Sistemi Operativi[1]

Mattia Monga

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

a.a. 2014/15

Lezione VIII: Shell 2

Astrazioni fornite dal s.o.

Per risolvere il suo problema Ada *deve* fare uso delle astrazioni fornite dal s.o.. Tipicamente:

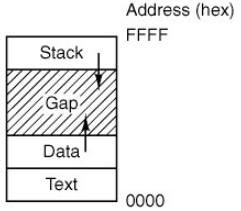
- System call
- Memoria virtuale
- Processo
- File
- Shell

L'insieme di queste costituisce una macchina virtuale piuttosto differente dal dispositivo elettronico `i386`.

Memoria virtuale

Il programmatore è libero di considerare un unico spazio di memoria, interamente dedicato al suo programma. Questo spazio può anche essere superiore alla memoria fisicamente disponibile.

Generalmente la memoria virtuale è divisa in *segmenti:* testo (codice), dati inizializzati, stack e heap.



Processo

Programma

Un programma è la codifica di un algoritmo in una forma eseguibile da una macchina specifica.

Processo

Un processo è un programma in esecuzione.

Thread

Un thread (*filo conduttore*) è una sequenza di istruzioni in esecuzione: piú thread possono condividere lo spazio di memoria in cui le istruzioni lavorano. Il termine assume anche un'accezione tecnica nei sistemi operativi che distinguono le due astrazioni.

Ogni processo dà vita ad **almeno** un thread. Ogni CPU in un dato istante può eseguire **al piú** un thread.

POSIX Syscall (process mgt)

pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = wait(&status)	Old version of waitpid
s = execve(name, argv, envp)	Replace a process core image
exit(status)	Terminate process execution and return status
size = brk(addr)	Set the size of the data segment
pid = getpid()	Return the caller's process id
pid = getpgrp()	Return the id of the caller's process group
pid = setsid()	Create a new session and return its process group id
l = ptrace(req, pid, addr, data)	Used for debugging

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`
7. Modificare l'assembly in modo che usi `scanf` per ottenere il numero di saluti.

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

```
1  int main(){
2          pid_t pid;
3          int f, off;
4          char string[] = "Hello, world!\n";
5
6          lsofd("padre (senza figli)");
7          printf("padre (senza figli) open *\n");
8          f = open("provaxxx.dat", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
9          if (f == -1){
10                 perror("open");
11                 exit(1);
12         }
13         lsofd("padre (senza figli)");
14         if (write(f, string, (strlen(string))) != (strlen(string)) ){
15                 perror("write");
16                 exit(1);
17         }
18
19         off = lseek(f, 0, SEEK_CUR);
20         printf("padre (senza figli) seek: %d\n", off);
21
22         printf("padre (senza figli) fork *\n");
23         if ( (pid = fork()) < 0){
24                 perror("fork");
25                 exit(1);
26         }
```


File (cont.)

```
1          if (pid > 0){
2                  lsofd("padre");
3                  printf("padre write & close *\n");
4                  off = lseek(f, 0, SEEK_CUR);
5                  printf("padre seek prima: %d\n", off);
6                  if (write(f, string, (strlen(string))) != (strlen(string)) ){
7                          perror("write");
8                          exit(1);
9                  }
10                 lsofd("padre");
11                 off = lseek(f, 0, SEEK_CUR);
12                 printf("padre seek dopo: %d\n", off);
13                 close(f);
14                 exit(0);
15         }
16         else {
17                 lsofd("figlio");
18                 printf("figlio write & close *\n");
19                 off = lseek(f, 0, SEEK_CUR);
20                 printf("figlio seek prima: %d\n", off);
21                 if (write(f, string, (strlen(string))) != (strlen(string)) ){
22                         perror("write");
23                         exit(1);
24                 }
25                 lsofd("figlio");
26                 off = lseek(f, 0, SEEK_CUR);
27                 printf("figlio seek dopo: %d\n", off);
28                 close(f);
29                 exit(0);
30         }
31 }
```


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

```
1  #include <stdio.h>
2  #include <sys/stat.h>
3  #define _POSIX_SOURCE
4  #include <limits.h>
5
6  void lsofd(const char* name){
7          int i;
8          for (i=0; i<_POSIX_OPEN_MAX; i++){
9                  struct stat buf;
10                 if (fstat(i, &buf) == 0){
11                     printf("%s fd:%d i-node: %d\n", name, i, (int)buf.st_ino);
12                 }
13         }
14 }
```