Controllo dei processi in UNIX

Capitolo 8 -- Stevens

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.

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

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

Funzione waitpid

- *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

- 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 puo’ 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

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

Variabile *status*

La variabile ***status*** contiene

Exit status negli 8 bit meno significativi se la terminazione è **normale**

WIFEXITED(status) + WEXITSTATUS(status)

Tipo di segnale che ha causato la terminazione in caso di uscita **anormale**

WIFSIGNALED(status) + WTERMSIG(status)

Tipo di segnale che ha causato lo **stop**

WIFSTOPPED(status) + WSTOPSIG(status)

Cosa succede quando un processo termina?

- Un processo terminato, il cui padre non ha ancora invocato la wait(), è uno 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 un orfano
- In Linux, init() diventa il padre e invoca periodicamente la wait() in modo da raccogliere lo stato di uscita del processo, rilasciando il suo pid e la sua entry nella tabella dei processi

esempio: terminazione normale

```
pid_t pid;

int status;



  pid=fork();

  if (pid==0)  /* figlio */

    exit(128);   /* qualsiasi numero */

  if (wait(&status) == pid)

     printf("terminazione normale\n: %d”,
   status);
```

vedi fig. 8.2 per le macro per la verifica di status e per stampare lo stato di terminazione

esempio: terminazione con abort

```
pid_t pid;

int status;


  pid = fork();

 if (pid==0)   /* figlio  */

     abort();   /* genera il segnale SIGABRT */

 if (wait(&status) == pid)

    printf("terminazione anormale con abort\n: %d”status);
```

zombie.c

```
int main()
{ pid_t pid;


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

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

esempio di race conditions

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

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

}

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

}
```

output dell'esempio

```
/home/studente > a.out

output from child

output from parent



/home/studente > a.out

oouuttppuutt ffrroomm cphairlednt

...
```

Race Conditions

Volendo sincronizzare un processo padre ed un processo figlio possiamo tentare di utilizzare strumenti che abbiamo già introdotto

Se un processo padre vuole aspettare che un figlio termini deve usare una delle wait

figlio esegue prima del padre

```
pid = fork();

if (!pid){                          /* figlio  */

   /* il figlio fa quello che deve fare */

      }

else{   /* padre */

   wait();

        /* il padre fa quello che deve fare */

}
```

Esercizio

- Scrivere un programma C in cui un processo genera un processo figlio
  - il figlio scrive sullo standard output il proprio pid
  - Successivamente, il padre scrive il proprio pid

Race Conditions

Volendo sincronizzare un processo padre ed un processo figlio possiamo tentare di utilizzare strumenti che abbiamo già introdotto

Se un processo padre vuole aspettare che un figlio termini deve usare una delle wait

Se un processo figlio vuole aspettare che il padre termini si può

- Tentare di utilizzare i segnali

padre esegue prima del figlio

```
#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”);
}
```

Race Conditions

Volendo sincronizzare un processo padre ed un processo figlio possiamo tentare di utilizzare strumenti che abbiamo già introdotto

Se un processo padre vuole aspettare che un figlio termini deve usare una delle wait

Se un processo figlio vuole aspettare che il padre termini si può

- Tentare di utilizzare i segnali
  - Difetti: il processo padre può mandare il segnale ancor prima che il figlio ha eseguito la signal, quindi il figlio è raggiunto dal segnale ma viene applicata l'azione di default
- Usare il fatto che il processo figlio viene adottato da init quando il suo padre naturale muore

padre esegue prima del figlio

```
#include <signal.h>

void catch(int);

int main (void) {

pid = fork();

if (!pid){      /* figlio  */
       while ( getppid() != 1) ;

      /* il figlio fa quello che deve fare */

} else{       /* padre */

     /* il padre fa quello che deve fare */


}
```

padre esegue prima del figlio

```
#include <signal.h>

void catch(int);

int main (void) {

aaa=getpid();

pid = fork();

if (!pid){      /* figlio  */
       while ( getppid() = aaa) ;

      /* il figlio fa quello che deve fare */

} else{       /* padre */

      /* il padre fa quello che deve fare */


}
```

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

Funzioni exec

- L’unico modo per creare un processo è attraverso la fork
- L’unico modo per eseguire un eseguibile (o comando) 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

