Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2017-18

Pietro Frasca

Lezione 6

Martedì 24-10-2017

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

File nuovo.c

```
#include <stdio.h>
#include <stdlib.h>
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 di *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.

```
main(){
   pid_t pid; int stato;
   pid=fork();
   if (pid==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 (pid > 0){
        //codice del padre
       pid=wait(&stato);
        printf("processo figlio pid: %d terminato\n",
          pid);
       if (stato<256)
          printf("terminaz. forzata: segnale n = %d",
                    stato);
       else
          printf("terminaz. volontaria stato: %d \n",
                  stato>>8);
  } else
        printf("fork fallita");
}
```