Segnali

I segnali costituiscono una forma (molto semplice) di comunicazione tra processi: tecnicamente sono interruzioni software causati da svariati eventi:

- Generati da terminale. Ad esempio il classico `ctrl-c` (`SIGINT`).
- Eccezioni dovute ad errori in esecuzione: es. divisione per 0, riferimento "sbagliato" in memoria, ecc…
- segnali esplicitamente inviati da un processo all'altro.
- eventi asincroni che vengono notificati ai processi: esempio `SIGALARM`.

Cosa possiamo fare quando arriva un segnale?

- Ignorarlo
- Gestirlo
- Lasciare il compito al gestore di sistema

Vediamo alcuni segnali POSIX (Portable Operating System Interface) supportati in Linux. La lista qui sotto è contenuta in `'man 7 signal'`

Vediamo alcuni segnali POSIX (Portable Operating System Interface) supportati in Linux. La lista qui sotto è contenuta in `'man 7 signal'`

```
First the signals described in the original POSIX.1-1990 standard.

Signal     Value     Action   Comment
----------------------------------------------------------------------
SIGHUP        1       Term    Hangup detected on controlling terminal
                              or death of controlling process
SIGINT        2       Term    Interrupt from keyboard     <== ctrl-C
SIGQUIT       3       Core    Quit from keyboard          <== ctrl-\
SIGILL        4       Core    Illegal Instruction
SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating point exception
SIGKILL       9       Term    Kill signal                 <== kill -9 (da shell)
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no readers
SIGALRM      14       Term    Timer signal from alarm(2)
SIGTERM      15       Term    Termination signal          <== kill (da shell)
SIGUSR1   30,10,16    Term    User-defined signal 1
SIGUSR2   31,12,17    Term    User-defined signal 2
SIGCHLD   20,17,18    Ign     Child stopped or terminated <== gestito da wait()
SIGCONT   19,18,25    Cont    Continue if stopped
SIGSTOP   17,19,23    Stop    Stop process
SIGTSTP   18,20,24    Stop    Stop typed at tty
SIGTTIN   21,21,26    Stop    tty input for background process
SIGTTOU   22,22,27    Stop    tty output for background process

The  signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

Azioni possibili:

```
The  entries  in  the  "Action"  column of the tables below specify the
default disposition for each signal, as follows:

Term   Default action is to terminate the process.

Ign    Default action is to ignore the signal.

Core   Default action is to terminate the process and dump core (see core(5)).

Stop   Default action is to stop the process.

Cont   Default action is to continue the process if it is currently stopped.
```

Un esempio semplice: alarm

`alarm` manda un `SIGALRM` dopo n secondi. Il default handler scrive "alarm clock" e poi termina il programma (serve per dare un timeout).

Vediamo un semplice esempio di programma che setta l'allarme dopo 3 secondi e poi va in loop infinito:

```
main()
{
        alarm(3);
        while(1){}
}
```

Il programma "punta una sveglia" dopo 3 secondi e attende in un ciclo infinito. Allo scadere dei 3 secondi viene inviato un segnale `SIGALRM` al processo. Il comportamento di default è quello di terminare il processo: se proviamo ad eseguire il programma osserviamo, infatti, che dopo 3 secondi il processo termina (viene anche scritto a terminale il motivo della terminazione).

```
$ a.out
<... dopo 3 secondi....>
Alarm clock
$
```

Impostare il gestore dei segnali tramite 'signal'

Tramite la system call `signal` è possibile cambiare il gestore dei segnali. La system call prende come parametri un segnale e una funzione che da quel momento diventerà il nuovo gestore del segnale. Vediamo un semplice esempio:

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

void alarmHandler()
{
        printf("questo me lo gestisco io!\n");
        alarm(3); // ri-setta il timer a 3 secondi
}

main() {
        signal(SIGALRM, alarmHandler);
        alarm(3);
        while(1){}
}
```

Dopo tre secondi arriva il segnale `SIGALRM`. Viene invocato `alarmHandler` che stampa a video una frase e reimposta l'allarme dopo 3 secondi. Il controllo ritorna al punto in cui il programma è stato interrotto (nel ciclo while quindi) e il programma attende altri 3 secondi che arrivi il successivo segnale:

```
questo me lo gestisco io!       <=== dopo 3 secondi
questo me lo gestisco io!       <=== dopo 3 secondi
questo me lo gestisco io!       <=== dopo 3 secondi
........
```