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[ ]);
```

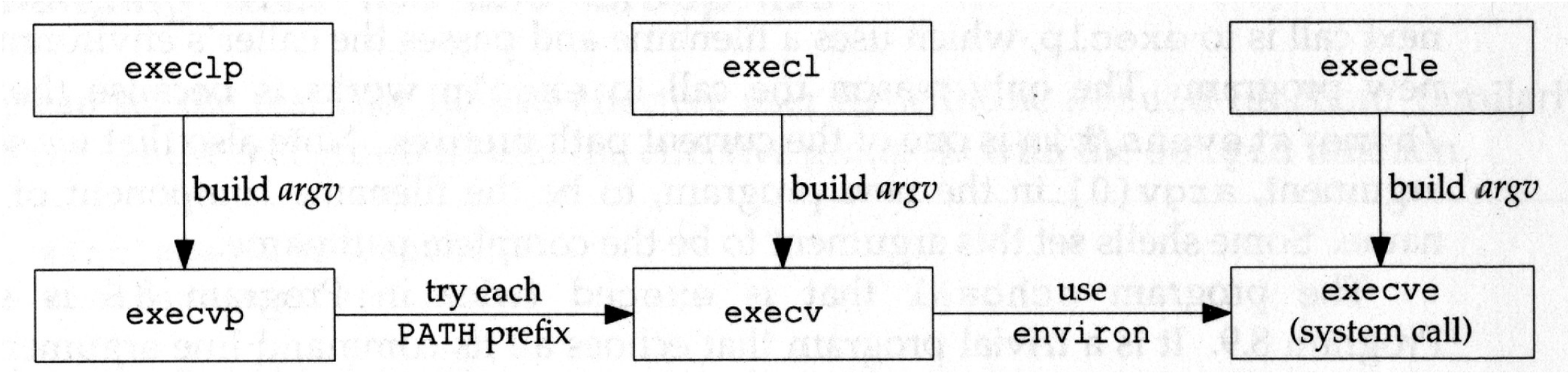
Restituiscono: -1 in caso di errore

non ritornano se OK.

Funzioni exec - *differenze*

- Nel nome delle exec l sta per list mentre v sta per vector
  - execl, execlp,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 significa che il file da eseguire e' ricercato in una delle directory specificate in 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)

Relazione tra le funzioni exec



primitive di controllo

- con la exec è chiuso il ciclo delle primitive di controllo dei processi UNIX

fork → creazione nuovi processi

exec → esecuzione nuovi programmi

exit → trattamento fine processo

wait/waitpid→trattamento attesa fine processo

```
#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);
}
```

```
$a.out
Sopra la panca
la capra campa
$
```

```
#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);

printf(“sotto la panca crepa \n”);

exit(0);

}
```

```
$a.out
Sopra la panca
la capra campa
$
```

```
#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);

execl(“/bin/echo”,”echo”,“sotto”,”la”,”panca”,”crepa”, NULL);

exit(0);

}
```

```
$a.out
Sopra la panca
la capra campa
$
```

fork + exec

La chiamata ad una funzione exec deve essere delegata ad un processo figlio

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


int main (void) {
printf("Sopra la panca \n");
pid = fork();
if (pid==0){
   execl("/bin/echo","echo","la","capra","campa",NULL);}
printf("sotto la panca crepa \n");
exit(0);
}
```

perché?

```
$a.out
Sopra la panca
la capra campa
sotto la panca crepa
$
```

```
$a.out
Sopra la panca
sotto la panca crepa
la capra campa
$
```

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


int main (void) {
printf(“Sopra la panca \n”);
pid = fork();
if (!pid){
   execl(“/bin/echo”,”echo”,”la”,”capra“,”campa”,NULL);}
wait( );
printf(“sotto la panca crepa \n”);
exit(0);
}
```

```
$a.out
Sopra la panca
la capra campa
sotto la panca crepa
$
```

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>
char    *env_init[ ]={"USER=studente", "PATH=/tmp", NULL };
int main(void){
  pid_t pid;
  pid = fork();
  if (!pid) {   /* figlio */
   execle("/home/studente/echoenv","echoenv",(char*) 0,env_init);
   }
   waitpid(pid, NULL, 0);
   printf("sono qui \n\n\n");
   pid = fork();
   if (pid == 0){/* specify filename, inherit environment */
       execl("/home/studente/echoenv", "echoenv", (char *) 0);
   }
   exit(0);
}
```

Start-up di un programma

L’esecuzione di un programma tramite fork+exec ha le seguenti caratteristiche

- Se un segnale è **ignorato** nel processo padre viene ignorato anche nel processo figlio
- Se un segnale è **catturato** nel processo padre viene assegnata l’azione di default nel processo figlio

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 è implementata attraverso la chiamata di fork, exec e waitpid

Esercizio 7.1

Scrivere un programma che crei un processo zombie.

Fare in modo che un processo figlio diventi figlio del processo ***init***.

Esercizio 7.2

Dire cosa fa il codice seguente

```
void main()
{ int j, ret;
     j = 10;
    ret = fork();
   printf("Child created \n");
    j = j * 2;
   if (ret == 0) {
            j = j * 2;
           printf("The value of j is %d \n", j);}
   else {
         ret = fork();
          j = j * 3;
          printf("The value of j is %d \n", j);}
  printf("Done \n");
}
```

Esercizio 7.3

Si assuma di avere nella cwd un file di nome File1

Scrivere un codice C che:

- scriva su standard output la parola “ordina”
- mandi in esecuzione il comando sort File1
- scriva su standard output la parola “fine”

Esercizio 7.4

La successione di Fibonacci (0,1,1,2,3,5,8,…) è definita ricorsivamente da

- $f_0 = 0$
- $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$

- Scrivere un programma C, che usi la chiamata di sistema fork(), per generare la successione di Fibonacci all'interno del processo figlio.
- Il processo figlio produrrà anche le relative stampe.
- Il padre dovrà rimanere in attesa tramite wait() fino alla terminazione del figlio.
- Il numero di termini da generare sarà specificato a riga di comando. Implementare i necessari controlli per garantire che il valore in ingresso sia un numero intero non negativo.