

Sincronizzazione tra processi: I Segnali

I segnali sono interrupt software, che segnalano il verificarsi di **eventi asincroni**:

- *generati da terminale* (combinazioni di tasti, es. ctrl-C)
- *eccezioni hardware* (divisioni per 0, accessi non autorizzati in memoria, ecc.)
- *spedizione esplicita di segnali* (tramite kill)
- *condizioni software* (scrittura su pipe chiusa, alarm)

Quando un processo riceve un segnale può:

- *ignorarlo*
- eseguire una qualche *azione* definita dall'utente
- mettere in atto una azione di *default*

Attenzione: i segnali hanno tutti la **stessa priorità**

Esempi

- 1) # programma
.....
 <CTRL>-C
segnale SIGINT al processo
azione default → terminazione
- 2) # programma
.....
Illegal instruction - 
core dumped
programma (corrotto)
ha eseguito una
istruzione illegale
segnale SIGILL al processo
azione default → terminazione
con produzione di un file core
core → immagine del processo
- 3) \$ programma &
<PID>
\$...
\$ kill <PID>  segnale SIGTERM al processo
azione default → terminazione

<PID> Terminated
- 4) \$ programma &
<PID>
\$ kill -9 <PID>  segnale SIGKILL al processo
(non ignorabile, né gestibile)
azione default → terminazione

<PID> Terminated

Elenco dei Segnali (file /usr/include/signal.h)

Name	Value	Default	Event
SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt
SIGQUIT	3	Core	Quit
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal1
SIGUSR2	17	Exit	User Signal2
SIGCHLD	18	Ignore	Child Status Changed
SIGPWR	19	Ignore	Power Fail/Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	21	Ignore	Urgent Socket Condition
SIGPOLL	22	Exit	Pollable Event
SIGSTOP	23	Stop	Stopped
SIGTSTP	24	Stop	Stopped
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped(tty input)
SIGTTOU	27	Stop	Stopped(tty output)
SIGPROF	29	Exit	Profiling
SIGXCPU	30	Core	CPU time limit exceeded
SIGXFSZ	31	Core	File size limit exceeded
SIGLWP	33	Ignore	Inter-LWP signal reserved

Attenzione: dipende dalle macchine (vedere diff. BSD System V)

Azioni al ricevimento di un segnale

Quando riceve un segnale, un processo può avere diversi comportamenti:

- *ignorare* il segnale (solo 2 segnali NON possono essere ignorati, il SIGKILL e il SIGSTOP)
- *gestire* il segnale (il processo può definire una funzione da mettere in esecuzione nel momento che riceve un determinato segnale)
- mettere in atto l'azione di *default* (ogni segnale ha una azione di default, es. la terminazione del processo)

Come specificare l'azione?

```
SIGNAL          #include <signal.h>
                  void (*signal (int sig, void (*func)(int)))(int);
```

definisce il comportamento del processo al ricevimento del segnale. Specifica il segnale (**sig**) e il suo trattamento (**func**). Restituisce il trattamento precedente del segnale sig oppure -1.

- per *ignorare* il segnale → SIG_IGN
- per *gestire* segnale attraverso handler → void (*func)()
- per riportare all'azione di *default* → SIG_DFL

Notare che:

```
#define SIG_DFL    void (*)() 0
#define SIG_IGN    void (*)() 1
#define SIG_ERR    void (*)() -1
```

Esempi

1) *ignorare* un segnale

```
....
signal(SIGUSR1,SIG_IGN);
.... Da qui in avanti il processo ignora il SIGUSR1
```

2) definire un *gestore* del segnale

```
....
void handler(segnale)
int segnale;
{
....
printf("Ricevuto segnale %d \n", segnale);
..../* Qui ci può essere una exit(), oppure il
gestore restituisce il controllo al chiamante*/
}

main()
....
signal(SIGINT, handler);
.... Da qui in avanti il processo mette in
esecuzione la funzione handler ogni volta che
viene premuto il CTRL-C da tastiera
```