Parametri particolari e valore di ritorno

È possibile passare alla `signal` le costanti `SIG_IGN` o `SIG_DFL` al posto della funzione handler per indicare, rispettivamente:

- che il segnale va ignorato
- che l'handler è quello di default di sistema

Il valore di ritorno di `signal` è

- `SIG_ERR` in caso di errore
- l'handler precedente, in caso di successo

NOTA: `sigaction` rimpiazza `signal` con un'implementazione più stabile nelle varie versioni UNIX. Viene raccomandata se si vuole portabilità. Ad esempio, la `signal` originale UNIX, e la sua versione System V, fa il reset del gestore a `SIG_DFL` ogni volta che viene ricevuto il segnale. In Linux, si ottiene questo comportamento particolare compilando con l'opzione `--ansi`. Per l'uso di `sigaction` si faccia riferimento al manuale.

Esempio: Proteggersi da ctrl-c

Vediamo ora un altro esempio di gestione dei segnali. Se modifichiamo il gestore del segnale `SIGINT` possiamo evitare che un programma venga interrotto tramite ctrl-c da terminale.

```
#include<signal.h>
#include<stdio.h>
main() {
        void (*old)(int);

        old = signal(SIGINT,SIG_IGN);
        printf("Sono protetto!\n");
        sleep(3);

        signal(SIGINT,old);
        printf("Non sono più protetto!\n");
        sleep(3);
}
```

Notare l'uso del valore di ritorno della signal per reimpostare il gestore originale. La signal, quando va a buon fine, ritorna il gestore precedente del segnale, che salviamo nella variabile `old`. Quando vogliamo reimpostare tale gestore è sufficiente passare `old` come secondo parametro a `signal`.

Se eseguiamo il programma possiamo osservare che per 3 secondi ctrl-c non ha alcun effetto. Appena viene reimpostato il vecchio gestore, invece, ctrl-c interrompe il programma.

Se eseguiamo il programma possiamo osservare che per 3 secondi ctrl-c non ha alcun effetto. Appena viene reimpostato il vecchio gestore, invece, ctrl-c interrompe il programma.

```
> a.out
Sono protetto!
<ctrl-c>
<ctrl-c>
<ctrl-c>                <==== nessun effetto
Non sono più protetto!
<ctrl-c>                <==== esce!
>
```

Sospensione e ripristino di processi tramite kill

La chiamata a sistema `kill` manda un segnale a un processo.

In questo esempio mostriamo come la chiamata a sistema `kill` possa essere utilizzata per sospendere, ripristinare e terminare un processo.

In questo esempio mostriamo come la chiamata a sistema `kill` possa essere utilizzata per sospendere, ripristinare e terminare un processo.

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <signal.h>
main(){
        pid_t pid1,pid2;

        if ( (pid1 = fork()) < 0 ) {
                perror("errore fork"); exit(EXIT_FAILURE);
        } else if (pid1 == 0)
                while(1) { // primo figlio
                        printf("%d è vivo !\n",getpid());
                        sleep(1);
                        }

        if ( (pid2 = fork()) < 0 ) {
                perror("errore fork"); exit(EXIT_FAILURE);
        } else if (pid2 == 0)
                while(1) { // secondo figlio
                        printf("%d è vivo !\n",getpid());
                        sleep(1);
                        }
        // processo padre
        sleep(2);
        kill(pid1,SIGSTOP); // sospende il primo figlio
        sleep(5);
        kill(pid1,SIGCONT); // risveglia il primo figlio
        sleep(2);
        kill(pid1,SIGINT); // termina il primo figlio
        kill(pid2,SIGINT); // termina il secondo figlio
}
```

Quando eseguiamo il programma notiamo che il primo figlio viene sospeso per 5 secondi e che alla fine entrambi i processi figli sono terminati:

```
> a.out
6720 è vivo !
6721 è vivo !
6720 è vivo !
6721 è vivo !
6720 è vivo !
6721 è vivo !    <==== sospende 6720
6721 è vivo !
6721 è vivo !
6721 è vivo !
6721 è vivo !
6720 è vivo !    <==== risveglia 6720
6721 è vivo !
6720 è vivo !
6721 è vivo !
>
```

Mascherare i segnali

A volte risulta utile bloccare temporaneamente la ricezione dei segnali per poi riattivarli. Tali segnali non sono ignorati ma solamente 'posticipati'.

NOTA POSIX non specifica se più occorrenze dello stesso segnale debbano essere memorizzate (accodate) oppure no. Tipicamente se più segnali uguali vengono generati, solamente uno verrà "recapitato" quando il blocco viene tolto.

La chiamata a sistema `sigprocmask(int action, sigset_t *newmask, sigset_t *oldmask)` compie azioni differenti a seconda del valore del primo parametro `action`:

- `SIG_BLOCK`: l'insieme dei segnali `newmask` viene unito all'insieme dei segnali attualmente bloccati, che sono restituiti in `oldmask`;
- `SIG_UNBLOCK`: l'insieme dei segnali `newmask` viene sottratto dai segnali attualmente bloccati, sempre restituiti in `oldmask`;
- `SIG_SETMASK`: l'insieme dei segnali `newmask` sostituisce quello dei segnali attualmente bloccati (`oldmask`).

Per gestire gli insiemi di segnali (di tipo `sigset_t`) si utilizzano:

La chiamata a sistema `sigprocmask(int action, sigset_t *newmask, sigset_t *oldmask)` compie azioni differenti a seconda del valore del primo parametro `action`:

- `SIG_BLOCK`: l'insieme dei segnali `newmask` viene unito all'insieme dei segnali attualmente bloccati, che sono restituiti in `oldmask`;
- `SIG_UNBLOCK`: l'insieme dei segnali `newmask` viene sottratto dai segnali attualmente bloccati, sempre restituiti in `oldmask`;
- `SIG_SETMASK`: l'insieme dei segnali `newmask` sostituisce quello dei segnali attualmente bloccati (`oldmask`).

Per gestire gli insiemi di segnali (di tipo `sigset_t`) si utilizzano:

- `sigemptyset(sigset_t *set)` che inizializza l'insieme `set` all'insieme vuoto
- `sigaddset(sigset_t *set, int signum)` che aggiunge il segnale `signum` all'insieme set

L'esempio mostra come bloccare `SIGINT` e poi ripristinarlo.

```c
#include<signal.h>
#include<stdio.h>
#include<stdlib.h>
main() {
        sigset_t newmask,oldmask;

        sigemptyset(&newmask);          // insieme vuoto
        sigaddset(&newmask, SIGINT);    // aggiunge SIGINT alla "maschera"
        // setta la nuova maschera e memorizza la vecchia
        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
                perror("errore settaggio maschera"); exit(1); }

        printf("Sono protetto!\n");
        sleep(3);

        // reimposta la vecchia maschera
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
                perror("errore settaggio maschera"); exit(1); }

        printf("Non sono piu' protetto!\n");
        sleep(3);
}
```

Se digitiamo ctrl-c mentre il segnale è mascherato esso viene sospeso. Appena la maschera viene rimossa, il segnale è ricevuto dal processo che viene immediatamente interrotto. Non stiamo quindi modificando il gestore del segnale ma solo sospendendone la ricezione per un periodo di tempo.

```
> a.out
Sono protetto!
<ctrl-c>
<ctrl-c>
<ctrl-c>                <==== per 3 secondi nessun effetto
                        <==== esce appena la maschera viene tolta
>                               (senza dare ulteriori ctrl-c)!
```

Tramite `sigpending` è possibile ottenere l'insieme dei segnali "pendenti" (vedere il manuale per maggiori informazioni)

NOTA: `sigprocmask` non aderisce allo standard ANSI (ISO C99) ma solamente allo standard POSIX. La gestione dei segnali da parte di ANSI-C è molto povera e si riduce solamente a `signal`, `raise(sig)`, che equivale a `kill(getpid(),sig)`, e `abort()`, cioè "abnormal termination". ANSI-C non prevede multi-processing e quindi non prevede l'invio di segnali ad altri processi (sezione 7.14.2.1).

Attendere un segnale tramite 'pause'

Negli esempi abbiamo sempre usato `while(1){}` per attendere un segnale (*busy-waiting*). La system call `pause()` attende un segnale senza consumare tempo di CPU. Vediamo un esempio:

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

void alarmHandler()
{
        printf("questo me lo gestisco io!\n");
}

main()
{
        signal(SIGALRM, alarmHandler);
        alarm(3);
        pause();
        printf("termino!\n");
}
```

Interferenze e funzioni 'safe' POSIX

L'uso della `printf` nell'handler è rischioso perchè usa *dati globali*: Se anche il programma interrotto stava facendo I/O i due potrebbero interferire! printf non è "safe".

