Segnali

I processi possono talvolta avere la necessità di gestire situazioni inaspettate.

Per esempio:

- un errore di floating point
- un segnale di "allarme" prodotto da un altro processo
- la morte di un processo figlio
- una richiesta di terminazione da parte dell'utene (ctrl-C)

Questi eventi sono chiamati **interruzioni**, perché interrompono il flusso regolare del programma.

Quando Unix riconosce che un evento del genere si è verificato, invia al processo interessato un segnale ("signal"). Il nucleo, comunque, non è l'unico che può inviare segnali, anche processi di utente possono farlo, purché abbiamo l'autorizzazione.

Esiste un unico segnale per ogni tipo di evento, e a ciascuno di questi è associato un numero identificativo (nell'Unix BSD, per esempio ci sono 31 segnali diversi).

Ad ogni segnale è inoltre assegnato una routine ("signal handler") che si occupa, per default, di gestire il segnale

Segnali predefiniti

Per ogni tipo di segnale, il programmatore può scegliere di far eseguire un particolare signal-handler definito dal programmatore stesso, di far attivare il signal-handler di default, o di far ignorare il segnale.

Le routine per la gestione dei segnali di default, in genere possono realizzano una delle seguenti operazioni:

- interrompono il processo e generano un file chiamato "core" (dunp)
- interrompono il processo senza generare il file core (quit)
- ignorano il segnale (ignore)
- sospendono il processo (suspend)

Di seguito sono riportati alcuni dei segnali predefiniti:

Nome	*Numero*	*Core*	*Commento*
SIGINT	2	quit	interrupt da terminale
SIGILL	4	dump	istruzione illegale
SIGFPE	8	dump	errore floating point
SIGBUS	10	dump	errore sul bus
SIGSEGV	11	dump	violazione di segmento
SIGPIPE	13	quit	scrittura su una pipe che nessuno può leggere
SIGALRM	14	quit	allarme di "clock"
SIGCHLD	20	ignore	il processo figlio ha cambiato stato
SIGXCPU	24	quit	tempo di CPU ecceduto

La primitiva `alarm`

La primitiva di sistema `alarm` fa sì che il nucleo invii un segnale SIGALRM al processo chiamante dopo un determinato numero di secondi.

La sua sintassi è la seguente:

```
alarm( unsigned int count )
```

dove il parametro `count` specifica il numero di secondi.

L'azione di default intrapresa dal signal-handler è quella di visualizzare "Alarm clock" e di interrompere il processo.

Esempio:

```
$ more alarm.c          ...listo il programma
#include <stdio.h>
#include <signal.h>

main()
{
  alarm(3)
  printf("Ciclo infinito...\n");
  while(1);
  printf("Questo non dovrebbe apparire\n");
}

$ alarm.exe             ...eseguo il programma
Ciclo infinito...
Alarm clock             ...dopo 3 secondi
$ _
```

Gestione dei segnali: la primitiva **`signal`**

Dato un particolare segnale, la primitiva di sistema `signal` consente di specificare un signal-handler diverso da quello di default.

La sua sintassi è:

```
void (*signal( int sigCode, void (*func)() )) ()
```

Il parametro `sigCode` specifica il numero del segnale che deve essere "riprogrammato".

La funzione restituisce il valore precedente di `func`, se termina con successo; -1 altrimenti.

Il secondo parametro può essere:

- SIG_IGN: indica che il segnale deve essere ignorato
- SIG_DFL: ripristina il signal-handler di default
- l'indirizzo di una funzione definita dall'utente che deve essere invocata all'arrivo del segnale

Alcuni segnali (come SIGKILL e SIGTSTP) non possono essere riprogrammati.

La primitiva `pause` sospende il processo chiamante fino a quando non riceve un segnale.

La sua sintassi è:

```
pause()
```

Essa è spesso usata insieme alla primitiva `signal`.

La primitiva `signal`:   Esempio

```
$ more handler.c           ...lista il programma
#include <stdio.h>
#include <signal.h>

int alarmFlag = 0;
alarmHandler();

main() {
  signal( SIGALRM, alarmHandler);
  alarm(3);
  printf("Loop...\n");
  while( !alarmFlag )  {
    pause();
    }
  printf("Il loop termina...\n");
}

alarmHandler()  {
  printf("Ricevuto un allarme di clock\n");
  alarmFlag=1;
}

$ handler.exe              ...esegue il programma
Loop...
Ricevuto un allarme di clock        ...dopo 3 sec.
Il loop termina...
$ _
```

Protezione di codice critico

La stessa tecnica vista in precedenza può essere impiegata per proteggere porzioni di codice da segnali come quelli generati da un `Ctrl-C`.

