Creazione di processi

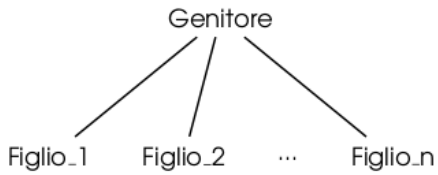
La creazione di un processo richiede alcune operazioni da parte del Sistema Operativo:

- Creazione nuovo ID (PID – Process Identifier),
- Allocazione Memoria (Codice, Dati),
- Allocazione altre risorse (stdin, stdout, dispositivi I/O in genere),
- Gestione informazioni sul nuovo processo (es. priorita),
- Creazione PCB (Process Control Block) contenente le informazioni precedenti.

Processi in Unix

Un processo è sempre creato da un altro processo tramite un'opportuna chiamata a sistema. Fa eccezione `init` (pid = 1) che viene creato al momento del boot.

Il processo creante e detto **parent** (genitore), mentre il processo creato **child** (figlio). Si genera una struttura di "parentela" ad albero.



Relazioni Dinamiche

Cosa accade dopo la creazione?

- Il processo genitore attende il processo figlio.
  Esempio: Esecuzione di un programma da shell

```
> xcalc
<la shell attende la terminazione del programma>
```

  La shell è genitore del nuovo processo `xcalc`
  NOTA: il terminale e associato al nuovo programma: `ctrl-c`, ad esempio, termina la calcolatrice.

- Il processo genitore continua
  esempio: Esecuzione in *background* di un programma da shell:

```
> xcalc &
[1] 17589    <-- PID del processo figlio
> ps
 PID TTY          TIME CMD
14695 pts/1    00:00:00 bash
17589 pts/1    00:00:00 xcalc
17590 pts/1    00:00:00 ps
>
```

  In questo caso i due processi procedono in modo *concorrente*. Tramite `ps` possiamo osservare i processi in esecuzione.

  NOTA: `ps` di default mostra solo i processi associati al terminale da cui viene lanciato. In questo caso abbiamo la shell `bash`, il programma `xcalc` e lo stesso `ps`.

  Anche in esecuzione rimane un legame genitore-figlio. Ad esempio, se chiudiamo la calcolatrice la shell viene notificata:

```
>
[1]+  Done                    xcalc
>
```

  Può essere utile che un processo si dissoci dal processo genitore. Ad esempio i *Daemon* sono processi che restano attivi fino allo shutdown del sistema: devono, ad esempio, dissociarsi dalla shell per non essere terminati alla chiusura del terminale stesso. Quando si dissociano diventano figli del processo `init` (vedremo come questo sia possibile).

Per visualizzare il PID del genitore basta passare gli opportuni parametri al programma ps (PPID significa Parent process ID):

Per visualizzare il PID del genitore basta passare gli opportuni parametri al programma ps (PPID significa Parent process ID):

```
> ps -o pid,ppid,command
  PID  PPID COMMAND
14695 14201 bash
17679 14695 xcalc
17680 14695 ps -o pid,ppid,command
>
```

Si osservi che xcalc e ps sono entrambi figli di bash.

Relazioni di contenuto

Due possibilità:

- Il figlio è un duplicato del genitore (ad esempio in UNIX)
- Il figlio esegue un programma differente (ad esempio nei sistemi Windows)

Questo è il comportamento standard ma ovviamente è possibile anche l'altra modalità in entrambi i sistemi.

System Call "fork"

La chiamata a sistema `fork` permette di creare un processo duplicato del processo genitore.

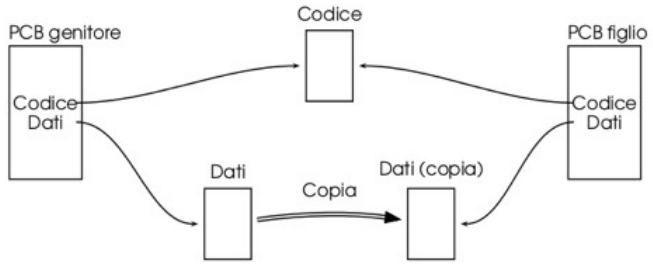
NOTA `fork`, come la maggior parte delle chiamate a sistema che discuteremo, appartiene allo standard POSIX (Portable Operating System Interface) di IEEE (Institute of Electrical and Electronics Engineers). È utilizzabile in qualsiasi sistema che supporti POSIX.

La `fork` crea un nuovo processo che

- condivide l'area codice del processo genitore
- utilizza una copia dell'area dati del processo genitore

Come fanno i processi a differenziarsi? Si utilizza il valore di ritorno della `fork`:

- <0 Errore
- =0 Processo figlio
- >0 Processo genitore: il valore di ritorno è il PID del figlio.



