & file_status.st_mode)
5880285             printf ("r");
5880286           else
5880287             printf ("-");
5880288           if (S_IWOTH & file_status.st_mode)
5880289             printf ("w");
5880290           else
5880291             printf ("-");
5880292           if (S_IXOTH & file_status.st_mode)
5880293             printf ("x");
5880294           else
5880295             printf ("-");
5880296           //
5880297           // Print links.
5880298           //
5880299           printf (" %3i", (int) file_status.st_nlink);
```

```
5880300             //
5880301             // Print owner.
5880302             //
5880303             pws = getpwuid (file_status.st_uid);
5880304             if (pws == NULL)
5880305               {
5880306                 printf (" %i", (int) file_status.st_uid);
5880307               }
5880308             else
5880309               {
5880310                 printf (" %s", pws->pw_name);
5880311               }
5880312             //
5880313             // Print group.
5880314             //
5880315             grs = getgrgid (file_status.st_gid);
5880316             if (grs == NULL)
5880317               {
5880318                 printf (" %i", (int) file_status.st_gid);
5880319               }
5880320             else
5880321               {
5880322                 printf (" %s", grs->gr_name);
5880323               }
5880324             //
5880325             // Print file size or device major-minor.
5880326             //
5880327             if (S_ISBLK (file_status.st_mode)
5880328                 || S_ISCHR (file_status.st_mode))
5880329               {
5880330                 printf (" %3i,",
5880331                         (int) major (file_status.st_rdev));
5880332                 printf (" %3i",
5880333                         (int) minor (file_status.st_rdev));
5880334               }
5880335             else
5880336               {
```

```
5880337                 printf (" %8i", (int) file_status.st_size);
5880338               }
5880339             //
5880340             // Print modification date and time.
5880341             //
5880342             tms = localtime (&(file_status.st_mtime));
5880343             printf (" %4u-%02u-%02u %02u:%02u",
5880344                     tms->tm_year, tms->tm_mon,
5880345                     tms->tm_mday, tms->tm_hour, tms->tm_min);
5880346             //
5880347             // Print file name, but with no additional
5880348             // path.
5880349             //
5880350             printf (" %s\n", argv[optind]);
5880351           }
5880352         else
5880353           {
5880354             //
5880355             // Just show the file name and go to the
5880356             // next line.
5880357             //
5880358             printf ("%s\n", argv[optind]);
5880359           }
5880360       }
5880361     //
5880362     // All done.
5880363     //
5880364     return (0);
5880365   }
5880366
5880367   //---------------------------------------------------------
5880368   static void
5880369   usage (void)
5880370   {
5880371     fprintf (stderr, "Usage: ls [OPTION] [FILE]...\n");
5880372   }
5880373
```

```
5880374 //-------------------------------------------------------
5880375 static int
5880376 compare (const void *p1, const void *p2)
5880377 {
5880378   return (strcmp (*((char **) p1), *((char **) p2)));
5880379 }
```

96.1.24 applic/man.c

Si veda la sezione 86.16.

```
5890001 #include <unistd.h>
5890002 #include <stdlib.h>
5890003 #include <errno.h>
5890004 //-------------------------------------------------------
5890005 #define MAX_LINES   20
5890006 #define MAX_COLUMNS 80
5890007 //-------------------------------------------------------
5890008 static char *man_page_directory = "/usr/share/man";
5890009 //-------------------------------------------------------
5890010 static void usage (void);
5890011 static FILE *open_man_page (int section, char *name);
5890012 static void build_path_name (int section, char *name,
5890013                              char *path);
5890014 //-------------------------------------------------------
5890015 int
5890016 main (int argc, char *argv[], char *envp[])
5890017 {
5890018   FILE *fp;
5890019   char *name;
5890020   int section;
5890021   int c;
5890022   int line = 1; // Line internal counter.
5890023   int column = 1;       // Column internal counter.
5890024   int loop;
5890025   //
5890026   // There must be minimum an argument, and maximum
```

```
5890027     // two.
5890028     //
5890029     if (argc < 2 || argc > 3)
5890030       {
5890031         usage ();
5890032         return (1);
5890033       }
5890034     //
5890035     // If there are two arguments, there must be the
5890036     // section number.
5890037     //
5890038     if (argc == 3)
5890039       {
5890040         section = atoi (argv[1]);
5890041         name = argv[2];
5890042       }
5890043     else
5890044       {
5890045         section = 0;
5890046         name = argv[1];
5890047       }
5890048     //
5890049     // Try to open the manual page.
5890050     //
5890051     fp = open_man_page (section, name);
5890052     //
5890053     if (fp == NULL)
5890054       {
5890055         //
5890056         // Error opening file.
5890057         //
5890058         return (1);
5890059       }
5890060
5890061     //
5890062     // The following loop continues while the file
5890063     // gives characters, or when a command to change
```

```
5890064     // file or to quit is given.
5890065     //
5890066     for (loop = 1; loop;)
5890067       {
5890068         //
5890069         // Read a single character.
5890070         //
5890071         c = getc (fp);
5890072         //
5890073         if (c == EOF)
5890074           {
5890075             loop = 0;
5890076             break;
5890077           }
5890078         //
5890079         // If the character read is a special one,
5890080         // the line/column calculation is modified,
5890081         // so that it is known when to stop scrolling.
5890082         //
5890083         switch (c)
5890084           {
5890085           case '\r':
5890086             //
5890087             // Displaying this character, the cursor
5890088             // should go
5890089             // back to the first column. So the column
5890090             // counter
5890091             // is reset.
5890092             //
5890093             column = 1;
5890094             break;
5890095           case '\n':
5890096             //
5890097             // Displaying this character, the cursor
5890098             // should go
5890099             // back to the next line, at the first
5890100             // column.
```

```
5890101             // So the column counter is reset and the
5890102             // line
5890103             // counter is incremented.
5890104             //
5890105             line++;
5890106             column = 1;
5890107             break;
5890108           case '\b':
5890109             //
5890110             // Displaying this character, the cursor
5890111             // should go
5890112             // back one position, unless it is already
5890113             // at the
5890114             // beginning.
5890115             //
5890116             if (column > 1)
5890117               {
5890118                 column--;
5890119               }
5890120             break;
5890121           default:
5890122             //
5890123             // Any other character must increase the
5890124             // column
5890125             // counter.
5890126             //
5890127             column++;
5890128           }
5890129         //
5890130         // Display the character, even if it is a
5890131         // special one:
5890132         // it is responsibility of the screen device
5890133         // management
5890134         // to do something good with special characters.
5890135         //
5890136         putchar (c);
5890137         //
```

```
5890138         // If the column counter is gone beyond the
5890139         // screen columns,
5890140         // then adjust the column counter and increment
5890141         // the line
5890142         // counter.
5890143         //
5890144         if (column > MAX_COLUMNS)
5890145           {
5890146             column -= MAX_COLUMNS;
5890147             line++;
5890148           }
5890149         //
5890150         // Check if there is space for scrolling.
5890151         //
5890152         if (line < MAX_LINES)
5890153           {
5890154             continue;
5890155           }
5890156         //
5890157         // Here, displayed lines are MAX_LINES.
5890158         //
5890159         if (column > 1)
5890160           {
5890161             //
5890162             // Something was printed at the current
5890163             // line: must
5890164             // do a new line.
5890165             //
5890166             putchar ('\n');
5890167           }
5890168         //
5890169         // Show the more prompt.
5890170         //
5890171         printf ("--More--");
5890172         fflush (stdout);
5890173         //
5890174         // Read a character from standard input.
```

```
5890175         //
5890176         c = getchar ();
5890177         //
5890178         // Consider command 'q', but any other character
5890179         // can be introduced, to let show the next page.
5890180         //
5890181         switch (c)
5890182           {
5890183           case 'Q':
5890184           case 'q':
5890185             //
5890186             // Quit. But must erase the '--More--'
5890187             // prompt.
5890188             //
5890189             printf ("\b \b\b \b\b \b\b \b\b \b");
5890190             printf ("\b \b\b \b\b \b\b \b");
5890191             fclose (fp);
5890192             return (0);
5890193           }
5890194         //
5890195         // Backspace to overwrite '--More--' and the
5890196         // character
5890197         // pressed.
5890198         //
5890199         printf
5890200           ("\b \b\b \b\b \b\b \b\b "
5890201            "\b\b \b\b \b\b \b\b \b");
5890202         //
5890203         // Reset line/column counters.
5890204         //
5890205         column = 1;
5890206         line = 1;
5890207       }
5890208     //
5890209     // Close the file pointer if it is still open.
5890210     //
5890211     if (fp != NULL)
```

```
5890212       {
5890213         fclose (fp);
5890214       }
5890215     //
5890216     return (0);
5890217   }
5890218
5890219   //-----------------------------------------------------------
5890220   static void
5890221   usage (void)
5890222   {
5890223     fprintf (stderr, "Usage: man [SECTION] NAME\n");
5890224   }
5890225
5890226   //-----------------------------------------------------------
5890227   FILE *
5890228   open_man_page (int section, char *name)
5890229   {
5890230     FILE *fp;
5890231     char path[PATH_MAX];
5890232     struct stat file_status;
5890233     //
5890234     //
5890235     //
5890236     if (section > 0)
5890237       {
5890238         build_path_name (section, name, path);
5890239         //
5890240         // Check if file exists.
5890241         //
5890242         if (stat (path, &file_status) != 0)
5890243           {
5890244             fprintf (stderr,
5890245                      "Man page %s(%i) does not exist!\n",
5890246                      name, section);
5890247             return (NULL);
5890248           }
```

```
5890249       }
5890250     else
5890251       {
5890252         //
5890253         // Must try a section.
5890254         //
5890255         for (section = 1; section < 9; section++)
5890256           {
5890257             build_path_name (section, name, path);
5890258             //
5890259             // Check if file exists.
5890260             //
5890261             if (stat (path, &file_status) == 0)
5890262               {
5890263                 //
5890264                 // Found.
5890265                 //
5890266                 break;
5890267               }
5890268           }
5890269       }
5890270     //
5890271     // Check if a file was found.
5890272     //
5890273     if (section < 9)
5890274       {
5890275         fp = fopen (path, "r");
5890276         //
5890277         if (fp == NULL)
5890278           {
5890279             //
5890280             // Error opening file.
5890281             //
5890282             perror (path);
5890283             return (NULL);
5890284           }
5890285         else
```

```
5890286           {
5890287             //
5890288             // Opened right.
5890289             //
5890290             return (fp);
5890291           }
5890292       }
5890293     else
5890294       {
5890295         fprintf (stderr, "Man page %s does not exist!\n",
5890296                  name);
5890297         return (NULL);
5890298       }
5890299   }
5890300
5890301   //-------------------------------------------------------
5890302   void
5890303   build_path_name (int section, char *name, char *path)
5890304   {
5890305     char string_section[10];
5890306     //
5890307     // Convert the section number into a string.
5890308     //
5890309     sprintf (string_section, "%i", section);
5890310     //
5890311     // Prepare the path to the man file.
5890312     //
5890313     path[0] = 0;
5890314     strcat (path, man_page_directory);
5890315     strcat (path, "/");
5890316     strcat (path, name);
5890317     strcat (path, ".");
5890318     strcat (path, string_section);
5890319   }
```

96.1.25 applic/mkdir.c

«

Si veda la sezione 86.17.

```
5900001 #include <sys/os32.h>
5900002 #include <sys/stat.h>
5900003 #include <sys/types.h>
5900004 #include <unistd.h>
5900005 #include <stdlib.h>
5900006 #include <fcntl.h>
5900007 #include <errno.h>
5900008 #include <signal.h>
5900009 #include <stdio.h>
5900010 #include <string.h>
5900011 #include <limits.h>
5900012 #include <libgen.h>
5900013 //-------------------------------------------------------
5900014 static int mkdir_parents (const char *path, mode_t mode);
5900015 static void usage (void);
5900016 //-------------------------------------------------------
5900017 int
5900018 main (int argc, char *argv[], char *envp[])
5900019 {
5900020   sysmsg_uarea_t msg;
5900021   int status;
5900022   mode_t mode = 0;
5900023   int m;          // Index inside mode argument.
5900024   int digit;
5900025   char **dir;
5900026   int d;          // Directory index.
5900027   int option_p = 0;
5900028   int option_m = 0;
5900029   int opt;
5900030   extern char *optarg;
5900031   extern int optind;
5900032   extern int optopt;
5900033   //
5900034   // There must be at least an argument, plus the
```

```
5900035     // program name.
5900036     //
5900037     if (argc < 2)
5900038       {
5900039         usage ();
5900040         return (1);
5900041       }
5900042     //
5900043     // Check for options, starting from `p'. The `dir'
5900044     // pointer is used
5900045     // to calculate the argument pointer to the first
5900046     // directory [1].
5900047     // The macroinstruction `max()' is declared inside
5900048     // <sys/os32.h>
5900049     // and does the expected thing.
5900050     //
5900051     while ((opt = getopt (argc, argv, ":pm:")) != -1)
5900052       {
5900053         switch (opt)
5900054           {
5900055           case 'm':
5900056             option_m = 1;
5900057             for (m = 0; m < strlen (optarg); m++)
5900058               {
5900059                 digit = (optarg[m] - '0');
5900060                 if (digit < 0 || digit > 7)
5900061                   {
5900062                     usage ();
5900063                     return (2);
5900064                   }
5900065                 mode = mode * 8 + digit;
5900066               }
5900067             break;
5900068           case 'p':
5900069             option_p = 1;
5900070             break;
5900071           case '?':
```

```
5900072             printf ("Unknown option -%c.\n", optopt);
5900073             usage ();
5900074             return (1);
5900075             break;
5900076           case ':':
5900077             printf ("Missing argument for option -%c\n",
5900078                     optopt);
5900079             usage ();
5900080             return (2);
5900081             break;
5900082           default:
5900083             printf
5900084               ("Getopt problem: unknown option %c\n", opt);
5900085             return (3);
5900086           }
5900087       }
5900088     //
5900089     dir = argv + optind;
5900090     //
5900091     // Check if the mode is to be set to a default
5900092     // value.
5900093     //
5900094     if (!option_m)
5900095       {
5900096         //
5900097         // Default mode.
5900098         //
5900099         sys (SYS_UAREA, &msg, (sizeof msg));
5900100         mode = 0777 & ~msg.umask;
5900101       }
5900102     //
5900103     // Directory creation.
5900104     //
5900105     for (d = 0; dir[d] != NULL; d++)
5900106       {
5900107         if (option_p)
5900108           {
```

```
5900109             status = mkdir_parents (dir[d], mode);
5900110             if (status != 0)
5900111               {
5900112                 perror (dir[d]);
5900113                 return (3);
5900114               }
5900115           }
5900116         else
5900117           {
5900118             status = mkdir (dir[d], mode);
5900119             if (status != 0)
5900120               {
5900121                 perror (dir[d]);
5900122                 return (4);
5900123               }
5900124           }
5900125       }
5900126     //
5900127     // All done.
5900128     //
5900129     return (0);
5900130   }
5900131
5900132   //-----------------------------------------------------------
5900133   static int
5900134   mkdir_parents (const char *path, mode_t mode)
5900135   {
5900136     char path_copy[PATH_MAX];
5900137     char *path_parent;
5900138     struct stat fst;
5900139     int status;
5900140     //
5900141     // Check if the path is empty.
5900142     //
5900143     if (path == NULL || strlen (path) == 0)
5900144       {
5900145         //
```

```
5900146         // Recursion ends here.
5900147         //
5900148         return (0);
5900149       }
5900150     //
5900151     // Check if it does already exists.
5900152     //
5900153     status = stat (path, &fst);
5900154     if (status == 0 && fst.st_mode & S_IFDIR)
5900155       {
5900156         //
5900157         // The path exists and is a directory.
5900158         //
5900159         return (0);
5900160       }
5900161     else if (status == 0 && !(fst.st_mode & S_IFDIR))
5900162       {
5900163         //
5900164         // The path exists but is not a directory.
5900165         //
5900166         errno = ENOTDIR;  // Not a directory.
5900167         return (-1);
5900168       }
5900169     //
5900170     // Get the directory path.
5900171     //
5900172     strncpy (path_copy, path, PATH_MAX);
5900173     path_parent = dirname (path_copy);
5900174     //
5900175     // If it is `.', or `/', the recursion is
5900176     // terminated.
5900177     //
5900178     if (strncmp (path_parent, ".", PATH_MAX) == 0 ||
5900179         strncmp (path_parent, "/", PATH_MAX) == 0)
5900180       {
5900181         return (0);
5900182       }
```

```
5900183     //
5900184     // Otherwise, continue the recursion.
5900185     //
5900186     status = mkdir_parents (path_parent, mode);
5900187     if (status != 0)
5900188       {
5900189         return (-1);
5900190       }
5900191     //
5900192     // Previous directories are there: create the
5900193     // current one.
5900194     //
5900195     status = mkdir (path, mode);
5900196     if (status)
5900197       {
5900198         perror (path);
5900199         return (-1);
5900200       }
5900201
5900202     return (0);
5900203   }
5900204
5900205   //-------------------------------------------------------
5900206   static void
5900207   usage (void)
5900208   {
5900209     fprintf
5900210       (stderr, "Usage: mkdir [-p] [-m OCTAL_MODE] DIR...\n");
5900211   }
```

96.1.26 applic/mmcheck.c

Si veda la sezione 86.18.