Facendo `man 7 signal` (in sistemi Linux recenti), troviamo la lista di funzioni safe che sicuramente **non** creano problemi di interferenza:

```
       _Exit(),  _exit(),  abort(),  accept(),  access(),  aio_error(),  aio-
       _return(),  aio_suspend(),  alarm(),  bind(),  cfgetispeed(),  cfget-
       ospeed(),  cfsetispeed(),  cfsetospeed(),  chdir(),  chmod(),  chown(),
       clock_gettime(), close(), connect(), creat(), dup(), dup2(), execle(),
       execve(),  fchmod(),  fchown(),  fcntl(),  fdatasync(),  fork(), fpath-
       conf(), fstat(), fsync(), ftruncate(), getegid(), geteuid(), getgid(),
       getgroups(),  getpeername(),  getpgrp(),  getpid(),  getppid(),  get-
       sockname(), getsockopt(), getuid(), kill(), link(), listen(), lseek(),
       lstat(),  mkdir(),  mkfifo(),  open(),  pathconf(),  pause(),  pipe(),
       poll(), posix_trace_event(), pselect(), raise(), read(), readlink(),
       recv(), recvfrom(), recvmsg(), rename(), rmdir(), select(), sem_post(),
       send(), sendmsg(), sendto(), setgid(), setpgid(), setsid(), setsock-
       opt(),  setuid(),  shutdown(),  sigaction(),  sigaddset(), sigdelset(),
       sigemptyset(), sigfillset(), sigismember(), signal(), sigpause(), sig-
       pending(),  sigprocmask(), sigqueue(), sigset(), sigsuspend(), sleep(),
       socket(),  socketpair(),  stat(),  symlink(),  sysconf(),  tcdrain(),
       tcflow(),  tcflush(),  tcgetattr(),  tcgetpgrp(), tcsendbreak(), tcset-
       attr(),  tcsetpgrp(),  time(),  timer_getoverrun(),  timer_gettime(),
       timer_settime(),  times(), umask(), uname(), unlink(), utime(), wait(),
       waitpid(), write().
```

Il seguente esempio cerca di far interferire le `printf` aggiungendo nel ciclo while la stampa di una stringa. Se il programma viene interrotto proprio durante la stampa, la printf del gestore potrebbe interferire con quella del programma.

Il seguente esempio cerca di far interferire le `printf` aggiungendo nel ciclo while la stampa di una stringa. Se il programma viene interrotto proprio durante la stampa, la printf del gestore potrebbe interferire con quella del programma.

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

void alarmHandler();    // gestore
static int i=0;         // contatore globale `volatile'

main() {
        signal(SIGALRM, alarmHandler);
        alarm(1);
        while(1){
                printf("prova\n");
        }
}
void alarmHandler()
{
        printf("questo me lo gestisco io %d!\n",i++);
        alarm(1);       // ri-setta il timer a 1 secondo
}
```

Tipicamente le stampe sono troncate o mischiate. In alcuni casi si possono addirittura perdere alcune `printf` (e/o alcuni a-capo) perché eseguite a metà di un'altra `printf`.

NOTA è necessario eseguire il comando con in coda " `| grep io`" per evitare di osservare tutte le stampe "prova". Il comando '`grep`' stampa solo le linee contenenti la stringa "io" (che compare in "questo me lo gestisco io"). la *pipe* '|' fa si che l'output del comando venga reindirizzato al comando successivo come input (è un modo per creare 'al volo' un canale di comunicazione *message passing* tra due processi, che approfondiremo la prossima volta).

NOTA è necessario eseguire il comando con in coda " | `grep io`" per evitare di osservare tutte le stampe "prova". Il comando '`grep`' stampa solo le linee contenenti la stringa "io" (che compare in "questo me lo gestisco io"). la *pipe* '|' fa si che l'output del comando venga reindirizzato al comando successivo come input (è un modo per creare 'al volo' un canale di comunicazione *message passing* tra due processi, che approfondiremo la prossima volta).

```
> a.out | grep io
questo me lo gestisco io 0!
questo me lo gestisco io 2!
questo me lo gestisco io 4!
questo me lo gestisco io 6!
questo me lo gestisco io 7!
questo me lo gestisco io 8!
questo me lo gestisco io 12!
questo me lo gestisco io 14!
questo me lo gestisco io 15!
questo me lo gestisco io 16!
questo me lo gestisco io 17!
provaquesto me lo gestisco io 18!
provaquesto me lo gestisco io 19!
```