Esempio:

```
$ more critical.c                   ...lista il codice
#include <stdio.h>
#include <signal.h>

main()  {

  int (*oldHandler)();

  printf("Posso essere interrotto\n");
  sleep(3);
  oldHandler = signal(SIGINT,SIG_IGN);
  printf("Ora sono protetto dai Ctrl-C\n");
  sleep(3);
  signal(SIGINT, oldHandler);
  printf("Ora posso essere interrotto\n");
  sleep(3)
  printf("Ciao!\n");
}

$ critical.exe
Posso essere interrotto
^C                          ...^C funziona
$ critical.exe
Posso essere interrotto
Ora sono protetto
^C                          ...^C e' ignorato
Ora posso essere interrotto
Ciao!
$_
```

La primitiva `kill`

Un processo può inviare un segnale ad un altro processo usando la primitiva `kill`.

La sua sintassi è la seguente:

```
kill( int pid, int sigCode )
```

La funzione invia il segnale `sigCode` al processo con PID `pid`.

Perché la funzione venga usata correttamente è necessario che almeno una delle seguenti condizioni sia verificata:

- il processo che invia il segnale abbia lo stesso proprietario di quello che lo riceve
- il processo che invia il segnale è di proprietà di un "super-utente"

La funzione restituisce 0 se il segnale è inviato correttamente; altrimenti restituisce -1.

Sospensione e riattivazione di processi

I segnali SIGSTOP e SIGCONT sospendono e riattivano un processo, rispettivamente.

Nel seguente esempio, il main crea due processi figli ed entrambi entrano in un loop infinito. Il main attende 3 secondi e quindi sospende il primo figlio, e dopo 3 secondi lo riattiva.

Esempio:

```
$ more pulse.c
#include <stdio.h>
#include <signal.h>

main() {
  int pid1, pid2;

  pid1=fork();
  if( pid1==0) {  /* 1o figlio */
    while(1)  {
      printf("pid1 is alive\n");
      sleep(1);
      }
    }
  pid2=fork();
  if( pid2==0) {  /* 2o figlio */
    while(1)  {
      printf("pid2 is alive\n");
      sleep(1);
      }
    }
  sleep(3);
  kill(pid1,SIGSTOP);  /* sospendi 1o figlio */
  sleep(3);
  kill(pid1,SIGCONT);  /* riattiva 1o figlio */
  sleep(3);
  kill(pid1,SIGINT);  /* uccidi 1o figlio */
  kill(pid2,SIGINT);  /* uccidi 2o figlio */
}

$ pulse                    ...esegue il programma
pid1 is alive              ...entrambi "vivono"
pid2 is alive
...
pid2 is alive              ...solo il 1o "vive"
pid2 is alive

...
pid1 is alive              ...di nuovo entrambi
pid2 is alive
$ _
```

Gruppi di processi e terminali di controllo

Quando si esegue un programma che genera dei processi figli, un singolo `Ctrl-C` da tastiera provoca la terminazione non solo del programma genitore, ma anche dei suoi figli.

Per supportare questo tipo di comportamento, in Unix sono stati introdotti alcuni concetti nuovi:

- ogni processo, oltre ad avere un proprio PID, è anche membro di un **gruppo di processi**.
  Quando un processo genera un figlio (con una fork), questi eredita il gruppo dal processo padre. Lo stesso accade quando si esegue una `exec`.
  Un processo può modificare il suo grupp mediante il comando `setpgrp`.

- Ogni processo può avere associato un **terminale di controllo**.
  Quando un processe genera un figlio, questi eredita il terminale di controllo dal padre, e lo stesso accade quando si esegue una `exec`.

- Ogni terminale può essere associato con un unico **processo di controllo**. Quando un meta-carattere come `Ctrl-C` viene inviato, il terminale invia il segnale appropriato a tutti i processi del gruppo a cui appartiene il suo processo di controllo.

- Se un processo cerca di legger dal suo terminale di controllo ma non è membro dello stesso gruppo del processo di controllo del terminale, il nucleo gli invia un segnale (SIGTTIN) che sospende il processo.