```
5910001   #include <sys/os32.h>
5910002   #include <kernel/memory.h>
5910003   #include <kernel/proc.h>
```

```
5910004 #include <unistd.h>
5910005 #include <stdio.h>
5910006 #include <fcntl.h>
5910007 #include <unistd.h>
5910008 #include <stdlib.h>
5910009 //-------------------------------------------------------
5910010 uint32_t mb_table[MEM_MAX_BLOCKS / 32]; // Memory
5910011                                        // blocks map.
5910012 unsigned int mb_max = MEM_MAX_BLOCKS;   // Memory
5910013                                        // blocks max.
5910014 proc_t process;
5910015 //-------------------------------------------------------
5910016 static int mb_block_set0 (int block);
5910017 static void mb_check (pid_t pid, addr_t address,
5910018                       size_t size);
5910019 static void mb_residual (void);
5910020 //-------------------------------------------------------
5910021 int
5910022 main (int argc, char *argv[], char *envp[])
5910023 {
5910024   int i;
5910025   int fd;
5910026   ssize_t size_read;
5910027   char *buffer;
5910028   pid_t pid;
5910029   proc_t *ps;
5910030   //
5910031   // Get memory map.
5910032   //
5910033   fd = open ("/dev/kmem_map", O_RDONLY);
5910034   if (fd < 0)
5910035     {
5910036       printf ("[%s] Cannot open \"/dev/kmem_map\" ",
5910037               argv[0]);
5910038       perror (NULL);
5910039       return (0);
5910040     }
```

```
5910041     //
5910042     buffer = (char *) mb_table;
5910043     lseek (fd, (off_t) 0, SEEK_SET);
5910044     for (i = 0; i < (MEM_MAX_BLOCKS / 8); i += size_read)
5910045       {
5910046         size_read = read (fd, &buffer[i], BUFSIZ);
5910047         if (size_read < 0)
5910048           {
5910049             printf
5910050               ("[%s] Cannot read "
5910051                "\"/dev/kmem_map\" %i %i  ",
5910052                argv[0], size_read, sizeof (mb_table));
5910053             perror (NULL);
5910054             return (0);
5910055           }
5910056       }
5910057     //
5910058     close (fd);
5910059     //
5910060     // Scan processes
5910061     //
5910062     buffer = (char *) &process;
5910063     //
5910064     fd = open ("/dev/kmem_ps", O_RDONLY);
5910065     if (fd < 0)
5910066       {
5910067         printf ("[%s] Cannot open \"/dev/kmem_ps\" ",
5910068                 argv[0]);
5910069         perror (NULL);
5910070         exit (0);
5910071       }
5910072     //
5910073     // Scan processes.
5910074     //
5910075     for (pid = 0; pid < PROCESS_MAX; pid++)
5910076       {
5910077         lseek (fd, (off_t) pid, SEEK_SET);
```

```
5910078         size_read = read (fd, buffer, sizeof (proc_t));
5910079         if (size_read < sizeof (proc_t))
5910080           {
5910081             printf
5910082               ("[%s] Cannot read "
5910083                "\"/dev/kmem_ps\" pid %i ", argv[0], pid);
5910084             perror (NULL);
5910085             continue;
5910086           }
5910087         ps = (proc_t *) buffer;
5910088         if (ps->status > 0)
5910089           {
5910090             //
5910091             //
5910092             //
5910093             if (ps->domain_data == 0)
5910094               {
5910095                 mb_check (pid, ps->address_text,
5910096                           ps->domain_text + ps->extra_data);
5910097               }
5910098             else
5910099               {
5910100                 mb_check (pid, ps->address_text,
5910101                           ps->domain_text);
5910102                 mb_check (pid, ps->address_data,
5910103                           ps->domain_data + ps->extra_data);
5910104               }
5910105           }
5910106       }
5910107     close (fd);
5910108     //
5910109     // Check residual allocation, if any.
5910110     //
5910111     mb_residual ();
5910112     //
5910113     return (0);
5910114   }
```

```
5910115
5910116 //-----------------------------------------------------------
5910117 static void
5910118 mb_check (pid_t pid, addr_t address, size_t size)
5910119 {
5910120   unsigned int bstart;
5910121   unsigned int bsize;
5910122   unsigned int bend;
5910123   unsigned int i;
5910124   addr_t block_address;
5910125   //
5910126   // k_printf ("releasing 0x%x, size 0x%x\n", (int)
5910127   // address,
5910128   // (int) size);
5910129   //
5910130   if (size == 0)
5910131     {
5910132       //
5910133       // Zero means nothing.
5910134       //
5910135       return;
5910136     }
5910137   //
5910138   if (size % MEM_BLOCK_SIZE)
5910139     {
5910140       bsize = size / MEM_BLOCK_SIZE + 1;
5910141     }
5910142   else
5910143     {
5910144       bsize = size / MEM_BLOCK_SIZE;
5910145     }
5910146   //
5910147   bstart = address / MEM_BLOCK_SIZE;
5910148   bend = bstart + bsize;
5910149   //
5910150   //
5910151   //
```

```
5910152     for (i = bstart; i < bend; i++)
5910153       {
5910154         if (mb_block_set0 (i))
5910155           {
5910156             ;
5910157           }
5910158         else
5910159           {
5910160             block_address = i;
5910161             block_address *= MEM_BLOCK_SIZE;
5910162             printf
5910163               ("PID %i is using memory address 0x%x, "
5910164                "but it is reported free or "
5910165                "already used by "
5910166                "another process!\n", (int) pid,
5910167                (unsigned int) block_address);
5910168           }
5910169       }
5910170   }
5910171
5910172   //-----------------------------------------------------
5910173   static int
5910174   mb_block_set0 (int block)
5910175   {
5910176     int i = block / 32;
5910177     int j = block % 32;
5910178     uint32_t mask = 0x80000000 >> j;
5910179     if (mb_table[i] & mask)
5910180       {
5910181         mb_table[i] = mb_table[i] & ~mask;
5910182         return (1);
5910183       }
5910184     else
5910185       {
5910186         return (0);         // The block is already set to
5910187         // 0 inside the map!
5910188       }
```

```
5910189 }
5910190
5910191 //-------------------------------------------------------------
5910192 static void
5910193 mb_residual (void)
5910194 {
5910195   unsigned int block;
5910196   unsigned int blocks = MEM_MAX_BLOCKS;
5910197   int i;
5910198   int j;
5910199   uint32_t mask;
5910200   unsigned int start = 0;
5910201   unsigned int stop = 0;
5910202   unsigned int status = 0;
5910203   //
5910204   // Show residual allocated memory.
5910205   //
5910206   for (block = 0; block < blocks; block++)
5910207     {
5910208       i = block / 32;
5910209       j = block % 32;
5910210       mask = 0x80000000 >> j;
5910211       if (mb_table[i] & mask)
5910212         {
5910213           //
5910214           // Allocated block
5910215           //
5910216           if (status == 0)
5910217             {
5910218               status = 1;
5910219               start = block;
5910220             }
5910221         }
5910222       else
5910223         {
5910224           //
5910225           // Not allocated block.
```

```
5910226             //
5910227             if (status == 1)
5910228               {
5910229                 status = 0;
5910230                 stop = block;
5910231               }
5910232           }
5910233         //
5910234         //
5910235         //
5910236         if (stop > 0)
5910237           {
5910238             printf ("residual allocation: %x-%x  ",
5910239                     start, stop);
5910240             start = 0;
5910241             stop = 0;
5910242           }
5910243       }
5910244     printf ("\n");
5910245   }
```

96.1.27 applic/more.c

«

Si veda la sezione 86.19.

```
5920001 #include <unistd.h>
5920002 #include <errno.h>
5920003 //---------------------------------------------------------
5920004 #define MAX_LINES   20
5920005 #define MAX_COLUMNS 80
5920006 //---------------------------------------------------------
5920007 static void usage (void);
5920008 //---------------------------------------------------------
5920009 int
5920010 main (int argc, char *argv[], char *envp[])
5920011 {
5920012   FILE *fp;
```

```
5920013     char *name;
5920014     int c;
5920015     int line = 1; // Line internal counter.
5920016     int column = 1;       // Column internal counter.
5920017     int a;        // Index inside arguments.
5920018     int loop;
5920019     //
5920020     // There must be at least an argument, plus the
5920021     // program name.
5920022     //
5920023     if (argc < 2)
5920024       {
5920025         usage ();
5920026         return (1);
5920027       }
5920028     //
5920029     // No options are allowed.
5920030     //
5920031     for (a = 1; a < argc; a++)
5920032       {
5920033         //
5920034         // Get next name from arguments.
5920035         //
5920036         name = argv[a];
5920037         //
5920038         // Try to open the file, read only.
5920039         //
5920040         fp = fopen (name, "r");
5920041         //
5920042         if (fp == NULL)
5920043           {
5920044             //
5920045             // Error opening file.
5920046             //
5920047             perror (name);
5920048             return (1);
5920049           }
```

```
5920050         //
5920051         // Print the file name to be displayed.
5920052         //
5920053         printf ("== %s ==\n", name);
5920054         line++;
5920055         //
5920056         // The following loop continues while the file
5920057         // gives characters, or when a command to change
5920058         // file or to quit is given.
5920059         //
5920060         for (loop = 1; loop;)
5920061           {
5920062             //
5920063             // Read a single character.
5920064             //
5920065             c = getc (fp);
5920066             //
5920067             if (c == EOF)
5920068               {
5920069                 loop = 0;
5920070                 break;
5920071               }
5920072             //
5920073             // If the character read is a special one,
5920074             // the line/column calculation is modified,
5920075             // so that it is known when to stop
5920076             // scrolling.
5920077             //
5920078             switch (c)
5920079               {
5920080               case '\r':
5920081                 //
5920082                 // Displaying this character, the cursor
5920083                 // should go
5920084                 // back to the first column. So the
5920085                 // column counter
5920086                 // is reset.
```

```
5920087                    //
5920088                    column = 1;
5920089                    break;
5920090                  case '\n':
5920091                    //
5920092                    // Displaying this character, the cursor
5920093                    // should go
5920094                    // back to the next line, at the first
5920095                    // column.
5920096                    // So the column counter is reset and
5920097                    // the line
5920098                    // counter is incremented.
5920099                    //
5920100                    line++;
5920101                    column = 1;
5920102                    break;
5920103                  case '\b':
5920104                    //
5920105                    // Displaying this character, the cursor
5920106                    // should go
5920107                    // back one position, unless it is
5920108                    // already at the
5920109                    // beginning.
5920110                    //
5920111                    if (column > 1)
5920112                      {
5920113                        column--;
5920114                      }
5920115                    break;
5920116                  default:
5920117                    //
5920118                    // Any other character must increase the
5920119                    // column
5920120                    // counter.
5920121                    //
5920122                    column++;
5920123                  }
```

```
5920124             //
5920125             // Display the character, even if it is a
5920126             // special one:
5920127             // it is responsibility of the screen device
5920128             // management
5920129             // to do something good with special
5920130             // characters.
5920131             //
5920132             putchar (c);
5920133             //
5920134             // If the column counter is gone beyond the
5920135             // screen columns,
5920136             // then adjust the column counter and
5920137             // increment the line
5920138             // counter.
5920139             //
5920140             if (column > MAX_COLUMNS)
5920141               {
5920142                 column -= MAX_COLUMNS;
5920143                 line++;
5920144               }
5920145             //
5920146             // Check if there is space for scrolling.
5920147             //
5920148             if (line < MAX_LINES)
5920149               {
5920150                 continue;
5920151               }
5920152             //
5920153             // Here, displayed lines are MAX_LINES.
5920154             //
5920155             if (column > 1)
5920156               {
5920157                 //
5920158                 // Something was printed at the current
5920159                 // line: must
5920160                 // do a new line.
```

```
5920161                 //
5920162                 putchar ('\n');
5920163               }
5920164             //
5920165             // Show the more prompt.
5920166             //
5920167             printf ("--More--");
5920168             fflush (stdout);
5920169             //
5920170             // Read a character from standard input.
5920171             //
5920172             c = getchar ();
5920173             //
5920174             // Consider commands `n' and `q', but any
5920175             // other character
5920176             // can be introduced, to let show the next
5920177             // page.
5920178             //
5920179             switch (c)
5920180               {
5920181               case 'N':
5920182               case 'n':
5920183                 //
5920184                 // Go to the next file, if any.
5920185                 //
5920186                 fclose (fp);
5920187                 fp = NULL;
5920188                 loop = 0;
5920189                 break;
5920190               case 'Q':
5920191               case 'q':
5920192                 // //
5920193                 // // Quit. But must erase the
5920194                 // `--More--' prompt.
5920195                 // //
5920196                 // printf ("\b \b\b \b\b \b\b \b\b \b");
5920197                 // printf ("\b \b\b \b\b \b\b \b");
```

```
5920198                 fclose (fp);
5920199                 return (0);
5920200               }
5920201             //
5920202             // Backspace to overwrite '--More--' and the
5920203             // character
5920204             // pressed.
5920205             //
5920206             // printf ("\b \b\b \b\b \b\b \b\b \b\b \b\b
5920207             // \b\b \b\b \b");
5920208             //
5920209             // Reset line/column counters.
5920210             //
5920211             column = 1;
5920212             line = 1;
5920213           }
5920214         //
5920215         // Close the file pointer if it is still open.
5920216         //
5920217         if (fp != NULL)
5920218           {
5920219             fclose (fp);
5920220           }
5920221       }
5920222     //
5920223     return (0);
5920224   }
5920225
5920226   //-------------------------------------------------------
5920227   static void
5920228   usage (void)
5920229   {
5920230     fprintf (stderr, "Usage: more FILE...\n");
5920231   }
```

96.1.28 applic/mount.c

Si veda la sezione 92.7.

