Programmazione di Sistema – 4

Lucidi per il corso di Laboratorio di Sistemi Operativi tenuto da Paolo Baldan presso l'Università Ca' Foscari di Venezia, anno accademico 2004/2005. Parte di questo materiale è rielaborato dai lucidi del Corso di Laboratorio di Sistemi Operativi tenuto da Rosario Pugliese presso l'Università di Firenze, anno accademico 2001/02.

Gestione dei Segnali

Segnali

I processi talvolta devono gestire eventi inaspettati o impredicibili quali

- *errori run-time (es. divisione per 0)*
- *mancanza di corrente elettrica*
- *terminazione di un processo figlio*
- *richiesta di terminazione da parte di un utente (`<Ctrl-c>`)*
- *richiesta di sospensione da parte di un utente (`<Ctrl-z>`)*

Per la gestione di eventi asincroni di questo tipo Unix offre il meccanismo dei segnali.

Sono chiamati interrupt software: la loro elaborazione può comportare l'interruzione del flusso regolare del processo.

Segnali: Gestione

Ogni segnale ha un identificatore numerico

- **`signal.h`** *definisce delle costanti simboliche* **`SIGxxx`** *che identificano i vari segnali.*
  - *es.* **`SIGCHLD`** *rappresenta la terminazione di un figlio*
- *il numero di segnali predefiniti varia da 15 a 35 a seconda della versione di Unix*

Quando Unix si rende conto che si è verificato l'evento associato ad un segnale **`SIGxxx`** *invia il segnale in questione al processo corrispondente.*

- *Il descrittore di un processo (PCB) nella process table include un bitmap, con una entry per ogni segnale, che memorizza i segnali pendenti.*
- *L'invio del segnale consiste nell'attivazione del bit corrispondente nel bitmap.*

Segnali: Gestione

Il kernel verifica la presenza di segnali pendenti per un processo quando questo

- *passa da kernel a user mode (es. ritorno da sys call).*
- *abbandona lo stato sleep o vi entra.*

Debolezze dei segnali

- *un segnale può essere gestito con un certo ritardo (problema per applicazioni real-time)*
- *non si tiene conto del numero di segnali dello stesso tipo pendenti (se vari* **`SIGINT`** *arrivano in successione rapida, può accadere che uno solo di questi venga rilevato).*

Segnali: Handler

Quando un processo "riceve" un segnale può reagire in uno dei seguenti modi

Ignore: il segnale viene ignorato e non ha alcun effetto

SIGKILL *e* **SIGSTOP** *non possono essere ignorati*

questo permette al superuser di terminare processi.

Catch: esegue un signal handler, una funzione che gestisce l'evento associato al segnale

sospende il flusso di esecuzione corrente

esegue il signal handler

riprende il flusso di controllo originario quando il signal handler termina.

Segnali: Handler

La reazione ad un segnale può essere definita dal programmatore o può essere quella di default stabilita dal kernel.

La reazione di default, in genere, consiste in una delle azioni seguenti

- *termina il processo*
  - *generando un file core (dump)*
  - *senza generare un file core (quit)*
- *ignora il segnale e lo cancella (ignore)*
- *sospende il processo (suspend)*
- *riprende l'esecuzione del processo (resume).*

Condizioni che generano un segnale

Pressione di tasti speciali sul terminale

- **<Ctrl-c>** *invia il segnale* **SIGINT**

Eccezioni hardware

- *l'interrupt hardware viene catturato dal kernel che invia un segnale corrispondente al processo*
  - *divisione per 0 (***SIGFPE***)*
  - *riferimento non valido alla memoria (***SIGSEGV***)*

System call **kill**

- *un processo può spedire un segnale ad altri processi*
- *deve aver il diritto di farlo: l'***uid** *del processo che esegue* **kill** *deve essere*
  - *lo stesso del processo a cui si spedisce il segnale*
  - *l'***uid** *di root (0)*

