Lezione 8: Processi(2)

8.1 Terminazione di processi in Unix

La terminazione di un processo consiste in una serie di operazioni che lasciano il sistema in stato coerente, chiusura di file aperti, rimozione dell'immagine dalla memoria ed eventuale segnalazione al processo padre.

Per gestire quest'ultimo aspetto unic impiega la funzione exit e la SC wait in modo coordinato.

Termina l'esecuzione di un processo exit e poi attesta della terminazione di un processo da parte del processo che lo ha creato.

8.2 Le chiamate di sistema wait e waitpid

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

La funzione **wait** sospende il processo invocante fincheè un figlio ha terminato la propria esecuzione oppure riceve un segnale.

La funzione **waitpid** sospende il processo invocante finchè il processo pid ha terminato la propria esecuzione oppure riceve un segnale.

Se il processo è uno zombie le funzioni ritornano subito, entrambe ritornanano con un errore -1 se il processo non ha figli altrimenti è restituito il pid del processo figlio terminato.

Il kerne notifica al genitore la terminazione di un processo figlio mediante il segnale SIGCHLD, le differenze tra le due funzioni sono:

- Se il processo invocante ha più di un figlio wait ritorna uno qualsiasi di essi ha terminato; waitpid permette di controllare quale figlio aspettare.
- wait blocca il processo chiamante fino a quando il figlio è terminato, mentre waitpid può non farlo.

Per entrambe le funzioni l'argomento è NULL, lo stato di terminazione è conservato nella locazione puntata dall'argomento.

Lo stato di terminazione può essere rilevato utilizzando le macro definite in <sys/wait.h>

8.3 Rilevazione dello stato

Se il byte meno significativo di statloc è 0 il byte più significativo rappresenta lo stato di terminazione, in caso contrario il meno significativo di statloc descrive il segnale che ha termianto il figlio.

8.4 Argomenti di waitpid

L'argomento pid di waitpid ha la seguente interpretazione:

- pid ==-1 attende un qualsiasi figlio (uguale a wait)
- pid > 0 attende il processo che ha il process ID uguale a pid
- pid == 0 attende un qualsiasi figlio il cui process group ID è uguale a quello del processo chiamante
- pid < -1 attende un qualsiasi figlio il cui process group ID è uguale a quello del valore assoluto di pid

L'argomento options di waitpid è 0, oppure una combinazione in OR delle costanti.

WNOHUNG non bloccherà il processo invocante il pi pid del figlio non è immediatamente disponibile

WUNTRACED ritorna lo stato di un figlio sospeso.

8.4 La chiamata exit

```
void exit(int status);
```

Termina il processo chiamante, rende disponibile il valore di status al processo padre che lo otterrà tramite wait.

```
#include “apue.h”
#include <sys/wait.h>
```

```
void pr_exit (int status)
{
    if (WIFEXITED(status))
        printf(“term. normale, exit status =%d\n”,WEXITSTATUS(st
    else if (WIFSIGNALED(status))
        printf(“term. anomala”, num. segnale =%d\n”,WTERMSIG(sta
    else if (WIFSTOPPED(status))
        printf(“figlio fermato, num. segnale %d\n”, WSTOPSIG(sta
}
```

8.5 Processi zombie

Un processo che termina non scompare dal sistema fino a che il padre non accetta il suo codice di terminazione.

Un processo che sta aspettando che il padre accetti il suo codice di terminazione è chiamato processo zombie.

Se il padre non temrina e non esegue main na wait(), il codice di terminazione non sarà mai accettato ed il processo resterà sempre uno zombie.

Uno zombie non ha area codice, dati o pila allocate quindi non usa molte risorse di sistema ma continua ad avere un PCB nella process table.

8.6 Processi adottati

Se un processo padre termina prima di un figlio questo processo viene detto orfano.

Il kernel assicura che tutti i processi organi siano adottati da init() ed assegna loro PPID 1

Cosa accede quando un processo adottato da init finisce?

Il processo adottato non diventa zombie poichè init è scritto in moo tale che se un qualsiasi processo figlio termina, venga chiamata una delle funzioni wait per determinare lo stato di uscita. init previene la proliferazione di zombie.

Osserviamo che per i processi di init intendiamo sia i processi generati direttamente da inti che quelli rimesti orfani e da eso adottati successivamente

8.7 Race conditions

Si verificano quando più processi cercano di operare con dati condivisi, il risultato finale dipende dall'ordine in cui i processi sono eseguiti.

In generale non è possibile predire quale processo venga eseguito per primo, anche se lo sapessimo ciò che accade dopo che il processo inizia l'esclusione dipende dal carico del sistema e dall'algoritmo di scheduling del kernel.

Per evitare la race condition è necessaria qualche forma di segnalazione tra i vari processi coinvolti o varie forme di IPC.

8.8 Famiglia di funzioni exec

Avere due processi che eseguono esattamente lo stesso codice non è molto utile, allora nella pratica accade che:

- si genera un secondo processo per affidargli l'esecuzione di un compito specifico
- Gli si fa eseguire un altro programma

per quest'ultimo caso si usa la terza funzione forndamentale per la programmazione con i processi che è la exec.

8.8.1 Le funzioni exec

Il programma che un processo sta eseguendo si chiama immagine del processo, le funzioni della famiglia exec permettono di caricare un altro programma dal

disco sostituendo quest'ultimo all'immagine corrente; l'immagine precedente viene completamente cancellata.

Quando il nuovo programma termina anche il processo termina e non si può tornare alla precedente immagine.

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

Quando un processo chiama una funzione della famiglia esso viene completamente sostiuito dal nuovo programma.

Il pid del processo non cambia dato che non viene creato un nuovo processo, la funzione rimpiazza semplicemente stack, heap, dati e testo del processo corrente con un nuovo programma letto dal disco.

La funzione **execve** esegue il file o lo script indicato da filename, passandogli la lista di argomenti indicata da argv e come ambiente la lista di stringhe indicata da envp. Entrambe le liste devono essere terminate da un puntatore nullo.

Le differenze delle funzioni della famiglia exec sono riassunte dai suffissi v ed l che stanno per vector e list.

Nel primo caso gli argomenti sono passati tramite vettore di puntatori argv a stringhe che terminano con 0.

Nel secondo caso le strinche degli argomenti sono passate alla funzione come lista di puntatori nella forma ..

La seconda differernza fra le funzioni rigurada le modalità con cui si specifica il programma che si vuole eseguire.

COn il suffisso p si indicano le due funzioni che replicano il comportamento della shell nello specificare il comando da eseguire, quando l'argomento file non continene / esso viene considerato come un nome di programma e viene eseguita automaticamente una ricerca fra i file presenti.

Le altre 4 funzioni si limitano invece a cercare di eseguire il file indicato dall'argomento path, che viene interpetrato come il pathname del programma.

La terza differenza è come viene passata la lista delle variabili di ambiente, con il suffisso e vengono indicate quelle funzioni che necessitano di un vattore di parametri envp analogo a quello usato per gli argomenti a riga di comando.

Le atre usano il valore della variabile environ del processo di partenza per costuire l'ambiente.