

Segnali

Segnali

Talvolta **eventi inattesi** devono essere **comunicati** ai **processi**, oppure il **kernel** deve segnalare **eccezioni** intercettate **dall'hardware** ai processi che le hanno generate.

Ad esempio, se viene eseguita una divisione per zero o si verifica un overflow, l'hardware comunica tale situazione al **kernel**, il quale **segnala** tale situazione al **processo**.

Uno **strumento** per implementare tale **comunicazione** sono i **segnali**.

Segnali (cont.)

Ciascun segnale ha un nome che inizia con SIG definito in `<signal.h>`.

Se un **processo** effettua una chiamata alla funzione **abort**, il segnale generato è **SIGABRT**; **SIGALRM** viene generato quando si azzerava il timer della funzione **alarm**.

I **segnali** sono un esempio di **evento asincrono**: dal punto di vista dei processi, essi possono accadere in un qualsiasi momento.

Il processo deve istruire il kernel su cosa fare se e quando un certo segnale viene generato.

Segnali (cont.)

Segnali vengono emessi quando l'utente preme determinati tasti; ad esempio <Ctrl C> provoca la generazione del segnale SIGINT, che termina il processo in esecuzione.

Eccezioni hardware generano segnali: una divisione per zero, un riferimento non valido ad una locazione di memoria, ecc.; ad esempio il segnale SIGSEGV verrà comunicato al processo che ha effettuato un riferimento non valido alla memoria.

Condizioni software possono generare segnali; ad esempio se un processo cerca di scrivere su di una pipe dopo che il processo che legge ha terminato, viene generato il segnale SIGPIPE.

Segnali (cont.)

La chiamata di sistema **kill(2)** permette ad un processo di inviare un qualsiasi segnale ad un altro processo o gruppo di processi, se il processo chiamante ne ha i privilegi.

Il comando **kill(1)** permette di mandare segnali ad altri processi o gruppi di processi e viene utilizzato per terminare processi di cui si è perso il controllo.

Segnali (cont.)

Tre sono le cose che un processo può chiedere al kernel di fare (azione associata ad un segnale):

Ignorare il segnale: (tutti tranne **SIGKILL** e **SIGSTOP**); nel caso si decida di ignorare i segnali generati da eccezioni hardware (**SIGFPE**, **SIGILL**, e **SIGSEGV**), il comportamento del processo è indefinito.

Intercettare il segnale: fornire una funzione da eseguire per un determinato segnale.

Eseguire le azioni di default: ciascun segnale ha una azione di default associata; nella maggior parte dei casi, questa azione consiste nella terminazione del processo.

```
[mariog]$ kill -l
```

```
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL
5) SIGTRAP     6) SIGIOT     7) SIGBUS      8) SIGFPE
9) SIGKILL     10) SIGUSR1    11) SIGSEGV    12) SIGUSR2
13) SIGPIPE    14) SIGALRM   15) SIGTERM    17) SIGCHLD
18) SIGCONT    19) SIGSTOP   20) SIGTSTP    21) SIGTTIN
22) SIGTTOU    23) SIGURG    24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF   28) SIGWINCH   29) SIGIO
30) SIGPWR
```

```
[mariog]$z
```

Funzione signal

La funzione **signal** fornisce lo strumento per istruire il kernel ad eseguire una determinata azione quando il processo chiamante riceve un determinato segnale.

Tale azione può essere

1. **ignorare** (**SIG_IGN**) il segnale,
2. far **eseguire** al kernel l'azione di default definita per tale segnale (**SIG_DFL**),
3. **passare** al kernel **l'indirizzo** di una funzione da eseguire quando si presenta tale segnale.

La funzione ritorna le precedenti disposizioni per il segnale.

Funzione **signal** (cont.)

```
#include <signal.h>
void (*signal(int signum, void (*handler)(int)))(int);
```

La funzione **signal** ha due argomenti e ritorna un puntatore ad una funzione che non ritorna nulla (void).

signum è un intero, *handler* è il **puntatore** ad una **funzione** che prende come argomento un intero e non ritorna nulla.

La funzione di cui è ritornato l'indirizzo come valore della funzione **signal** ha un unico argomento intero (l'ultimo (int)).

Funzione signal (cont.)

In altre parole, alla funzione da eseguire (*signal handler*) viene passato un singolo argomento intero e non ritorna nulla.

Se si definisce il tipo:

```
typedef void Sigfunc(int);
```

allora il prototipo di **signal** diviene:

```
Sigfunc *signal(int, Sigfunc *);
```

```
#include<signal.h>

int main(void)
{
    typedef void Sigfunc(int);
    Sigfunc *signal(int, Sigfunc *);

    signal (SIGINT, SIG_IGN);
    while (1);
}
```

Non è possibile interrompere l'esecuzione con <CTRL C>.

