Lo shell

Le operazioni dello shell possono essere riassunte come segue:

1. il "main command loop"
2. analisi sintattica
3. esecuzione di comandi "built-in"
4. esecuzione di pipeline
5. esecuzione di sequenze
6. elaborazione in background
7. redirezione di file

```
/* Includes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
.
.
.



/* Macros */

#define MAX_STRING_LENGTH       200
#define MAX_TOKENS              100
#define MAX_TOKEN_LENGTH        30
#define MAX_SIMPLE              5
#define MAX_PIPES               5
#define DEFAULT_PERMISSION      0660
.
.
.

/* Enumerators */

enum{FALSE,TRUE};
enum pipeEnum{ READ,WRITE };
enum IOEnum{ NO_REDIRECT, FILE_REDIRECT };
enum descriptorEnum {STDIN, STDOUT, STDERR };
enum builtInEnum{ ECHO_BUILTIN, CD, .... }
.
.
.
```

```
/* Every simple command has one of these associated
   with it
*/
struct simple
{
char* token[MAX_TOKENS]; /*tokens of the command */
int count;               /* the number of tokens */
int outputRedirect;      /* set to IOEnum */
int inputRedirect;       /* set to IOEnum */
int append;              /* TRUE for append mode */
char *outputFile;        /* out file or NULL */
char *inputFile;         /* in file or NULL */
}


/* Every pipeline has one of these associated with
   it
*/
struct pipeline
{
struct simple simple[MAX_SIMPLE]; /*commands */
int count;    /* no. of simple commands */
}


/* Every command sequence has one of these
   associated with it
*/
struct sequence
{
struct pipeline pipeline[MAX_PIPES] /* Pipes */
int count; / * no. of pipes */
int background; /* TRUE if this is a background
                   sequence*/
}




/* Globals */

char *builtIns[] = { "echo", "cd", ....};
char line[MAX_STRING_LENGTH];
char tokens[MAX_TOKENS][MAX_TOKEN_LENGTH];
int tokenCount; /* no. of tokens in current line */
int errorFlag; /* true when error */
```

```
main( argc, argv )

int argc;
char * argv[];

{
     initialize();  /* Initialize some globals */
     commandLoop(); /* Accept and process commands*/
     return(0); /* exit success */
}


/*************************************************/

commandLoop()

{
struct sequence sequence;

/* Accept and process commands until a ^D occurs */
while(TRUE)
     {
     displayPrompt();
     if( gets(line) == NULL) break; /* get a line */
     tokenize(); /*break input line into tokens */
     errorFlag = FALSE;  /* reset error flag */

     if(tokenCount > 1)
          {
          sequence = parseSequence(); /*parse line*/

          /* If no errors occurred during the
             parsing, execute the command */
          if(!errorFlag) executeSequence(&sequence);
          }
     }
```

```
/*********
*
*  COMMAND EXECUTION ROUTINES
*
**********/


executeSequence(p)

struct sequence *p;

{
  int i;


  /* execute a sequence of statements (possibly
     just one) */
  if( p->background) /* execute in background */
    {
    if( fork() == 0)
      {
      printf([%d]\n",getpid()); /* display PID */
      for(i=0; i < p->count; i++)
        executePipeline(&p->pipeline[i]);
      exit(0); /* exit success */
      }
    }
  else /* execute in foreground */
    for(i=0; i < p->count; i++)
      executePipeline(&p->pipeline[i]);
}
```

```
executePipeline(p)

struct pipeline *p;

{
int pid;

/* execute every simple command in pipeline
   (possibly one) */
if(p->count==1 && builtIn(p->simple[0].token[0])
  executeSimple(&p->simple[0]); /* execute */

else
  {
  if((pid=fork()) == 0)
    {
    /* Child shell executes simple command */
    if(p->count == 1)
       executeSimple(&p->simple[0]);
    else
       executePipes(p);
    exit(0); /* exit success */
    }

  else
    {
    /* Parent shell waits for child to complete */
    waitForPID(pid);
    }
  }
}

/******************/


waitForPID(pid)

int pid;

{
int status;

/* Return when the child process with PID pid
   terminates */

while( wait(&status) != pid );
}
```

