

Laboratorio di Sistemi Operativi

Marzo-Giugno 2008
matricole congrue 0 mod 3

Controllo dei processi (2)

Funzioni **wait** e **waitpid**

- ▶ 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)
- ▶ in ogni caso il padre può chiedere informazioni sullo stato di uscita del figlio; questo è fatto chiamando le funzioni **wait** e **waitpid**.

Laboratorio di Sistemi Operativi

2

Funzione **wait**

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

```
pid_t wait (int *statloc);
```

Descrizione: chiamata da un processo padre ottiene in *statloc* lo stato di terminazione di un figlio

Restituisce: PID se OK,
-1 in caso di errore

Laboratorio di Sistemi Operativi

3

Funzione **waitpid**

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

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

Descrizione: chiamata da un processo padre chiede lo stato di terminazione in *statloc* del figlio specificato dal *pid* 1° argomento; tale processo padre si blocca in attesa o meno secondo il contenuto di *options*

Restituisce: PID se OK,
0 oppure -1 in caso di errore

Laboratorio di Sistemi Operativi

4

Funzione **waitpid**

- ▶ $pid == -1$ (qualsiasi figlio...la rende simile a **wait**)
 - ▶ $pid > 0$ (pid del figlio che si vuole aspettare)
 - ▶ $pid < -1$ (qualsiasi processo con $grp\ id = |pid|$)
- ▶ **options** =
- ▶ 0 (niente... come **wait**)
 - ▶ WNOHANG (non blocca se process pid non è disponibile)
 - ▶ WUNTRACED (status di un figlio *stopped*)
 - ▶ WCONTINUED (status di un figlio *continued*)

Laboratorio di Sistemi Operativi

5

differenze

- ▶ in generale con la funzione **wait**
 - ▶ il processo si blocca in attesa (se tutti i figli stanno girando)
 - ▶ ritorna immediatamente con lo stato di un figlio
 - ▶ ritorna immediatamente con un errore (se non ha figli)
- ▶ un processo può chiamare **wait** quando riceve SIGCHLD, in questo caso ritorna immediatamente con lo stato del figlio appena terminato
- ▶ **waitpid** può scegliere quale figlio aspettare (1° argomento)
- ▶ **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

Laboratorio di Sistemi Operativi

6

Terminazione di un processo

- ▶ Terminazione normale
 - ▶ ritorno dal **main**
 - ▶ chiamata a **exit**
 - ▶ chiamata a **_exit**
- ▶ Terminazione anormale
 - ▶ chiamata **abort**
 - ▶ arrivo di un segnale
 - interrupt generati da altri processi o dal kernel (p.e. quando si divide per zero)

Laboratorio di Sistemi Operativi

7

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 indica il motivo "anormale"
- ▶ in entrambi i casi il padre del processo ottiene questo stato da **wait** o **waitpid**

Laboratorio di Sistemi Operativi

8

Termination status

- » `Waitpid(pid, &status, options)`
- » `WIFEXITED(status)`
 - True se status fa riferimento ad un figlio terminato regolarmente
 - `WEXITSTATUS(status) -> exit/_exit value`
- » `WIFSTOPPED(status)`
 - True se status fa riferimento ad un figlio *stopped*
- » `WIFCONTINUED(status)`
 - True se status fa riferimento ad un figlio ripartito dopo essere stato in stato di *stopped*
- » `WIFSIGNALED(status)`
 - True se status fa riferimento ad un figlio terminato in maniera anormale in seguito ad un segnale

Laboratorio di Sistemi Operativi

9

Cosa succede quando un processo termina?

