

SEGNALI >

Sono delle notifiche inviate ai processi per notificare eventi.

Sono importabili con `<signal.h>` o `<sys/types.h>` come interi usando l'enum `SIGXXXX` (es: `SIGINT => CRTL + c`)

Si può mostrare una

descrizione con `strsignal()`:

```
// NSIG is a macro equal to the total number of signal  
for (i=0; i<NSIG; i++) {  
    printf("Signal %2d: %s\n", i, strsignal(i));  
}
```

CICLO DI VITA

Il ciclo di vita dei segnali è:

- generated: Il segnale è stato appena creato
- pending : Il momento intermedio tra creazione e consegna.
- delivered: Il segnale è stato consegnato al processo che lo eseguirà.

MASCHERE DI SEGNALI

I segnali in pending solitamente sono recapitati appena il processo è in esecuzione.

Se il processo non vuole essere interrotto, può registrare il segnale nella **maschera dei segnali** che blocca le ricezioni.

PROCESSAMENTO DEI SEGNALI

Quando il processo riceve un segnale può succedere che:

- Il segnale è ignorato: perché scartato dal kernel.
- Il processo viene terminato:
Per terminazione anomala (es: div. per 0, Istr. macchina anom)
- core dump + termination:
Viene salvata un immagine della memoria virtuale e poi viene terminato il processo.
- blocked:
esecuzione proc sospesa (es: per timer)

• resumed:

ESECUZIONE PROC. RIPRESA

GESTIONE SEGNALI SU UNIX

Su Unix i segnali sono gestiti con

- Interrupts: da eventi esterni
- Trap : da eventi interni

Il sistema di segnali in Unix: le trap

- Una classe di segnali sono le **trap**: segnali generati da eventi prodotti da un processo e inviati al processo stesso.
- Alcune **trap** sono cause di comportamenti errati del processo stesso, e immediatamente inviate al processo che normalmente reagisce terminando.
- per esempio tentativi di divisione per zero (**SIGFPE**), indirizzamento errato degli array (**SIGSEGV**), tentativo di eseguire istruzioni privilegiate (**SIGILL**), etc.

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

16

Il sistema di segnali in Unix: gli interrupt

- Gli interrupt sono segnali inviati ad un processo da un agente esterno: l'utente o un altro processo
- Utente (da terminale):
 - **CTRL-C** (invia **SIGINT**)
 - **CTRL-Z** (invia **SIGSTOP**)
 - Comando **kill -s name PID**
 - Comando **kill -l -s number PID**
- Altro processo:
 - System call **kill(kill(PID, SIGNAL))**

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

17

SIGNAL DISPOSITION >

I programmi possono modificare l'azione da fare al momento della delivery del segnale in:

- default action : per ripristinare il default
- ignore signal
- exec signal handler :

Si comunica al kernel di eseguire una funzione custom.

Se si usa, si dice che il segnale è caught o handled.

6 SIGABRT create core image abort program (formerly SIGIOT)
14 SIGALRM terminate process real-time timer expired

- **SIGABRT**. Un processo riceve questo segnale quando invoca la funzione **abort()**. Di default questo segnale termina il processo con un core dump. Questo produce l'effetto della chiamata **abort()**, che produce un core dump a fini di debug.

- **SIGALRM**. Il kernel genera questo segnale al momento del raggiungimento dello zero di un timer impostato da una chiamata ad **alarm()** o **setitimer()**.

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

20 SIGCHLD discard signal child status has changed
19 SIGCONT discard signal continue after stop

- **SIGCHLD**. Segnale inviato dal kernel a un processo genitore quando uno dei figli termina (chiamando **exit()**, o uscito da un qualche segnale). Può essere inviato a un processo quando uno dei suoi figli è bloccato o risvegliato da un segnale.

