Controllo dei Processi[1]

I processi

Nei sistemi Unix/Linux ogni processo ne può generare altri. Il processo che li genera è detto *processo padre* (*parent process*), mentre i processi generati sono detti *processi figli* (*child process*). Ogni processo è identificato da un numero univoco chiamato **process identifier** (PID) che viene assegnato alla creazione in modo progressivo.
Tutti processi sono quindi generati da altri processi e di conseguenza ogni processo ha un processo padre. L'unico processo che non ha un padre è il primo, `init` (o `systemd`), che viene creato dal kernel e a cui è assegnato PID 1.
Ogni processo può conoscere il proprio PID e quello del proprio padre utilizzando rispettivamente le funzioni `getpid()` e `getppid()`.

I controllo dei processi corrisponde a un insieme di operazioni che permettono la creazione, l'esecuzione e la terminazione dei processi. Queste operazioni corrispondono a delle **system call**: `fork()`, `exit()`, `wait()` ed `execve()`, che descriveremo nelle prossime sezioni.

Creazione di processi

La funzione `fork()` crea un nuovo processo. Il suo prototipo è:

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void);
```

Il processo figlio è una copia *quasi* esatte del processo padre: riceve, infatti, una copia dei segmenti di testo, stack, dati e dei descrittori dei file, ed esegue esattamente lo stesso codice del padre. L'unica differenza è nel valore di ritorno della `fork()`. In caso di successo essa restituisce il PID del figlio al padre e zero al figlio, permettendo così di distinguere i due processi. In caso di errore (il figlio non è stato creato) restituisce -1.
Dopo l'esecuzione della `fork()` entrambi i processi continuano la loro esecuzione dall'istruzione successiva. Non è possibile stabilire a priori quale processo sarà eseguito prima: sono necessarie tecniche di sincronizzazione per garantire un particolare ordine di esecuzione.
Inoltre, dopo la `fork()`, ogni processo può modificare le variabili nei propri segmenti senza influenzare l'altro processo, mentre i descrittori dei file sono condivisi tra padre e figlio e le modifiche attuate da uno dei processi si ripercuotono sull'altro.

[1] Dispense per il corso di Sistemi di Elaborazione Informazione I, Scuola Interfacoltà di Scienze Strategiche, Università di Torino
Docente: Alessia Visconti, http://di.unito.it/~visconti

Schema di utilizzo

```
#include <unistd.h>
#include <sys/types.h>

pid_t padre;
padre = fork();
if ( padre == -1 )
        exit(1);

if (padre != 0)
{
        // codice del padre
}
else
{
        // codice del figlio
}
```

Terminazione di processi

Un processo termina in due modi:

- in modo anormale a causa di un segnale di sistema (es. `kill`);
- in modo normale usando le funzioni `_exit()` (di sistema) o `exit()` (di libreria) i cui prototipi sono:

```
#include <stdlib.h>
#include <unistd.h>
void _exit(int status);
void exit(int status);
```

Entrambe le funzioni richiedono un parametro di input (un intero) che rappresenta le condizioni di uscita del processo che sta terminando. Tali condizioni di uscita saranno passate al processo padre. Per convenzione un processo che termina correttamente ha come condizione di uscita 0. Notate che nessuna delle due funzioni ha un valore di ritorno.

La funzione `exit()` esegue alcune operazioni (che non commenteremo in questa sede) per eseguire tutte le funzioni che sono registrate come da terminare all'uscita e per chiudere gli stream sospesi. Successivamente passa il controllo al kernel invocando la funzione `_exit()`.

La funzione `_exit()` chiude i descrittori dei file, assegna eventuali figli del processo in terminazione al processo `init`, indica al processo padre che il processo sta per terminare e memorizza lo stato di uscita che viene recuperato dal padre con la funzione `wait()`, che analizzeremo nella prossima sezione.

Monitoraggio di processi

Spesso il processo padre deve essere informato quando uno dei figli termina (o cambia stato, o è bloccato). Per questo scopo utilizza la system call `wait()` o la system call `waitpid()` i cui prototipi sono:

```
#include <sys/wait.h>
#include <sys/types.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Entrambe restituiscono il PID del figlio o -1 in caso di errore (per esempio il processo che esegue la `wait()` non ha figli).
Ci sono due possibili scenari dopo l'esecuzione di una `wait()`:

- nessun figlio ha terminato: la chiamata si blocca;
- almeno un figlio ha terminato: la chiamata restituisce immediatamente.

