



# Sistemi Operativi<sup>1</sup>

Mattia Monga

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

a.a. 2017/18

<sup>1</sup>© 2008–18 M. Monga. Creative Commons Attribuzione — Condividi allo stesso modo 4.0 Internazionale. <http://creativecommons.org/licenses/by-sa/4.0/deed.it>. Immagini tratte da [2] e da Wikipedia.



# Lezione XI: Processi, shell, file



# 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: più thread possono condividere lo spazio di memoria in cui le istruzioni lavorano. Ogni processo dà vita ad **almeno** un thread d'esecuzione. Ogni CPU in un dato istante può eseguire **al più** un thread.

I termini hanno un'accezione generale e una tecnica: spesso corrispondono a specifiche strutture dati nei sistemi operativi.



# POSIX Syscall (process mgt)

## UNIX originario: process $\mapsto$ PCB

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

## Il meccanismo fondamentale della fork



Sistemi Operativi

Bruschi Monga Re

Processi e Thread

Shell

Esercizi

File

(ora passiamo al C, per gestire meglio la complessità, ma non cambia in assembly)

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(void){
6     int x = fork();
7     if (x < 0){
8         perror("Errore nella fork:");
9         exit(1);
10    }
11    if (x != 0){
12        while(1) printf("Processo padre (x == %d)\n", x);
13    }
14    else { // x == 0
15        while(1) printf("Processo figlio (x == %d)\n", x);
16    }
17    return 0;
18 }
```

216

## Esercizi



Sistemi Operativi

Bruschi Monga Re

Processi e Thread

Shell

Esercizi

File

- 1 Scrivere un programma che produca 3 processi.
- 2 Scrivere un programma saluti che stampa sullo schermo "Hello world! (numero)" per 10 volte alla distanza di 1 secondo l'una dall'altra (`sleep(int)`).
- 3 Spiegare succede se la `fork` viene usata in un ciclo come il seguente:

```
while ((p = fork()) != 0) {
    if (p > 0) {
        /* do something */
    } else {
        exit(0);
    }
}
```

217

## Shell



Sistemi Operativi

Bruschi Monga Re

Processi e Thread

Shell

Esercizi

File

### 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 più 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*.

218

## shell (pseudo codice)



Sistemi Operativi

Bruschi Monga Re

Processi e Thread

Shell

Esercizi

File

```
while (1){
    display_prompt();
    read_command(command, command_parameters);
    if (fork() > 0){
        /* Parent */
        waitpid(1, &status, 0);
    } else {
        execve(command, command_parameters, environment);
    }
}
```

La `execve` sostituisce il processo con quello che si genera dal programma (un *file*) passato come primo argomento.

219



Dopo aver letto il manuale `man execve`:

- 1 Implementare una *shell* che permetta di eseguire (senza parametri, usando NULL come environment) i programmi in `/bin` scrivendone il nome.

220



- Per iniziare l'esecuzione di un programma basta scrivere il nome del file
  - `/bin/ls` oppure `./ls` (o `ls` se `bin` è nel PATH di ricerca)
- Il programma 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 - ~> opzioni
  - `/bin/ls /usr`  
`argv[0]="/bin/ls" argv[1]="/usr"`
  - `/bin/ls piripacchio`  
`argv[0]="/bin/ls" argv[1]="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`

221



- 1 Usare il programma precedente per sperimentare l'esecuzione in sequenza e in parallelo
- 2 Il valore di ritorno dell'ultimo programma eseguito è conservato dalla shell nella *variabile d'ambiente*? (il nome è il punto di domanda... Si accede al suo valore con `$?`). Controllare il valore di ritorno con `/bin/echo $?`
- 3 Tradurre il programma in assembly con `gcc -S -masm=intel nome.c`
- 4 Modificare l'assembly affinché il programma esca con valore di ritorno 3 e controllare con `/bin/echo $?` dopo aver compilato con `gcc -o nome nome.s`

222



### File

Una sequenza di byte che esistono indipendentemente dall'esecuzione dei programmi (e quindi sono persistenti rispetto all'attivazione dei processi)

Sono identificati da nomi (link nel gergo di Unix) organizzati gerarchicamente in un *file system*.

223

## POSIX Syscall (file mgt)



Sistemi Operativi

Bruschi  
Monga Re

Processi e Thread

Shell

Esercizi

File

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

224

## POSIX Syscall (file mgt cont.)



Sistemi Operativi

Bruschi  
Monga Re

Processi e Thread

Shell

Esercizi

File

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

225

## File



Sistemi Operativi

Bruschi  
Monga Re

Processi e Thread

Shell

Esercizi

File

Che succede se un file viene manipolato da processi diversi?  
(`lsofd` è definito più avanti)

```
23 int main(){
24     pid_t pid;
25     int f, off;
26     char string[] = "Hello, world!\n";
27
28     lsofd("padre (senza figli)");
29     printf("padre (senza figli) open *\n");
30     f = open("provaxxx.dat", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
31     if (f == -1){
32         perror("open");
33         exit(1);
34     }
35     lsofd("padre (senza figli)");
36     if (write(f, string, (strlen(string))) != (strlen(string)) ){
37         perror("write");
38         exit(1);
39     }
40
41     off = lseek(f, 0, SEEK_CUR);
42     printf("padre (senza figli) seek: %d\n", off);
43
44     printf("padre (senza figli) fork *\n");
45     if ( (pid = fork()) < 0){
46         perror("fork");
47         exit(1);
48     }
```

226

## File (cont.)



Sistemi Operativi

Bruschi  
Monga Re

Processi e Thread

Shell

Esercizi

File

```
49     if (pid > 0){
50         lsofd("padre");
51         printf("padre write & close *\n");
52         off = lseek(f, 0, SEEK_CUR);
53         printf("padre seek prima: %d\n", off);
54         if (write(f, string, (strlen(string))) != (strlen(string)) ){
55             perror("write");
56             exit(1);
57         }
58         lsofd("padre");
59         off = lseek(f, 0, SEEK_CUR);
60         printf("padre seek dopo: %d\n", off);
61         close(f);
62         exit(0);
63     }
64     else {
65         lsofd("figlio");
66         printf("figlio write & close *\n");
67         off = lseek(f, 0, SEEK_CUR);
68         printf("figlio seek prima: %d\n", off);
69         if (write(f, string, (strlen(string))) != (strlen(string)) ){
70             perror("write");
71             exit(1);
72         }
73         lsofd("figlio");
74         off = lseek(f, 0, SEEK_CUR);
75         printf("figlio seek dopo: %d\n", off);
76         close(f);
77         exit(0);
78     }
79 }
80 }
```

227



Sistemi  
Operativi

Bruschi  
Monga Re

Processi e  
Thread

Shell  
Esercizi

File

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <fcntl.h>
5  #include <sys/stat.h>
6  #define _USE_POSIX
7  #include <limits.h>
8
9  void lsofd(const char* name){
10     int i;
11     for (i=0; i<_POSIX_OPEN_MAX; i++){
12         struct stat buf;
13         if (fstat(i, &buf) == 0){
14             printf("%s fd:%d i-node: %d\n",
15                 name, i, (int)buf.st_ino);
16         }
17     }
18 }
```

228



Sistemi  
Operativi

Bruschi  
Monga Re

Processi e  
Thread

Shell  
Esercizi

File

415