E' necessario eseguire un kill(1) da un'altra shell!

```
# include <signal.h>
# include "ourhdr.h"
static void sig_usr(int); /* one handler for all signals */

int main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");
    if (signal(SIGINT, sig_usr) == SIG_ERR) /* <CTRL C> */
        err_sys("can't catch SIGINT");
    if (signal(SIGTSTP, sig_usr) == SIG_ERR) /* <CTRL Z> */
        err_sys("can't catch SIGTSTP");

    for ( ; ; ) pause();
}
```

```
static void
sig_usr(int signo)      /* argument is signal number */
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else if (signo == SIGINT)
        printf("received SIGINT\n");
    else if (signo == SIGTSTP)
        printf("received SIGTSTP\n");
    else
        err_dump("received signal %d\n", signo);
    return;
}
```

Esempio 2 (cont.)

Eseguendo l'esempio su Linux, si ottiene:

```
[mariog]$ a.out &
[1] 774
[mariog]$ kill -USR1 774
[mariog]$ received SIGUSR1

[mariog]$ kill -USR2 774
[mariog]$ received SIGUSR2

[mariog]$ kill 774
[mariog]$ jobs
[1]+  Terminated                a.out
[mariog]$ a.out
received SIGINT
received SIGTSTP
```

La seconda volta vengono digitati <Ctrl C> e <Ctrl Z>.

Per terminare la seconda esecuzione è adesso necessario eseguire un `kill(1)` da un'altra shell!

Attivazione dei programmi

Quando un **eseguibile** viene attivato con una **exec**, lo stato di tutti i **segnali** o è quello di **default**, o è quello di **ignora**.

Di norma l'occorrenza di un segnale comporta l'esecuzione dell'azione di default, a meno che il processo che ha eseguito la **exec** non stia ignorando quel segnale.

Un **segnale intercettato** in un processo, **non** può essere **intercettato** da un eseguibile attivato con **exec**, perché per quest'ultimo l'indirizzo della funzione di gestione del segnale non ha senso.

Attivazione dei programmi (cont.)

Quella descritta è una situazione frequente. Infatti, ogni volta che mandiamo in esecuzione un **processo** in **background**, non siamo in grado di terminarlo con <Ctrl C> (SIGINT) o fermarlo con <Ctrl Z> (SIGTSTP) in quanto la shell dispone che per i processi in background questi **segnali** siano **ignorati**.

I **processi** generati con **fork** si comportano, rispetto ai segnali, esattamente **come** i **genitori**, in quanto essi hanno a disposizione un'**immagine** della memoria del genitore.

Funzioni rientranti

Non tutte le funzioni possono essere utilizzate in un signal handler.

Supponiamo, ad esempio, che una funzione riceve un segnale mentre sta allocando con **malloc** della memoria e che la funzione di signal handling a sua volta allochi memoria dinamicamente.

Poiché **malloc** mantiene una lista di tutte le aree allocate, è possibile che il segnale venga lanciato mentre essa sta aggiornando tale lista e l'esecuzione della funzione di signal handling intervenga proprio su tale lista. In tal caso il risultato è imprevedibile.

Funzioni rientranti (openBSD 3.0)

**_exit(), access(), alarm(), cfgetispeed(), cfgetospeed(),
cfsetispeed(), cfsetospeed(), chdir(), chmod(), chown(),
close(), creat(), dup(), dup2(), execl(), execve(), fcntl(),
fork(), fpathconf(), fstat(), fsync(), getegid(), geteuid(),
getgid(), getgroups(), getpgrp(), getpid(), getppid(),
getuid(), kill(), link(), lseek(), mkdir(), mkfifo(), open(),
pathconf(), pause(), pipe(), raise(), read(), rename(),
rmdir(), setgid(), setpgid(), setsid(), setuid(), sigaction(),
sigaddset(), sigdelset(), sigemptyset(), sigfillset(),
sigismember(), signal(), sigpending(), sigprocmask(),
sigsuspend(), sleep(), stat(), sysconf(), tcdrain(), tcflow(),
tcflush(), tcgetattr(), tcgetpgrp(), tcsendbreak(), tcsetattr(),
tcsetpgrp(), time(), times(), umask(), uname(), unlink(),
utime(), wait(), waitpid(), write()**

Alcune definizioni

Quando l'evento che genera un segnale accade, si dice che un segnale è stato generato (generated) per il processo, oppure che è stato mandato.

Quando un'azione è presa in corrispondenza di un determinato segnale si dice che il segnale è stato consegnato (delivered).

Nel periodo di tempo che interviene tra la generazione del segnale e la sua consegna si dice che il segnale è pendente (pending).

Come un segnale generato più di una volta venga consegnato ad un processo, dipende dalla particolare implementazione.