Alcuni segnali predefiniti

Macro	#	Default	Description
SIGHUP	1	quit	hang up
SIGINT	2	quit	interrupt
SIGQUIT	3	dump	quit
SIGILL	4	dump	illegal instruction
SIGTRAP	5	dump	trace trap
SIGABRT	6	dump	abort
SIGEMT	7	dump	emulator trap instruction
SIGFPE	8	dump	arithmetic execption
SIGKILL	9	quit	kill (cannot be caught, blocked or ignored)
SIGBUS	10	dump	bus error (bad format address)
SIGSEGV	11	dump	segmentation violation (out-of-range address)
SIGSYS	12	dump	bad argument to system call
SIGPIPE	13	quit	write on a pipe/socket with no one to read it
SIGALRM	14	quit	alarm clock
SIGTERM	15	quit	software termination signal

Alcuni segnali predefiniti

Macro	#	Default	Description
SIGUSR1	16	quit	user signal 1
SIGUSR2	17	quit	user signal 2
SIGCHLD	18	ignore	child status changed
SIGPWR	19	ignore	power fail or restart
SIGWINCH	20	ignore	window size change
SIGURG	21	ignore	urgent socket condition
SIGPOLL	22	exit	pollable event
SIGSTOP	23	quit	stopped (signal)
SIGSTP	24	quit	stopped (user)
SIGCONT	25	ignore	continued
SIGTTIN	26	quit	stopped (tty input)
SIGTTOU	27	quit	stopped (tty output)
SIGVTALRM	28	quit	virtual timer expired
SIGPROF	29	quit	profiling timer expired
SIGXCPU	30	dump	CPU time limit exceeded
SIGXFSZ	31	dump	file size limit exceeded

Segnali da tastiera

È possibile inviare un segnale ad un processo in foreground premendo **<Ctrl-c>** *o* **<Ctrl-z>** *dalla tastiera.*

Quando il driver di un terminale riconosce che è stato premuto **<Ctrl-c>** *(*<**Ctrl-z**>*) invia un segnale* **SIGINT** *(***SIGSTP***) a tutti i processi nel gruppo del processo in foreground.*

Alcuni segnali collegati

SIGCHLD*: inviato al padre da un figlio che termina;*

SIGSTOP*: sospensione da dentro un programma;*

SIGCONT*: riprende l'esecuzione di un programma dopo una sospensione.*

Richiedere un segnale di allarme: alarm()

`unsigned int alarm (unsigned int count)`

Istruisce il nucleo a spedire il segnale **`SIGALRM`** *al processo invocante dopo* **`count`** *secondi.*

Un eventuale **`alarm()`***già schedulato viene sovrascritto col nuovo.*

Se **`count`** *è 0, non schedula nessun nuovo* **`alarm()`** *(e cancella quello eventualmente già schedulato).*

Restituisce il numero di secondi rimanenti prima dell'invio dell'allarme, oppure 0 se non è schedulato nessun **`alarm()`**.

Nota: l'allarme è inviato dopo almeno **`count`** *secondi, ma il meccanismo di scheduling può ritardare ulteriormente la ricezione del segnale.*

Gestire i segnali: signal()

```
int (*signal(int signum, void (*handler)(int)))(void)
```

oppure

```
typedef void (*sighandler_t)(int)
sighandler_t signal(int signum, sighandler_t handler)
```

Installa un nuovo signal handler, **`handler`***, per il segnale con numero* **`signum`***.*

Restituisce il precedente signal handler associato a **`signum`***, se ha successo; altrimenti, -1.*

Gestire i segnali: signal()

L'handler può essere

- indirizzo di una funzione **handler** definita dall'utente
  - La funzione **handler** ha un argomento intero che rappresenta il numero del segnale. Questo permette di utilizzare lo stesso handler per segnali differenti.
- uno dei seguenti valori
  - **SIGN_IGN**: indica che il segnale dev'essere ignorato;
  - **SIGN_DFL**: indica che deve essere usato l'handler di default fornito dal nucleo.