```
5930001 #include <unistd.h>
5930002 #include <stdlib.h>
5930003 #include <sys/stat.h>
5930004 #include <sys/types.h>
5930005 #include <fcntl.h>
5930006 #include <errno.h>
5930007 #include <signal.h>
5930008 #include <stdio.h>
5930009 #include <sys/wait.h>
5930010 #include <stdio.h>
5930011 #include <string.h>
5930012 #include <limits.h>
5930013 #include <sys/os32.h>
5930014 //-------------------------------------------------------
5930015 static void usage (void);
5930016 //-------------------------------------------------------
5930017 int
5930018 main (int argc, char *argv[], char *envp[])
5930019 {
5930020   int options;
5930021   int status;
5930022   //
5930023   //
5930024   //
5930025   if (argc < 3 || argc > 4)
5930026     {
5930027       usage ();
5930028       return (1);
5930029     }
5930030   //
5930031   // Set options.
5930032   //
5930033   if (argc == 4)
5930034     {
```

```
5930035         if (strcmp (argv[3], "rw") == 0)
5930036           {
5930037             options = MOUNT_DEFAULT;
5930038           }
5930039         else if (strcmp (argv[3], "ro") == 0)
5930040           {
5930041             options = MOUNT_RO;
5930042           }
5930043         else
5930044           {
5930045             printf
5930046               ("Invalid mount option: "
5930047                "only \"ro\" or \"rw\" " "are allowed\n");
5930048             return (2);
5930049           }
5930050       }
5930051     else
5930052       {
5930053         options = MOUNT_DEFAULT;
5930054       }
5930055     //
5930056     // System call.
5930057     //
5930058     status = mount (argv[1], argv[2], options);
5930059     if (status != 0)
5930060       {
5930061         perror (NULL);
5930062         return (2);
5930063       }
5930064     //
5930065     return (0);
5930066   }
5930067
5930068   //---------------------------------------------------------
5930069   static void
5930070   usage (void)
5930071   {
```

```
5930072     fprintf (stderr, "Usage: mount DEVICE MOUNT_POINT "
5930073              "[MOUNT_OPTIONS]\n");
5930074   }
```

96.1.29 applic/nc.c

Si veda la sezione 86.20.

```
5940001   #include <sys/stat.h>
5940002   #include <sys/types.h>
5940003   #include <unistd.h>
5940004   #include <stdlib.h>
5940005   #include <fcntl.h>
5940006   #include <errno.h>
5940007   #include <signal.h>
5940008   #include <stdio.h>
5940009   #include <string.h>
5940010   #include <limits.h>
5940011   #include <libgen.h>
5940012   #include <arpa/inet.h>
5940013   #include <sys/socket.h>
5940014   #include <stdint.h>
5940015   #include <stdbool.h>
5940016   #include <fcntl.h>
5940017   //-------------------------------------------------------
5940018   static void usage (void);
5940019   char buffer[BUFSIZ];
5940020   //-------------------------------------------------------
5940021   int
5940022   main (int argc, char *argv[], char *envp[])
5940023   {
5940024     bool option_l = 0;
5940025     bool option_u = 0;
5940026     int opt;
5940027   //extern char *optarg;                // not used.
5940028     extern int optind;
5940029     extern int optopt;
```

```
5940030     //
5940031     int status;
5940032     int sfdn;
5940033     int sfdn2;
5940034     struct sockaddr_in sa_local;
5940035     struct sockaddr_in sa_remote;
5940036     socklen_t sa_remote_size = sizeof (struct sockaddr_in);
5940037     ssize_t read_size;
5940038     ssize_t sent_size;
5940039     ssize_t recv_size;
5940040     char *addr = NULL;
5940041     char *port = NULL;
5940042     bool can_rx = 1;
5940043     bool can_tx = 1;
5940044     //
5940045     // Check for options.
5940046     //
5940047     while ((opt = getopt (argc, argv, ":ul")) != -1)
5940048       {
5940049         switch (opt)
5940050           {
5940051           case 'l':
5940052             option_l = 1;
5940053             break;
5940054           case 'u':
5940055             option_u = 1;
5940056             break;
5940057           case '?':
5940058             fprintf (stderr, "Unknown option -%c.\n", optopt);
5940059             usage ();
5940060             return (1);
5940061             break;
5940062           case ':':
5940063             fprintf (stderr,
5940064                      "Missing argument for option -%c\n",
5940065                      optopt);
5940066             usage ();
```

```
5940067             return (1);
5940068             break;
5940069           default:
5940070             fprintf (stderr,
5940071                      "Getopt problem: "
5940072                      "unknown option %c\n", opt);
5940073             usage ();
5940074             return (1);
5940075           }
5940076       }
5940077     //
5940078     // Arguments.
5940079     //
5940080     if (optind == (argc - 2))
5940081       {
5940082         //
5940083         // There are exactly two arguments: destination
5940084         // address and port.
5940085         //
5940086         addr = argv[argc - 2];
5940087         port = argv[argc - 1];
5940088       }
5940089     else
5940090       {
5940091         //
5940092         // Arguments wrong!
5940093         //
5940094         usage ();
5940095         return (2);
5940096       }
5940097     //
5940098     // Set the local or the remote address.
5940099     //
5940100     if (option_l)
5940101       {
5940102         //
5940103         // Address and port are local.
```

```
5940104         //
5940105         sa_local.sin_family = AF_INET;
5940106         sa_local.sin_port = htons (atoi (port));
5940107         inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
5940108       }
5940109     else
5940110       {
5940111         //
5940112         // Address and port are remote.
5940113         //
5940114         sa_remote.sin_family = AF_INET;
5940115         sa_remote.sin_port = htons (atoi (port));
5940116         inet_pton (AF_INET, addr, &sa_remote.sin_addr.s_addr);
5940117       }
5940118     //
5940119     // Open the socket.
5940120     //
5940121     if (option_u)
5940122       {
5940123         sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
5940124       }
5940125     else
5940126       {
5940127         sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
5940128       }
5940129     if (sfdn < 0)
5940130       {
5940131         perror (NULL);
5940132         return (3);
5940133       }
5940134     //
5940135     // Set it listening or connect.
5940136     //
5940137     if (option_l)
5940138       {
5940139         //
5940140         // Bind the local 'sa' location.
```

```
5940141         //
5940142         status =
5940143           bind (sfdn, (struct sockaddr *) &sa_local,
5940144                 sizeof (sa_local));
5940145         if (status < 0)
5940146           {
5940147             perror (NULL);
5940148             close (sfdn);
5940149             return (4);
5940150           }
5940151         //
5940152         // Listen (TCP) or wait the first packet (UDP).
5940153         //
5940154         if (option_u)
5940155           {
5940156             //
5940157             // Instead of listening, we use the function
5940158             // 'recvfrom()',
5940159             // to get the remote address and port.
5940160             //
5940161             recv_size =
5940162               recvfrom (sfdn, &buffer,
5940163                         (size_t) BUFSIZ - 1, 0,
5940164                         (struct sockaddr *) &sa_remote,
5940165                         &sa_remote_size);
5940166             if (recv_size < 0)
5940167               {
5940168                 perror (NULL);
5940169                 close (sfdn);
5940170                 return (4);
5940171               }
5940172             //
5940173             // Now connect the remote destination
5940174             //
5940175             status =
5940176               connect (sfdn,
5940177                        (struct sockaddr *) &sa_remote,
```

```
5940178                         sizeof (sa_remote));
5940179             if (status < 0)
5940180               {
5940181                 perror (NULL);
5940182                 close (sfdn);
5940183                 return (7);
5940184               }
5940185             //
5940186             // And show what was received as a first
5940187             // packet.
5940188             //
5940189             buffer[recv_size] = 0;
5940190             printf ("%s", buffer);
5940191           }
5940192         else
5940193           {
5940194             //
5940195             // TCP: listen.
5940196             //
5940197             status = listen (sfdn, 1);
5940198             if (status < 0)
5940199               {
5940200                 perror (NULL);
5940201                 close (sfdn);
5940202                 return (5);
5940203               }
5940204             //
5940205             // Accept.
5940206             //
5940207             sfdn2 =
5940208               accept (sfdn,
5940209                       (struct sockaddr *) &sa_remote,
5940210                       &sa_remote_size);
5940211             if (sfdn2 < 0)
5940212               {
5940213                 perror (NULL);
5940214                 close (sfdn);
```

```
5940215                 return (6);
5940216               }
5940217             //
5940218             // Close listening socket.
5940219             //
5940220             close (sfdn);
5940221             //
5940222             // Variable 'sfdn' will be the new socket.
5940223             //
5940224             sfdn = sfdn2;
5940225           }
5940226       }
5940227     else
5940228       {
5940229         //
5940230         // Connect the remote destination.
5940231         //
5940232         status =
5940233           connect (sfdn, (struct sockaddr *) &sa_remote,
5940234                    sizeof (sa_remote));
5940235         if (status < 0)
5940236           {
5940237             perror (NULL);
5940238             close (sfdn);
5940239             return (7);
5940240           }
5940241       }
5940242     //
5940243     // Define the standard input non blocking.
5940244     //
5940245     status = fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
5940246     if (status < 0)
5940247       {
5940248         perror (NULL);
5940249         return (8);
5940250       }
5940251     //
```

```
5940252     // Define the socket non blocking.
5940253     //
5940254     status = fcntl (sfdn, F_SETFL, O_NONBLOCK);
5940255     if (status < 0)
5940256       {
5940257         perror (NULL);
5940258         return (9);
5940259       }
5940260     //
5940261     // Will read from the remote and show to the screen.
5940262     //
5940263     while (can_rx || can_tx)
5940264       {
5940265         if (can_rx)
5940266           {
5940267             recv_size =
5940268               recv (sfdn, &buffer, (size_t) BUFSIZ - 1, 0);
5940269 //          recv_size = read (sfdn, &buffer,
5940270 //                            (size_t) BUFSIZ-1);
5940271             if (recv_size < 0)
5940272               {
5940273                 if (errno == EAGAIN || errno == EWOULDBLOCK)
5940274                   {
5940275                     ;
5940276                   }
5940277                 else
5940278                   {
5940279                     perror (NULL);
5940280                     close (sfdn);
5940281                     return (10);
5940282                   }
5940283               }
5940284             else if (recv_size == 0)
5940285               {
5940286                 //
5940287                 // End of stream.
5940288                 //
```

```
5940289                 can_rx = 0;
5940290                 printf ("--end of receive stream--\n");
5940291               }
5940292             else
5940293               {
5940294                 buffer[recv_size] = 0;
5940295                 printf ("%s", buffer);
5940296               }
5940297           }
5940298         if (can_tx)
5940299           {
5940300             read_size = read (STDIN_FILENO, buffer, BUFSIZ);
5940301             if (read_size < 0)
5940302               {
5940303                 if (errno == EAGAIN || errno == EWOULDBLOCK)
5940304                   {
5940305                     ;
5940306                   }
5940307                 else
5940308                   {
5940309                     perror (NULL);
5940310                     close (sfdn);
5940311                     return (11);
5940312                   }
5940313               }
5940314             else if (read_size == 0)
5940315               {
5940316                 //
5940317                 // End of input.
5940318                 //
5940319                 printf ("--closing send stream--\n");
5940320                 can_tx = 0;
5940321               }
5940322             else
5940323               {
5940324                 //
5940325                 // Send it.
```

```
5940326                 //
5940327                 sent_size =
5940328                   send (sfdn, &buffer, (size_t) read_size, 0);
5940329                 if (sent_size < 0)
5940330                   {
5940331                     if (errno == EAGAIN
5940332                         || errno == EWOULDBLOCK)
5940333                       {
5940334                         ;
5940335                       }
5940336                     else
5940337                       {
5940338                         perror (NULL);
5940339                         close (sfdn);
5940340                         return (12);
5940341                       }
5940342                   }
5940343               }
5940344           }
5940345       }
5940346     //
5940347     // All done.
5940348     //
5940349     close (sfdn);
5940350     return (0);
5940351   }
5940352
5940353   //-------------------------------------------------------------
5940354   static void
5940355   usage (void)
5940356   {
5940357     fprintf
5940358       (stderr,
5940359        "os32 netcat usage:\n"
5940360        "\n"
5940361        "nc [-u][-l] ADDRESS PORT\n"
5940362        "\n"
```

```
5940363         "-u      Use UDP protocol instead of TCP.\n"
5940364         "-l      Listen for incoming connection \n"
5940365         "        requests.\n"
5940366         "ADDRESS IPv4 numeric address; if option -l is\n"
5940367         "        used, this\n"
5940368         "        is the local address, otherwise it is\n"
5940369         "        the remote address.\n"
5940370         "PORT    TCP or UDP port; if option -l is used,\n"
5940371         "        this is local address, otherwise it is\n"
5940372         "        the remote address.\n");
5940373 }
```

96.1.30 applic/ping.c

Si veda la sezione 92.8.