Funzioni **kill** e **raise**

```
# include <sys/types.h>
# include <signal.h>
int kill(pid_t pid, int sig);
int raise (int sig);
```

La funzione **kill** permette ad un processo di inviare il segnale *sig* ad al processo *pid*, **raise** a sé stesso.

Ritorna 0, se ok, <0 in caso di errore.

Funzioni kill e raise

```
# include <sys/types.h>
# include <signal.h>
int kill(pid_t pid, int sig);
int raise (int sig);
```

pid può assumere i seguenti valori:

pid > 0 il segnale viene inviato al processo con process ID uguale a *pid*.

pid == 0 il segnale viene inviato a tutti i processi con lo stesso process group

pid < 0 il segnale viene inviato a tutti i processi con process group ID uguale al modulo di *pid*

Funzioni kill e raise

Il **segnale 0** è il segnale nullo e può essere utilizzato per sapere se **esiste** un **processo** con un determinato **pid**.

Un processo può mandare un segnale ad un altro processo

- se entrambi sono in esecuzione o con gli stessi real o effective user ID;
- sempre, se è in esecuzione con i permessi di superutente.

Esempio 3

```
# include <unistd.h>
# include <signal.h>
# include <sys/types.h>
static void sig_hdl();
int main (void)
{
typedef void Sigfun(int);
Sigfun *signal(int, Sigfun *);
pid_t pid;
```

```
if(signal(SIGUSR1,sig_hdl)==SIG_ERR) perror("signal");
/* signal handler */
```

```
if((pid=fork())<0) perror("fork"); /* Creamo un figlio */
```

```
if( pid==0) pause(); /* il figlio aspetta un segnale */
```

```
Sigfun  *signal(int, Sigfun *);
```

```
pid_t  pid;
```

```
if(signal(SIGUSR1,sig_hdl)==SIG_ERR) perror("signal");  
/* signal handler */
```

```
if((pid=fork())<0) perror("fork");  /* Creamo un figlio */
```

```
if( pid==0)  pause();  /* il figlio aspetta un segnale */
```

```
else{
```

```
    sleep(1);
```

```
    kill(pid,SIGUSR1);  /* il genitore invia SIGUSR1 */
```

```
}
```

```
printf("Pid: %d\n", getpid());
```

```
exit(0);
```

```
,
```

```
else{
    sleep(1);
    kill(pid,SIGUSR1);          /* il genitore invia SIGUSR1 */
}
```

```
printf("Pid: %d\n", getpid());
exit(0);
}
```

```
static void sig_hdl()
{
    char buf[]="segnale intercettato\n";
    write(STDOUT_FILENO, buf, sizeof(buf)-1);
    return;
}
```

Eseguendo l'esempio su Linux, si ottiene:

```
[mariog]$ a.out
segnale intercettato
Pid: 12398
Pid: 12399
[mariog]$
```

E' necessario sospendere il genitore (**sleep**) affinché il figlio possa eseguire la funzione **pause**; in caso contrario quest'ultimo intercetta il segnale e poi esegue **pause**, che lo pone in attesa.

Infatti, commentando la **sleep** e ricompilando il programma si ottiene la seguente esecuzione:

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

```
Pid: 32128
```

```
segnale intercettato
```

```
[mariog]$ ps -efa |grep a.out
```

```
mariog 32129 1 0 16:24 pts/10 00:00:00 a.out
```

```
mariog 32131 32092 0 16:24 pts/10 00:00:00 grep a.out
```

```
[mariog]$ kill -USR1 32129
```

```
segnale intercettato
```

```
Pid: 32129
```

```
[mariog]$
```

Funzione alarm

```
# include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

La funzione **alarm** permette di inizializzare un timer che terminerà dopo *seconds* secondi. Il segnale generato è SIGALRM, che prevede come azione di default la terminazione del processo.

Ci può essere un solo timer per processo; se la funzione viene richiamata una seconda volta, ritorna il numero di secondi che mancano alla fine del conto alla rovescia.

Se viene chiamata una seconda volta con argomento nullo, l'allarme precedente viene cancellato.

Funzione `pause`

```
# include <unistd.h>  
  
int pause(void);
```

La funzione **pause** non fa niente: mette in attesa un processo fin quando non gli viene invitato un segnale. In questo caso ritorna `-1` e pone `errno` uguale ad **EINTR** (interrupted system call)

Sommario

I segnali trovano applicazione in molte applicazioni complesse.

Permettono di gestire situazioni di errore.

La loro conoscenza è essenziale nella programmazione di sistema.

Esercizi

Scrivere un programma per vedere se, con Linux, intercettando SIGCHLD si evitano i processi zombie (*SIGCHLD non si può ignorare*)

Cosa succede se un segnale intercettato arriva ripetutamente?
Scrivere un programma per verificare quante volte va in esecuzione il signal handler.

Scrivere un programma che ignori SIGUSR1 e resti in pausa fino a quando non riceve il segnale SIGUSR2