Nota

- Il nome di una funzione è un valore puntatore a funzione.
- Una funzione in una dichiarazione di parametro viene interpretata dal compilatore come un puntatore.
- Quindi nel prototipo di **signal()** si può togliere "*".

Gestire i segnali: signal()

I segnali **SIGKILL** *e* **SIGSTP** *non sono riprogrammabili.*

Con la creazione di nuovi processi ...

Dopo una **fork()**, *il processo figlio eredita le politiche di gestione dei segnali del padre.*

Se il figlio esegue una **exec()**

i segnali precedentemente ignorati continuano ad essere ignorati

i segnali i cui handler erano stati ridefiniti sono ora gestiti dagli handler di default.

Ad eccezione di **SIGCHLD**, *i segnali non sono accodati*

quando più segnali dello stesso tipo arrivano "contemporaneamente", solo uno viene trattato.

Esempio: critical.c

Il programma **`critical.c`** *suggerisce come si possono proteggere pezzi di codice "critici" da interruzioni dovute a* **`<Ctrl-c>`** *(segnale* **`SIGINT`***) o ad altri segnali simili che possono essere ignorati.*

Procede nel modo seguente

- *salva il precedente valore dell'handler, indicando che il segnale deve essere ignorato;*
- *esegue la regione di codice protetta;*
- *ripristina il valore originale dell'handler.*

Esempio: critical.c

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

int main (void) {
  void (*oldHandler) (int);  /* To hold old handler value */
  printf ("I can be Control-C'ed\n");
  sleep (3);
  oldHandler = signal (SIGINT, SIG_IGN);  /* Ignore Control-C */
  printf ("I'm protected from Control-C now\n");
  sleep (3);
  signal (SIGINT, oldHandler);  /* Restore old handler */
  printf ("I can be Control-C'ed again\n");
  sleep (3);
  printf ("Bye!\n");
  return 0;
}
```

Esempio: critical.c

```
$ critical
I can be Control-C'ed
I'm protected from Control-C now
I can be Control-C'ed again
Bye!
$
```

Protezione dai segnali: Mascheramento

Ignorare i segnali non è sempre una buona soluzione... possono portare informazioni rilevanti.

Un processo può "bloccare" la ricezione di segnali (la cui azione di default non sia ignore).

Un segnale bloccato che venga inviato ad un processo rimane pendente fino a che

- *il processo sblocca il segnale;*
- *il processo cambia l'azione associata in "ignore".*

I segnali pendenti possono essere analizzati...

System call: **`sigprocmask`**, **`sigpending`**

Inviare segnali: kill()

`int kill(pid_t pid, int signum)`

spedisce il segnale con valore **`signum`** *al processo con PID* **`pid`**.

Permessi

è possibile inviare il segnale nei seguenti casi

- *i processi mittente e destinatario hanno lo stesso proprietario; più precisamente real o effective uid del mittente coincide con real o effective uid del destinatario.*
- *il processo mittente ha come proprietario il superuser.*

Inviare segnali: kill()

Il comportamento di **`kill`** *varia a seconda di* **`pid`**

`pid > 0`

il segnale è inviato al processo pid;

`pid = 0`

invia il segnale a tutti i processi nel gruppo del mittente;

`pid = -1`

se il mittente ha per proprietario il superuser, invia il segnale a tutti i processi, mittente incluso;

se il mittente non ha per proprietario un superuser, invia il segnale a tutti i processi nello stesso gruppo del mittente, con esclusione del mittente;

`pid < -1`

invia il segnale a tutti i processi nel gruppo **`|pid|`**.

Restituisce valore 0, se invia con successo almeno un segnale; altrimenti, restituisce -1.

Esempio: limit.c (terminazione figlio)

Il programma **`limit.c`** *permette all'utente di limitare il tempo impiegato da un comando per l'esecuzione.*

`limit` ***`nsec cmd args`***
esegue il comando **`cmd`** *con gli argomenti* **`args`** *indicati, dedicandovi al massimo* **`nsec`** *secondi.*

Il programma definisce un handler per il segnale **`SIGCHLD`**.

Esempio: limit.c (terminazione figlio)

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

int delay;
void childHandler (int);   /* death-of-child handler (see later) */

int main (int argc, char *argv[]) {
  int pid;
  signal (SIGCHLD, childHandler); /* Install death-of-child handler */
  pid = fork (); /* Duplicate */
  if (pid == 0) { /* Child */
    execvp (argv[2], &argv[2]); /* Execute command */
    perror ("limit"); /* Should never execute */
  }
  else { /* Parent */
    sscanf (argv[1], "%d", &delay); /* Read delay from command line */
    sleep (delay); /* Sleep for the specified number of seconds */
    printf ("Child %d exceeded limit and is being killed\n", pid);
    kill (pid, SIGINT); /* Kill the child */
  }
  return 0;
}
```