```
5950001 #include <stdio.h>
5950002 #include <sys/types.h>
5950003 #include <arpa/inet.h>
5950004 #include <sys/socket.h>
5950005 #include <netinet/icmp.h>
5950006 #include <unistd.h>
5950007 #include <stdlib.h>
5950008 #include <stdint.h>
5950009 #include <errno.h>
5950010 //----------------------------------------------------------
5950011 struct ip_pkt
5950012 {
5950013   struct iphdr ip;       // <netinet/ip.h>
5950014   struct icmphdr icmp;   // <netinet/icmp.h>
5950015   char data[60];
5950016 } __attribute__ ((packed));
5950017
5950018 struct icmp_pkt
5950019 {
5950020   struct icmphdr icmp;   // <netinet/icmp.h>
5950021   char data[60];
```

```
5950022 } __attribute__ ((packed));
5950023 //-----------------------------------------------------------
5950024 static uint16_t ip_chk (uint16_t * data, size_t size);
5950025 static void usage (void);
5950026 //-----------------------------------------------------------
5950027 int
5950028 main (int argc, char *argv[], char *envp[])
5950029 {
5950030   int sfdn;
5950031   struct sockaddr_in sa;
5950032   ssize_t sent;
5950033   ssize_t received;
5950034   int status;
5950035   char *destination;
5950036   uint16_t checksum;
5950037   struct icmp_pkt icmp_pkt_send;
5950038   struct ip_pkt ip_pkt_receive;
5950039 //int                 retry = 3;     // Max send retry.
5950040   clock_t clock_ping;
5950041   clock_t clock_pong;
5950042   clock_t clock_time;
5950043   //
5950044   // No options are known, but an argument must be
5950045   // given.
5950046   //
5950047   if (argc < 2)
5950048     {
5950049       usage ();
5950050       return (1);
5950051     }
5950052   destination = argv[1];
5950053   //
5950054   // Define the destination 'sa'
5950055   //
5950056   sa.sin_family = AF_INET;
5950057   sa.sin_port = 0;
5950058   inet_pton (AF_INET, destination, &sa.sin_addr.s_addr);
```

```
5950059     //
5950060     // Put some data inside the packet header.
5950061     //
5950062     icmp_pkt_send.icmp.un.echo.id = (uint16_t) rand ();
5950063     icmp_pkt_send.icmp.un.echo.sequence = 0;
5950064     icmp_pkt_send.icmp.type = 8;  // Echo request.
5950065     icmp_pkt_send.icmp.code = 0;
5950066     icmp_pkt_send.icmp.checksum = 0;
5950067     //
5950068     // Calculate the ICMP checksum.
5950069     //
5950070     checksum = ~(ip_chk ((void *) &icmp_pkt_send,
5950071                          sizeof (struct icmp_pkt)));
5950072     icmp_pkt_send.icmp.checksum = htons (checksum);
5950073     //
5950074     // Open the socket.
5950075     //
5950076     sfdn = socket (AF_INET, SOCK_RAW, IPPROTO_ICMP);
5950077     if (sfdn < 0)
5950078       {
5950079         perror (NULL);
5950080         return (0);
5950081       }
5950082     //
5950083     // Connect the 'sa' destination
5950084     //
5950085     status =
5950086       connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
5950087     if (status < 0)
5950088       {
5950089         perror (NULL);
5950090         close (sfdn);
5950091         return (0);
5950092       }
5950093     //
5950094     // Send one single packet: please notice that we
5950095     // send an ICMP,
```

```
5950096     // but we receive a full IP.
5950097     //
5950098     clock_ping = clock ();
5950099     sent = send (sfdn, &icmp_pkt_send,
5950100                  sizeof (struct icmp_pkt), 0);
5950101     if (sent < 0)
5950102       {
5950103         perror (NULL);
5950104         return (1);
5950105       }
5950106     //
5950107     // Packet sent.
5950108     //
5950109     printf ("ping: ");
5950110     //
5950111     // Now receive all ICMP raw packets, and select the
5950112     // one with the same
5950113     // identifier.
5950114     //
5950115     while (1)
5950116       {
5950117         received =
5950118           read (sfdn, &ip_pkt_receive,
5950119                 sizeof (struct ip_pkt));
5950120         clock_pong = clock ();
5950121
5950122         if (ip_pkt_receive.icmp.un.echo.id
5950123             == icmp_pkt_send.icmp.un.echo.id)
5950124           {
5950125             clock_time = (clock_pong - clock_ping);
5950126             printf ("pong %llu.%03llu s\n",
5950127                     clock_time / CLOCKS_PER_SEC,
5950128                     (clock_time % CLOCKS_PER_SEC) *
5950129                     1000 / CLOCKS_PER_SEC);
5950130             break;
5950131           }
5950132       }
```

```
5950133
5950134     close (sfdn);
5950135     return (0);
5950136   }
5950137
5950138   //-------------------------------------------------------
5950139   static uint16_t
5950140   ip_chk (uint16_t * data, size_t size)
5950141   {
5950142     int i;
5950143     uint32_t sum;
5950144     uint16_t carry;
5950145     uint16_t checksum;
5950146     uint16_t last;
5950147     uint8_t *octet = (uint8_t *) data;
5950148     //
5950149     // 2's complement sum.
5950150     //
5950151     for (i = 0, sum = 0; i < (size / 2); i++)
5950152       {
5950153         sum += ntohs (data[i]);
5950154       }
5950155     //
5950156     if (size % 2)
5950157       {
5950158         //
5950159         // The size is odd, and the last octet must be
5950160         // accounted too.
5950161         //
5950162         last = octet[size - 1];
5950163         last = last << 8;
5950164         //
5950165         sum += last;
5950166       }
5950167     //
5950168     // Extract the carries and make the checksum.
5950169     //
```

```
5950170     carry = sum >> 16;
5950171     checksum = sum & 0x0000FFFF;
5950172     checksum += carry;
5950173     //
5950174     // End of job.
5950175     //
5950176     return (checksum);
5950177   }
5950178
5950179   //---------------------------------------------------------
5950180   static void
5950181   usage (void)
5950182   {
5950183     fprintf (stderr, "Usage: ping IPv4\n");
5950184   }
```

96.1.31 applic/ps.c

«

Si veda la sezione 86.21.

```
5960001   #include <sys/os32.h>
5960002   #include <kernel/proc.h>
5960003   #include <unistd.h>
5960004   #include <stdio.h>
5960005   #include <fcntl.h>
5960006   #include <unistd.h>
5960007   #include <stdlib.h>
5960008   //---------------------------------------------------------
5960009   static void usage (void);
5960010   //---------------------------------------------------------
5960011   int
5960012   main (int argc, char *argv[], char *envp[])
5960013   {
5960014     pid_t pid;
5960015     proc_t *ps;
5960016     int fd;
5960017     char stat;
```

```
5960018     ssize_t size_read;
5960019     char buffer[sizeof (proc_t)];
5960020     unsigned int min;
5960021     unsigned int sec;
5960022     size_t stack_size;
5960023     addr_t stack_bottom;
5960024     int stack_usage;
5960025     //
5960026     int opt;
5960027   //  extern char   *optarg;              // not used.
5960028   //  extern int     optind;
5960029     extern int optopt;
5960030     int option_u = 0;
5960031     int option_g = 0;
5960032     //
5960033     // Check for options.
5960034     //
5960035     while ((opt = getopt (argc, argv, ":ug")) != -1)
5960036       {
5960037         switch (opt)
5960038           {
5960039           case 'u':
5960040             option_u = 1;
5960041             break;
5960042           case 'g':
5960043             option_g = 1;
5960044             break;
5960045           case '?':
5960046             fprintf (stderr, "Unknown option -%c.\n", optopt);
5960047             usage ();
5960048             return (1);
5960049             break;
5960050           case ':':
5960051             fprintf (stderr,
5960052                      "Missing argument for option -%c\n",
5960053                      optopt);
5960054             usage ();
```

```
5960055             return (1);
5960056             break;
5960057           default:
5960058             fprintf (stderr,
5960059                      "Getopt problem: "
5960060                      "unknown option %c\n", opt);
5960061             return (1);
5960062           }
5960063       }
5960064     //
5960065     // Fix options '-u' or '-g'.
5960066     //
5960067     if (option_u)
5960068       option_g = 0;
5960069     if (!option_g)
5960070       option_u = 1;
5960071     //
5960072     // Open '/dev/kmem_ps', to get the process table.
5960073     //
5960074     fd = open ("/dev/kmem_ps", O_RDONLY);
5960075     if (fd < 0)
5960076       {
5960077         printf ("[%s] Cannot open \"/dev/kmem_ps\" ",
5960078                 argv[0]);
5960079         perror (NULL);
5960080         exit (0);
5960081       }
5960082     //
5960083     // Print head.
5960084     //
5960085     if (option_u)
5960086       {
5960087         printf ("pp  p pg                                  "
5960088                 "T * 0x1000 D * 0x1000 stack        \n"
5960089                 "id id rp  tty  uid euid suid usage s "
5960090                 "addr  size addr  size usage    name\n");
5960091       }
```

```
5960092     else
5960093       {
5960094         printf ("pp  p pg                                  "
5960095                 "T * 0x1000 D * 0x1000 stack         \n"
5960096                 "id id rp  tty  gid egid sgid usage s "
5960097                 "addr  size addr  size usage    name\n");
5960098       }
5960099     //
5960100     // Scan processes and then print body.
5960101     //
5960102     for (pid = 0; pid < PROCESS_MAX; pid++)
5960103       {
5960104         lseek (fd, (off_t) pid, SEEK_SET);
5960105         size_read = read (fd, buffer, sizeof (proc_t));
5960106         if (size_read < sizeof (proc_t))
5960107           {
5960108             printf
5960109               ("[%s] Cannot read "
5960110                "\"/dev/kmem_ps\" pid %i ", argv[0], pid);
5960111             perror (NULL);
5960112             continue;
5960113           }
5960114         ps = (proc_t *) buffer;
5960115         if (ps->status > 0)
5960116           {
5960117             ps->name[PATH_MAX - 1] = 0;    // Terminated
5960118             // string.
5960119             //
5960120             // Check the current stack size.
5960121             //
5960122             if (ps->domain_data == 0)
5960123               {
5960124                 stack_bottom = ps->domain_text;
5960125               }
5960126             else
5960127               {
5960128                 stack_bottom = ps->domain_data;
```

```
5960129             }
5960130           //
5960131           stack_size = stack_bottom - ps->sp;
5960132           //
5960133           stack_usage = 100 * stack_size / ps->domain_stack;
5960134           //
5960135           switch (ps->status)
5960136             {
5960137             case PROC_EMPTY:
5960138               stat = '-';
5960139               break;
5960140             case PROC_CREATED:
5960141               stat = 'c';
5960142               break;
5960143             case PROC_READY:
5960144               stat = 'r';
5960145               break;
5960146             case PROC_RUNNING:
5960147               stat = 'R';
5960148               break;
5960149             case PROC_SLEEPING:
5960150               stat = 's';
5960151               break;
5960152             case PROC_ZOMBIE:
5960153               stat = 'z';
5960154               break;
5960155             default:
5960156               stat = '?';
5960157               break;
5960158             }
5960159           //
5960160           min = ((ps->usage / CLOCKS_PER_SEC) / 60);
5960161           sec = ((ps->usage / CLOCKS_PER_SEC) % 60);
5960162           //
5960163           // Print the line.
5960164           //
5960165           //
```

```
5960166             // Addresses and sizes are multiple of 4096
5960167             // (0x1000);
5960168             // for the stack pointer is shown only the
5960169             // last five
5960170             // hexadecimal digits.
5960171             //
5960172             if (ps->domain_data > 0)
5960173               {
5960174                 if (option_u)
5960175                   {
5960176                     printf
5960177                       ("%2i %2i %2i %04x %4i %4i %4i "
5960178                        "%02i.%02i %c %05x %04x %05x "
5960179                        "%04x % 3i%% %15s\n",
5960180                        (unsigned int) ps->ppid,
5960181                        (unsigned int) pid,
5960182                        (unsigned int) ps->pgrp,
5960183                        (unsigned int) ps->device_tty,
5960184                        (unsigned int) ps->uid,
5960185                        (unsigned int) ps->euid,
5960186                        (unsigned int) ps->suid, min, sec,
5960187                        stat,
5960188                        (unsigned int) ps->address_text /
5960189                        MEM_BLOCK_SIZE,
5960190                        (unsigned int) ps->domain_text /
5960191                        MEM_BLOCK_SIZE,
5960192                        (unsigned int) ps->address_data /
5960193                        MEM_BLOCK_SIZE,
5960194                        (unsigned int) (ps->domain_data +
5960195                                        ps->extra_data) /
5960196                        MEM_BLOCK_SIZE,
5960197                        (unsigned int) stack_usage, ps->name);
5960198                   }
5960199                 else
5960200                   {
5960201                     printf
5960202                       ("%2i %2i %2i %04x %4i %4i "
```

```
5960203                         "%4i %02i.%02i %c %05x %04x "
5960204                         "%05x %04x % 3i%% %15s\n",
5960205                         (unsigned int) ps->ppid,
5960206                         (unsigned int) pid,
5960207                         (unsigned int) ps->pgrp,
5960208                         (unsigned int) ps->device_tty,
5960209                         (unsigned int) ps->gid,
5960210                         (unsigned int) ps->egid,
5960211                         (unsigned int) ps->sgid, min, sec,
5960212                         stat,
5960213                         (unsigned int) ps->address_text /
5960214                         MEM_BLOCK_SIZE,
5960215                         (unsigned int) ps->domain_text /
5960216                         MEM_BLOCK_SIZE,
5960217                         (unsigned int) ps->address_data /
5960218                         MEM_BLOCK_SIZE,
5960219                         (unsigned int) (ps->domain_data +
5960220                                         ps->extra_data) /
5960221                         MEM_BLOCK_SIZE,
5960222                         (unsigned int) stack_usage, ps->name);
5960223                   }
5960224               }
5960225             else
5960226               {
5960227                 if (option_u)
5960228                   {
5960229                     printf
5960230                       ("%2i %2i %2i %04x %4i %4i %4i "
5960231                        "%02i.%02i %c %05x %04x %05x "
5960232                        "%04x % 3i%% %15s\n",
5960233                        (unsigned int) ps->ppid,
5960234                        (unsigned int) pid,
5960235                        (unsigned int) ps->pgrp,
5960236                        (unsigned int) ps->device_tty,
5960237                        (unsigned int) ps->uid,
5960238                        (unsigned int) ps->euid,
5960239                        (unsigned int) ps->suid, min, sec,
```

```
5960240                         stat,
5960241                         (unsigned int) ps->address_text /
5960242                         MEM_BLOCK_SIZE,
5960243                         (unsigned int) (ps->domain_text +
5960244                                         ps->extra_data) /
5960245                         MEM_BLOCK_SIZE,
5960246                         (unsigned int) ps->address_data /
5960247                         MEM_BLOCK_SIZE,
5960248                         (unsigned int) ps->domain_data /
5960249                         MEM_BLOCK_SIZE,
5960250                         (unsigned int) stack_usage, ps->name);
5960251                   }
5960252                 else
5960253                   {
5960254                     printf
5960255                       ("%2i %2i %2i %04x %4i %4i %4i "
5960256                        "%02i.%02i %c %05x %04x %05x "
5960257                        "%04x % 3i%% %15s\n",
5960258                        (unsigned int) ps->ppid,
5960259                        (unsigned int) pid,
5960260                        (unsigned int) ps->pgrp,
5960261                        (unsigned int) ps->device_tty,
5960262                        (unsigned int) ps->gid,
5960263                        (unsigned int) ps->egid,
5960264                        (unsigned int) ps->sgid, min, sec,
5960265                        stat,
5960266                        (unsigned int) ps->address_text /
5960267                        MEM_BLOCK_SIZE,
5960268                        (unsigned int) (ps->domain_text +
5960269                                        ps->extra_data) /
5960270                        MEM_BLOCK_SIZE,
5960271                        (unsigned int) ps->address_data /
5960272                        MEM_BLOCK_SIZE,
5960273                        (unsigned int) ps->domain_data /
5960274                        MEM_BLOCK_SIZE,
5960275                        (unsigned int) stack_usage, ps->name);
5960276                   }
```

```
5960277                 }
5960278               }
5960279             }
5960280           close (fd);
5960281           return (0);
5960282         }
5960283
5960284         //-------------------------------------------------------
5960285         static void
5960286         usage (void)
5960287         {
5960288           fprintf (stderr, "Usage: ps [-u|g]\n");
5960289         }
```

96.1.32 applic/rm.c

«

Si veda la sezione 86.22.

