Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2020-2021

Pietro Frasca

Lezione 5

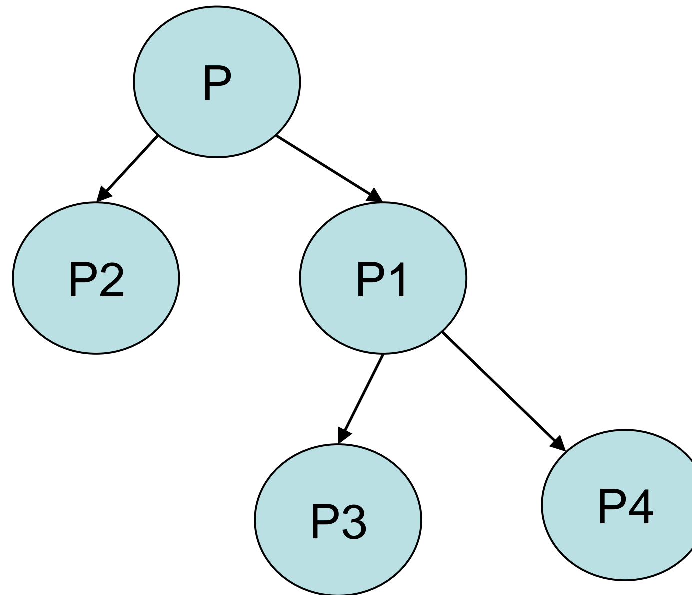
Martedì 20-10-2020

Operazioni sui processi

- Nella maggior parte dei sistemi, i processi sono eseguiti concorretemene, e possono essere creati e terminati dinamicamente.
- Tuttavia, in alcuni sistemi si possono avere applicazioni nelle quali il numero dei processi è definito inizialmente e non è più modificato durante il tempo di vita dell'applicazione. Questa gestione dei processi può essere implementata in sistemi in tempo reale, ad esempio per il controllo di impianti fisici, in cui tutti i processi sono creati all'avvio dell'applicazione *(**creazione statica**).*
- Così, i kernel di questi sistemi devono fornire funzioni per la creazione e la terminazione dei processi

Creazione e terminazione dei processi

- In generale, durante la sua esecuzione un processo può creare altri processi utilizzando opportune chiamate di sistema fornite dal kernel.
- Il processo genitore prende il nome di processo ***padre*** ed il processo creato il nome di processo ***figlio.*** Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando così un albero di processi.



- Un processo è univocamente identificato con un numero intero detto ***PID (Process IDentifier)***.
- Quando un processo crea nuovi processi, esistono due possibilità per l’esecuzione del padre. La prima prevede che esso continui la sua esecuzione concorretemene con i suoi figli; la seconda che si sospenda fino a che alcuni o tutti i suoi figli siano terminati.
- La condivisione di dati e risorse tra processi padri e figli e della loro sincronizzazione variano da sistema a sistema. Anche nel caso di terminazione di un processo, questa può avvenire secondo diverse politiche di segnalazione al processo padre.
- Generalmente, il kernel offre System Call (SC) per la ***creazione*** e ***terminazione*** dei processi. La SC di creazione dovrà inizializzare il descrittore del processo da creare ed inserirlo nella coda dei processi pronti. Analogamente, la funzione di terminazione provocherà l’eliminazione del descrittore dalla tabella dei descrittori di processo e la notifica che l'area di memoria può essere recuperata dal sistema operativo.

