Sistemi Operativi[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

a.a. 2015/16

Lezione XI: Processi, shell, file

Shell

Shell

La *shell* è l'*interprete dei comandi* che l'utente dà al sistema operativo. Ne esistono grafiche e testuali.

In ambito GNU/Linux la piú diffusa è una shell testuale `bash`, che fornisce i costrutti base di un linguaggio di programmazione (variabili, strutture di controllo) e primitive per la gestione dei processi e dei *file*.

shell (pseudo codice)


```
1  while (1){ /* repeat forever */
2        type_prompt(); /* display prompt on the screen */
3        read_command(command, parameters); /* read input from terminal */
4        if (fork() > 0){ /* fork off child process */
5              /* Parent code. */
6              waitpid(1, &status, 0); /* wait for child to exit */
7        } else {
8             /* Child code. */
9             execve(command, parameters, 0); /* execute command */
10       }
11 }
```

Lanciare programmi con la shell

- Per iniziare l'esecuzione di un programma basta scrivere il nome del file
  - /bin/ls
- Il programma è trattato come una *funzione*, che prende dei parametri e ritorna un intero (`int main(int argc, char*argv[])`). Convenzione: 0 significa "non ci sono stati errori", > 0 errori (2 errore nei parametri), parametri - $\rightsquigarrow$ opzioni
  - /bin/ls /usr
  - /bin/ls piripacchio
- Si può evitare che il padre aspetti la terminazione del figlio
  - /bin/ls /usr &
- Due programmi in sequenza
  - /bin/ls /usr ; /bin/ls /usr
- Due programmi in parallelo
  - /bin/ls /usr & /bin/ls /usr

Esercizi

1. Scrivere, compilare (cc −o nome nome.c) ed eseguire un programma che *forca* un nuovo processo.
2. Scrivere un programma che stampi sullo schermo ``Hello world! (numero)'' per 10 volte alla distanza di 1 secondo l'una dall'altra (`sleep(int)`). Terminare il programma con una chiamata `exit(0)`
3. Usare il programma precedente per sperimentare l'esecuzione in sequenza e in parallelo
4. Controllare il valore di ritorno con `/bin/echo $?`
5. Tradurre il programma in assembly con cc −S −masm=intel nome.c
6. Modificare l'assembly affinché il programmi esca con valore di ritorno 3 e controllare con `echo $?` dopo aver compilato con `cc -o nome nome.s`

POSIX Syscall (file mgt)

fd = creat(name, mode)	Obsolete way to create a new file
fd = mknod(name, mode, addr)	Create a regular, special, or directory i-node
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
pos = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information
s = fstat(fd, &buf)	Get a file's status information
fd = dup(fd)	Allocate a new file descriptor for an open file
s = pipe(&fd[0])	Create a pipe
s = ioctl(fd, request, argp)	Perform special operations on a file
s = access(name, amode)	Check a file's accessibility
s = rename(old, new)	Give a file a new name
s = fcntl(fd, cmd, ...)	File locking and other operations

POSIX Syscall (file mgt cont.)

s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system
s = sync()	Flush all cached blocks to the disk
s = chdir(dirname)	Change the working directory
s = chroot(dirname)	Change the root directory

File

```c
int main(){
        pid_t pid;
        int f, off;
        char string[] = "Hello, world!\n";

        lsofd("padre (senza figli)");
        printf("padre (senza figli) open *\n");
        f = open("provaxxx.dat", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
        if (f == -1){
                perror("open");
                exit(1);
        }
        lsofd("padre (senza figli)");
        if (write(f, string, (strlen(string))) != (strlen(string)) ){
                perror("write");
                exit(1);
        }

        off = lseek(f, 0, SEEK_CUR);
        printf("padre (senza figli) seek: %d\n", off);

        printf("padre (senza figli) fork *\n");
        if ( (pid = fork()) < 0){
                perror("fork");
                exit(1);
        }
```


File (cont.)

```c
        if (pid > 0){
                lsofd("padre");
                printf("padre write & close *\n");
                off = lseek(f, 0, SEEK_CUR);
                printf("padre seek prima: %d\n", off);
                if (write(f, string, (strlen(string))) != (strlen(string)) ){
                        perror("write");
                        exit(1);
                }
                lsofd("padre");
                off = lseek(f, 0, SEEK_CUR);
                printf("padre seek dopo: %d\n", off);
                close(f);
                exit(0);
        }
        else {
                lsofd("figlio");
                printf("figlio write & close *\n");
                off = lseek(f, 0, SEEK_CUR);
                printf("figlio seek prima: %d\n", off);
                if (write(f, string, (strlen(string))) != (strlen(string)) ){
                        perror("write");
                        exit(1);
                }
                lsofd("figlio");
                off = lseek(f, 0, SEEK_CUR);
                printf("figlio seek dopo: %d\n", off);
                close(f);
                exit(0);
        }
}
```