```
5970001   #include <fcntl.h>
5970002   #include <sys/stat.h>
5970003   #include <stddef.h>
5970004   #include <unistd.h>
5970005   #include <errno.h>
5970006   //-------------------------------------------------------
5970007   static void usage (void);
5970008   //-------------------------------------------------------
5970009   int
5970010   main (int argc, char *argv[], char *envp[])
5970011   {
5970012     int a;          // Argument index.
5970013     int status;
5970014     struct stat file_status;
5970015     //
5970016     // No options are known, but at least an argument
5970017     // must be given.
5970018     //
5970019     if (argc < 2)
```

```
5970020       {
5970021         usage ();
5970022         return (1);
5970023       }
5970024     //
5970025     // Scan arguments.
5970026     //
5970027     for (a = 1; a < argc; a++)
5970028       {
5970029         //
5970030         // Verify if the file exists.
5970031         //
5970032         if (stat (argv[a], &file_status) != 0)
5970033           {
5970034             fprintf (stderr,
5970035                      "File \"%s\" does not exist!\n",
5970036                      argv[a]);
5970037             continue;
5970038           }
5970039         //
5970040         // File exists: check the file type.
5970041         //
5970042         if (S_ISDIR (file_status.st_mode))
5970043           {
5970044             fprintf (stderr,
5970045                      "Cannot remove directory \"%s\"!\n",
5970046                      argv[a]);
5970047             continue;
5970048           }
5970049         //
5970050         // Can remove it.
5970051         //
5970052         status = unlink (argv[a]);
5970053         if (status != 0)
5970054           {
5970055             perror (NULL);
5970056             return (2);
```

```
5970057           }
5970058       }
5970059     return (0);
5970060   }
5970061
5970062   //-------------------------------------------------------
5970063   static void
5970064   usage (void)
5970065   {
5970066     fprintf (stderr, "Usage: rm FILE...\n");
5970067   }
```

96.1.33 applic/rmdir.c

«

Si veda la sezione 86.23.

```
5980001   #include <fcntl.h>
5980002   #include <sys/stat.h>
5980003   #include <stddef.h>
5980004   #include <unistd.h>
5980005   #include <errno.h>
5980006   //-------------------------------------------------------
5980007   static void usage (void);
5980008   //-------------------------------------------------------
5980009   int
5980010   main (int argc, char *argv[], char *envp[])
5980011   {
5980012     int a;          // Argument index.
5980013     int status;
5980014     struct stat file_status;
5980015     //
5980016     // No options are known, but at least an argument
5980017     // must be given.
5980018     //
5980019     if (argc < 2)
5980020       {
5980021         usage ();
```

```
5980022         return (1);
5980023       }
5980024     //
5980025     // Scan arguments.
5980026     //
5980027     for (a = 1; a < argc; a++)
5980028       {
5980029         //
5980030         // Verify if the file exists.
5980031         //
5980032         if (stat (argv[a], &file_status) != 0)
5980033           {
5980034             fprintf (stderr,
5980035                      "File \"%s\" does not exist!\n",
5980036                      argv[a]);
5980037             continue;
5980038           }
5980039         //
5980040         // File exists: check the file type.
5980041         //
5980042         if (!S_ISDIR (file_status.st_mode))
5980043           {
5980044             fprintf (stderr,
5980045                      "Cannot remove file \"%s\"!\n", argv[a]);
5980046             continue;
5980047           }
5980048         //
5980049         // Can try to remove it.
5980050         //
5980051         status = rmdir (argv[a]);
5980052         if (status != 0)
5980053           {
5980054             perror (NULL);
5980055             return (2);
5980056           }
5980057       }
5980058     return (0);
```

```
5980059 }
5980060
5980061 //---------------------------------------------------------
5980062 static void
5980063 usage (void)
5980064 {
5980065   fprintf (stderr, "Usage: rmdir DIR...\n");
5980066 }
```

96.1.34 applic/route.c

«

Si veda la sezione 92.9.

```
5990001 #include <sys/os32.h>
5990002 #include <kernel/net/route.h>
5990003 #include <kernel/net.h>
5990004 #include <unistd.h>
5990005 #include <stdio.h>
5990006 #include <fcntl.h>
5990007 #include <unistd.h>
5990008 #include <stdlib.h>
5990009 //---------------------------------------------------------
5990010 int
5990011 main (int argc, char *argv[], char *envp[])
5990012 {
5990013   int fd;
5990014   ssize_t size_read;
5990015   char buffer[sizeof (route_t)];
5990016   int n;
5990017   route_t *route_table_item;
5990018   char string[80];
5990019
5990020   //
5990021   // All options are ignored, at the moment
5990022   //
5990023
5990024   //
```

```
5990025     // Open '/dev/kmem_route', to get the routing table.
5990026     //
5990027     fd = open ("/dev/kmem_route", O_RDONLY);
5990028     if (fd < 0)
5990029       {
5990030         printf ("[%s] Cannot open \"/dev/kmem_route\" ",
5990031                 argv[0]);
5990032         perror (NULL);
5990033         exit (0);
5990034       }
5990035     //
5990036     // Print header.
5990037     //
5990038     printf ("Destination/mask    "
5990039             "Router          " "Interface\n");
5990040     //
5990041     // Scan route table items and then print body.
5990042     //
5990043     for (n = 0; n < ROUTE_MAX_ROUTES; n++)
5990044       {
5990045         lseek (fd, (off_t) n, SEEK_SET);
5990046         size_read = read (fd, buffer, sizeof (route_t));
5990047         if (size_read < sizeof (route_t))
5990048           {
5990049             printf
5990050               ("[%s] Cannot read "
5990051                "\"/dev/kmem_route\" item %i ", argv[0], n);
5990052             perror (NULL);
5990053             continue;
5990054           }
5990055         //
5990056         route_table_item = (route_t *) buffer;
5990057         //
5990058         if (route_table_item->network == 0xFFFFFFFF)
5990059           {
5990060             //
5990061             // Empty item.
```

```
5990062             //
5990063             continue;
5990064           }
5990065         //
5990066         sprintf (string, "%i.%i.%i.%i/%i"
5990067                  "                    ",
5990068                  route_table_item->network >> 24 & 0x000000FF,
5990069                  route_table_item->network >> 16 & 0x000000FF,
5990070                  route_table_item->network >> 8 & 0x000000FF,
5990071                  route_table_item->network >> 0 & 0x000000FF,
5990072                  route_table_item->m);
5990073         string[19] = '\0';
5990074         printf ("%s", string);
5990075         //
5990076         if (route_table_item->router == 0)
5990077           {
5990078             printf ("               ");
5990079           }
5990080         else
5990081           {
5990082             sprintf (string, "%i.%i.%i.%i"
5990083                      "           ",
5990084                      route_table_item->router >> 24 &
5990085                      0x000000FF,
5990086                      route_table_item->router >> 16 &
5990087                      0x000000FF,
5990088                      route_table_item->router >> 8 &
5990089                      0x000000FF,
5990090                      route_table_item->router >> 0 &
5990091                      0x000000FF);
5990092             string[16] = '\0';
5990093             printf ("%s", string);
5990094           }
5990095         //
5990096         printf ("net%i\n", route_table_item->interface);
5990097       }
5990098     close (fd);
```

```
5990099   return (0);
5990100 }
```

96.1.35 applic/shell.c

«

Si veda la sezione 86.24.

```
6000001 #include <unistd.h>
6000002 #include <stdlib.h>
6000003 #include <sys/stat.h>
6000004 #include <sys/types.h>
6000005 #include <fcntl.h>
6000006 #include <errno.h>
6000007 #include <unistd.h>
6000008 #include <signal.h>
6000009 #include <stdio.h>
6000010 #include <sys/wait.h>
6000011 #include <stdio.h>
6000012 #include <string.h>
6000013 #include <limits.h>
6000014 #include <sys/os32.h>
6000015 //-------------------------------------------------------
6000016 #define PROMPT_SIZE     16
6000017 //-------------------------------------------------------
6000018 static void sh_cd (int argc, char *argv[]);
6000019 static void sh_pwd (int argc, char *argv[]);
6000020 static void sh_umask (int argc, char *argv[]);
6000021 //-------------------------------------------------------
6000022 int
6000023 main (int argc, char *argv[], char *envp[])
6000024 {
6000025   char buffer_cmd[ARG_MAX / 2];
6000026   char *argv_cmd[ARG_MAX / 16];
6000027 //char   prompt[PROMPT_SIZE];
6000028   uid_t uid;
6000029   int argc_cmd;
6000030   pid_t pid_cmd;
```

```
6000031     pid_t pid_dead;
6000032     int status;
6000033     void *pstatus;
6000034     int i;
6000035     //
6000036     //
6000037     //
6000038     uid = geteuid ();
6000039     //
6000040     // Load processes, reading the keyboard.
6000041     //
6000042     while (1)
6000043       {
6000044         if (uid == 0)
6000045           {
6000046             printf ("# ");
6000047           }
6000048         else
6000049           {
6000050             printf ("$ ");
6000051           }
6000052         //
6000053         pstatus = fgets (buffer_cmd, (ARG_MAX / 2), stdin);
6000054         if (pstatus == NULL)
6000055           {
6000056             if (errno)
6000057               {
6000058                 perror (NULL);
6000059                 continue;
6000060               }
6000061             else
6000062               {
6000063                 //
6000064                 // End of file, like ^D.
6000065                 //
6000066                 return (0);
6000067               }
```

```
6000068           }
6000069         //
6000070         i = strlen (buffer_cmd);
6000071         if (i > 0 && buffer_cmd[i - 1] == '\n')
6000072           {
6000073             buffer_cmd[i - 1] = '\0';
6000074           }
6000075         //
6000076         // Clear 'argv_cmd[]';
6000077         //
6000078         for (argc_cmd = 0; argc_cmd < (ARG_MAX / 16);
6000079              argc_cmd++)
6000080           {
6000081             argv_cmd[argc_cmd] = NULL;
6000082           }
6000083         //
6000084         // Initialize the command scan.
6000085         //
6000086         argv_cmd[0] = strtok (buffer_cmd, " \t");
6000087         //
6000088         // Verify: if the input is not valid, loop
6000089         // again.
6000090         //
6000091         if (argv_cmd[0] == NULL)
6000092           {
6000093             continue;
6000094           }
6000095         //
6000096         // Find the arguments.
6000097         //
6000098         for (argc_cmd = 1;
6000099              argc_cmd < ((ARG_MAX / 16) - 1)
6000100              && argv_cmd[argc_cmd - 1] != NULL; argc_cmd++)
6000101           {
6000102             argv_cmd[argc_cmd] = strtok (NULL, " \t");
6000103           }
6000104         //
```

```
6000105         // If there are too many arguments, show a
6000106         // message and continue.
6000107         //
6000108         if (argv_cmd[argc_cmd - 1] != NULL)
6000109           {
6000110             errset (E2BIG);       // Argument list too
6000111             // long.
6000112             perror (NULL);
6000113             continue;
6000114           }
6000115         //
6000116         // Correct the value for `argc_cmd', because
6000117         // actually
6000118         // it counts also the NULL element.
6000119         //
6000120         argc_cmd--;
6000121         //
6000122         // Verify if it is an internal command.
6000123         //
6000124         if (strcmp (argv_cmd[0], "exit") == 0)
6000125           {
6000126             return (0);
6000127           }
6000128         else if (strcmp (argv_cmd[0], "cd") == 0)
6000129           {
6000130             sh_cd (argc_cmd, argv_cmd);
6000131             continue;
6000132           }
6000133         else if (strcmp (argv_cmd[0], "pwd") == 0)
6000134           {
6000135             sh_pwd (argc_cmd, argv_cmd);
6000136             continue;
6000137           }
6000138         else if (strcmp (argv_cmd[0], "umask") == 0)
6000139           {
6000140             sh_umask (argc_cmd, argv_cmd);
6000141             continue;
```

```
6000142           }
6000143         //
6000144         // It should be a program to run.
6000145         //
6000146         pid_cmd = fork ();
6000147         if (pid_cmd == -1)
6000148           {
6000149             printf ("%s: cannot run command", argv[0]);
6000150             perror (NULL);
6000151           }
6000152         else if (pid_cmd == 0)
6000153           {
6000154             execvp (argv_cmd[0], argv_cmd);
6000155             perror (NULL);
6000156             exit (0);
6000157           }
6000158         while (1)
6000159           {
6000160             pid_dead = wait (&status);
6000161             if (pid_dead == pid_cmd)
6000162               {
6000163                 break;
6000164               }
6000165           }
6000166         printf ("pid %i terminated with status %i.\n",
6000167                 (int) pid_dead, status);
6000168       }
6000169   }
6000170
6000171   //---------------------------------------------------
6000172   static void
6000173   sh_cd (int argc, char *argv[])
6000174   {
6000175     int status;
6000176     //
6000177     if (argc != 2)
6000178       {
```

```
6000179         errset (EINVAL);  // Invalid argument.
6000180         perror (NULL);
6000181         return;
6000182       }
6000183     //
6000184     status = chdir (argv[1]);
6000185     if (status != 0)
6000186       {
6000187         perror (NULL);
6000188       }
6000189     return;
6000190   }
6000191
6000192   //-------------------------------------------------------
6000193   static void
6000194   sh_pwd (int argc, char *argv[])
6000195   {
6000196     char path[PATH_MAX];
6000197     void *pstatus;
6000198     //
6000199     if (argc != 1)
6000200       {
6000201         errset (EINVAL);  // Invalid argument.
6000202         perror (NULL);
6000203         return;
6000204       }
6000205     //
6000206     // Get the current directory.
6000207     //
6000208     pstatus = getcwd (path, (size_t) PATH_MAX);
6000209     if (pstatus == NULL)
6000210       {
6000211         perror (NULL);
6000212       }
6000213     else
6000214       {
6000215         printf ("%s\n", path);
```

```
6000216       }
6000217     return;
6000218   }
6000219
6000220   //-------------------------------------------------------
6000221   static void
6000222   sh_umask (int argc, char *argv[])
6000223   {
6000224     sysmsg_uarea_t msg;
6000225     char *m;       // Index inside the umask octal
6000226     // string.
6000227     int mask;
6000228     int digit;
6000229     //
6000230     if (argc > 2)
6000231       {
6000232         errset (EINVAL);  // Invalid argument.
6000233         perror (NULL);
6000234         return;
6000235       }
6000236     //
6000237     // If no argument is available, the umask is shown,
6000238     // with a direct
6000239     // system call.
6000240     //
6000241     if (argc == 1)
6000242       {
6000243         sys (SYS_UAREA, &msg, (sizeof msg));
6000244         printf ("%04o\n", msg.umask);
6000245         return;
6000246       }
6000247     //
6000248     // Get the mask: must be the first argument.
6000249     //
6000250     for (mask = 0, m = argv[1]; *m != 0; m++)
6000251       {
6000252         digit = (*m - '0');
```

```
6000253         if (digit < 0 || digit > 7)
6000254           {
6000255             errset (EINVAL);       // Invalid argument.
6000256             perror (NULL);
6000257             return;
6000258           }
6000259         mask = mask * 8 + digit;
6000260       }
6000261     //
6000262     // Set the umask and return.
6000263     //
6000264     umask (mask);
6000265     return;
6000266   }
```

96.1.36 applic/t_fcntl.c

«

Si veda la sezione 86.25.