Il valore contenuto nella variabile `status` permette di capire quale evento si è manifestato:

- il figlio ha terminato la sua esecuzione chiamando `exit()` e specificando un codice di uscita;
- il figlio è stato terminato da un segnale;
- il figlio è stato bloccato/svegliato da un segnale.

Non vedremo come *decodificare* lo stato di uscita.

La funzione `wait()` non permette al processo padre di attendere un dato figlio: possiamo farlo con la `waitpid()`, settando il parametro `pid` con il valore del PID del figlio di cui vogliamo attendere la terminazione.

Orfani e Zombie

Se il processo padre termina prima dei propri figli essi (divenuti **processi orfani**) vengono adottati da `init`.

Se un processo figlio termina prima che il padre abbia eseguito una `wait()` il suo descrittore viene conservato nella tabella dei processi, in modo che il padre possa sapere come il figlio è terminato: il kernel lo trasforma in uno **zombie**. La funzione `wait()` scandisce poi la tabella dei processi alla ricerca di figli zombie e (se essi esistono) ne legge lo stato di uscita e li rimuove. Se il processo padre non esegue la `wait()` o se questa fallisce, una voce relativa allo zombie sarà mantenuta indefinitamente nella tabella dei processi (che ha un dimensione finita!). L'unico modo per rimuoverli dal sistema è uccidere il processo padre (o attenderne la terminazione): gli zombie saranno adottati da `init` e rimossi.

Esecuzione di processi

Esistono diverse funzioni (il cui nome inizia con `exec`) che permettono di eseguire un processo, ma l'unica vera chiamata di sistema è la `execve()`. Essa carica un programma nella memoria del processo, abbandonando quello in esecuzione; lo stack, i dati, e lo heap del processo sono sostituiti da quelli del nuovo programma.
Il suo prototipo è:

```
#include <unistd.h>
int execve(const char *pathname, const char *arg, .../* ,
     (char *) NULL, char *const envp[] */);
```

L'argomento `pathname` contiene il pathname del programma; `argv` contiene gli argomenti della linea di comando da passare al nuovo programma. Si tratta di una lista di puntatori a stringa, terminati da puntatore a `NULL`. Il valore fornito per `argv[0]` corrisponde al nome del comando (ultimo elemento del pathname). `envp` specifica la lista *environment list* per il nuovo programma; è una lista di puntatori a stringa, terminati da puntatore a `NULL`, nella forma nome = valore.

Poiché sostituisce il programma chiamante, `execve()` non restituisce nulla se va a buon fine. Se, invece, viene restituito un valore di ritorno, allora è occorso qualche problema. Determinare la causa di errore non è argomento di questo corso.

Esempio

```
/* saluta.c */

#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
   int j;
   for (j = 0; j < argc; j++)
           printf("%s ", argv[j]);
   printf("\n");
   exit(0);
}
```

```
/* esegui.c */

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

int main(int argc, char *argv[])
{
   char *newargv[] = { "hello", "world", NULL };
   char *newenviron[] = { NULL };

   execve("./saluta", newargv, newenviron);
   printf("non eseguo");
   exit(-1);
}
```

Letture consigliate

Non ci sono libri in italiano che trattano l'argomento, quindi se siete interessati dovrete usare due manuali inglesi: [B1, B2]. Tuttavia diversi tutorial sono disponibili on-line, anche in italiano.

Bibliografia

[B1] Michael Kerrisk, *The Linux Programming interface - a Linux and UNIX System Programminng Handbook*, 2010, No Starch Press.

[B2] Richard Stevens e Stephen A. Rago, *Advanced Programming in the UNIX Environment*, 2005, seconda edizione, Addison-Wesley.