3) ripristinare l'azione di *default* per il segnale

```
....
signal(SIGINT, handler);
....
signal(SIGINT, SIG_DFL);
.... Da qui in avanti il processo ripristina
l'azione di default per il CTRL-C da tastiera
```

Esempio: gestore del SIGCHLD

```
#include <sys/errno.h>
#include <fcntl.h>
#include <signal.h>

extern int errno;

void handler(signo)
int signo;
{
    int stato, salva_errno;
    salva_errno=errno;
    perror("handler perror inizio");
    psignal(signo,"handler psignal");
    wait(&stato);
    printf("handler stato uscita figlio%d\n",stato>>8);
    open("file_non_esiste", O_WRONLY);
    perror("handler perror fine");
    errno=salva_errno;
}

main(argc, argv)
int argc;
char *argv[];
{
    int pid, rit,i;
    pid=fork();
    if(pid>0){ /* codice eseguito dal padre */
        signal(SIGCHLD,handler);
        signal(SIGKILL,handler);/*ERRORE!!!
                                SIGKILL non gestibile*/
        perror("padre perror inizio");
        for(i=1;i<10000000;i++);
        perror("padre perror fine");
        exit(0);
    } else { /* codice eseguito dal figlio */
        sleep(1); exit(1);
    }
}
```

Segnali e fork

Un processo figlio eredita tutta la disposizione dei segnali del padre:

- *ignora* gli stessi segnali ignorati dal padre
- *gestisce* nello stesso modo i segnali gestiti dal padre
- stessa azione di *default* del padre

Segnali ed exec

exec cambia la disposizione dei segnali:

- segnali *gestiti* prima di **exec** ritornano alla condizione di *default*
- processo dopo **exec** *ignora* gli stessi segnali ignorati prima

Esempio di comando in background

```
$ sleep(10) &  
shell:  
...  
fork();  
...  
signal(SIGINT, SIG_IGN);  
signal(SIGQUIT, SIG_IGN);  
exec("/usr/bin/sleep", "sleep", ...);
```

Segnali unreliable

Molte implementazioni UNIX hanno segnali non affidabili (unreliable)

Problemi:

- ➔ **segnale può essere perso** e non ricevuto da un processo, esempio:
se più occorrenze “contemporanee” di uno **stesso** segnale sono dirette a **un** processo, tale processo ne riceve solo **una**, le altre NON sono registrate
- ➔ **scarso controllo sulla gestione** dei segnali, esempio:
gestore segnale non rimane installato
- ➔ possibile **innestamento** della routine di gestione del segnale (stack)
- ➔ possibile **interruzione delle system call**

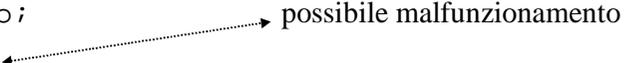
Caso System V

- gestore segnale non rimane installato (riportato a default)
- anche nel caso si riinstalli possibile malfunzionamento
- non blocca il segnale durante la sua gestione
- possibile innestamento multiplo delle routine di gestione

Esempio:

```
handler(signo) /* routine gestione segnale*/
int signo;
{
    signal(SIGINT, handler);
    printf ("handler: signo=%d\n", signo);
}

main()
...
signal (SIGINT, handler); /*aggancia segnale*/
...
```



Caso BSD

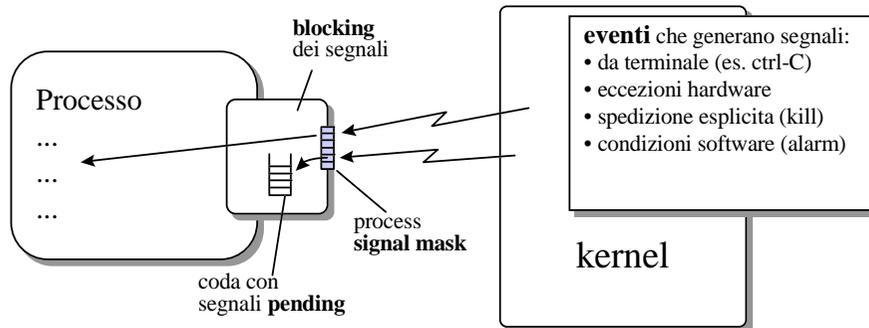
- gestore segnale rimane installato
- durante la routine di gestione del segnale, il segnale stesso è bloccato (cioè bufferizzato)
- occorrenze “contemporanee” di uno **stesso** segnale → processo ne riceve solo **una**