- » Se un figlio termina prima del padre, allora il padre ottiene lo status del figlio con **wait**
- » Se un padre termina prima del figlio, allora il processo **init** diventa il nuovo padre
- » Se un figlio termina prima del padre, ma il padre non utilizza **wait**, allora il figlio diventa uno **zombie**

Laboratorio di Sistemi Operativi

10

esempio: terminazione normale

```
pid_t pid;
int status;

pid=fork();
if (!pid) /* figlio */
    exit(128); /* qualsiasi numero */
if (wait(&status) == pid)
    printf("terminazione normale\n");
```

Laboratorio di Sistemi Operativi

11

esempio: terminazione con abort

```
pid_t pid;
int status;

pid = fork();
if (!pid) /* figlio */
    abort(); /* genera il segnale SIGABRT */
if (wait(&status) == pid)
    printf("terminazione anormale con abort\n");
```

Laboratorio di Sistemi Operativi

12

zombie.c

```
int main()
{ pid_t pid;

  if ((pid=fork()) < 0)
    fprintf(stderr, "fork error");
  else if (!pid) { /* figlio */
    printf("pid figlio= %d",getpid());
    exit(0);
  }
  sleep(2); /* padre */
  system("ps T"); /* dice che il figlio è
                  zombie... STAT Z*/
  exit(0);
}
```

Laboratorio di Sistemi Operativi

13

Race Conditions

» ogni volta che dei processi tentano di fare qualcosa con dati condivisi e il risultato finale dipende dall'ordine in cui i processi sono eseguiti sorgono delle **race conditions**

☞ se si vuole che un figlio aspetti che il padre termini si può usare:

```
while ( getppid() != 1)
  sleep(1);
```

☞ se un processo vuole aspettare che un figlio termini deve usare una delle **wait**

Laboratorio di Sistemi Operativi

14

evitare le Race Conditions

» la prima soluzione spreca molta CPU, per evitare ciò si devono usare i segnali oppure qualche forma di IPC (interprocess communication).

Laboratorio di Sistemi Operativi

15

esempio di race conditions

```
int main(void) {
  pid_t pid;

  pid = fork();
  if (!pid) {charatime("output dal figlio\n"); }
  else { charatime("output dal padre\n"); }
  exit(0);
}

static void charatime(char *str)
{char *ptr;
 int c;
 setbuf(stdout, NULL); /* set unbuffered */
 for (ptr = str; c = *ptr++; )
  putc(c, stdout);
}
```

Laboratorio di Sistemi Operativi

16

output dell'esempio

```
/home/studente > a.out
output from child
output from parent

/home/studente > a.out
oouuttppuutt ffrroomm cphairlednt
...
```

Laboratorio di Sistemi Operativi

17

figlio esegue prima del padre

```
TELL_WAIT();          /* inizializzazione */
pid = fork();
if (!pid){             /* figlio */
    /* il figlio fa quello che deve fare */
    TELL_PARENT( getpid() );
} else{                /* padre */
    WAIT_CHILD();
    /* il padre fa quello che deve fare */
}
```

Laboratorio di Sistemi Operativi

18

figlio esegue prima del padre (esempio)

```
pid = fork();
if (!pid){             /* figlio */
    /* il figlio fa quello che deve fare */
}
else{                 /* padre */
    wait();
    /* il padre fa quello che deve fare */
}
```

Laboratorio di Sistemi Operativi

19

padre esegue prima del figlio

```
TELL_WAIT();          /* inizializzazione */
pid = fork();
if (!pid){             /* figlio */
    WAIT_PARENT();
    /* il figlio fa quello che deve fare */
} else{                /* padre */
    /* il padre fa quello che deve fare */
    TELL_CHILD(pid);
}
```

Laboratorio di Sistemi Operativi

20

padre esegue prima del figlio (esempio)

```
#include <signal.h>
void catch(int);
int main (void) {
    pid = fork();
    if (!pid) { /* figlio */
        signal(SIGALRM, catch);
        pause();
        /* il figlio fa quello che deve fare */
    } else { /* padre */
        /* il padre fa quello che deve fare */
        kill(pid, SIGALRM);
    }

    void catch(int signo) {
        printf("parte il figlio");
    }
}
```

Laboratorio di Sistemi Operativi

21

primitive di controllo

- 👉 `fork` → creazione nuovi processi
- 👉 `exec` → esecuzione nuovi programmi
- 👉 `exit` → trattamento fine processo
- 👉 `wait/waitpid` → trattamento attesa fine processo

