SmallShell

Piccolo processore comandi

```
/* file smallsh.h */
#include <stdio.h>
#include <fcntl.h>

#define  EOL 1
#define  ARG 2
#define  AMPERSAND 3
#define  SEMICOLON 4
#define  RIDIRIN 5
#define  RIDIROUT 6
#define  PIPE 7

#define  MAXARG  112
#define  MAXBUF  112

#define  FOREGROUND 0
#define  BACKGROUND 1

#define  MASK 0777
#define  NOARG 0
```

```
/* file smallsh.c */
#include "smallsh.h"

char *prompt = "Comando>";
char *ver="Small Shell version 26 november 2001";
char *shellInput = "/dev/tty";
char *shellOutput = "/dev/tty";

static char *inpfile, *outfile; /* file di in e out */
char inpbuf [MAXBUF], tokbuf [2*MAXBUF]; /* buffer di
                                            input */
int i, j;

void setfile ()
{ strcpy(inpfile, shellInput);
  strcpy(outfile, shellOutput);
}

extern int userin();
extern int inarg ();
extern void processalinea();
extern int runcomando ();

main (argc, argv)
    int argc; char *argv[];
{
   int m, n;
   for (m=1; m < argc; m++)
   {
     if (argv[m][0] == '-') /* e' una opzione */
     {
       switch (argv[m][1])
       {
         case 'v': /* versione */
              printf("%s\n\n", ver); break;
         case 'h': /* help */
              printf("%s\n", ver);
                        printf(" -v print version
information\n"); break;
```

```
        }
      }
    }
    inpfile = (char *)malloc(255);
    outfile = (char *)malloc(255);
    setfile();
    while (userin(prompt) != EOF)  processalinea ();
}


int userin (p)
    char *p; /* prompt */
{  int c, count;

    printf ("%s ", p);
    i = j = count = 0;

    c = getchar ();
    while  ((c != '\n') && (count < MAXBUF)
             && (c != EOF))
    {   inpbuf [count++] = c;
        c = getchar ();
    }
    if (count >= MAXBUF)
        {   fprintf (stderr, "line too long\n");
            count = 0;
                        inpbuf [count++] ='\n';
                        inpbuf [count] ='\0';
            return (count);
        };
      if (c == EOF) return (EOF);
      if ((c == '\n') && (count < MAXBUF))
        {   inpbuf [count++] = '\n';
            inpbuf [count] = '\0';
                    return (count);
    /*  si inseriscono sia il fine linea, sia il
        carattere di fine stringa  */
        }
}
```

```
int gettoken (outptr)
    char **outptr;
/* preleva i caratteri dal buffer di ingresso e li
trasferisce
   nell'area di outptr  */
{  int type;

    *outptr = &tokbuf[j];

    /* togli blanks dal buffer di input */
    while (inpbuf[i] == ' ' || inpbuf[i] == '\t')
        i++;

    tokbuf[j++] = inpbuf[i];
    switch (inpbuf[i++])
    {   case '\n': type = EOL; break;
        case '&':  type = AMPERSAND; break;
        case ';':  type = SEMICOLON; break;
        default:   type = ARG;
                   while (inarg(inpbuf[i]))
                       tokbuf[j++] = inpbuf[i++];
       /* inarg ricerca un carattere speciale:
          ritorna vero per caratteri normali,
          falso se si trova un carattere speciale  */
    }

    tokbuf[j++] = '\0'; /* fine stringa */
    return (type);
}

int gettoken2 (clineEL)
    char *clineEL;
/* dice che tipo di token e' l'elemento
   passato come parametro  */
{  int type;

    if (clineEL == NULL)
      return EOL;
```

```
    switch (clineEL[0])
    {
        case '<':   type = RIDIRIN; break;
        case '>':   type = RIDIROUT; break;
        case '|':   type = PIPE; break;
        case '\0':
        case '\n':  type = EOL; break;
        default:   type = ARG;
    }

    return (type);
}

static char special [] = {' ', '\t', '&', ';', '<', '>',
'\n', '|', '\0'};

int inarg (c)
char c;
{  char *wrk;

    for (wrk = special; *wrk != '\0'; wrk++)
        if (c == *wrk) return (0);
    return (1);
}


void processalinea ()
{
    char *arg[MAXARG + 1];  /* array di stringhe
                               che contengono
                               le parti del comando */