```
6010001 #include <stdio.h>
6010002 #include <unistd.h>
6010003 #include <stdlib.h>
6010004 #include <sys/wait.h>
6010005 #include <fcntl.h>
6010006 //-------------------------------------------------------
6010007 int
6010008 main (void)
6010009 {
6010010   int status;
6010011
6010012
6010013   printf ("opzione O_NONBLOCK=%08x\n", O_NONBLOCK);
6010014
6010015   fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
6010016
6010017   status = fcntl (STDIN_FILENO, F_GETFL);
6010018
```

```
6010019
6010020
6010021     printf ("fcntl (STDIN_FILENO, F_GETFL) == %08x\n",
6010022             status);
6010023
6010024     return (0);
6010025
6010026   }
```

96.1.37 applic/t_fifo.c

«

Si veda la sezione 86.25.

```
6020001   #include <stdio.h>
6020002   #include <unistd.h>
6020003   #include <stdlib.h>
6020004   #include <sys/wait.h>
6020005   #include <signal.h>
6020006   #include <sys/wait.h>
6020007   #include <string.h>
6020008   #include <fcntl.h>
6020009   #include <sys/stat.h>
6020010   //-------------------------------------------------------
6020011   int
6020012   main (void)
6020013   {
6020014     int fd;
6020015     pid_t child;
6020016     char buffer;
6020017     char *message =
6020018       "ciao a tutti voi amici vicini e lontani\n";
6020019     int i;
6020020     size_t size;
6020021     ssize_t written;
6020022     int status;
6020023     //
6020024     //
```

```
6020025     //
6020026     unlink ("/tmp/fifo");
6020027     //
6020028     status =
6020029       mknod ("/tmp/fifo", S_IFIFO | S_IRUSR | S_IWUSR, 0);
6020030     if (status != 0)
6020031       {
6020032         perror ("mknod fifo");
6020033         exit (EXIT_FAILURE);
6020034       }
6020035     //
6020036     //
6020037     //
6020038     child = fork ();
6020039     if (child == -1)
6020040       {
6020041         perror ("fork");
6020042         exit (EXIT_FAILURE);
6020043       }
6020044     //
6020045     //
6020046     //
6020047     if (child == 0)
6020048       {
6020049         //
6020050         // This is the child and it have to read the
6020051         // fifo.
6020052         //
6020053         fd = open ("/tmp/fifo", O_RDONLY);
6020054         if (fd < 0)
6020055           {
6020056             perror ("fifo read open");
6020057             exit (EXIT_FAILURE);
6020058           }
6020059         //
6020060         // Read one byte at the time, as long as there
6020061         // is
```

```
6020062         // something to read.
6020063         //
6020064         while (read (fd, &buffer, 1) > 0)
6020065           {
6020066             write (STDOUT_FILENO, &buffer, 1);
6020067           }
6020068         //
6020069         // Close the fifo and exit the child.
6020070         //
6020071         close (fd);
6020072         //
6020073         exit (EXIT_SUCCESS);
6020074       }
6020075     else
6020076       {
6020077         //
6020078         // This is the parent process and it writes to
6020079         // the FIFO.
6020080         //
6020081         fd = open ("/tmp/fifo", O_WRONLY);
6020082         if (fd < 0)
6020083           {
6020084             perror ("fifo write open");
6020085             exit (EXIT_FAILURE);
6020086           }
6020087         //
6020088         while (1)
6020089           {
6020090             for (i = 0, written = 0, size =
6020091                  strlen (message); i < strlen (message);
6020092                  i += written, size -= written)
6020093               {
6020094                 written = write (fd, &message[i], size);
6020095                 if (written < 0)
6020096                   {
6020097                     perror ("pipe");
6020098                     close (fd);
```

```
6020099                     wait (NULL);  // Wait for child.
6020100                     exit (EXIT_FAILURE);
6020101                   }
6020102                 }
6020103               }
6020104             close (fd);         /* Reader will see EOF */
6020105             wait (NULL);        /* Wait for child */
6020106             exit (EXIT_SUCCESS);
6020107           }
6020108         //
6020109         return (0);
6020110       }
```

96.1.38 applic/t_grp.c

«

Si veda la sezione 86.25.

```
6030001  #include <stdio.h>
6030002  #include <grp.h>
6030003  #include <pwd.h>
6030004  #include <unistd.h>
6030005  #include <stdlib.h>
6030006  #include <sys/wait.h>
6030007  #include <signal.h>
6030008  #include <sys/wait.h>
6030009  #include <string.h>
6030010  #include <fcntl.h>
6030011  #include <sys/stat.h>
6030012  //-------------------------------------------------------
6030013  int
6030014  main (void)
6030015  {
6030016    struct passwd *pw;
6030017    struct group *gr;
6030018    int i;
6030019
6030020    pw = getpwuid ((uid_t) 1001);
```

```
6030021
6030022     if (pw == NULL)
6030023       {
6030024         perror (NULL);
6030025         exit (0);
6030026       }
6030027
6030028     printf ("%s:%s:%i:%i:\n", pw->pw_name, pw->pw_passwd,
6030029             pw->pw_uid, pw->pw_gid);
6030030
6030031     gr = getgrgid ((gid_t) 233);
6030032
6030033     if (gr == NULL)
6030034       {
6030035         perror (NULL);
6030036         exit (0);
6030037       }
6030038
6030039     printf ("%s:%s:%i:", gr->gr_name, gr->gr_passwd,
6030040             gr->gr_gid);
6030041     for (i = 0; i < 32 && gr->gr_mem[i] != NULL; i++)
6030042       {
6030043         printf ("%s,", gr->gr_mem[i]);
6030044       }
6030045
6030046     return (0);
6030047   }
```

96.1.39 applic/t_nc.c

Si veda la sezione 86.25.

```
6040001   #include <sys/stat.h>
6040002   #include <sys/types.h>
6040003   #include <unistd.h>
6040004   #include <stdlib.h>
6040005   #include <fcntl.h>
```

```
6040006 #include <errno.h>
6040007 #include <signal.h>
6040008 #include <stdio.h>
6040009 #include <string.h>
6040010 #include <limits.h>
6040011 #include <libgen.h>
6040012 #include <arpa/inet.h>
6040013 #include <sys/socket.h>
6040014 #include <stdint.h>
6040015 #include <stdbool.h>
6040016 #include <fcntl.h>
6040017 //------------------------------------------------------------
6040018 static void usage (void);
6040019 char buffer[BUFSIZ];
6040020 //------------------------------------------------------------
6040021 int
6040022 main (int argc, char *argv[], char *envp[])
6040023 {
6040024   bool option_l = 0;
6040025   bool option_u = 0;
6040026   int opt;
6040027 //extern char *optarg;                // not used.
6040028   extern int optind;
6040029   extern int optopt;
6040030   //
6040031   int status;
6040032   int sfdn;
6040033   int sfdn2;
6040034   struct sockaddr_in sa_local;
6040035   struct sockaddr_in sa_remote;
6040036   socklen_t sa_remote_size = sizeof (struct sockaddr_in);
6040037   ssize_t read_size;
6040038   ssize_t sent_size;
6040039   ssize_t recv_size;
6040040   char *addr = NULL;
6040041   char *port = NULL;
6040042   bool can_rx = 1;
```

```
6040043     bool can_tx = 1;
6040044     //
6040045     // Check for options.
6040046     //
6040047     while ((opt = getopt (argc, argv, ":ul")) != -1)
6040048       {
6040049         switch (opt)
6040050           {
6040051           case 'l':
6040052             option_l = 1;
6040053             break;
6040054           case 'u':
6040055             option_u = 1;
6040056             break;
6040057           case '?':
6040058             fprintf (stderr, "Unknown option -%c.\n", optopt);
6040059             usage ();
6040060             return (1);
6040061             break;
6040062           case ':':
6040063             fprintf (stderr,
6040064                      "Missing argument for option -%c\n",
6040065                      optopt);
6040066             usage ();
6040067             return (1);
6040068             break;
6040069           default:
6040070             fprintf (stderr,
6040071                      "Getopt problem: unknown option %c\n",
6040072                      opt);
6040073             usage ();
6040074             return (1);
6040075           }
6040076       }
6040077     //
6040078     // Arguments.
6040079     //
```

```
6040080     if (optind == (argc - 2))
6040081       {
6040082         //
6040083         // There are exactly two arguments: destination
6040084         // address and port.
6040085         //
6040086         addr = argv[argc - 2];
6040087         port = argv[argc - 1];
6040088       }
6040089     else
6040090       {
6040091         //
6040092         // Arguments wrong!
6040093         //
6040094         usage ();
6040095         return (2);
6040096       }
6040097     //
6040098     // Set the local or the remote address.
6040099     //
6040100     if (option_l)
6040101       {
6040102         //
6040103         // Address and port are local.
6040104         //
6040105         sa_local.sin_family = AF_INET;
6040106         sa_local.sin_port = htons (atoi (port));
6040107         inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
6040108       }
6040109     else
6040110       {
6040111         //
6040112         // Address and port are remote.
6040113         //
6040114         sa_remote.sin_family = AF_INET;
6040115         sa_remote.sin_port = htons (atoi (port));
6040116         inet_pton (AF_INET, addr, &sa_remote.sin_addr.s_addr);
```

```
6040117       }
6040118     //
6040119     // Open the socket.
6040120     //
6040121     if (option_u)
6040122       {
6040123         sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
6040124       }
6040125     else
6040126       {
6040127         sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
6040128       }
6040129     if (sfdn < 0)
6040130       {
6040131         perror (NULL);
6040132         return (3);
6040133       }
6040134     //
6040135     // Set it listening or connect.
6040136     //
6040137     if (option_l)
6040138       {
6040139         //
6040140         // Bind the local 'sa' location.
6040141         //
6040142         status =
6040143           bind (sfdn, (struct sockaddr *) &sa_local,
6040144                 sizeof (sa_local));
6040145         if (status < 0)
6040146           {
6040147             perror (NULL);
6040148             close (sfdn);
6040149             return (4);
6040150           }
6040151         //
6040152         // Listen (TCP) or wait the first packet (UDP).
6040153         //
```

```
6040154         if (option_u)
6040155           {
6040156             //
6040157             // Instead of listening, we use the function
6040158             // 'recvfrom()',
6040159             // to get the remote address and port.
6040160             //
6040161             recv_size =
6040162               recvfrom (sfdn, &buffer,
6040163                         (size_t) BUFSIZ - 1, 0,
6040164                         (struct sockaddr *) &sa_remote,
6040165                         &sa_remote_size);
6040166             if (recv_size < 0)
6040167               {
6040168                 perror (NULL);
6040169                 close (sfdn);
6040170                 return (4);
6040171               }
6040172             //
6040173             // Now connect the remote destination
6040174             //
6040175             status =
6040176               connect (sfdn,
6040177                        (struct sockaddr *) &sa_remote,
6040178                        sizeof (sa_remote));
6040179             if (status < 0)
6040180               {
6040181                 perror (NULL);
6040182                 close (sfdn);
6040183                 return (7);
6040184               }
6040185             //
6040186             // And show what was received as a first
6040187             // packet.
6040188             //
6040189             buffer[recv_size] = 0;
6040190             printf ("%s", buffer);
```

```
6040191           }
6040192         else
6040193           {
6040194             //
6040195             // TCP: listen.
6040196             //
6040197             status = listen (sfdn, 1);
6040198             if (status < 0)
6040199               {
6040200                 perror (NULL);
6040201                 close (sfdn);
6040202                 return (5);
6040203               }
6040204             //
6040205             // Accept.
6040206             //
6040207             sfdn2 =
6040208               accept (sfdn,
6040209                       (struct sockaddr *) &sa_remote,
6040210                       &sa_remote_size);
6040211             if (sfdn2 < 0)
6040212               {
6040213                 perror (NULL);
6040214                 close (sfdn);
6040215                 return (6);
6040216               }
6040217             //
6040218             // Close listening socket.
6040219             //
6040220             close (sfdn);
6040221             //
6040222             // Variable 'sfdn' will be the new socket.
6040223             //
6040224             sfdn = sfdn2;
6040225           }
6040226       }
6040227     else
```

```
6040228       {
6040229         //
6040230         // Connect the remote destination.
6040231         //
6040232         status =
6040233           connect (sfdn, (struct sockaddr *) &sa_remote,
6040234                    sizeof (sa_remote));
6040235         if (status < 0)
6040236           {
6040237             perror (NULL);
6040238             close (sfdn);
6040239             return (7);
6040240           }
6040241       }
6040242     //
6040243     // Define the standard input non blocking.
6040244     //
6040245     status = fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
6040246     if (status < 0)
6040247       {
6040248         perror (NULL);
6040249         return (8);
6040250       }
6040251     //
6040252     // Will read from the remote and show to the screen.
6040253     //
6040254     while (can_rx || can_tx)
6040255       {
6040256         if (can_rx)
6040257           {
6040258             recv_size =
6040259               recv (sfdn, &buffer, (size_t) BUFSIZ - 1, 0);
6040260 //          recv_size = read (sfdn, &buffer, (size_t) BUFSIZ-1);
6040261             if (recv_size < 0)
6040262               {
6040263                 if (errno == EAGAIN || errno == EWOULDBLOCK)
6040264                   {
```

```
6040265                     ;
6040266                   }
6040267                 else
6040268                   {
6040269                     perror (NULL);
6040270                     close (sfdn);
6040271                     return (10);
6040272                   }
6040273               }
6040274             else if (recv_size == 0)
6040275               {
6040276                 //
6040277                 // End of stream.
6040278                 //
6040279                 can_rx = 0;
6040280                 printf ("--end of receive stream--\n");
6040281               }
6040282             else
6040283               {
6040284                 buffer[recv_size] = 0;
6040285                 printf ("%s", buffer);
6040286               }
6040287           }
6040288         if (can_tx)
6040289           {
6040290             read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6040291             if (read_size < 0)
6040292               {
6040293                 if (errno == EAGAIN || errno == EWOULDBLOCK)
6040294                   {
6040295                     ;
6040296                   }
6040297                 else
6040298                   {
6040299                     perror (NULL);
6040300                     close (sfdn);
6040301                     return (11);
```

```
6040302                     }
6040303                 }
6040304               else if (read_size == 0)
6040305                 {
6040306                   //
6040307                   // End of input.
6040308                   //
6040309                   printf ("--closing send stream--\n");
6040310                   can_tx = 0;
6040311                 }
6040312               else
6040313                 {
6040314                   //
6040315                   // Send it.
6040316                   //
6040317                   sent_size =
6040318                     send (sfdn, &buffer, (size_t) read_size, 0);
6040319                   if (sent_size < 0)
6040320                     {
6040321                       if (errno == EAGAIN
6040322                           || errno == EWOULDBLOCK)
6040323                         {
6040324                           ;
6040325                         }
6040326                       else
6040327                         {
6040328                           perror (NULL);
6040329                           close (sfdn);
6040330                           return (12);
6040331                         }
6040332                     }
6040333                 }
6040334             }
6040335         }
6040336       //
6040337       // All done.
6040338       //
```

```
6040339   close (sfdn);
6040340   return (0);
6040341 }
6040342
6040343 //-------------------------------------------------------
6040344 static void
6040345 usage (void)
6040346 {
6040347   fprintf (stderr,
6040348            "os32 netcat usage:\n"
6040349            "\n"
6040350            "nc [-u][-l] ADDRESS PORT\n"
6040351            "\n"
6040352            "-u      Use UDP protocol instead of TCP.\n"
6040353            "-l      Listen for incoming connection requests.\n"
6040354            "ADDRESS IPv4 numeric address; if option -l is used, th
6040355            "        is the local address, otherwise it is the remo
6040356            "        address.\n"
6040357            "PORT    TCP or UDP port; if option -l is used, this is
6040358            "        local address, otherwise it is the remote\n"
6040359            "        address.\n");
6040360 }
```

96.1.40 applic/t_ping2.c

«

Si veda la sezione 86.25.