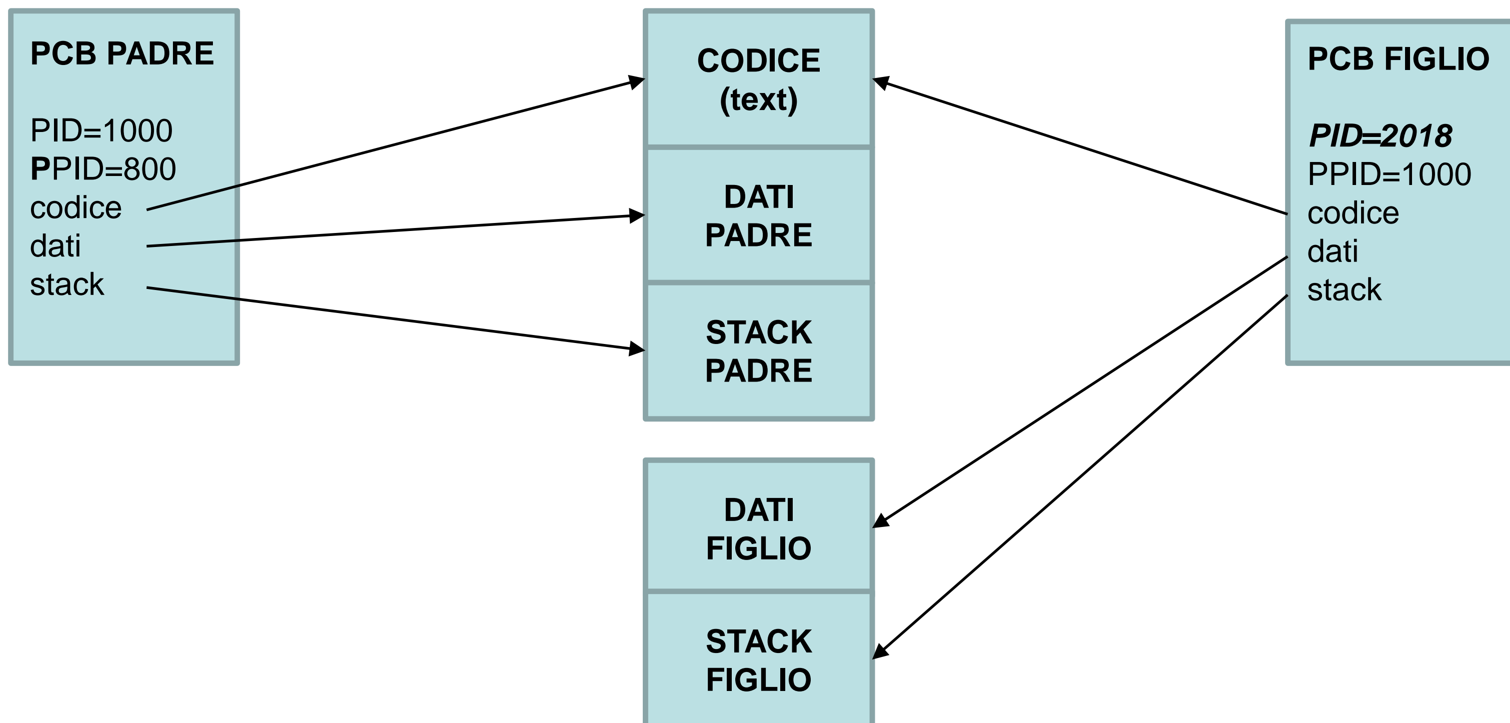
Creazione di processi in POSIX

- **POSIX** (**P**ortable **O**perating **S**ystem **I**nterface for Uni**x**), indica la famiglia degli standard definiti dall'IEEE denominati formalmente **IEEE 1003**.
- In POSIX, un nuovo processo si crea con la chiamata di sistema *fork()*.

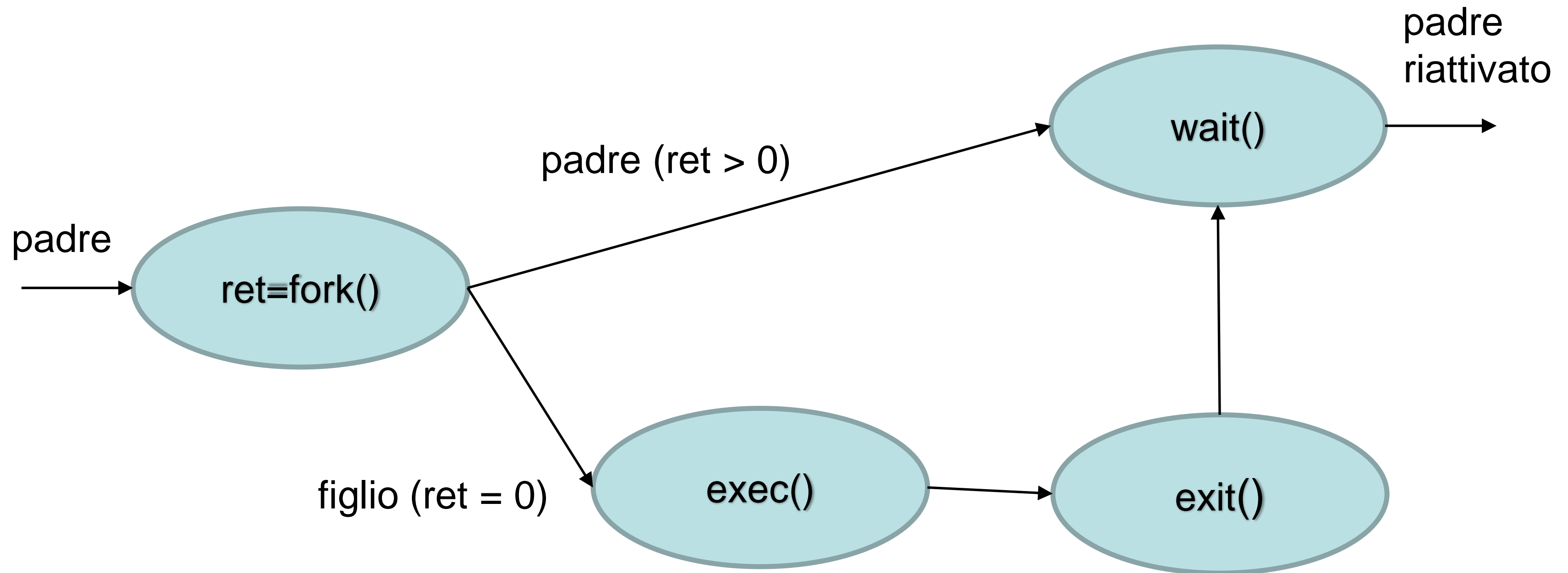
```
#include <unistd.h>
pid_t pid;
pid=fork();
```