Esempio: limit.c (terminazione figlio)

```
void childHandler (int sig) {   /* Executed if the child dies */
  int childPid, childStat;    /* before the parent */
  childPid = wait (&childStat); /* Accept child's termination code */
  printf ("Child %d terminated within %d seconds\n", childPid, delay);
  exit (/* EXITSUCCESS */ 0);
}
```

```
$ limit 3 find / -name filechenonce
find: /root/.links: Permission denied
find: /root/.ssh: Permission denied
...
Child 828 exceeded limit and is being killed

$ limit 1 ls
count.c          myexec.c          redirect.c      critical.c
limit.c          limit             myfork.c           lez7.ps
Child 828 terminated within 0 seconds
$
```

Esempio: pulse.c (sospensione&ripresa)

Il programma **`pulse.c`** *crea due figli che entrano in un ciclo infinito e mostrano un messaggio ogni secondo.*

Il padre aspetta 2 secondi e quindi sospende il primo figlio, mentre il secondo figlio continua l'esecuzione.

Dopo altri 2 secondi il padre riattiva il primo figlio, aspetta altri 2 secondi e quindi termina entrambi i figli.

Esempio: pulse.c (sospensione&ripresa)

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

int main (void) {
  int pid1;
  int pid2;
  pid1 = fork ();
  if (pid1 == 0) { /* First child */
    while (1) { /* Infinite loop */
        printf ("pid1 is alive\n");
        sleep (1);
      }
  }
  pid2 = fork ();
  if (pid2 == 0) { /* Second child */
    while (1) { /* Infinite loop */
        printf ("pid2 is alive\n");
        sleep (1);
      }
  }
```

Esempio: pulse.c (sospensione&ripresa)

```
/* ... continue ... */

  sleep (2);
  kill (pid1, SIGSTOP); /* Suspend first child */
  sleep (2);
  kill (pid1, SIGCONT); /* Resume first child */
  sleep (2);
  kill (pid1, SIGINT); /* Kill first child */
  kill (pid2, SIGINT); /* Kill second child */
  return 0;
}
```

Esempio: pulse.c (sospensione&ripresa)

... funzionamento ...