```
6050001 #include <stdio.h>
6050002 #include <sys/types.h>
6050003 //#include <arpa/inet.h>
6050004 #include <sys/socket.h>
6050005 #include <unistd.h>
6050006 #include <errno.h>
6050007 //-------------------------------------------------------
6050008 int
6050009 main (void)
6050010 {
```

```
6050011     int i;
6050012     int sfdn;
6050013     struct sockaddr_in sa;
6050014     ssize_t spediti;
6050015     ssize_t ricevuti;
6050016     int status;
6050017     uint8_t buffer[100];
6050018     uint8_t packet[] =
6050019       { 0x45, 0x00, 0x00, 0x22, 0x00, 0x00, 0x40, 0x00,
6050020       0x40, 0x01, 0x3c, 0xd9, 0x7f, 0x00, 0x00, 0x01,
6050021       0x7f, 0x00, 0x00, 0x01, 'c', 'i', 'a', 'o', ' ',
6050022       'a', 'm', 'o', 'r', 'e', ' ', 'm', 'i', 'o'
6050023     };
6050024
6050025     sa.sin_family = AF_INET;
6050026     sa.sin_port = 0;
6050027     // sa.sin_addr.s_addr=htonl (0xAC15FEFE); //172.21.254.254
6050028     sa.sin_addr.s_addr = htonl (0xAC150B12);  // 172.21.11.18
6050029
6050030     errno = 0;
6050031     sfdn = socket (AF_INET, SOCK_RAW, IPPROTO_ICMP);
6050032     perror (NULL);
6050033
6050034     errno = 0;
6050035     status =
6050036       connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
6050037     perror (NULL);
6050038
6050039     errno = 0;
6050040     spediti = send (sfdn, packet, 34, 0);
6050041     printf ("scritti %i\n", spediti);
6050042     perror (NULL);
6050043
6050044     ricevuti = 10;
6050045     while (ricevuti > 0)
6050046       {
6050047         errno = 0;
```

```
6050048         ricevuti = recv (sfdn, buffer, (size_t) 30, 0);
6050049         printf ("ricevuti=%i\n", (int) ricevuti);
6050050         perror (NULL);
6050051
6050052         if (ricevuti > 0)
6050053           {
6050054             for (i = 0; i < ricevuti; i++)
6050055               {
6050056                 printf ("%02x", (unsigned int) buffer[i]);
6050057               }
6050058           }
6050059       }
6050060
6050061     close (sfdn);
6050062     return (0);
6050063   }
```

96.1.41 applic/t_pipe.c

Si veda la sezione 86.25.

```
6060001   #include <stdio.h>
6060002   #include <unistd.h>
6060003   #include <stdlib.h>
6060004   #include <sys/wait.h>
6060005   #include <signal.h>
6060006   #include <sys/wait.h>
6060007   #include <stdio.h>
6060008   #include <stdlib.h>
6060009   #include <string.h>
6060010   //-----------------------------------------------------
6060011   int
6060012   main (void)
6060013   {
6060014     int pipefd[2];
6060015     pid_t child;
6060016     char buffer;
```

```
6060017     char *message =
6060018       "ciao a tutti voi amici vicini e lontani\n";
6060019     int i;
6060020     size_t size;
6060021     ssize_t written;
6060022     //
6060023     //
6060024     //
6060025     if (pipe (pipefd) == -1)
6060026       {
6060027         perror ("pipe");
6060028         exit (EXIT_FAILURE);
6060029       }
6060030     //
6060031     //
6060032     //
6060033     child = fork ();
6060034     if (child == -1)
6060035       {
6060036         perror ("fork");
6060037         exit (EXIT_FAILURE);
6060038       }
6060039     //
6060040     //
6060041     //
6060042     if (child == 0)
6060043       {
6060044         //
6060045         // This is the child and it have to read the
6060046         // pipe:
6060047         // close the write end of the pipe.
6060048         //
6060049         close (pipefd[1]);
6060050         //
6060051         // Read one byte at the time, as long as there
6060052         // is
6060053         // something to read.
```

```
6060054         //
6060055         while (read (pipefd[0], &buffer, 1) > 0)
6060056           {
6060057             write (STDOUT_FILENO, &buffer, 1);
6060058           }
6060059         //
6060060         // Close the pipe and exit the child.
6060061         //
6060062         close (pipefd[0]);
6060063         //
6060064         exit (EXIT_SUCCESS);
6060065       }
6060066     else
6060067       {
6060068         //
6060069         // This is the parent process, and the read end
6060070         // of
6060071         // pipe is closed.
6060072         //
6060073         close (pipefd[0]);
6060074         //
6060075         while (1)
6060076           {
6060077             for (i = 0, written = 0, size =
6060078                  strlen (message); i < strlen (message);
6060079                  i += written, size -= written)
6060080               {
6060081                 written =
6060082                   write (pipefd[1], &message[i], size);
6060083                 if (written < 0)
6060084                   {
6060085                     perror ("pipe");
6060086                     close (pipefd[1]);
6060087                     wait (NULL);  // Wait for child.
6060088                     exit (EXIT_FAILURE);
6060089                   }
6060090               }
```

```
6060091           }
6060092         close (pipefd[1]);          /* Reader will see EOF */
6060093         wait (NULL);        /* Wait for child */
6060094         exit (EXIT_SUCCESS);
6060095       }
6060096     //
6060097     return (0);
6060098   }
```

96.1.42 applic/t_read.c

«

Si veda la sezione 86.25.

```
6070001  #include <sys/stat.h>
6070002  #include <sys/types.h>
6070003  #include <unistd.h>
6070004  #include <stdlib.h>
6070005  #include <fcntl.h>
6070006  #include <errno.h>
6070007  #include <signal.h>
6070008  #include <stdio.h>
6070009  #include <string.h>
6070010  #include <limits.h>
6070011  #include <libgen.h>
6070012  #include <arpa/inet.h>
6070013  #include <sys/socket.h>
6070014  #include <stdint.h>
6070015  #include <stdbool.h>
6070016  #include <fcntl.h>
6070017
6070018  char buffer[BUFSIZ];
6070019
6070020  //-----------------------------------------------------
6070021  int
6070022  main (int argc, char *argv[], char *envp[])
6070023  {
6070024    int status;
```

```
6070025     ssize_t read_size;
6070026
6070027
6070028     //
6070029     // Define the standard input non blocking.
6070030     //
6070031     status = fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
6070032     if (status < 0)
6070033       {
6070034         perror (NULL);
6070035         return (2);
6070036       }
6070037
6070038
6070039     read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6070040     if (read_size < 0)
6070041       {
6070042         if (errno == EAGAIN || errno == EWOULDBLOCK)
6070043           {
6070044             printf ("nulla da leggere per ora\n");
6070045           }
6070046         else
6070047           {
6070048             perror (NULL);
6070049             return (0);
6070050           }
6070051       }
6070052     else
6070053       {
6070054         buffer[read_size] = 0;
6070055         printf ("letto: %s\n", buffer);
6070056       }
6070057     printf ("finito\n");
6070058     return (0);
6070059   }
```

96.1.43 applic/t_ret.c

«

Si veda la sezione 86.25.

```
6080001 #include <stdlib.h>
6080002 //-------------------------------------------------------
6080003 int
6080004 main (void)
6080005 {
6080006 //    exit (1);
6080007   return (1);
6080008 }
```

96.1.44 applic/t_rx_udp.c

«

Si veda la sezione 86.25.

```
6090001 #include <sys/stat.h>
6090002 #include <sys/types.h>
6090003 #include <unistd.h>
6090004 #include <stdlib.h>
6090005 #include <fcntl.h>
6090006 #include <errno.h>
6090007 #include <signal.h>
6090008 #include <stdio.h>
6090009 #include <string.h>
6090010 #include <limits.h>
6090011 #include <libgen.h>
6090012 #include <arpa/inet.h>
6090013 #include <sys/socket.h>
6090014 #include <stdint.h>
6090015 #include <stdbool.h>
6090016 //-------------------------------------------------------
6090017 static void usage (void);
6090018 //-------------------------------------------------------
6090019 int
6090020 main (int argc, char *argv[], char *envp[])
6090021 {
```

```
6090022     int status;
6090023     int sfdn;
6090024     struct sockaddr_in sa_local;
6090025     ssize_t recv_size;
6090026     char buffer[BUFSIZ];
6090027     char *addr = NULL;
6090028     char *port = NULL;
6090029     //
6090030     // Arguments.
6090031     //
6090032     if (argc == 3)
6090033       {
6090034         //
6090035         // There are exactly two arguments: destination
6090036         // address and port.
6090037         //
6090038         addr = argv[1];
6090039         port = argv[2];
6090040       }
6090041     else
6090042       {
6090043         //
6090044         // Arguments wrong!
6090045         //
6090046         usage ();
6090047         return (4);
6090048       }
6090049     //
6090050     // Define the destination `sa_local'
6090051     //
6090052     sa_local.sin_family = AF_INET;
6090053     sa_local.sin_port = htons (atoi (port));
6090054     inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
6090055     //
6090056     // Open the socket.
6090057     //
6090058     sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

```
6090059     if (sfdn < 0)
6090060       {
6090061         perror (NULL);
6090062         return (5);
6090063       }
6090064     //
6090065     // Bind the local 'sa' location.
6090066     //
6090067     status = bind (sfdn, (struct sockaddr *) &sa_local,
6090068                    sizeof (sa_local));
6090069     if (status < 0)
6090070       {
6090071         perror (NULL);
6090072         close (sfdn);
6090073         return (7);
6090074       }
6090075     //
6090076     // Will read from the remote and show to the screen.
6090077     //
6090078     while (1)
6090079       {
6090080         recv_size = read (sfdn, &buffer, (size_t) BUFSIZ - 1);
6090081         if (recv_size < 0)
6090082           {
6090083             perror (NULL);
6090084             close (sfdn);
6090085             return (10);
6090086           }
6090087         buffer[recv_size] = 0;
6090088         printf ("%s", buffer);
6090089       }
6090090     //
6090091     // All done.
6090092     //
6090093     return (0);
6090094   }
6090095
```

```
6090096 //-------------------------------------------------------
6090097 static void
6090098 usage (void)
6090099 {
6090100   fprintf (stderr, "Usage: rx_udp LOCAL_ADDR LOCAL_PORT\n");
6090101 }
```

96.1.45 applic/t_scr.c

«

Si veda la sezione 86.25.

```
6100001 #include <unistd.h>
6100002 #include <stdio.h>
6100003 #include <fcntl.h>
6100004 #include <unistd.h>
6100005 #include <stdlib.h>
6100006 //-------------------------------------------------------
6100007 int
6100008 main (int argc, char *argv[], char *envp[])
6100009 {
6100010   FILE *screen;
6100011   int status;
6100012
6100013   screen = fopen ("/dev/tty", "w");
6100014   if (screen == NULL)
6100015     {
6100016       printf ("[%s] Cannot open \"/dev/tty\" ", argv[0]);
6100017       perror (NULL);
6100018       exit (0);
6100019     }
6100020
6100021   status = fseek (screen, (long) 1000, SEEK_SET);
6100022
6100023   fprintf (screen, "ciao status: %i ciao", status);
6100024   perror (NULL);
6100025
6100026   fclose (screen);
```

```
6100027     return (0);
6100028 }
```

96.1.46 applic/t_setjmp.c

«

Si veda la sezione 86.25.

```
6110001 #include <stdio.h>
6110002 #include <setjmp.h>
6110003 //-------------------------------------------------------
6110004
6110005 jmp_buf env;
6110006
6110007 void
6110008 prova3 (void)
6110009 {
6110010   printf ("funzione prova3\n");
6110011   longjmp (env, 1);
6110012   printf ("funzione prova3 post\n");
6110013 }
6110014
6110015 void
6110016 prova2 (void)
6110017 {
6110018   printf ("funzione prova2\n");
6110019   prova3 ();
6110020   printf ("funzione prova2 post\n");
6110021 }
6110022
6110023 void
6110024 prova1 (void)
6110025 {
6110026   printf ("funzione prova1\n");
6110027   prova2 ();
6110028   printf ("funzione prova1 post\n");
6110029 }
6110030
```

```
6110031 int
6110032 main (int argc, char *argv[], char *envp[])
6110033 {
6110034   int val;
6110035   //
6110036   printf ("prima\n");
6110037   //
6110038   val = setjmp (env);
6110039   //
6110040   printf ("dopo setjmp val=%i\n", val);
6110041   //
6110042   if (val != 0)
6110043     return (0);
6110044
6110045   prova1 ();
6110046
6110047   return (0);
6110048 }
```

96.1.47 applic/t_sig.c

«

Si veda la sezione 86.25.

```
6120001 #include <stdio.h>
6120002 #include <unistd.h>
6120003 #include <stdlib.h>
6120004 #include <sys/wait.h>
6120005 #include <signal.h>
6120006 //-------------------------------------------------------
6120007 void
6120008 signal_handler (int signal)
6120009 {
6120010   printf ("Hello! I have caught the signal %i.\n", signal);
6120011 }
6120012
6120013 //-------------------------------------------------------
6120014 int
```

```
6120015 main (void)
6120016 {
6120017   signal (SIGHUP, signal_handler);
6120018   signal (SIGINT, signal_handler);
6120019   signal (SIGQUIT, signal_handler);
6120020   signal (SIGILL, signal_handler);
6120021   signal (SIGABRT, signal_handler);
6120022   signal (SIGFPE, signal_handler);
6120023   signal (SIGKILL, signal_handler);
6120024   signal (SIGSEGV, signal_handler);
6120025   signal (SIGPIPE, signal_handler);
6120026   signal (SIGALRM, signal_handler);
6120027   signal (SIGTERM, signal_handler);
6120028   signal (SIGSTOP, signal_handler);
6120029   signal (SIGTSTP, signal_handler);
6120030   signal (SIGCONT, signal_handler);
6120031   signal (SIGCHLD, signal_handler);
6120032   signal (SIGTTIN, signal_handler);
6120033   signal (SIGTTOU, signal_handler);
6120034   signal (SIGUSR1, signal_handler);
6120035   signal (SIGUSR2, signal_handler);
6120036   //
6120037   while (1)
6120038     {
6120039       sleep (1);
6120040       printf ("ciao!\n");
6120041     }
6120042   //
6120043   return (0);
6120044 }
```