Schema di base di utilizzo della fork:

```
if ( (pid = fork() ) < 0 )
        perror("fork error"); // stampa la descrizione dell'errore
else if (pid == 0) {
        // codice figlio
} else {
        // codice genitore, (pid > 0)
}
// codice del genitore e del figlio: da usare con cautela!
```

NOTA: `pid_t` è un signed integer, cioè `int` in molti sistemi ma potrebbe essere di dimensioni differenti, es. `long`, ed è quindi bene usare sempre il tipo `pid_t`

Un esempio concreto:

Un esempio concreto:

```
// Esempio di utilizzo della fork.
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
main() {
        pid_t pid;

        printf("Prima della fork. pid = %d, pid del genitore = %d\n",getpid(), getppid());

        if ( (pid = fork()) < 0 )
                perror("fork error"); // stampa la descrizione dell'errore
        else if (pid == 0) {
                // figlio
                printf("[Figlio] pid = %d, pid del genitore = %d\n",getpid(), getppid());
        } else {
                // genitore
                printf("[Genitore] pid = %d, pid del mio genitore = %d\n",getpid(), getppid());
                printf("[Genitore] Mio figlio ha pid = %d\n",pid);
                sleep(1); // attende 1 secondo
        }
        // entrambi i processi
        printf("PID %d termina.\n", getpid());
}
```

che dà il seguente output:

```
> a.out
Prima della fork. pid = 25267, pid del genitore = 329
[Genitore] pid = 25267, pid del mio genitore = 329
[Genitore] Mio figlio ha pid = 25268
[Figlio] pid = 25268, pid del genitore = 25267
PID 25268 termina.
PID 25267 termina.
>
```

Esempio Come possiamo visualizzare l'alternanza nell'esecuzione dei processi genitore e figlio? Un modo è mettere le printf dentro un `while(true)`:

```
// visualizza l'esecuzione concorrente di genitore e figlio
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdbool.h>
main() {
        pid_t pid;
        int i;

        if ( (pid = fork()) < 0 )
                perror("fork error"); // stampa la descrizione dell'errore
        else if (pid == 0) {
                while(true) {
                        for (i=0;i<10000;i++) {} // riduce il numero di printf
                        printf("Figlio: pid = %d, pid del genitore = %d\n",getpid(), getppid());
                }
        } else {
                while(true) {
                        for (i=0;i<10000;i++) {} // riduce il numero di printf
                        printf("genitore: pid = %d, pid di mio genitore = %d\n",getpid(), getppid());
                }
        }
}
```

In output avremo un'alternanza (non stretta a causa dell'output bufferizzato!) degli output dei due processi. Il ciclo for "vuoto" prima della `printf` riduce il numero di stampe e permette di visualizzare meglio l'alternanza (ed evita di "saturare" il terminale di output). Variare, se necessario, il limite.

NOTA Su computer a più core i processi vengono eseguiti in parallelo su core differenti. In tale case non è necessario rallentarli con il ciclo for per vedere l'alternanza nell'output. Per vedere il carico sui vari core si può lanciare 'top' da un altro terminale. Il Linux, premendo 1, viene visualizzato il carico delle differenti CPU. Provare per esercizio.

Fallimento della fork

Quando fallisce una `fork`? Quando non è possibile creare un processo e tipicamente questo accade quando non c'è memoria per il processo o per il kernel. Ecco un piccolo test.

NOTA BENE. Il test potrebbe bloccare tutto il sistema perché i processi generati vanno ad occupare tutte le risorse e il sistema non può più prendere il controllo. È necessario limitare in numero di processi utenti tramite il comando `ulimit -u 300` (al massimo 300 processi sono ammessi per l'utente sul presente terminale). Attenzione che il limite è per terminale quindi il test va eseguito dalla stessa finestra su cui avete impostato il limite a 300.

```
main() {
        while(1)
                if (fork() < 0)
                        perror ("errore fork");
}
```

Esercizio. Quanti processi eseguono la `fork` al loop i-esimo? (pensare a quanti processi ci sono alla prima fork, quanti alla seconda, e cosi' via).

Processi orfani e processi zombie

Se metto una `sleep` subito prima della printf nel figlio lo rendo *orfano* perché termina il genitore prima di lui: viene adottato da init (processo con pid = 1):

```
// figlio
sleep(5);
printf("[Figlio] pid = %d, pid del genitore = %d\n",getpid(), getppid());
```

ottengo:

```
> a.out
Prima della fork. pid = 8127, pid del genitore = 5835
genitore: pid = 8127, pid di mio genitore = 5835
Mio figlio ha pid = 8128
PID 8127 termina.
>                                          <== 5 secondi
Figlio: pid = 8128, pid del genitore = 1        <== processo adottato!
PID 8128 termina.
```

dove si nota, appunto, l'adozione da parte di init.

NOTA Un processo orfano non viene più terminato da ctrl-c o dalla chiusura della shell (provare).

Gli *zombie* sono processi terminati ma in attesa che il genitore rilevi il loro stato di terminazione (vedremo come fare). Per osservare la generazione di un processo zombie ci basta porre la sleep prima della printf del processo genitore:

```
// genitore
sleep(5);
printf("[genitore] pid = %d, pid di mio genitore = %d\n",getpid(),getppid());
```

che dà il seguente output:

```
> a.out &                       <== notare l'esecuzione in background
[1] 8270
Prima della fork. pid = 8270, pid del genitore = 5835
Figlio: pid = 8271, pid del genitore = 8270
PID 8271 termina.               <== termina figlio e diventa zombie
> ps                            <== andiamo ad osservare i processi
  PID TTY          TIME CMD
 5835 pts/1    00:00:00 bash
 8270 pts/1    00:00:00 a.out                  <== genitore in esecuzione
 8271 pts/1    00:00:00 a.out <defunct>        <== zombie
 8272 pts/1    00:00:00 ps
>                                          <== 5 secondi
 genitore: pid = 8270, pid di mio genitore = 5835
Mio figlio ha pid = 8271
PID 8270 termina.
[1]+  Exit 18                 a.out
```