Processi

Tutorato del corso di Sistemi Operativi

Prof. Giuseppe Cattaneo

Vincenzo Apicella

Processi

- Un processo può essere definito come "ogni singolo programma in esecuzione".
- Può essere un programma di sistema, oppure programmi lanciati dall'utente.
- Quando UNIX esegue un processo, assegna ad ognuno un numero unico ed univoco, cioe' un "process ID" o "pid".
- Il comando UNIX "ps" elenca tutti i processi in esecuzione in quel momento sulla propria macchina, elencandone anche i pid.

Identificatori dei processi

- Ogni processo ha un identificatore unico non negativo

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

pid_t getpid ();          process ID del processo chiamante.
pid_t getppid ();         process ID del padre del processo chiamante.
uid_t getuid ();          real user ID del processo chiamante.
uid_t geteuid ();         effective user ID del processo chiamante.
gid_t getgid ();          real group ID del processo chiamante.
gid_t getegid ();         effective group ID del processo chiamante.
```

Creazione di processi

- La creazione di nuovi processi avviene tramite una chiamata alla funzione **fork** da parte di un processo già esistente.
- Dopo la chiamata, la funzione **fork** crea un nuovo processo detto **figlio**.
- Dalla **fork** si ritorna due volte:
  - Viene restituito 0 al processo figlio
  - Viene restituito il pid del figlio al processo padre
- Il processo padre ed il processo figlio proseguono nell'esecuzione delle istruzioni successive alla chiamata della **fork**.

Funzione fork

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

pid_t fork();
```

Restituisce:

0	al processo figlio;
pid del figlio	al processo padre;
-1	in caso di errore.

Esempio funzione fork

```
int pid; /* process identifier */

   pid=fork();
   if (pid < 0)
  {
            printf("Cannot fork!!\n");
            exit(1);
  }
   if (pid == 0)
            {/* child process */ ...}
   else
            {/* parent process pid is child's pid */ ...}
```

Sincronizzazione ed utilizzo della fork

- Tutti i file aperti del padre sono condivisi dai figli.
- Il padre aspetta che il figlio termini.
- La posizione nei file condivisi viene eventualmente aggiornata dal figlio ed il padre si adegua.
- Un processo che attende richieste si duplica, il figlio gestisce la richiesta, il padre ne attende di nuove.
- Un processo che vuole eseguire un nuovo programma si duplica per farlo eseguire al figlio.

Terminazione di un processo

- Terminazione normale:
  - Ritorno dal **main**;
  - Chiamata ad **exit**;
  - Chiamata ad **_exit**.

- Terminazione anormale:
  - Chiamata **abort**;
  - Arrivo di un **segnale**.

Funzione exit

#include <stdlib.h>

void **exit** (int status);

Restituisce la variabile status al processo che chiama il programma che utilizza la exit;
effettua una pulizia e poi ritorna al kernel (fclose, flush dell'output)

Funzione _exit

#include <unistd.h>

void **_exit** (int status);

Ritorna immediatamente al kernel

Funzione atexit

#include <stdlib.h>

int **atexit** (void ‘‘*funzione*’’ (void));

Restituisce:
0 in caso di successo.
!=0 in caso si sia verificato un errore

funzione = punta ad una funzione chiamata per effettuare operazioni di cleanup per il processo alla sua normale terminazione.

Terminazione di un processo

- Quando un processo termina il kernel manda al padre il segnale SIGCHLD
- Il padre può ignorare il segnale (default) oppure lanciare una funzione (signal handler)
- Inoltre, il padre può chiedere informazioni sullo stato di uscita del figlio chiamando le funzioni **wait** e **waitpid**.

Funzione wait

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

pid_t wait (int *statloc);
```

Descrizione: funzione chiamata da un processo padre, in statloc viene memorizzato lo stato di terminazione di un figlio.

Restituisce:
pid in caso di successo,
-1 in caso di errore.

Funzione waitpid

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

pid_t waitpid (pid_t pid, int *statloc, int options);
```

Descrizione: funzione chiamata da un processo padre, viene memorizzato lo stato di terminazione in statloc del figlio specificato dal *pid*;
Il processo padre si blocca in attesa o meno a seconda del contenuto di *options*.

Restituisce:
pid in caso di successo se OK,
0 oppure **-1** in caso di errore.