96.1.48 applic/t_sig2.c

«

Si veda la sezione 86.25.

```
6130001 #include <stdio.h>
6130002 #include <unistd.h>
```

```
6130003 #include <stdlib.h>
6130004 #include <sys/wait.h>
6130005 #include <signal.h>
6130006 //-------------------------------------------------------
6130007 void
6130008 signal_handler (int signal)
6130009 {
6130010   printf ("Hello! I have caught the signal %i.\n", signal);
6130011 }
6130012
6130013 //-------------------------------------------------------
6130014 int
6130015 main (void)
6130016 {
6130017   //
6130018   while (1)
6130019     {
6130020       signal (SIGHUP, signal_handler);
6130021       signal (SIGINT, signal_handler);
6130022       signal (SIGQUIT, signal_handler);
6130023       signal (SIGILL, signal_handler);
6130024       signal (SIGABRT, signal_handler);
6130025       signal (SIGFPE, signal_handler);
6130026       signal (SIGKILL, signal_handler);
6130027       signal (SIGSEGV, signal_handler);
6130028       signal (SIGPIPE, signal_handler);
6130029       signal (SIGALRM, signal_handler);
6130030       signal (SIGTERM, signal_handler);
6130031       signal (SIGSTOP, signal_handler);
6130032       signal (SIGTSTP, signal_handler);
6130033       signal (SIGCONT, signal_handler);
6130034       signal (SIGCHLD, signal_handler);
6130035       signal (SIGTTIN, signal_handler);
6130036       signal (SIGTTOU, signal_handler);
6130037       signal (SIGUSR1, signal_handler);
6130038       signal (SIGUSR2, signal_handler);
6130039       printf ("ciao!\n");
```

```
6130040         sleep (1);
6130041       }
6130042     //
6130043     return (0);
6130044   }
```

96.1.49 applic/t_tx_tcp.c

«

Si veda la sezione 86.25.

```
6140001   #include <sys/stat.h>
6140002   #include <sys/types.h>
6140003   #include <unistd.h>
6140004   #include <stdlib.h>
6140005   #include <fcntl.h>
6140006   #include <errno.h>
6140007   #include <signal.h>
6140008   #include <stdio.h>
6140009   #include <string.h>
6140010   #include <limits.h>
6140011   #include <libgen.h>
6140012   #include <arpa/inet.h>
6140013   #include <sys/socket.h>
6140014   #include <stdint.h>
6140015   #include <stdbool.h>
6140016   //-------------------------------------------------------
6140017   static void usage (void);
6140018   //-------------------------------------------------------
6140019   int
6140020   main (int argc, char *argv[], char *envp[])
6140021   {
6140022     int status;
6140023     int sfdn;
6140024     struct sockaddr_in sa;
6140025     ssize_t read_size;
6140026     ssize_t sent_size;
6140027     char buffer[BUFSIZ];
```

```
6140028     char *addr = NULL;
6140029     char *port = NULL;
6140030     //
6140031     // Arguments.
6140032     //
6140033     if (argc == 3)
6140034       {
6140035         //
6140036         // There are exactly two arguments: destination
6140037         // address and port.
6140038         //
6140039         addr = argv[1];
6140040         port = argv[2];
6140041       }
6140042     else
6140043       {
6140044         //
6140045         // Arguments wrong!
6140046         //
6140047         usage ();
6140048         return (4);
6140049       }
6140050     //
6140051     // Define the destination `sa'
6140052     //
6140053     sa.sin_family = AF_INET;
6140054     sa.sin_port = htons (atoi (port));
6140055     inet_pton (AF_INET, addr, &sa.sin_addr.s_addr);
6140056     //
6140057     //
6140058     // Open the socket.
6140059     //
6140060     sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
6140061     if (sfdn < 0)
6140062       {
6140063         perror (NULL);
6140064         return (5);
```

```
6140065       }
6140066     //
6140067     // Connect the 'sa' destination
6140068     //
6140069     status =
6140070       connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
6140071     if (status < 0)
6140072       {
6140073         perror (NULL);
6140074         close (sfdn);
6140075         return (7);
6140076       }
6140077     //
6140078     // Will read from the standard input and send to the
6140079     // other
6140080     // side.
6140081     //
6140082     while (1)
6140083       {
6140084         read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6140085         if (read_size < 0)
6140086           {
6140087             perror (NULL);
6140088             close (sfdn);
6140089             return (8);
6140090           }
6140091         if (read_size == 0)
6140092           {
6140093             close (sfdn);
6140094             return (0);
6140095           }
6140096         //
6140097         // Verify the 'stop' command.
6140098         //
6140099         if (strncmp (buffer, "stop\n", read_size) == 0)
6140100           {
6140101             printf ("closing send...\n");
```

```
6140102             close (sfdn);
6140103             return (0);
6140104           }
6140105         //
6140106         sent_size =
6140107           send (sfdn, &buffer, (size_t) read_size, 0);
6140108         if (sent_size < 0)
6140109           {
6140110             perror (NULL);
6140111             close (sfdn);
6140112             return (9);
6140113           }
6140114         printf ("sent %i bytes\n", (int) sent_size);
6140115       }
6140116     //
6140117     // All done.
6140118     //
6140119     return (0);
6140120   }
6140121
6140122   //---------------------------------------------------------
6140123   static void
6140124   usage (void)
6140125   {
6140126     fprintf (stderr, "Usage: tx_tcp DEST_ADDR DEST_PORT\n");
6140127   }
```

96.1.50 applic/t_tx_udp.c

«

Si veda la sezione 86.25.

```
6150001   #include <sys/stat.h>
6150002   #include <sys/types.h>
6150003   #include <unistd.h>
6150004   #include <stdlib.h>
6150005   #include <fcntl.h>
6150006   #include <errno.h>
```

```
6150007 #include <signal.h>
6150008 #include <stdio.h>
6150009 #include <string.h>
6150010 #include <limits.h>
6150011 #include <libgen.h>
6150012 #include <arpa/inet.h>
6150013 #include <sys/socket.h>
6150014 #include <stdint.h>
6150015 #include <stdbool.h>
6150016 //-------------------------------------------------------
6150017 static void usage (void);
6150018 //-------------------------------------------------------
6150019 int
6150020 main (int argc, char *argv[], char *envp[])
6150021 {
6150022   int status;
6150023   int sfdn;
6150024   struct sockaddr_in sa;
6150025   ssize_t read_size;
6150026   ssize_t sent_size;
6150027   char buffer[BUFSIZ];
6150028   char *addr = NULL;
6150029   char *port = NULL;
6150030   //
6150031   // Arguments.
6150032   //
6150033   if (argc == 3)
6150034     {
6150035       //
6150036       // There are exactly two arguments: destination
6150037       // address and port.
6150038       //
6150039       addr = argv[1];
6150040       port = argv[2];
6150041     }
6150042   else
6150043     {
```

```
6150044         //
6150045         // Arguments wrong!
6150046         //
6150047         usage ();
6150048         return (4);
6150049       }
6150050     //
6150051     // Define the destination 'sa'
6150052     //
6150053     sa.sin_family = AF_INET;
6150054     sa.sin_port = htons (atoi (port));
6150055     inet_pton (AF_INET, addr, &sa.sin_addr.s_addr);
6150056     //
6150057     //
6150058     // Open the socket.
6150059     //
6150060     sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
6150061     if (sfdn < 0)
6150062       {
6150063         perror (NULL);
6150064         return (5);
6150065       }
6150066     //
6150067     // Connect the 'sa' destination
6150068     //
6150069     status =
6150070       connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
6150071     if (status < 0)
6150072       {
6150073         perror (NULL);
6150074         close (sfdn);
6150075         return (7);
6150076       }
6150077     //
6150078     // Will read from the standard input and send to the
6150079     // other
6150080     // side.
```

```
6150081     //
6150082     while (1)
6150083       {
6150084         read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6150085         if (read_size < 0)
6150086           {
6150087             perror (NULL);
6150088             close (sfdn);
6150089             return (8);
6150090           }
6150091         if (read_size == 0)
6150092           {
6150093             close (sfdn);
6150094             return (0);
6150095           }
6150096         //
6150097         sent_size =
6150098           send (sfdn, &buffer, (size_t) read_size, 0);
6150099         if (sent_size < 0)
6150100           {
6150101             perror (NULL);
6150102             close (sfdn);
6150103             return (9);
6150104           }
6150105         printf ("sent %i bytes\n", (int) sent_size);
6150106       }
6150107     //
6150108     // All done.
6150109     //
6150110     return (0);
6150111   }
6150112
6150113   //-------------------------------------------------------
6150114   static void
6150115   usage (void)
6150116   {
6150117     fprintf (stderr, "Usage: tx_udp DEST_ADDR DEST_PORT\n");
```

```
6150118 }
```

96.1.51 applic/touch.c

Si veda la sezione 86.26.

```
6160001 #include <fcntl.h>
6160002 #include <sys/stat.h>
6160003 #include <utime.h>
6160004 #include <stddef.h>
6160005 #include <unistd.h>
6160006 #include <errno.h>
6160007 //------------------------------------------------------------
6160008 static void usage (void);
6160009 //------------------------------------------------------------
6160010 int
6160011 main (int argc, char *argv[], char *envp[])
6160012 {
6160013   int a;         // Argument index.
6160014   int status;
6160015   struct stat file_status;
6160016   //
6160017   // No options are known, but at least an argument
6160018   // must be given.
6160019   //
6160020   if (argc < 2)
6160021     {
6160022       usage ();
6160023       return (1);
6160024     }
6160025   //
6160026   // Scan arguments.
6160027   //
6160028   for (a = 1; a < argc; a++)
6160029     {
6160030       //
6160031       // Verify if the file exists, through the return
```

```
6160032         // value of
6160033         // 'stat()'. No other checks are made.
6160034         //
6160035         if (stat (argv[a], &file_status) == 0)
6160036           {
6160037             //
6160038             // File exists: should be updated the times.
6160039             //
6160040             status = utime (argv[a], NULL);
6160041             if (status != 0)
6160042               {
6160043                 perror (NULL);
6160044                 return (2);
6160045               }
6160046           }
6160047         else
6160048           {
6160049             //
6160050             // File does not exist: should be created.
6160051             //
6160052             status =
6160053               open (argv[a],
6160054                     O_WRONLY | O_CREAT | O_TRUNC, 0666);
6160055             //
6160056             if (status >= 0)
6160057               {
6160058                 //
6160059                 // Here, the variable 'status' is the
6160060                 // file
6160061                 // descriptor to be closed.
6160062                 //
6160063                 status = close (status);
6160064                 if (status != 0)
6160065                   {
6160066                     perror (NULL);
6160067                     return (3);
6160068                   }
```

```
6160069                 }
6160070               else
6160071                 {
6160072                   perror (NULL);
6160073                   return (4);
6160074                 }
6160075             }
6160076         }
6160077     return (0);
6160078   }
6160079
6160080   //-------------------------------------------------------
6160081   static void
6160082   usage (void)
6160083   {
6160084     fprintf (stderr, "Usage: touch FILE...\n");
6160085   }
```

96.1.52 applic/tty.c

«

Si veda la sezione 86.27.

```
6170001   #include <fcntl.h>
6170002   #include <sys/stat.h>
6170003   #include <utime.h>
6170004   #include <stddef.h>
6170005   #include <unistd.h>
6170006   #include <errno.h>
6170007   #include <sys/os32.h>
6170008   #include <sys/types.h>
6170009   //-------------------------------------------------------
6170010   static void usage (void);
6170011   //-------------------------------------------------------
6170012   int
6170013   main (int argc, char *argv[], char *envp[])
6170014   {
6170015     int dev_minor;
```

```
6170016     struct stat file_status;
6170017     //
6170018     // No options and no arguments.
6170019     //
6170020     if (argc > 1)
6170021       {
6170022         usage ();
6170023         return (1);
6170024       }
6170025     //
6170026     // Verify the standard input.
6170027     //
6170028     if (fstat (STDIN_FILENO, &file_status) == 0)
6170029       {
6170030         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
6170031           {
6170032             dev_minor = minor (file_status.st_rdev);
6170033             //
6170034             // If minor is equal to 0xFF, it is
6170035             // '/dev/console'
6170036             // that is not a controlling terminal, but
6170037             // just
6170038             // a reference for the current virtual
6170039             // console.
6170040             //
6170041             if (dev_minor < 0xFF)
6170042               {
6170043                 printf ("/dev/console%i\n", dev_minor);
6170044               }
6170045           }
6170046       }
6170047     else
6170048       {
6170049         perror ("Cannot get standard input file status");
6170050         return (2);
6170051       }
6170052     //
```

```
6170053     return (0);
6170054   }
6170055
6170056   //-------------------------------------------------------
6170057   static void
6170058   usage (void)
6170059   {
6170060     fprintf (stderr, "Usage: tty\n");
6170061   }
```

96.1.53 applic/umount.c

«

Si veda la sezione 92.7.

```
6180001   #include <unistd.h>
6180002   #include <stdlib.h>
6180003   #include <sys/stat.h>
6180004   #include <sys/types.h>
6180005   #include <fcntl.h>
6180006   #include <errno.h>
6180007   #include <signal.h>
6180008   #include <stdio.h>
6180009   #include <sys/wait.h>
6180010   #include <stdio.h>
6180011   #include <string.h>
6180012   #include <limits.h>
6180013   #include <sys/os32.h>
6180014   //-------------------------------------------------------
6180015   static void usage (void);
6180016   //-------------------------------------------------------
6180017   int
6180018   main (int argc, char *argv[], char *envp[])
6180019   {
6180020     int status;
6180021     //
6180022     // One argument is mandatory.
6180023     //
```

```
6180024     if (argc != 2)
6180025       {
6180026         usage ();
6180027         return (1);
6180028       }
6180029     //
6180030     // System call.
6180031     //
6180032     status = umount (argv[1]);
6180033     if (status != 0)
6180034       {
6180035         perror (argv[1]);
6180036         return (2);
6180037       }
6180038     //
6180039     return (0);
6180040   }
6180041
6180042   //-------------------------------------------------------
6180043   static void
6180044   usage (void)
6180045   {
6180046     fprintf (stderr, "Usage: umount MOUNT_POINT\n");
6180047   }
```

96.1.54 applic/yes.c

«

Si veda la sezione 86.28.

```
6190001   #include <stdio.h>
6190002   //-------------------------------------------------------
6190003   int
6190004   main (int argc, char *argv[], char *envp[])
6190005   {
6190006     int i;
6190007     //
6190008     if (argc > 1)
```

```
6190009       {
6190010         while (1)
6190011           {
6190012             printf ("%s", argv[1]);
6190013             for (i = 2; i < argc; i++)
6190014               {
6190015                 printf (" %s", argv[i]);
6190016               }
6190017             printf ("\n");
6190018           }
6190019       }
6190020     else
6190021       {
6190022         while (1)
6190023           {
6190024             printf ("y\n");
6190025           }
6190026       }
6190027     return (0);
6190028   }
```