- **SIGCONT**. Quando viene inviato a un processo bloccato (**stopped**), questo segnale causa il risveglio del processo (**resume**), cioè che il processo venga schedulato per successivamente essere eseguito. Quando è ricevuto da un processo che non è bloccato, questo segnale è ignorato di default. Un processo può intercettarlo questo segnale, in modo da eseguire qualche azione particolare al momento della ripresa dell'esecuzione.

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

2 SIGINT terminate process interrupt program
9 SIGKILL terminate process kill program
13 SIGPIPE terminate process write on a pipe with no reader

- **SIGINT**. Quanto l'utente digita il carattere di interrupt (**Control-C**), il terminale invia questo segnale al gruppo del processo in **foreground**. L'azione di default per questo segnale è terminare il processo.

- **SIGKILL**. È il segnale sicuro di **kill**. Non può essere bloccato, ignorato, o intercettato da un **handler**; e quindi termina sempre un processo.

- **SIGPIPE**. Segnale generato quando un processo tenta di scrivere su un **pipe** o un **FIFO** per il quale non c'è un corrispondente processo lettore. Questo normalmente occorre perché il **processo lettore ha chiuso il proprio file descriptor** per il canale **IPC**.

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

11 SIGSEGV create core image segmentation violation

- **SIGSEGV**. Segnale generato quando un programma tenta un riferimento in memoria non valido. Il riferimento può non essere valido perché la pagina riferita non esiste (per esempio, giace in un'area non mappata, fra lo heap e lo stack), oppure il processo ha tentato di modificare una locazione in **read-only memory** (il segmento di testo del programma o una regione di memoria marcata come disponibili in sola lettura), o il processo ha tentato di accedere a una parte della memoria del kernel durante l'esecuzione in **user mode**.

- In C, questi eventi spesso derivano dalla dereferenziazione di un puntatore che contiene un 'bad address' (come un puntatore non initializzato) o dal passaggio di un argomento non valido in una chiamata a funzione.

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

5 SIGTRAP create core image trace trap
18 SIGSTP stop process stop signal generated from keyboard

- **SIGTRAP**. Segnale utilizzato per implementare i breakpoint in fase di debugging e per la tracciatura delle system call.

- Cercare **ptrace()** sul manuale per ulteriori informazioni.

- **SIGSTP**. Segnale per lo stop, inviato per bloccare il gruppo di processi in foreground quando l'utente digita il carattere di sospensione (**Control-Z**) sulla tastiera.

- il nome di questo segnale deriva da "terminal stop".

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

30 SIGUSR1 terminate process User defined signal 1
31 SIGUSR2 terminate process User defined signal 2

- **SIGUSR1**. Questo segnale e **SIGUSR2** sono disponibili per fini specificati dal programmatore. Il kernel non genera mai questi segnali per un processo.

- I processi possono utilizzare questi segnali per notificarsi a vicenda eventi, o per sincronizzarsi.

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

SIGNAL DISPOSITION UNIX

Si possono usare:

- **signal**

Più semplice ma può verificare il comportamento

```
#include <signal.h>
void (*signal(int sig, void (*handler)(int)) ) (int);
    Returns previous signal disposition on success, or
        SIG_ERR on error
```

- Il primo argomento, **sig**, identifica il segnale di cui vogliamo modificare la disposizione.
- Il secondo argomento, **handler**, è l'**indirizzo** della funzione che dovrebbe essere chiamata quando questo segnale è inviato (e **consegnato, delivered**).
- Questa funzione non restituisce (**è void**) e prende un intero. Quindi un handler di segnali ha la seguente forma generale:

```
void handler(int sig) {
    /* codice dell'handler */
```



- Il valore di ritorno di **signal()** è la precedente disposizione del segnale. Come l'argomento dell'handler, questo è un **puntatore a funzione** che non restituisce nulla e che prende un intero come argomento.

