Esecuzione e terminazione

Esercizio. Lo scopo del seguente esercizio è la comprensione approfondita del funzionamento della `fork` e, in particolare, del fatto che dopo ogni `fork` esiste un processo identico al processo genitore (tranne che per il valore di ritorno della `fork`) in esecuzione nello stesso punto del programma.

Considerare il seguente programma:

```
#include <unistd.h>
#include <stdio.h>

main() {
  pid_t f1,f2,f3;

  f1=fork();
  f2=fork();
  f3=fork();

  printf("%i%i%i ", (f1 > 0),(f2 > 0),(f3 > 0));
}
```

Domanda: che output dà? Perchè?

Soluzione: guardarla solo **dopo** aver provato a risolvere l'esercizio da soli!

System call exec

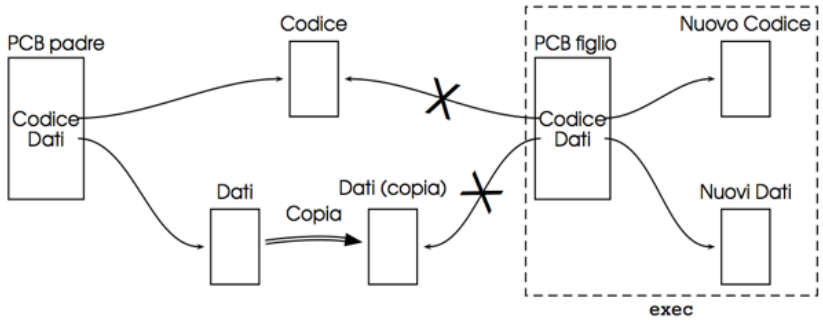
Come si fa ad eseguire un programma diverso da quello che ha effettuato la `fork`? Esiste una chiamata a sistema apposita: `exec`. Tale chiamata a sistema **sostituisce** *codice* e *dati* di un processo con quelli di un programma differente.

Lo schema seguente mostra `fork` ed `exec` assieme.

System call exec

Come si fa ad eseguire un programma diverso da quello che ha effettuato la `fork`? Esiste una chiamata a sistema apposita: `exec`. Tale chiamata a sistema **sostituisce** *codice* e *dati* di un processo con quelli di un programma differente.

Lo schema seguente mostra `fork` ed `exec` assieme.



Copy-on-write

Notare che la `exec` "butta via" la copia dei dati creata dalla `fork`. Questo è chiaramente inefficiente, soprattutto quando la `exec` viene eseguita immediatamente dopo la `fork`. Per ovviare a questo problema, viene copiata solamente la page-table, e le pagine (quelle contenenti i dati, che dovrebbero essere state copiate) sono invece etichettate come *read-only*. Un tentativo di scrittura, quindi, genera un errore che viene gestito dal kernel:

1. copiando al volo (**copy-on-write**, appunto) la pagina fisica e aggiornando opportunamente la page-table in modo che punti alla nuova copia;
2. impostando la modalità a read-write: da quel momento in poi le due copie sono indipendenti.

Quindi se si fa `fork` e subito `exec` non avviene nessuna scrittura e quindi nessuna pagina viene effettivamente copiata.

Nota storica: Precedentemente alla tecnica Copy-on-write veniva utilizzata la `vfork`, che condivideva i dati con il processo genitore in attesa di eseguire la `exec`. Eseguire `man vfork` per maggiori informazioni.

Sintassi

La `exec` ha diverse varianti che si differenziano in base al

- formato degli argomenti (lista o array `argv[]`)
- utilizzo o meno del path della shell

```
execl("/bin/<programma>", arg0, arg1, ..., NULL);
execlp("<programma>", arg0, arg1, ..., NULL);
execv("/bin/<programma>", argv);
execvp("<programma>", argv);
```

Le prime due varianti prendono una lista di argomenti terminata da NULL. Le altre due, invece, prendono i parametri sotto forma di un array di puntatori a stringhe, sempre terminato da NULL. La presenza della 'p' nel nome della exec indica che viene utilizzato il path della shell (quindi, ad esempio, non è necessario specificare /bin perché già nel path).