Per fare esperimenti con i file descriptor può essere utile una funzione come la seguente

```c
#include <stdio.h>
#include <sys/stat.h>
#define _POSIX_SOURCE
#include <limits.h>

void lsofd(const char* name){
        int i;
        for (i=0; i<_POSIX_OPEN_MAX; i++){
                struct stat buf;
                if (fstat(i, &buf) == 0){
                    printf("%s fd:%d i-node: %d\n", name, i, (int)buf.st_ino);
                }
        }
}
```


Un vero linguaggio di programmazione

La shell è un vero (Turing-completo) linguaggio di programmazione (interpretato)

- Variabili (create al primo assegnamento, uso con $, **export** in un'altra shell).
  - x="ciao"; y=2 ; /bin/**echo** "$x $y $x"
- Istruzioni condizionali (valore di ritorno 0 $\rightsquigarrow$ true)
  - **if** /bin/ls piripacchio; **then** /bin/**echo** ciao; **else** /bin/**echo** buonasera; **fi**
- Iterazioni su insiemi
  - **for** i **in** a b c d e; **do** /bin/**echo** $i; **done**
- Cicli
  - ```
    /usr/bin/touch piripacchio
    while /bin/ls piripacchio; do
     /usr/bin/sleep 2
     /bin/echo ciao
    done & ( /usr/bin/sleep 10 ; /bin/rm piripacchio )
    ```
    ```

Esercizi

1. Per ciascuno dei file `dog`, `cat`, `fish` controllare se esistono nella directory `bin` (hint: usare `/bin/ls` e nel caso scrivere ‘‘`Trovato`’’)
2. Consultare il manuale (programma `/usr/bin/man`) del programma `/bin/test` (per il manuale man **test**)
3. Riscrivere il primo esercizio facendo uso di `test`

Input e Output

In generale il paradigma UNIX permette alle applicazioni di fare I/O tramite:

Input

- Parametri al momento del lancio
- Variabili *d'ambiente*
- File (tutto ciò che può essere gestito con le syscall `open, read, write, close`)
  - Terminale (interfaccia testuale)
  - Device (per es. il mouse potrebbe essere `/dev/mouse`)
  - Rete (socket)

Output

- Valore di ritorno
- Variabili *d'ambiente*
- File (tutto ciò che può essere gestito con le syscall `open, read, write, close`)
  - Terminale (interfaccia testuale)
  - Device (per es. lo schermo in modalità grafica potrebbe essere `/dev/fb`)
  - Rete (socket)

Redirezioni

Ad ogni processo sono sempre associati tre file (già aperti)

- Standard input (Terminale, tastiera)
- Standard output (Terminale, video)
- Standard error (Terminale, video, usato per le segnalazione d'errore)

Possono essere *rediretti*

- /usr/bin/sort < lista Lo stdin è il file `lista`
- /bin/ls > lista Lo stdout è il file `lista`
- /bin/ls piripacchio 2> lista Lo stderr è il file `lista`
- (**echo** ciao & date ; ls piripacchio) 2> errori 1>output

Pipe

ls | sort

```
1  int main(void){
2          int fd[2], nbytes; pid_t childpid;
3          char string[] = "Hello, world!\n";
4          char readbuffer[80];
5
6          pipe(fd);
7          if(fork() == 0){
8                  /* Child process closes up input side of pipe */
9                  close(fd[0]);
10                 write(fd[1], string, (strlen(string)+1));
11                 exit(0);
12         } else {
13                 /* Parent process closes up output side of pipe */
14                 close(fd[1]);
15                 nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
16                 printf("Received string: %s", readbuffer);
17         }
18         return(0);}
```

Pipe (cont.)


```
1       if(fork() == 0)
2       {
3               /* Close up standard input of the child */
4               close(0);
5
6               /* Duplicate the input side of pipe to stdin */
7               dup(fd[0]);
8               execlp("sort", "sort", NULL);
9       }
```


Pipe

La pipe è un canale, analogo ad un file, bufferizzato in cui un processo scrive e un altro legge. Con la shell è possibile collegare due processi tramite una pipe anonima.
Lo stdout del primo diventa lo stdin del secondo
/bin/ls | sort
ls −lR / | sort | more
funzionalmente equivalente a
ls −lR >tmp1; sort <tmp1 >tmp2; more<tmp2; rm tmp∗
Molti programmi copiano lo stdin su stdout dopo averlo elaborato: sono detti filtri.