    int  narg = 0;          /* numero di argomenti */
    int  toktype = ARG;     /* tipo del token */
    int  type = FOREGROUND; /* BACKGROUND,
                               FOREGROUND o PIPE */

while (toktype != EOL)
 {
```

```
   /* a seconda del tipo di token */
   switch (toktype = gettoken (&arg[narg]))
   {
    case ARG:       if (narg < MAXARG)  narg++; break;

    case AMPERSAND: type = BACKGROUND; /* background */
    case EOL: /* stessa azione per '&', EOL, ';' */
    case SEMICOLON: if (narg != 0) {
                       arg [narg] = NULL;
                       runcomando (arg, type, -1, -1);
                       }
                    narg = 0;
                    break;

     } /* fine switch */
 } /* fine while */
}
```

```
int runcomando ( cline, where, pipeOut,
pipeToClose)
    char **cline;  /* linea di comando da eseguire */
    int  where;/* in background, foreground o pipe */
    int  pipeOut;  /* pipe di output, se > 0 */
    int  pipeToClose; /* pipe da chiudere, se > 0 */

{  int pid, exitst, returnst;
   int toktype = ARG; /* tipo di token */
   int narg;          /* numero di argomenti */
   int posPipe = -1;  /* posizione dell'ultima pipe */
   int pipeIn = -1;   /* pipe in lettura */
   int append = 0;    /* indica se c'e' append */

  /* trattamento differenziato del change directory  */
   if (strcmp (cline [0] , "cd")== 0)
   { returnst = chdir (cline [1]);
     if (returnst < 0)
       perror(cline[1]);
     return (returnst); }
```

```
/* trattamento dell'uscita dal processore comandi */
 if (strcmp (cline [0], "lo") ==0) exit ();
/* trattamento differenziato della versione  */
 if (strcmp (cline [0] , "ver")== 0)
 { printf("%s\n", ver);
   return (0); }

 /* creazione del figlio che eseguira' il comando */
 if ((pid = fork () ) < 0)
  /* errore nella generazione del figlio */
 {  perror ("smallsh: cannot fork");   return (-1);  }

 if (pid == 0)  /* figlio */
 {
    narg = 0;
    while (toktype != EOL)
    {
      /* a seconda del tipo di token */
      /* usa le gettoken2! */
      toktype = gettoken2 (cline[narg]);

      switch (toktype)
      {  int outfd; int pipearray [2];
         int i;
         int k;
         char **precCline; /* stadio precedente
                              della pipe */

        case ARG: if (narg < MAXARG)  narg ++; break;

       /* le seguenti stringhe rappresentano token */
       /* per cui al loro posto viene messo un NULL */
       /* per terminare la linea di comando */

        case RIDIRIN:  /* elimina il '<' */
                       cline[narg] = NULL;
                       /* file di input */
                       inpfile =  cline[++narg];
                       break;
```

```
            case RIDIROUT: /* elimina il '>' */
                           cline[narg] = NULL;
                     if (strcmp(cline[narg+1],">") == 0)
                     { /* caso di append */
                       /* elimina il secondo '>' */
                       cline[++narg] = NULL;
                       append = 1;
                     }
                     /* nome del file di output */
                     outfile =  cline[++narg];
                     break;

               /* NOTA: in caso di piu' ridirezioni, viene
                  considerata solo l'ultima; */
               /* in caso di ridirezione in out, solo
                  l'ultimo file viene creato */

            /* la pipe non viene eliminata subito
               perche' in caso di piu' pipe,
               verranno elaborate dai processi figli */

            case PIPE:   posPipe = narg;
           /* memorizza l'ultima posizione della pipe */
                        narg ++;
                        break;

            case EOL:    if (posPipe == 0)
                         {
                      fprintf(stderr, "smallshell: syntax