Esempio:

```
handler(signo) /* routine gestione segnale*/
int signo;
{
    printf ("handler: signo=%d\n", signo);
}

main()
...
signal (SIGINT, handler); /*aggancia segnale*/
...
```

Modello reliable segnali



- occorrenze “contemporanee” di uno **stesso** segnale sono memorizzate:
 - ➔ processo le riceve **tutte** (1 contatore per ogni segnale)
 - ➔ non si perdono segnali
- è possibile **bloccare** un segnale diretto a un processo P, e gestirlo successivamente ➔ regioni critiche

I segnali in attesa sono detti **pending**

La **process signal mask** definisce quali segnali bloccare per un processo P (i segnali sono messi in attesa)

Segnali in Solaris 2.5 (System V)

Problema della **signal()**:

- gestione dei segnali non rimane installata (ripristina gestione di default)
- non blocca il segnale durante la sua gestione

Invece di **signal()** utilizzare **sigset**:

```
SIGSET #include <signal.h>  
void (*sigset (int sig, void (*func)(int)))(int);
```

Come la **signal**, specifica come deve essere gestito il segnale sig. Ma con **sigset**:

- gestore segnale rimane installato
- il kernel blocca il segnale sig mentre il processo lo sta servendo (aggiunge sig alla signal mask del processo per il tempo di gestione del segnale)
- occorrenze “contemporanee” di uno **stesso** segnale ➔ processo ne riceve solo **una**

Ritorna il trattamento precedente del segnale oppure **SIG_ERR** (-1)

Esempio Solaris 2.5

Il processo definisce un gestore per il segnale **CTRL-C**; ad ogni CTRL-C ricevuto produce una stampa a video

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

void gestore(segnale)
int segnale;
{
int i;
/*signal(SIGINT,gestore);*/
printf("Ricevuto segnale %d\n", segnale);
sleep(2);
}

main(argc,argv)
int argc;
char **argv;
{
int i;
/* signal(SIGINT,gestore); */
sigset(SIGINT,gestore);
for(i=1;i<10000000;i++);
}
```

Provare ad utilizzare:

1. sigset
2. signal solo nel main
3. signal sia nel main sia nel gestore

Interrupted System Call

Alcune system call possono essere molto lente o bloccarsi (I/O su terminali o via rete, non su file). E' possibile interrompere le system call.

System call interrotte da un segnale ritornano **-1**
ed **errno** vale **EINTR**

Comportamento delle system call interrotte varia a seconda della implementazione:

- applicazione deve esplicitamente trattare il caso

```
do {
    read(0, buf, 20)
} while (errno==EINTR)
```
- automatic restart delle system call
- scelta esplicita del comportamento per ogni segnale

Altre Primitive

Sospensione temporizzata

```
SLEEP      #include <unistd.h>
              unsigned int sleep (numsecondi);
              unsigned int numsecondi;
```

Sospende il processo per numsecondi.

Il processo può **ripartire** prima di numsecondi se riceve un **segnale**.

Installazione di un allarme

```
ALARM      #include <unistd.h>
              unsigned int alarm (numsecondi);
              unsigned int numsecondi;
```

Fa partire un **timer**. Dopo numsecondi il processo riceve un **SIGALRM**.

C'è un solo timer per processo: nel caso di due chiamate successive il valore della seconda sovrascrive il primo.

Sospensione in attesa di un qualunque segnale

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

Sospende il processo chiamante fino al ricevimento di un qualunque segnale → ritorna **sempre -1** ed **errno=EINTR**

Esempio: Uso di alarm e pause (sleep)