```
executePipes(p)

struct pipeline *p;

{
int pipes, status, i;
int pipefd [MAX_PIPES][2]

/* Execute two or more simple commands connected
   by pipes
*/

pipes = p->count - 1; /* no. of pipes to build */
for(i=0; i < pipes; i++) /* build the pipes */
   pipe(pipefd[i]);

for(i=0; i < p->count; i++) /* build one process
                               per pipe */
  {
  if( fork() != 0 ) continue;

    /* Child shell */
    /* Connect stdin to pipe if not the 1st cmd */
    if(i != 0)
      dup2(pipefd[i-1][READ],STDIN);

    /* Connect stdout to pipe if not the last */
    if(i != p->count - 1)
      dup2(pipefd[i][WRITE],STDOUT);

    /* Close all of the pipes' file descr. */
    closeAllPipes(pipefd,pipes);

    /* Execute the simple command */
    executeSimple(&p->simple[i]);
    exit(0);  /* exit success */
  }

  /* The parent shell comes here after forking the
     children */
  closeAllPipes(pipefd,pipes);
  for(i=0; i < p->count; i++)
    wait(&status); /* wait for children to finish*/
}
```

```
closeAllPipes(pipefd,pipes)

int pipefd[][2];
int pipes;

{
int i;

/* Close every pipe's file descriptors */

for(i=0; i < pipes; i++)
  {
  close(pipefd[i][READ]);
  close(pipefd[i][WRITE]);
  }
}

/******************/


executeSimple(p)

struct simple *p;

{
int copyStdin,copyStdout;

/* Execute a simple command */
if(builtIn(p->token[0]))   /* Built in command */
  {
  /* The parent shell is executing this, so
     remember stdin/out in case of built-in
     redirection */
  copyStdin = dup(STDIN);
  copyStdout = dup(STDOUT);
  if(redirect(p))
    executeBuiltin(p);  /* Execute built-in */
  /* Restore stdin and stdout */
  dup2(copyStdin,STDIN);
  dup2(copyStdout,STDOUT);
  close(copyStdin);
  close(copyStdout);
  }

else if(redirect(p))  /* Redirect if necessary */
  executePrimitive(p);
}
```

```
executePrimitive(p)

struct simple *p;

{
/* Execute a command by exec'ing */
if( execvp(p->token[0],p->token) == -1)
  exit(1); /* exit failure */
}




/**********
*
*  BUILT-IN COMMANDS
*
**********/


builtInCode(token)

char *token;

{
/* Return the index fo token in the builtIns array
*/

return( findString(builtIns,token) );
}



/*******************/


builtIn(token)

char *token;

{
/* Return TRUE if token is a built-in */
return( builtInCode(token) != NOT_FOUND );
}
```

```
executeBuiltIn(p)

struct simple *p;

{
/* execute a single built-in command */

switch( builtInCode(p->token[0]) )

  {
  case CD:
    executeCd(p);
    break;

  case ECHO_BUILTIN:
    executeEcho(p);
    break;

  .
  .
  .

  }

}


/******************/

executeEcho(p)

struct simple *p;

{
int i;

/* Echo the tokens in this command */
for (i=1; i < p->count; i++)
  printf("%s ", p->token[i]);

printf("\n");

}
```

```
executeCd(p)

struct simple *p;

{
/* Change directory */

if( p->count !=2 )
  error("...");
else if (chdir(p->token[1]) == -1)
  error("...");

}





/**********
*
*  REDIRECTION
*
**********/

redirect(p)

struct simple *p;

{
int mask;

/* Perform input redirection */
if( p->inputRedirect == FILE_REDIRECT )
  if( !dupFd( p->inputFile,O_RDONLY, STDIN) )
    return(FALSE);

/* Perform output redirection */
mask = O_CREAT | O_WRONLY |
       (p->append? : O_APPEND : O_TRUNC );
if( p->outputRedirect == FILE_REDIRECT )
  if( !dupFd( p->outputFile,mask, STDOUT) )
    return(FALSE);

/* If I got here, then everything went OK */
return(TRUE);
}
```

```
dupFd( name, mask, stdFd )

char *name;
int mask,stdFd;

{

int fd;

/* Duplicate a new file descriptor over stdin/out
*/
fd = open(name,mask,DEFAULT_PERMISSION);

if(fd == -1)
  {
  error("Cannot redirect\n);
  return(FALSE);
  }

dup2(fd,stdFd); /* copy over standard fd */
close(fd);  /* Close other one */
return(TRUE);
}



/***************************/

error (str)

char *str;

{
/* Display str as an error to the standard error
channel
*/
fprintf(stderr,"%s", str);
errorFlag=TRUE;

}
```