

Processi

Introduzione

Come possiamo creare nuovi processi?

Come possiamo eseguire un programma?

Come possiamo terminare un processo?

Introduzione

Unix fornisce le primitive per la creazione e terminazione di processi, l'esecuzione di programmi:

- **fork** crea nuovi processi;
- **exec** attiva nuovi programmi;
- **exit** e **wait** termina e attende la terminazione di un processo.

Con queste funzioni si possono costruire altre funzioni di controllo.

Funzione fork

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

La funzione viene chiamata una volta, ma ritorna due:

- nel processo genitore restituisce l'identificativo del figlio,
- nel processo figlio restituisce 0.

Il figlio è una copia del genitore (*data space, heap e stack*), cioè essi non condividono parti di memoria.

Esempio 1

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

int main ()
{
    pid_t pid;

    if((pid=fork())==0)    /* sono il figlio */
        printf("Ciao, sono il figlio\n");
    else                  /* sono il padre */
        printf("Ciao, sono il padre\n");

    exit(0);
}
```

Identificativi di processo

Ciascun processo è identificato da un intero non negativo.

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

pid_t getpid(void);      identificativo di processo
pid_t getppid(void);   identificativo del padre
```

Queste funzioni ritornano gli identificativi di processo.

```
# include <sys/types.h>
# include "ourhdr.h"
```

```
int    glob = 6;                /* external variable */
```

```
int main(void) {
    int    var;                /* automatic variable */
    pid_t  pid;
```

```
    var = 88;
```

```
    if ( (pid = fork()) < 0) err_sys("fork error");
    else if (pid == 0) {        /* child */
        glob++;                /* modify variables */
        var++;
    } else sleep(2);           /* parent */
```

```
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
```

```
}
```

Esempio 2 (cont.)

Eseguendo l'esempio, si ottiene:

```
[mariog]$ a.out
```

```
pid = 686, glob = 7, var = 89
```

```
pid = 685, glob = 6, var = 88
```

```
[mariog]$
```

*Eseguito un **fork**, non si può sapere se il figlio va in esecuzione prima o dopo del genitore*

Funzione vfork

```
# include <unistd.h>
pid_t vfork(void);
```

vfork crea un nuovo processo, esattamente come **fork**, ma non copia lo spazio di indirizzamento.

Fino a che il figlio non esegue **exec** o **exit**, viene eseguito nello spazio di indirizzamento del genitore.

vfork assicura che il figlio venga eseguito per primo, fino a quando questi non chiama **exec** o **exit**.

vfork viene utilizzata quando il processo generato ha lo scopo di eseguire (**exec**) un nuovo programma.

```
# include <sys/types.h>
# include "ourhdr.h"
```

```
int glob = 6;                /* external variable */
```

```
int main(void) {
    int var;                 /* automatic variable */
    pid_t pid;
```

```
    var = 88;
```

```
    if ( (pid = vfork()) < 0) err_sys("vfork error");
    else if (pid == 0) {     /* child */
        glob++;             /* modify parent's variables */
        var++;
        exit(0);           /* child terminates */
    }
```

```
    /* parent */
```

```
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
```

```
}
```

Esempio 3 (cont.)

Eseguendo l'esempio, si ottiene:

```
[mariog]$ a.out  
before vfork  
pid = 2624, glob = 7, var = 89  
[mariog]$
```

L'incremento della variabile nel figlio cambia il valore della variabile nel genitore.

Funzione exit

Un processo termina normalmente se:

1. esegue **return** dal main (stesso di **exit**);
2. chiama **exit**, richiamando le eventuali funzioni registrate da **atexit(3)** e chiude gli stream di I/O;
3. chiama **_exit**, che termina il processo chiudendo tutti i descrittori dei file aperti.

Un processo non termina normalmente se:

1. chiama **abort**, generando un segnale di **SIGABRT**;
2. il processo riceve determinati segnali.

In ogni caso, il kernel chiude tutti i descrittori aperti dal processo e rilascia la memoria.

Funzione `exit` (cont.)

Un figlio che termina comunica al genitore il suo stato di terminazione

Se ha eseguito `exit` o `_exit`, esso passa uno stato di uscita al genitore mediante l'argomento di queste due funzioni.

Nel caso di terminazione non normale, il kernel genera uno stato di terminazione per indicare la ragione.

Funzione `exit` (cont.)

Il genitore è sempre in grado di ottenere lo stato di terminazione di un figlio mediante le funzioni **`wait`** e **`waitpid`**.

Se il genitore termina prima del figlio, il processo **`init`** eredita il figlio e il parent process ID di questo diventa 1.

Se il figlio termina prima che il genitore sia in grado di controllare la sua terminazione, il kernel conserva almeno il process ID e lo stato di terminazione.

Tali processi vengono detti zombie.

Funzioni `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 finché:

- un figlio ha terminato la propria esecuzione, oppure
- riceve un segnale

La funzione **waitpid** sospende il processo finché:

- il processo `pid` ha terminato la propria esecuzione, oppure
- riceve un segnale

Se il processo è uno zombie, le funzioni ritornano subito.