```
#include <stdio.h>
#include <signal.h>
#define TRUE 1
#define FALSE 0
#define BELLS "\007\007\007"

int alarm_flag = FALSE; /* variabile globale */

void setflag()          /* handler per SIGALRM */
{
    alarm_flag = TRUE; /*quando si verifica
                        l'allarme si setta questa variabile */
}

main (argc, argv)
int argc;
char *argv[];
{
    int nsecs;

    if(argc!=2){
        printf("Errore numero parametri\n");
        exit(1);
    }
    nsecs = atoi(argv[1]);

    signal(SIGALRM, setflag);

    alarm(nsecs); /* in pratica, una sleep .... */
    pause();

    if (alarm_flag == TRUE) {
        printf(BELLS);
        printf("\n");
    }
    exit(0);
}
```

Altre Primitive

Invio di segnali

```
KILL      #include <signal.h>
           #include <sys/types.h>
```

```
int kill (pid, signo);
pid_t pid;
int signo;
```

kill invia il segnale signo ad un processo oppure ad un gruppo di processi.

Quattro casi a seconda del valore di pid:

- **pid == 0** il segnale viene mandato a tutti i processi che appartengono allo stesso gruppo del processo che manda il segnale.
- **pid == -1** e l'effettivo **user-id** del processo non è super-user, il segnale è mandato a tutti i processi con user-id reale uguale all'effettivo user-id del processo che invia il segnale.
- **pid == -1** e l'effettivo **user-id** del processo è super-user, il segnale è mandato a tutti i processi (escluso alcuni processi di sistema).
- **pid < -1** il segnale è inviato a tutti i processi il cui **process group-id** è uguale in valore assoluto a pid (eventualmente incluso il processo che invia il segnale).

Esempio: padre e figlio si scambiano il controllo alternativamente, utilizzando kill e pause

```
#include<signal.h>
#define BELL "\007\007\007\007"
int ntimes=0; /*contatore, uno per ogni processo*/
void handler(); /*gestore di SIGUSR1 */
main ()
{ int pid, ppid;

  signal(SIGUSR1,handler); /*aggancio di SIGUSR1 */
  if ((pid=fork())<0) {exit (1);}

  if (pid == 0) /* figlio */
  { /*installazione handler viene ereditata */
    ppid=getppid (); /* PID del padre */
    printf("Figlio %d di padre %d\n",getpid(),ppid);
    for (;;) {
      printf("Figlio \n");
      sleep (1);
      kill (ppid, SIGUSR1); /* invio al padre */
      pause (); /* attesa del segnale */
    }
  } else { /* Padre*/
    printf("Padre %d figlio %d\n",getpid(),pid);
    for (;;) {
      printf ("Padre\n");
      pause (); /* attesa iniziale */
      sleep (1);
      kill (pid, SIGUSR1); /* invio al figlio */
    } }

  void handler() {
    printf("Proc %d, ricevuto segnale %d volte\n",
           getpid(), ++ntimes);
    printf(BELL);}
}
```

Che modifiche per **UNIX System V** ?

Funzioni rientranti

Un segnale interrompe un processo in modo asincrono, e dentro il gestore del segnale possono essere presenti qualunque funzione o System Call.

Problema: interrompere alcune funzioni può generare delle inconsistenze.

Esempio:

processo: malloc  **handler: malloc**

la malloc gestisce gli spazi di memoria allocati attraverso una lista linkata → modifiche contemporanee possono produrre situazioni inconsistenti.

Le funzioni che si possono usare senza problemi all'interno delle routine di gestione dei segnali sono dette funzioni rientranti.

Funzioni non rientranti
(malloc e funzioni della
standard I/O library)

Sono regioni di codice
critiche che possono
richiedere il **blocco** dei
segnali

Corse Critiche

Corsa critica: situazione in cui più processi modificano dei dati condivisi. ma l'esito finale dipende dall'**ordine relativo** con cui sono eseguite le operazioni dai processi.

Molte situazioni di corse critiche si possono risolvere utilizzando dei meccanismi di sincronizzazione e comunicazione tra i processi tipo segnali, pipe, etc.

Anche nel **caso dei segnali** attenzione:

Esempio: alarm e pause (sleep)