- Il seguente blocco di codice stabilisce un handler temporaneo per un segnale, e quindi setta nuovamente la disposizione del segnale come era impostata prima:

```
void (*oldHandler)(int);
oldHandler = signal(SIGINT, newHandler);
if (oldHandler == SIG_ERR)
    errExit("Signal error!");

/* altre istruzioni ...
   se durante l'esecuzione arriva un SIGINT,
   la sua gestione sarà affidata al newHandler */

if (signal(SIGINT, oldHandler) == SIG_ERR)
    errExit("Signal error!");
```

2 disposizioni particolari

- Se la disposizione è settata a **SIG_IGN**, il segnale è ignorato.
- Se la disposizione è settata a **SIG_DFL**, l'azione di default è associata con il segnale.

signal(SIGTERM, SIG_DFL);



Daniele Radocci - Laboratorio di Sistemi Operativi, corso B - turno T3

33

• **sigaction**

Sembra più verbosa ma più portabile

ESEMPI DI SIGNAL DISPOSITION

```
...  
#include <signal.h>  
  
static void dd_signal_handler(int sig) {  
    printf("ahiaaaaa!\n");  
}  
  
int main(int argc, char *argv[]) {  
    int j;  
  
    if (signal(SIGINT, dd_signal_handler) == SIG_ERR)  
        errExit("signal (SIGINT) error");  
  
    for (j = 0; ; j++) {  
        printf("%d\n", j);  
        sleep(1);  
    }  
}
```



Daniele Radocci - Laboratorio di Sistemi Operativi, corso B - turno T3

59

```
Ctrl+C  
... ./prova_segnali  
#include <signal.h>  
static void dd_signal_handler(int sig) {  
    printf("ahiaaaaa!\n");  
}  
  
int main(int argc, char *argv[]) {  
    int j;  
  
    if (signal(SIGINT, dd_signal_handler) == SIG_ERR)  
        errExit("signal (SIGINT) error");  
  
    for (j = 0; ; j++) {  
        printf("%d\n", j);  
        sleep(1);  
    }  
}
```



Daniele Radocci - Laboratorio di Sistemi Operativi, corso B - turno T3

60

persistenza dell'associazione

- Nei sistemi in cui l'associazione con il nuovo handler vale per un'unica ricezione del segnale, se si vuole rendere persistente l'associazione, bisogna ripeterla a ogni invocazione dell'handler.

```
void handler(int s) {  
    signal(SIGTERM, handler_int);  
    printf("\n ricevuto segnale SIGTERM \n");  
}
```

Daniele Radocci - Laboratorio di Sistemi Operativi, corso B - turno T3

64

KILL > DA CODICE Proc. Segnale (mau ≠ kill)

L'istruzione **kill(pid_t, int)** consente di inviare un segnale ad un processo.

1° ARGOMENTO

L'argomento **pid_t** definisce il/i processi target. può essere

- >0: Prende il PID con quel valore
- =0: Invia ad ogni processo dello stesso gruppo del chiamante includendo il chiamante.
- =-1: Invia a tutti i processi ai quali il processo ha permesso.
- <-1: Invia ai processi del gruppo che hanno ID uguale al valore assoluto.

2° ARGOMENTO

È il segnale da inviare.

Si può anche impostare

O (null signal) per verificare

l'invio.

kill() system call

```
int main() {
    printf("process with pid %d\n", (long) getpid());
    printf("going to sleep for 3 secs\n"
           "and then SIGKILLing...\n");
    sleep(3);

    // broadcast signal il segnale è inviato a tutti i
    // processi per i quali il processo ha i permessi di
    // inviare un segnale, eccetto init (che ha pid 1) ed
    // il chiamante. Se l'utente non è super user, il
    // segnale è inviato a tutti i processi con stesso
    // uid dell'utente, escluso il processo che invia il
    // segnale.
    kill(-1, SIGKILL);
}
```

