Sistemi Operativi[1]

Mattia Monga

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

a.a. 2019/20

Lezione XI: Processi, shell, file

Processo

Programma

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

Processo

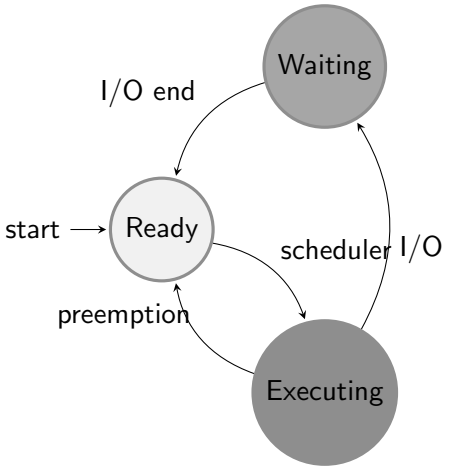
Un processo è un programma in esecuzione.

Thread

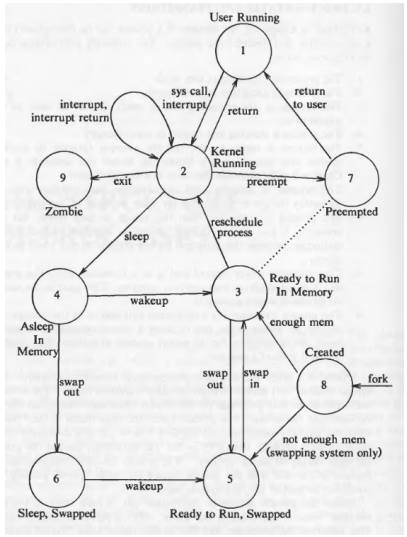
Un thread (*filo conduttore*) è una sequenza di istruzioni in esecuzione: piú thread possono condividere lo spazio di memoria in cui le istruzioni lavorano. Ogni processo dà vita ad **almeno** un thread d'esecuzione. Ogni CPU in un dato istante può eseguire **al piú** un thread.

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

Stati di un processo

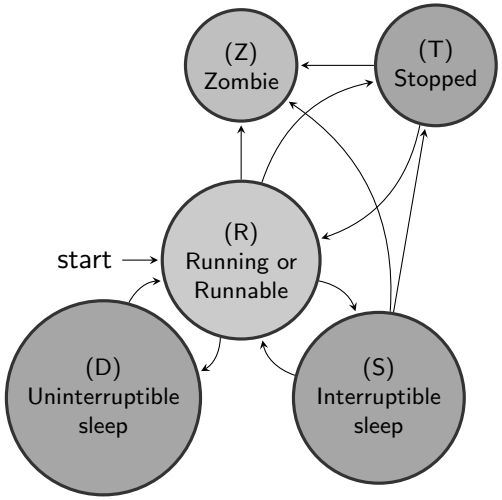


Stati di un processo (Unix)



Da:
Maurice Bach,
"The design of the UNIX operating system",
Prentice-Hall,
1986

Stati di un processo (Linux)



Le lettere corrispondono all'output del comando
`/usr/bin/ps -eo pid,stat`

POSIX Syscall (process mgt)

UNIX originario: process $\mapsto$ PCB

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

Il meccanismo fondamentale della fork

(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 }
```

Esercizi

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);
  }
}
```


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)


```
while (1){
     display_prompt();
     read_command(command, command_parameters);
     if (fork() > 0){
          /* Parent */
          int status;
          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.

Esercizio

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.

Lanciare programmi con la shell

- 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 - $\rightsquigarrow$ 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`

Esercizi

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 programmi esca con valore di ritorno 3 e controllare con `/bin/echo $?` dopo aver compilato con `gcc -o nome nome.s`

Un vero linguaggio di programmazione

La shell è un vero e proprio linguaggio di programmazione (interpretato)

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


Esercizi

1. Scrivere uno script che controlli se esiste nella directory `/bin` un file che si chiama: `dog`, `cat` o `fish`, scrivendo "Trovato: " e il nome del file. (hint: usare un ciclo `for` e `ls`)
2. Consultare il manuale (programma `man`) del programma `test` (per il manuale `man test`)
3. Ripetere il primo esercizio facendo uso di `test`
    ```