```
$ pulse
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
pid2 is alive
pid2 is alive
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
$
```

Gruppi di processi

Ogni processo è membro di un gruppo di processi.

Un gruppo di processi ha associato un identificatore pgid – process group id (da non confondere con il gid).

Un processo figlio eredita il gruppo di appartenenza dal padre.

La system call **`setpgid()`** *permette al processo invocante di cambiare gruppo di appartenenza (a se stesso o ad altri).*

Gruppi di processi e terminale di controllo

Ad ogni processo può essere associato un terminale di controllo

- *è tipicamente il terminale da cui il processo è lanciato;*
- *i figli ereditano il terminale di controllo del padre;*
- *se un processo esegue una* **exec()**, *il terminale di controllo non cambia.*

Ad ogni terminale è associato un processo di controllo

- *se il terminale individua un metacarattere come* **<Ctrl-c>** *spedisce il segnale appropriato a tutti i processi nel gruppo del processo di controllo;*
- *se un processo tenta di leggere dal suo terminale di controllo e non è membro del gruppo del processo di controllo di quel terminale, il processo riceve un segnale* **SIGTTIN** *che, normalmente, lo sospende.*

Gruppi e terminale: uso nella shell

All'avvio di una shell interattiva

- la shell è il processo di controllo del terminale da cui è lanciata
- il terminale in questione è il terminale di controllo della shell.

Se la shell esegue un comando in foreground

- la shell figlia si mette in un diverso gruppo, assume il controllo del terminale, esegue il comando
- così ogni segnale generato dal terminale viene indirizzato al comando e non alla shell originaria.
- quando il comando termina, la shell originaria riprende il controllo del terminale.

Se la shell esegue un comando in background

- la shell figlia si mette in un diverso gruppo ed esegue il comando, ma non non assume il controllo del terminale.
- così ogni segnale generato dal terminale continua ad essere indirizzato alla shell originaria.
- se il comando in background tenta di leggere dal suo terminale di controllo, viene sospeso da un segnale SIGTTIN.

Cambiare il gruppo: setpgid()

`int setpgid(pid_t pid, pid_t pgrpId)`

assegna valore **`pgrpId`** *al process group id del processo con* **PID** **`pid`**.

Se **`pid`** *è* **`0`**, *cambia il valore del process group id del processo invocante.*

Se **`pgrpId`** *è* **`0`**, *assegna il process group ID del processo con* **PID** **`pid`** *al processo invocante.*

Ha successo se

il processo invocante ed il processo specificato come primo argomento hanno lo stesso (effective) uid/gid;

il proprietario del processo invocante è il superuser.

Restituisce 0 se ha successo; -1 altrimenti.

Cambiare il gruppo: setgpid()

Quando un processo vuole creare un proprio gruppo di processi, distinto dagli altri gruppi del sistema, tipicamente passa il proprio **PID** *come argomento per* **setgpid()**

```
setgpid(0,getpid())
```

Ottenere il gruppo: getgpid()

`pid_t getpgid(pid_t pid)`

Restituisce il gruppo di processi a cui appartiene il processo con **`PID pid`**.

Se pid è 0, restituisce il gruppo di processi a cui appartiene il processo invocante.

Non fallisce mai.

Esempio: pgrp1.c

Il programma **`pgrp1.c`** *mostra come un terminale invia i segnali ad ogni processo appartenente al gruppo di processi del suo processo di controllo.*

Poiché un figlio eredita il gruppo di processi del padre, padre e figlio catturano il segnale **`SIGINT`**.

Esempio: pgrp1.c

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

void sigHandler (int sig) {
  printf ("Process %d got a %d signal \n", getpid (), sig);
}

int main (void) {
  signal (SIGINT, sigHandler); /* Handle Control-C */
  if (fork () == 0)
    printf ("Child PID %d PGRP %d waits\n", getpid (), getpgid (0));
  else
    printf ("Parent PID %d PGRP %d waits\n", getpid (), getpgid (0));
  pause (); /* Wait for a signal */
}
```

Esempio: pgrp1.c

```
$ pgrpl
Parent PID 24444 PGRP 24444 waits
Child PID 24445 PGRP 24444 waits
<^C> Process 24445 got a 2 signal
Process 24444 got a 2 signal
$
```

pause() *sospende il processo invocante fino all'arrivo di un segnale*

- *che ne causa la terminazione*
- *che causa l'esecuzione di un signal handler.*

Esempio: pgrp2.c

Il programma pgrp2.c mostra che se un processo lascia il gruppo del processo di controllo del terminale, non riceve più segnali dal terminale.