- Entrambi i processi, padre e il figlio, riprendono l'esecuzione dall'istruzione successiva alla *fork ()*.
- La *fork* è una funzione che esegue operazioni molto onerose poiché il sistema operativo per eseguirla deve svolgere molte attività.
- In particolare, alloca al processo figlio un segmento dati e un segmento stack privati e crea un PCB per il nuovo processo inserendolo nella tabella dei processi. Il processo figlio condivide il segmento del codice del processo padre.

- Inizialmente il segmento dati del figlio è una copia del segmento dati del padre. Pertanto, ogni variabile del figlio è inizializzata al valore che aveva nel processo padre prima che eseguisse la fork().
- La *fork()* ritorna due valori differenti al padre e al figlio. Più precisamente, ritorna il valore zero al nuovo processo figlio, mentre ritorna un valore maggiore di zero al processo padre, che è l'identificatore (PID) del processo figlio.



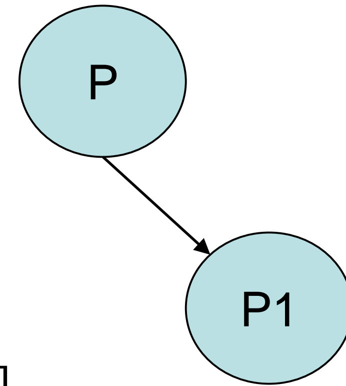
- In caso di fallimento ritorna -1. In base a questi diversi valori che la *fork* ritorna è possibile differenziare il comportamento del processo padre dal processo figlio.



Creazione di un processo con la system call fork()

Esempio di creazione di un processo in POSIX

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  pid_t ret;
  ret=fork();
    /*  fork ritorna il pid del figlio al padre,
    il valore 0 al figlio e -1 in caso di errore.
    In base a questo è possibile separare il codice del
    padre da quello del figlio.
    */
  if (ret==-1) {
    printf("Errore nella fork \n");
    exit(0);
  } else if (ret==0) {
   // codice del figlio
    printf("figlio con pid=%d \n",getpid());
  } else {
   // codice del padre
    printf("padre con pid=%d \n",getpid());
  }
  // codice eseguito da entrambi i processi, padre e figlio
  printf("Questa istruzione printf è eseguita da pid=%d \n", getpid());
}
```



Sostituzione del codice

- Tipicamente, dopo una *fork(),* uno dei due processi, padre o figlio, utilizza la chiamata di sistema ***exec()*** per sostituire lo spazio di memoria del processo con un nuovo programma.
- La *famiglia* di funzioni *exec()* sostituisce l'immagine del processo in corso con una nuova immagine di processo. La nuova immagine deve essere creata da un normale file eseguibile. Nel caso di successo, la *exec()* non ritorna alcun valore perché l'immagine del processo chiamante è sostituita dalla immagine del nuovo processo.
- La exec permette a un processo di eseguire un diverso programma. I tre segmenti codice, dati e stack del processo che esegue la *exec()* sono sostituiti con i segmenti del nuovo programma ma senza che sia creato un nuovo processo.
- Quindi, dopo la *fork()* nel sistema è presente un processo in più, mentre dopo la *exec()* il numero di processi non cambia ma il codice del processo chiamante la *exec()* è sostituito. Tuttavia, molti campi del *PCB* del processo originale restano invariati, come ad esempio il *PID* e il *PPID*.

- Inoltre, eventuali risorse allocate o file aperti nel processo chiamante la *exec()* restano accessibili al nuovo processo.
- La *exec()* è una famiglia di funzioni. Ha varie *firme*, tra le quali, due molto usate sono la *execl()* e la *execv()*.