Gruppi di processi e terminali di controllo (2)

Lo shell usa le caratteristiche precedenti nel seguente modo:

- Quando viene lanciato uno shell interattivo, questi è il processo di controllo del terminale ed ha quel terminale come terminale di controllo

- Quando uno shell esegue un processo in foreground, prima di lanciare la primitiva exec, lo shell figlio si pone in un gruppo diverso e prende il controllo del terminale.
  Questo vuol dire che tutti i segnali generati dal terminale sono inviati al processo in foreground e non allo shell.
  Alla terminazione del processo, il controllo del terminale ritorna allo shell

- Quando uno shell esegue un processo in background, prima di lanciare la primitiva exec, lo shell figlio si pone in un gruppo diverso, ma NON prende il controllo del terminale.
  I segnali generati dal terminale, quindi, sarà inviato allo shell e non al processo.
  Inoltre, se il processo cerca di legger dal suo terminale di controllo, viene sospeso mediante il segnale SIGTTIN

Il comando **setpgrp**

Il comando `setpgrp` modifica il gruppo di appartenenza di un processo.

La sua sintassi è la seguente:

```
int setpgrp( int pid, int pgrpId )
```

Il comando pone il processo il cui indentificatore è `pid` nel gruppo identificato dall'identificatore `pgrpId`.

Se `pid`=0, il processo chiamante viene posto nel gruppo `pgrpId`.

Perché il comando abbia successo, almeno una delle seguenti condizioni deve essere verificata:

- il chiamante ed il processo specificato devono appartenere allo stesso proprietario.
- il chiamante deve essere un super-utente

Se `setpgrp` fallisce, il processo restituisce un -1.

In genere, quando un processo vuole creare il proprio gruppo, usa il suo PID come identificatore del gruppo.

Per ottenere il gruppo di un processo si usa il comando `getpgrp`:

```
int getpgrp( int pid)
```

che restituisce il gruppo del processo con identificatore `pid`.
Se `pid = 0`, il comando restituisce il gruppo del processo chiamante.

Esempio (ver. 1)

In questo esempio vediamo come il terminale distribuisca i segnali a tutti i processi appartenenti al gruppo del processo di controllo.

```
$ more pgrp1.c          ...lista il programma
#include <stdio.h>
#include <signal.h>

sigintHandler();

main()
{
  signal( SIGINT, sigintHandler );
  if( fork() == 0 )
    printf("Child PID %d PGRP % waits\n",getpid(),
           getpgrp(0) );
  else
    printf("Parent PID %d PGRP % waits\n",getpid(),
           getpgrp(0) );
  pause();
}

sigintHandler()
{
  printf("Process PID %d got a SIGINT\n",getpid());
}
```

```
$ pgrp1                 ...esegue il programma
Parent PID 24783 PGRP 24783 waits
Child PID 24784 PGRP 24783 waits
^C                      ...digito ^C
Process 24783 got a SIGINT
Process 24784 got a SIGINT
$ _
```

Esempio (ver. 2)

In questo esempio vediamo come il processo figlio non riceve il segnale di interruzione.

```
$ more pgrp2.c         ...lista il programma
#include <stdio.h>
#include <signal.h>

sigintHandler();

main()  {
  int i;

  signal( SIGINT, sigintHandler );
  if( fork() == 0 )
    setpgrp(0, getpid());
  printf("Process PID %d PGRP % waits\n",getpid(),
           getpgrp(0) );
  for(i=1; i <=3; i++)  {
    printf("Process PID %d is alive\n",getpid());
    sleep(1);
    }
}

sigintHandler()  {
  printf("Process PID %d got a SIGINT\n",getpid());
}

$ pgrp2                ...esegue il programma
Process PID 24783 PGRP 24783 waits
Process PID 24784 PGRP 24784 waits
^C                     ...digito ^C
Process 24783 got a SIGINT      ...padre riceve ^C
Process 24784 is alive          ...figlio continua
Process 24784 is alive
Process 24784 is alive
$ _
```

Esempio (ver. 3)

In questo esempio vediamo come, se un processo cerca di leggere dal suo terminale di controllo dopo essersi "dissociato" dal gruppo del processo di controllo del terminale, il nucleo gli invia un segnale SIGTTIN.

```
$ more pgrp3.c         ...lista il programma
#include <stdio.h>
#include <signal.h>

sigttinHandler();

main()  {
  int status; char str[100];

  if( fork() == 0 )  {     /* figlio */
    signal( SIGTTIN, sigttinHandler );
    setpgrp(0, getpid());
    printf("Enter a string: ");
    scanf("%s",str); /* cerco di leggere */
    printf("You entered %s\n", str);
    }
  else                    /* padre */
    wait(&status);
}

sigttinHandler()  {
  printf("Illegal read from control terminal\n");
}


$ pgrp3                   ... esegue il programma
Enter a string: Illegal read from control terminal
$ _
```