Daniele Radicioni - Laboratorio di Sistemi Operativi, corso B - turno T3

#include <signal.h>
int kill(pid_t pid, int sig);
Returns 0 on success, or -1 on error
• Se nessun processo corrisponde al **pid** predefinito, **kill()** fallisce e setta errore a **ESRCH** ("No such process")
• Verifica dell'esistenza di un processo. Se l'argomento **sig** è settato a 0 (detto **null signal**), non è inviato alcun segnale.
- In questo caso **kill()** esegue unicamente un controllo degli errori per vedere se è possibile inviare segnali al processo: il **null signal** può essere utilizzato per testare se un processo esiste.
- Se la chiamata fallisce con errore **EPERM**, il processo esiste, ma non abbiamo i permessi per inviargli un segnale.
Se la chiamata va a buon fine, sappiamo che il processo esiste.

69

SIGACTION NEL DETTAGLIO >

È l'alternativa a **signal** per gestire la **signal disposition**.

sigaction() system call

```
#include <signal.h>

int sigaction(int signum,
             const struct sigaction *act,
             struct sigaction *oldact);

Returns 0 if successful; otherwise the value -1 is
returned and the global variable errno is set.
```

- syscall utilizzata per impostare un gestore di segnali
- **signum**: numero del segnale da gestire
- **act**: puntatore al nuovo gestore del segnale; se NULL il gestore resta invariato
- **oldact**: puntatore al vecchio gestore; se impostato a NULL non viene restituito alcun handler.

• NB: **sigaction** è sia una sys call sia una struct

Oltre al **signal**, prende come parametri dei puntatori alla struttura **sigaction** per poter

- ***act**: definire il nuovo comportamento
- ***oldact**: salvare in un puntatore il comportamento "vecchio"

sblocco del segnale durante l'handler

- quando un segnale è inviato a un processo durante l'esecuzione di un handler, il segnale è automaticamente bloccato fino a quando l'handler restituisce
- a meno che non sia settato il flag **SA_NODEFER**

```
void handle_signal(int signal);

int main() {
    struct sigaction sa;
    sigset(SIG_BLOCK, my_mask); // funct ptr
    sa.sa_handler = handle_signal;
    sa.sa_mask = my_mask;
    sa.sa_flags = SA_NODEFER; // allow nested invocations
    sigemptyset(&my_mask); // do not mask any signals;
    sa.sa_mask = my_mask;
    sigaction(SIGUSR1, &sa, NULL); // set the handler
    // further code
}
```

sigaction() system call

• (dichiarazione e) tre possibili invocazioni

```
struct sigaction new, old;

sigaction(signum, new, NULL); // set new handler to new
sigaction(signum, NULL, old); // current handler in old
sigaction(signum, new, old); // do both
```

the sigaction structure

```
#include <signal.h>

struct sigaction {
    void (*sa_handler)(int signum);
    sigset(SIG_BLOCK, my_mask);
    int sa_flags;
    // plus others (for advanced usage)
};

• sa_handler, puntatore alla funzione per la gestione del segnale;
• sa_mask, maschera dei segnali bloccati durante l'esecuzione dell'handler;
• sa_flags, insieme di flag messi in bitwse OR per modificare il comportamento del segnale.
```

MASCHERA DEL SEGNALI

I processi ereditano la maschera dai parent. Sono di tipo **sigset(SIG_BLOCK)** e per convenzione vengono gestiti con gli appositi metodi.

signal mask

```
int sigemptyset(sigset_t *set);
    initializes a signal set to be empty.

int sigfillset(sigset_t *set);
    initializes a signal set to contain all signals.

int sigaddset(sigset_t *set, int signum);
    adds the specified signal signum to the signal set.

int sigdelset(sigset_t *set, int signum);
    deletes the specified signal signum from the signal set.

int sigismember(const sigset_t *set, int signum);
    returns whether a specified signal signum is contained in the signal set.
```

```
void handle_signal(int signal) {
    printf("Got signal #%-d, signal, strsignal(signal));
}

int main()
{
    struct sigaction sa;
    sigset(SIG_BLOCK, my_mask);
    int i;

    sa.sa_handler = handle_signal;
    sigemptyset(&my_mask); // signal mask is now EMPTY
    sa.sa_mask = my_mask;
    sa.sa_flags = 0; // no special behavior

    for (i=0; i<SIGSTG; ++i) { // set the handler for all signals
        if (sigaction(i, &sa, NULL) == -1) {
            perror("Error setting up handler for signal %d: %s", i, strerror(errno));
        }
    }

    for (i;i) {
        print("Sleeping for 3 seconds\n");
        sleep(3);
    }
}
```