NOTA: Per convenzione, il primo argomento contiene il nome del file associato al programma da eseguire.

Valore di ritorno

La exec ritorna **solamente** in caso di errore (valore -1). In caso di successo il vecchio codice è completamente sostituito dal nuovo e non è più possibile tornare al programma originale. È estremamente importante capire questo punto. L'esempio successivo lo evidenzia e permette di fare alcuni test interessanti.

```
#include<stdio.h>
#include <unistd.h>
main() {
      printf("provo a eseguire ls\n");

      execl("/bin/ls","ls","-l",NULL);
      // oppure : execlp("ls","ls","-l",NULL);

      printf("non scrivo questo! \n");
      // questa printf non viene eseguita, se la exec va a buon fine
}
```

Dà il seguente output

```
provo a eseguire ls
total 16
-rwxr-xr-x 1 focardi focardi 6619 2008-03-05 17:02 a.out
-rw-r--r-- 1 focardi focardi  226 2008-03-05 17:02 exec1.c
-rw-r--r-- 1 focardi focardi  225 2008-03-05 17:02 exec1.c~
```

Si può quindi osservare che in effetti la exec **non ritorna** se il comando viene eseguito correttamente! Se sostituiamo il seguente comando alla exec del programma precedente generiamo un errore, in quanto ls2 non esiste:

```
      execlp("ls2","ls2","-l",NULL);
```

L'esecuzione in questo caso dà il seguente risultato:

```
provo a eseguire ls
non scrivo questo!
```

In questo caso la exec è andata in errore e il controllo ritorna al programma. Di conseguenza l'ultima `printf` viene eseguita stampando la stringa "non scrivo questo!"

È quindi buona norma verificare se la `exec` ritorna -1 e in tal caso gestire l'errore (in generale in C è sempre consigliabile **testare il valore di ritorno** delle chiamate a libreria e a sistema e stampare un messaggio di errore, se necessario, altrimenti diventa complesso capire dove il programma sta fallendo)

```
if (execlp("ls2","ls2","-l",NULL) == -1) {
    perror("errore durante la exec");
    // eventualmente si esce: exit(EXIT_FAILURE);
    }
```

In questo caso otteniamo:

```
provo a eseguire ls
errore durante la exec: No such file or directory
non scrivo questo!
```

che ci fornisce informazioni utili per il debug del programma.

Errori nei programmi eseguiti

Vediamo ora cosa accade se si prova ad invocare `ls` con un parametro errato. Questa situazione è nuova: il programma esiste e dovrebbe essere quindi eseguito dalla `exec` ma poi gli si passa un parametro che lo manda in errore.

```
if (execlp("ls","ls","-z",NULL) == -1) {
    perror("errore durante la exec");
    // eventualmente si esce: exit(EXIT_FAILURE);
    }
```

dà il seguente output

```
provo a eseguire ls
ls: invalid option -- z
Try `ls --help' for more information.
```

Nota: la `exec` **non** fallisce (non ritorna e non esegue nessuna istruzione del programma che l'ha invocata). L'errore a terminale è prodotto dalla `ls`. Quindi non è la `exec` a fallire ma il programma `ls` il quale ha già sovrascritto codice e dati del programma originale. Non c'è modo di gestire l'errore della `ls` dal programma chiamante.