```
....  
alarm(nsec);  
pause();  
....
```

Corsa critica: il SIGALRM potrebbe arrivare al processo PRIMA della pause () → blocco del processo (in dipendenza dal carico della macchina, dal numero nsec di secondi, etc.)

Segnali in Solaris 2.5 (System V)

Per **bloccare** i segnali:

```
SIGHOLD      #include <signal.h>
               int sighold (int sig);
```

Aggiunge il segnale sig alla process signal mask.

Il segnale sig diretto al processo è **bloccato**, cioè bufferizzato.

Un sig bloccato raggiungerà il processo quando sbloccato tramite sigrelse() oppure sigpause()

Per **sbloccare** i segnali:

```
SIGRELSE     #include <signal.h>
               int sigrelse (int sig);
```

Rimuove il segnale sig dalla process signal mask

```
SIGPAUSE     #include <signal.h>
               int sigpause (int sig);
```

Rimuove il segnale sig dalla process signal mask e sospende il processo (equivale a sigrelse + pause)

Esempio di corsa critica

```
.....
alarm(nsec);
pause( );
.....
```

il SIGALRM potrebbe arrivare al processo

Soluzione senza corsa critica

```
.....
sighold(SIGALRM);
alarm(secs);
sigpause(SIGALRM);
.....
```

Esempio: padre e figlio si scambiano il controllo alternativamente, utilizzando kill e pause

Figlio		Padre
...		...
kill (ppid, SIGUSR1);	→	pause ();
sleep (3);	←	sleep (1);
pause ();	←	kill (pid, SIGUSR1);
...		...

La sleep(3) nel figlio simula una condizione in cui si verifica una corsa critica, figlio riceve SIGUSR1 dal padre e DOPO va in pause (blocco)

Soluzione che evita la corsa critica (Solaris 2.5)

```
sigset(SIGUSR1, handler); /*aggancio di SIGUSR1 */
if ((pid=fork())==0) { /* figlio */
    sighold(SIGUSR1);
    .....
    printf("Figlio \n");
    kill (ppid, SIGUSR1); /* invio al padre */
    sleep(3);
    sigpause(SIGUSR1); /*attesa del segnale*/
} else { /* Padre*/
    sighold(SIGUSR1);
    .....
    printf ("Padre\n");
    sigpause(SIGUSR1); /*attesa del segnale*/
    kill (pid, SIGUSR1); /* invio al figlio */
    .....
}
```

Per qualunque numero di secondi di sleep non si verificano corse

Signal Process Mask

Esempio che stampa una parte della signal process mask

```
#include <signal.h>

main(argc, argv)
int argc;
char **argv;
{
    sigset_t      sigset;

    sighold(SIGQUIT); /*per esempio, blocco SIGQUIT*/

    if(sigprocmask(0, NULL, &sigset)<0)
        perror("sigprocmask");

    if(sigismember(&sigset, SIGINT))
        printf("SIGINT\n");
    if(sigismember(&sigset, SIGQUIT))
        printf("SIGQUIT\n");
    if(sigismember(&sigset, SIGALRM))
        printf("SIGALRM\n");
}
```

Esempio

padre-figlio-nipote, padre non si blocca su wait,
nipote ereditato da init → se muore non zombie

```
#include .....

main(argc, argv)
int argc;
char *argv[];
{
    pid_t pid;
    int stato;

    if ((pid=fork())<0) {perror("Fork"); exit(1);}
    if(pid==0){ /* Figlio */
        if((pid=fork())<0) {perror("Fork");exit(1);}
        if(pid==0){ /* Nipote */
            sleep(2);
            printf("nipote %d, figlio di %d\n",
                getpid(), getppid());
            exit(0);
        } else { /* Figlio */
            printf("figlio %d, son figlio di %d\n",
                getpid(), getppid());
            exit(0); /* figlio termina*/
        }
    } else { /* Padre */
        printf("padre %d\n", getpid());
        if(wait(&stato)!= pid) { printf("errore\n");
            exit(0); /* padre termina*/
        }
    }
}
```