Argomenti di waitpid

- **PID**
  - pid > 0 (pid del figlio che si vuole aspettare)
  - pid == -1 (qualsiasi figlio...la rende simile a wait)
  - pid == 0 (figlio con GroupID uguale al padre)
  - pid < 0 (figlio con GroupID uguale a abs(pid))

- **OPTIONS**
  - 0 (niente... come wait)
  - WNOHANG (non blocca se il figlio indicato non è disponibile)

Differenze tra wait e waitpid

- Con la funzione wait il processo si blocca in attesa (se tutti i figli stanno girando), ritorna immediatamente con lo stato di un figlio o ritorna immediatamente con un errore (se non ha figli).
- Un processo può chiamare wait quando riceve SIGCHLD, ed in caso ritorna con lo stato del figlio appena terminato.
- Waitpid può scegliere quale figlio aspettare (tramite argomento).
- Infine, wait può bloccare il processo chiamante (se non ha figli che hanno terminato), mentre waitpid ha una opzione (WNOHANG) per non farlo bloccare e ritornare immediatamente.

Terminazione

- In ogni caso il kernel esegue il codice del processo e determina lo ***stato di terminazione:***
  - se normale, lo stato è l'argomento di exit, return oppure _exit.
  - altrimenti il kernel genera uno ***stato di terminazione*** che ne indica il motivo anomalo.
- In entrambi i casi il padre del processo ottiene questo stato da wait o waitpid

Variabile status

- Contiene:
  - Exit status in caso di terminazione normale (WIFEXITED(status) + WEXITSTATUS(status)).
  - Tipo di segnale che ha causato la terminazione in caso di uscita anomala (WIFSIGNALED(status) + WTERMSIG(status)).
  - Tipo di segnale che ha causato lo stop (WIFSTOPPED(status) + WSTOPSIG(status)).

Terminazione di un processo

- Un processo terminato, il cui padre non ha ancora invocato la wait(), diventa un processo zombie.
- Dopo la chiamata alla wait(), il pid del processo zombie e la relativa voce nella tabella dei processi vengono rilasciati.
- Se il padre termina senza invocare la wait(), il processo è orfano.
- Su Linux, init() diventa il padre e invoca periodicamente la wait() in modo da raccogliere lo stato di uscita del processo, rilasciandone il pid e la entry nella tabella dei processi.

Sincronizzazione

- Ogni volta che i processi tentano di fare qualcosa con dati condivisi, il risultato finale dipende dall'ordine in cui i questi ultimi vengono eseguiti.
- Per sincronizzarsi, un processo padre aspetta che un figlio termini usando una delle wait.
- Se un processo figlio vuole aspettare che il padre termini si può provare ad utilizzare i segnali (anche se il processo padre può mandare il segnale ancor prima che il figlio ha eseguito la signal, quindi il figlio viene raggiunto dal segnale ma viene applicata l'azione di default) oppure sfruttare la situazione in cui il processo figlio viene adottato da init quando il suo processo padre termina.

Funzioni exec

- Esegue un eseguibile o un comando

```
#include <unistd.h>

int execl (const char *path, const char *arg0, ../* (char *) 0 */);
int execv (const char *path, char *const argv[ ]);
int execle (const char *path, const char *arg0, ../*(char *) 0, char *const envp[ ] */);
int execve (const char *path, char *const argv[ ], char *const envp[ ]);
int execlp (const char *fle, const char *arg0, ../*(char *)0 */);
int execvp (const char *fle, char *const argv[ ]);

Restituiscono:
-1 in caso di errore
Niente in caso di successo.
```

Funzioni exec

- **execl**, **execlp**, **execle** prendono come parametro la lista degli argomenti da passare al *file* da eseguire.
- **execv**, **execvp**, **execve** prendono come parametro l'array di puntatori agli argomenti da passare al *file* da eseguire
- **execlp** ed **execvp** prendono come primo argomento un *file* e non un *pathname*, questo signifca che il file da eseguire viene cercato in una delle directory specificate nella path.
- **execle** ed **execve** passano al *file* da eseguire la enviroment list; un processo che invece chiama le altre exec copia la sua variabile environ per il nuovo *file* (programma).

Esempio exec

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

int main (void) {

        printf(“Sopra la panca \n”);
        execl(“/bin/echo”,”echo”,”la”,”capra“,”campa”,NULL);
        exit(0);

}
```

Funzione system

#include <stdlib.h>

int system (const char **cmdstring*);

Serve ad eseguire un comando shell dall'interno di un programma (esempio: system("date > fle");).
Implementata attraverso la chiamata di fork, exec e waitpid