error: unexpected token '|'\n");
                      exit(-1);
                           }
                    /* se c'era una pipe */
                    if (posPipe > 0)
                    {
                      /* crea la pipe */
                      if (pipe ( pipearray ) < 0)
                      { perror("smallsh: cannot create
pipe");
                        exit(1);
```

```
                    }
                    /* elimina il '|' */
                    cline[posPipe] = NULL;
                    /* copia lo stadio precedente
                       della pipe */
                    k = 0;
                    precCline = (char **)malloc(posPipe *
sizeof(char *));
                    while (cline[k])
                    {
                      precCline[k] =
                      (char *)malloc(strlen(cline[k])+1);
                      strcpy(precCline[k], cline[k]);
                      k++;
                    }
                    precCline[k] = NULL;
                    /* delega ad un figlio l'esecuzione
                       dello stadio precedente
                       della pipe */
                    /* dandogli il lato scrittura
                       della pipe come stdout */
                    runcomando (precCline, PIPE,
pipearray[1], pipearray[0]);

                    /* prepara l'esecuzione
                       dell'ultimo stadio */
                    cline = &cline[posPipe+1];
                    pipeIn = pipearray [0];
                    pipeToClose = pipearray[1];
                  }
                  break;
        } /* fine switch */
      } /* fine while */

      if (strcmp(inpfile, shellInput) != 0)
      /* c'e' ridirezione in input */
      {
        close (0);
        if (open (inpfile, O_RDONLY) < 0)
        { perror("smallshell: cannot open input file");
```

```
          exit (-1);
        }
      }
      if (strcmp(outfile, shellOutput) != 0)
      /* c'e' ridirezione in output */
      {
        close (1);
        if (append == 1) /* ridirezione in append */
        {
          if (open (outfile, O_WRONLY) < 0)
          { /* se il file non esiste, lo creo */
            if (creat (outfile, MASK) < 0)
            {  perror("smallshell: cannot create output
file");
               exit (-1);
            }
          }
          else
            lseek (1, 0, 2);
        } /* fine if (append == 1) */
        else
        if (creat(outfile, MASK) < 0)
        {  perror("smallshell: cannot create output
file");
           exit (-1);
        }
      }
      /* chiude il lato scrittore della pipe */
      if (pipeToClose > 0)
        close(pipeToClose);
      if (pipeOut > 0)
      {
    close (1);
    if (dup (pipeOut) < 0)
    {
       perror("smallshell: dup OUT error");
       exit(-1);
    }
    close (pipeOut);
      }
```

```
      if (pipeIn > 0)
      {
    close (0); /* chiude lo stdin */
    /* e duplica il fd della pipe in 0) */
    if (dup (pipeIn) < 0)
    {
       perror("smallshell: dup IN error");
       exit(-1);
    }
    close (pipeIn);
  }

  execvp (*cline, cline);
  /* se la exec va a buon fine, non ritorna */
  perror (*cline);   exit (127);
 } /* fine if (pid = 0) */

   /* if ((pid != 0) padre */

   if (where == BACKGROUND)
   /* in caso di background stampa pid e ritorna */
   {  printf (" Background process ID %d\n", pid);
      return (0);  }
   else
     if (where == PIPE)
     /* in caso di pipe non dobbiamo aspettare
        il processo figlio */
     /* ma padre e figlio eseguono in parallelo */
       return(0);
     else
     {
     /* attesa del processo generato:
        il padre attende il figlio
        se e' il caso.  Si attende fino a che non
        termina il processo che e' stato generato  */
       while (((returnst = wait (&exitst)) != pid) &&
(returnst != -1));
       return (returnst == -1 ? -1 : exitst);
     }
}
```