Esempio: pgrp2.c

```
#include <signal.h>
#include <stdio.h>
void sigHandler (int sig) {
  printf ("Process %d got a SIGINT\n", getpid ());
  exit (1);
}
int main (void) {
  int i;
  signal (SIGINT, sigHandler); /* Install signal handler */
  if (fork () == 0)
    setpgid (0, getpid ()); /* Place child in its own process group */
  printf ("Process PID %d PGRP %d waits\n", getpid (), getpgid (0));
  for (i = 1; i <= 3; i++) { /* Loop three times */
    printf ("Process %d is alive\n", getpid ());
    sleep(2);
  }
  return 0;
}
```

Esempio: pgrp2.c

esecuzione...

```
$ pgrp2
Process PID 24535 PGRP 24535 waits
Process 24535 is alive
Process PID 24536 PGRP 24536 waits
Process 24536 is alive
<^C> Process 24535 got a SIGINT
$ Process 24536 is alive
Process 24536 is alive
<return>
$
```

Esempio: sigttin.c

Il programma sigttin.c mostra che se un processo lascia il gruppo del processo di controllo del terminale e quindi tenta di leggere dal terminale stesso, riceve un segnale **SIGTTIN** *che ne provoca la sospensione.*

Nell'esempio l'handler per **SIGTTIN** *(21) viene riprogrammato.*

Esempio: sigttin.c

```
#include <signal.h>
#include <stdio.h>
#include <sys/termio.h>
#include <fcntl.h>
void sigHandler (int sig) {
  printf ("%d Inappropriate read from control terminal\n", sig);
  exit (1);
}

int main (void) {
  int status;
  char str [100];
  if (fork () == 0) { /* Child */
    signal (SIGTTIN, sigHandler); /* Install handler */
    setpgid (0, getpid ()); /* Place myself in a new process group */
    printf ("Enter a string: ");
    scanf ("%s", str); /* Try to read from control terminal */
    printf ("You entered %s\n", str);
  } else { /* Parent */
    wait (&status); /* Wait for child to terminate */
  }
}
```

Esempio: sigttin.c

```
$ pgrp3
Enter a string: 21 Inappropriate read from control terminal
$
```

Segnali per IPC

I segnali possono essere utilizzati per (forme limitate di) IPC.

Esempio

- *il processo P apre un file*
- *si duplica, creando un figlio Q*
- *P scrive un intero, sempre all'inizio del file*
- *Q legge un intero, sempre all'inizio del file*

Funziona se P e Q si sincronizzano tramite segnali

- *P scrive ed invia un segnale a Q, quindi attende un segnale*
- *Q aspetta un segnale prima di leggere, legge ed invia un segnale a P*

Esempio: commSig.c

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int usr1rcv = 0; /* var per comunicazione tra handler e figlio :
                    segnala la ricezione  di un segnale SIGUSR1 */

void usr1Handler(int sig) { /* handler per SIGUSR1 */
  usr1rcv = (sig == SIGUSR1);
}

int main (void) {
  int fdTmp; /* file descriptor comune */
  pid_t father, son;
  char tmpFile[7] = "XXXXXX";
  int elem;

  fdTmp = mkstemp(tmpFile); /* Crea un file temporaneo */
  unlink(tmpFile);
  signal(SIGUSR1,usr1Handler); /* Installa l'handler */
```

Esempio: commSig.c

```
if ((son = fork()) != 0)
   {
     /* processo padre */
     for(elem=0; elem<10; elem++) {
        sleep(1);
        /* scrive un valore per il figlio */
        lseek(fdTmp, 0, SEEK_SET);
        write(fdTmp, &elem, sizeof(elem));
        printf("Padre: Inviato il valore %d\n", elem);
        /* segnala la presenza del valore */
        kill(son, SIGUSR1);
        /* attende che il figlio legga il valore */
        while (!(usr1rcv))
          pause();
        usr1rcv = 0;
     }
   }
```

Esempio: commSig.c

```
  else
    {
      /* processo figlio */
      father = getppid();
      while (1) {
        /* attende che il padre spedisca un valore */
        while (!(usr1rcv))
          pause();
        lseek(fdTmp, 0, SEEK_SET);
        read(fdTmp, &elem, sizeof(elem));
        printf("Figlio: Ricevuto il valore %d\n", elem);
        usr1rcv = 0;
        /* segnala al padre che ha letto il valore */
        kill(father, SIGUSR1);
       }
     }
}
```