Laboratorio di Sistemi Operativi

22

Funzioni `exec`

- ▶▶ `fork` di solito è usata per creare un nuovo processo (il figlio) che a sua volta esegue un programma chiamando la funzione `exec`.
- ▶▶ in questo caso il figlio è completamente rimpiazzato dal nuovo programma e questo inizia l'esecuzione con la sua funzione `main`
 - ▶ non è cambiato il pid... l'address space è sostituito da un nuovo programma che risiedeva sul disco

Laboratorio di Sistemi Operativi

23

Funzioni `exec`

- ▶▶ L'unico modo per creare un processo è attraverso la `fork`
- ▶▶ L'unico modo per eseguire un programma è attraverso la `exec`
- ▶▶ La chiamata ad `exec` reinizializza un processo: il segmento istruzioni ed il segmento dati utente cambiano (viene eseguito un nuovo programma) mentre il segmento dati di sistema rimane invariato

Laboratorio di Sistemi Operativi

24

Funzioni exec

```
#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[ ]);

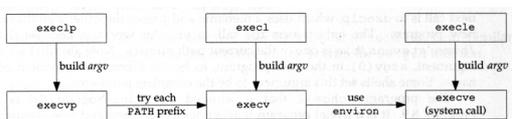
int execlp (const char *file, const char *arg0, ../*(char *)0 */);
int execvp (const char *file, char *const argv[ ]);

Restituiscono: -1 in caso di errore
non ritornano se OK.
```

Funzioni exec - differenze

- ▶ Nel nome delle exec **l** sta per list mentre **v** sta per vector
 - ▶ **execl**, **execlp**, **execl**e prendono come parametro la lista degli argomenti da passare al *file* da eseguire
 - ▶ **execv**, **execvp**, **execv**e 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 significa che il file da eseguire e' ricercato in una delle directory specificate in PATH
- ▶ **execl**e ed **execv**e 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)

Relazione tra le funzioni exec



Caratteristiche delle exec

Funzioni Args Env PATH

execl	lista	auto	no
execv	vettore	auto	no
execl e	lista	man	no
execv e	vettore	man	no
execl p	lista	auto	si
execv p	vettore	auto	si

```
.....
.....
printf("Sopra la panca \n");
execl("/bin/echo", "echo", "la", "capra", "campa", NULL);
.....
```

```
$a.out
Sopra la panca
la capra campa
$
$a.out>temp.txt
$cat temp.txt
la capra campa
```

esempio: echoenv.c

```
#include <stdlib.h>
extern **char environ;

int main() {
    int i = 0;

    while (environ[i])
        printf("%s\n", environ[i++]);

    return (0);
}
```

esempio di execl[ep]

```
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"
char *env_init[]={"USER=studente", "PATH=/tmp", NULL };
int main(void) {
    pid_t pid;
    pid = fork();
    if (!pid) { /* figlio */
        execl("/home/studente/echoenv", "echoenv", (char*) 0, env_init);
    }
    waitpid(pid, NULL, 0);
    printf("sono qui \n\n");
    pid = fork();
    if (pid == 0) { /* specify filename, inherit environment */
        execlp("echoenv", "echoenv", (char *) 0);
    }
    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 > file");
- ▶▶ essa e' implementata attraverso la chiamata di fork, exec e waitpid

Esercizio 1

- ❏ Scrivere un programma che crei un processo zombie.
- ❏ Fare in modo che un processo figlio diventi figlio del processo *init*.

Laboratorio di Sistemi Operativi

33

Esercizio 2

Sia ELENCO.TXT un file contenente dei record del tipo: **Cognome** \t **Nome**.

Scrivere un programma che utilizzi fork + exec per eseguire il programma ORDINA che creerà il file contenente i record in ordine alfabetico crescente. Tale programma avrà come parametri il nome del file da ordinare e il file da creare con i nomi ordinati.

Laboratorio di Sistemi Operativi

34

Esercizio 3

Scrivere un programma che effettui la copia di un file utilizzando 2 figli:

- ▶▶ uno specializzato nella copia delle vocali ed
- ▶▶ uno nella copia delle consonanti.

Per la sincronizzazione tra i processi figli utilizzare un semaforo implementato tramite file.

Laboratorio di Sistemi Operativi

35