```
execl(char *path,char *arg1,char *arg2,…char
    *argN,(char *)0)
execv(char *path,char *argv[])
```

- Alla *execl()* è possibile passare un numero variabile di parametri. L'ultimo parametro è il carattere nullo, che indica la fine della lista di parametri. Il primo parametro *path* della funzione è il nome del file da eseguire; arg_1, arg_2…arg_N sono i parametri da passare al programma specificato con il parametro *path*.

Esempio execl

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(){
  pid_t ret;
  int   stato;
  ret=fork();
  printf("pid=%d \n",getpid());
  if (ret==0) {
    //figlio
    execl("./nuovo", "Saluti", " dal processo"," padre",(char *)0);
    printf("exec fallita");
    exit(1);
  } else if (ret > 0){
    printf("sono il padre con pid=%d",getpid());
    ret=wait(&stato);
  } else
    printf ("Errore fork");
}
```

File nuovo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
Int main(int argc, char *argv[]){
  int i;
  printf("Processo chiamato da execl con PID = %d e  PPID = %d
       \n",getpid(),getppid());
  for (i=0;i<argc;i++)
    printf ("%s ",argv[i]); // visualizza i parametri d'ingresso
  printf("\n");
}
```

Terminazione di processi

- Un processo termina la sua esecuzione quando esegue la sua ultima istruzione oppure esegue la chiamata ***exit()***.

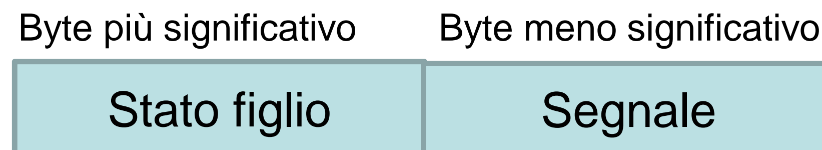
```
#include <stdlib.h>
void exit (int stato);
```

- Tutte le risorse del processo tra cui la memoria allocata, i file aperti, e buffer di I/O sono deallocati dal sistema operativo.
- Il parametro ***stato*** consente al processo figlio che chiama la exit di comunicare al processo padre, un valore di tipo intero che ne indica lo stato di uscita.
- Per rilevare la notifica del processo figlio, il padre deve sincronizzarsi con il processo figlio e attendere la sua terminazione.

- Per sincronizzarsi il padre può utilizzare le chiamate *wait* o *waitpid*.

```
#include <unistd.h>
pid_t wait(int *stato);
pid_t waitpid(pid_t pid, int *stato, int opzioni);
```

- La *wait* ritorna il *PID* di un qualsiasi figlio che è terminato, mentre *waitpid* permette, tramite il primo parametro *pid* di specificare il particolare figlio da attendere.
- L'argomento *stato*, in entrambe le funzioni, è un riferimento ad una variabile che conterrà lo stato del processo figlio quando termina. Più precisamente, nel caso di terminazione volontaria, la variabile *stato* conterrà nel suo byte più significativo il valore che il processo figlio ha passato al parametro stato chiamando *exit()*, mentre conterrà il numero del segnale che ha causato la terminazione nel caso di terminazione forzata.



- Il significato del valore *stato* passato nella funzione *exit()* è stabilito dal programmatore.
- Il terzo parametro *opzioni* in *waitpid()* stabilisce se il processo chiamante deve attendere la terminazione del figlio o invece deve continuare l'esecuzione.
- Se il valore del parametro *opzioni* è zero il processo chiamante si blocca, altrimenti il processo chiamante continua la sua esecuzione.
- Quindi, la *waitpid()* può avere sia un funzionamento bloccante che non bloccate. La *wait()*, invece, è solo bloccante e ritorna il *pid* del processo figlio che ha risvegliato il padre.
- Quando un processo termina, le sue risorse sono deallocate dal sistema operativo. Tuttavia, il suo PCB nella tabella dei processi deve rimanere fino a quando il processo padre chiama la *wait()*, in quanto il PCB contiene lo stato di uscita del processo.
- Un processo che è terminato, senza che il suo genitore non abbia ancora rilevato il suo stato di uscita con la *wait()*, entra in uno stato detto ***zombie***.

- Una volta che il padre chiama la wait(), l’identificatore di processo del processo zombie e il suo PCB sono cancellati.
- Nei sistemi Linux e Unix se un genitore è terminato senza chiamare la *wait()*, i suoi processi figli *orfani* sono *adottati* dal processo ***init*** il quale è il capostipite della gerarchia dei processi nei sistemi UNIX e Linux.

```
int main(){
   pid_t ret, pid; int stato;
   ret=fork();
   if (ret==0){
       // codice del figlio
       printf("sono il figlio pid: %d \n",getpid());
       sleep(10); // sospensione per 10 secondi
      exit(2);//valore che il padre leggerà in stato
   } else if (ret > 0){
       //codice del padre
      pid=wait(&stato);
       printf("processo figlio pid: %d terminato\n",
         pid);
      if (stato<256)
         printf("terminaz. forzata: segnale = %d \n",
                   stato);
      else
         printf("terminaz. Volontaria: stato = %d \n",
                stato>>8);
  } else
       printf("fork fallita");
}
```