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

int main ()
{
    pid_t pid;
    int status;

    if( (pid=fork()) == 0 )    /* sono il figlio */
        printf("Ciao, sono il figlio\n");
    else                      /* sono il padre */
        pid = wait(&status);

    exit(0);
}
```

Funzioni wait e waitpid (cont.)

Un processo che chiama **wait** o **waitpid** può

- bloccarsi se i figli sono in esecuzione, o
- ritornare subito con lo stato di terminazione di un figlio, o
- ritornare con un errore, se il processo non ha figli.

Funzioni wait e waitpid (cont.)

Il kernel notifica al genitore la terminazione di un processo figlio mediante il segnale SIGCHLD.

Le differenze tra le due funzioni sono:

- se il processo che chiama ha più di un figlio, **wait** ritorna quando uno di questi ha terminato, mentre **waitpid** permette di controllare quale figlio aspettare;
- **wait** blocca il processo chiamante fino a quando il figlio non è terminato, mentre **waitpid** può non farlo;

Funzioni wait e waitpid (cont.)

Per entrambe le funzioni l'argomento *statloc* é un puntatore ad un intero.

Se l'argomento non 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>`.

Funzioni wait e waitpid (cont.)

Macro

WIFEXITED(*status*)

Descrizione

vero se il figlio è terminato
normalmente

WEXITSTATUS(*status*) ritorna lo stato

WIFSIGNALED(*status*)

vero se il figlio è uscito per un
segnale

WTERMSIG(*status*) ritorna il segnale

WIFSTOPPED(*status*)

vero se il figlio è fermato

WSTOPSIG(*status*) ritorna il segnale

*Queste macro hanno per argomento
un intero, non un puntatore!*

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

int main ()
{
    pid_t pid;
    int status;

    if( (pid=fork()) == 0 )    /* sono il figlio */
        printf("Ciao, sono il figlio\n");
        exit(32);
    else                      /* sono il padre */
        pid = waitpid(pid, &status, 0);
        printf("Stato: %d \n", WEXITSTATUS(status));
        exit(0);
}
```

Funzioni wait e waitpid (cont.)

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

Funzioni wait e waitpid (cont.)

L'argomento *options* di **waitpid** o è 0, o una combinazione con OR delle costanti:

WNOHUNG	ritorna immediatamente con 0 se il figlio non è terminato
WUNTRACED	ritorna lo stato di un figlio sospeso

Esempio 5

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

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

```
    if ( (pid = fork()) < 0)  err_sys("fork error");
    else if (pid == 0) {      /* first child */
        if ( (pid = fork()) < 0)  err_sys("fork error");
        else if (pid > 0)
            exit(0);          /* parent from 2nd fork == first child */
```

```
    /* We're the second child; our parent becomes init as soon
       as our real parent calls exit() in the statement above.
       Here's where we'd continue executing, knowing that
       when we're done, init will reap our status. */
```

```
    sleep(2);
```

```
    printf("second child, parent pid = %d\n", getppid());
```

```
    exit(0);    /* parent from 2nd fork == first child */
```

/* We're the second child; our parent becomes init as soon as our real parent calls exit() in the statement above. Here's where we'd continue executing, knowing that when we're done, init will reap our status. */

```
    sleep(2);  
    printf("second child, parent pid = %d\n", getppid());  
    exit(0);  
}
```

```
if (waitpid(pid, NULL, 0) != pid)    /* wait for first child */  
    err_sys("waitpid error");
```

/* We're the original process; we continue executing, knowing that we're not the parent of the second child. */

```
exit(0);  
}
```

Eseguendo l'esempio, si ottiene:

```
[mariog]$ a.out
```

```
[mariog]$ second child, parent pid = 1
```

In questo modo è possibile non attendere la fine di un processo figlio ed evitare che questo diventi uno zombie.

Eseguendo un **fork** su **fork** otteniamo che il secondo è ereditato da `init`.

Se non ci fosse una **sleep**, e se avesse iniziato la sua esecuzione dopo il **fork** prima del padre, il **ppid** stampato sarebbe stato quello del padre, non 1.

Funzioni exec

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

int execl ( const char *path, const char *arg0, ... /* (char *) 0 */);
int execv ( const char *path, char *const argv[ ]);
int execl ( 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[ ]);
```

Le funzioni di **exec** sostituiscono il processo corrente con un nuovo programma, che inizia l'esecuzione del suo main.

L'identificativo del nuovo processo è lo stesso di quello sostituito.

Funzioni exec (cont.)

Le funzioni **execl**, **execlv**, **execle** e **execve** hanno come primo argomento un path. Le ultime due il nome di un file.

Le funzioni **execl**, **execle** e **execvp** richiedono che gli argomenti del nuovo programma siano specificati separatamente (l sta per list) e che la fine sia marcata da un puntatore nullo.

Le funzioni **execv**, **execvp** e **execve** (v sta per vector) richiedono che venga costruito un array di puntatori agli argomenti, e l'indirizzo di questo array è l'argomento di queste funzioni.

Funzioni exec (cont.)

Le funzioni **execle** e **execve** (e sta per environment) permettono di passare come argomento un puntatore ad un array di puntatori a stringhe d'ambiente. Le altre quattro funzioni utilizzano la variabile.

`environ` nel processo chiamante per copiare l'ambiente esistente nel nuovo programma.

Le funzioni **execvp** e **execvp** (p sta per path) cercano l'eseguibile file nella lista di directory specificate dalla variabile d'ambiente `PATH`.

Esempio 6

```
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"
int main(void) {
    char buf[MAXLINE];
    pid_t pid;
    int status;

    printf("%% "); /* print prompt */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ( (pid = fork()) < 0)
            err_sys("fork error");

        else if (pid == 0) { /* child */
            execl(buf, buf, (char *) 0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        } /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```