Un esempio completo: simulare una shell

Vediamo come si può simulare il comportamento di una shell semplificando un po' la gestione dei parametri (non ci interessa fare il parse dell'input ma solo mostrare la generazione del nuovo processo e la sua esecuzione)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
main() {
    int esito;
    char comando[128];
    while(1) {
        printf("myshell# ");
        scanf("%s", comando); //lettura rudimentale: niente argomenti separati

        if ((esito=fork()) < 0)
            perror("fallimento fork");
        else if (esito == 0) {
            execlp(comando,comando,NULL); // NOTA: non gestisce argomenti
            perror("Errore esecuzione:");
            exit(EXIT_FAILURE);
            }
        // il processo genitore (shell) torna immediatamente a leggere un altro comando
    }
}
```

Proviamo a compilarlo e eseguirlo:

```
focardi@sally$ myshell
myshell# ls
myshell# exec1.c   exec2.c~  fork-fork-fork.c   shell.c~
exec1.c~  exec3.c   fork-fork-fork.c~
```

Cosa si nota di anomalo?

Non attende la terminazione del processo figlio: è come avere un `&` implicito. Lo si nota anche dal prompt `myshell#` immediatamente prima della lista dei file generata da `ls`: la shell chiede subito il comando successivo senza aspettare il risultato di quello precedente. Per poter attendere e gestire la terminazione di un processo figlio dobbiamo vedere un po' più in dettaglio cosa accade quando un processo termina.

Terminazione di un processo

La terminazione di un processo rilascia le risorse allocate dal SO al momento della creazione (ad esempio la memoria e i file aperti) e "segnala" la terminazione al genitore: alcune informazioni di stato vengono messe a disposizione al processo genitore e devono rimanere memorizzate finché non vengono processate. Parte della informazioni contenute nella PCB vengono quindi mantenute dopo la terminazione, finché il processo genitore non ha eventualmente letto tali informazioni.

Il sistema mantiene almeno:

1. il PID,
2. lo stato di terminazione;
3. il tempo di CPU utilizzato dal processo.

Esistono due chiamate a sistema:

- `exit`: termina il processo (già usata negli esempi per i casi di errore);
- `wait`: attende una `exit` di un figlio (se uno dei figli è uno *zombie* ritorna subito senza bloccarsi).

NOTA. Un processo può anche terminare in modo anomalo a causa di un errore o perchè terminato dal SO o da altri processi.

Sintassi

- `exit(int stato)`: termina il processo ritornando lo `stato` al genitore; Si usano le costanti `EXIT_FAILURE` e `EXIT_SUCCESS` che normalmente sono uguali ad 1 e 0 rispettivamente;
- `pid = wait(int &stato)`: ritorna il `pid` e lo `stato` del figlio che ha terminato. Si invoca `wait(NULL)` se non interessa lo stato. Se non ci sono figli ritorna -1.

Valore di ritorno della wait

Lo stato titornato da `wait` va gestito con opportune macro:

- `WIFEXITED(status)==true` se il figlio è uscito normalmente con una `exit`.
  `WEXITSTATUS(status)` ritorna gli 8 bit di stato passati dalla `exit`.

Valore di ritorno della wait

Lo stato titornato da `wait` va gestito con opportune macro:

- `WIFEXITED(status)==true` se il figlio è uscito normalmente con una `exit`.
  `WEXITSTATUS(status)` ritorna gli 8 bit di stato passati dalla `exit`.

  Esempio di codice:

```
if (WIFEXITED(status))
    printf("OK: status = %d\n",WEXITSTATUS(status));
```

- `WIFSIGNALED(status)==true` se il figlio è stato terminato in maniera anomala.
  `WTERMSIG(status)` ritorna il "segnale" che ha causato la terminazione.

  Esempio di codice:

```
if (WIFSIGNALED(status))
    printf("ANOMALO: status = %d\n",WTERMSIG(status));
```

- macro analoghe per stop/resume, utili per il tracing dei processi (`WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`).

Variante per attendere un processo particolare

Se si vuole attendere un processo particolare (o processi appartenenti a un gruppo particolare), si può utilizzare la chiamata a sistema `pid = waitpid(pid2,&stato,options)`

- attende il processo `pid2` (valori di `pid2` minori o uguali a zero permettono di attendere gruppi di processi, vedere il manuale per maggiori dettagli);
- se `pid2 == -1` attende un qualsiasi figlio: diventa uguale alla `wait`;

Esempio completo con exit e wait

Vediamo le chiamate a sistema `exit` e `wait` in azione nel seguente codice. Vengono creati 2 figli: il primo termina normalmente restituendo un valore al genitore, il secondo dereferenzia l'indirizzo 0 generando un segmentation fault. Il processo genitore attende la terminazione dei figli e stampa le relative informazioni.

```
#include<wait.h>
#include<stdio.h>
#include<stdlib.h>
main() {
        int pid,status;

        printf("pid = %d e pid della shell = %d\n",getpid(), getppid());

        if ((pid = fork())<0) {
                perror("errore fork"); exit(1); }

        /* figlio 1: esce normalmente inviando al genitore lo stato "42" */
        else if (pid == 0) {
                printf("Sono il figlio1! pid=%d ppid=%d\n",getpid(), getppid());
                sleep(3);
                exit(42);}

        if ((pid = fork())<0) {
                perror("errore fork"); exit(1); }

        /* figlio 2: segfault, cerca di accedere alla locazione 0 */
        else if (pid == 0) {
                int *tmp=0;
                int a;
                printf("Sono il figlio2! pid=%d ppid=%d\n",getpid(), getppid());
                sleep(5);
                a = *tmp; // segfault}

        /* solo il genitore continua e attende tutti i figli ... */
        while((pid=wait(&status)) >= 0) {
        printf("ricevuta terminazione di pid=%d\n",pid);
        if (WIFEXITED(status))
             printf("OK: status = %d\n",WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
             printf("ANOMALO: status = %d\n",WTERMSIG(status));
        }
}
```

dà il seguente output:

```
pid = 10080 e pid della shell = 5884
Sono il figlio1! pid=10081 ppid=10080
Sono il figlio2! pid=10082 ppid=10080
ricevuta terminazione di pid=10081
OK: status = 42
ricevuta terminazione di pid=10082
ANOMALO: status = 11
```

Il codice 11 corrisponde al 'segnale' di violazione di segmento (vedremo più in dettaglio i segnali nella prossima lezione).

Esercizio. Aggiungere le opportune `wait`, con relativa gestione dello stato, al codice della shell visto precedentemente. Provare anche ad eseguire, tramite tale shell, programmi che effettuano errori run-time (come divisioni per zero).

Esercizio. Una tecnica per creare *demoni* (programmi in esecuzione che non sono sotto il controllo degli utenti, come i servizi di sistema) è quella di eseguire una doppia `fork` e far terminare il primo figlio: in questo modo il processo "nipote" viene adottato da `init` e si distacca dal processo "nonno" definitivamente.
Scrivere il codice che realizzi questa tecnica facendo attenzione che

1. la seconda `fork` vegna effettuata solo dal primo figlio e non dal genitore, altrimenti si generano due nuovi processi;
2. la terminazione del primo figlio venga correttamente processata dal genitore tramite una opportuna `wait` (meglio ancora una `waitpid`), onde evitare processi *zombie*.

Soluzione esercizio sulla fork

L'output è una permutazione qualunque del seguente:

```
000 001 100 101 010 011 110 111
```

cioè tutti i numeri binari di 3 cifre in qualche ordine (dipende dallo scheduling).

Nota: esecuzioni diverse possono dare ordinamenti diversi: provare a eseguire più volte il programma e osservare l'output.

Nota: esecuzioni diverse possono dare ordinamenti diversi: provare a eseguire più volte il programma e osservare l'output.

Perchè succede? Possiamo visualizzarlo con un albero binario in cui mettiamo a destra il processo genitore (stesso id del nodo genitore) e a sinistra il processo figlio generato dalla fork.
Il valore di f1, f2 ed f3 sono quindi 0 sul ramo di sinistra e >0 sul ramo di destra