80

Per inizializzare la mask si può

- Partire da **sigemptyset** e aggiungere
- Partire da **sigfillset** e rimuovere.

Si può anche impostare la signal mask durante l'esecuzione

N.B: Gli handler

reimpresano la

maschera dopo l'esec.

setting the signal mask for a process

```
#include <signal.h>

int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *oldset);
    Returns 0 if successful; otherwise the value -1 is returned
    • Per impostare la maschera di bloccaggio durante l'esecuzione dei processi si usa la syscall sigprocmask();
    • l'argomento how può assumere i seguenti valori:
      - SIG_BLOCK: segnali nel set sono bloccati;
      - SIG_UNBLOCK: segnali nel set sono rimossi dalla maschera esistente;
      - SIG_SETMASK: il set diventa la nuova maschera del segnale;
    • oldset è la vecchia maschera.
```

utilizzando la **sigprocmask()**

1. mascheriamo (=blocciamo) SIGTERM per 20 ms
2. dopo 20 sec il segnale viene sbloccato e il segnale gestito.

```
static int segnale_ricevuto = 0;
int main(int argc, char *argv[])
{
    sigset(SIG_BLOCK, my_mask);
    static void mio_handler(int signo)
    {
        static int segnale_ricevuto = 0;
        printf("segnalet ricevuto: %d\n", signo);
        if (signo == SIG_BLOCK, signo == 0) {
            sigemptyset(&my_mask); // inizializzo maschera vuota
            sigadd(SIG_BLOCK, SIGTERM);
            if (sigprocmask(SIG_BLOCK, &my_mask, NULL) < 0) {
                perror("Error setting up handler for signal %d: %s", i, strerror(errno));
            }
        }
    }
}
```

delivery di segnali a processi sospesi

- all'arrivo (asincrono) di un segnale
- 1. Lo stato del processo è salvato (registers, etc)
- 2. La funzione dell'handler è eseguita
- 3. Lo stato del processo è restored
- per processi in attesa su `wait()`, o sospesi con `pause()` o `sleep()` sono possibili 2 comportamenti:
 - il processo non è in esecuzione (è sospeso su qualche system call); oppure
 - la funzione dell'handler è eseguita normalmente.
- quando l'handler ritorna:
 - A. la syscall restituisce un errore, con `errno` settato a `EINTR`; o
 - B. la syscall viene automaticamente ripresa.
- A o B dipende dal sistema operativo, e dal flag `SA_RESTART` nella syscall `sigaction()`.

writing handlers

- l'handler deve essere associato al corrispondente segnale **prima** che il segnale possa essere (lanciato e) ricevuto dal processo: immaginando di dovere gestire un segnale `SIGCHLD`, è necessario associare il gestore al segnale prima della `fork()`.
- spesso una funzione può servire a gestire vari tipi di segnale.

```
void handle_signal(int signum) {  
    switch (signum) {  
        case SIGINT:  
            // gestione SIGINT  
            break;  
        case SIGALRM:  
            // gestione SIGNALRM  
            break;  
            // gestione altri segnali  
    }  
}
```

88

89

STAMPA DEGLI ERRORI >

Si possono stampare gli errori impostati su `errno` così:

- `strerrorname_np(iut err)` `<string.h>`

Stampa il nome simbolico dell'errore (es: EPERM).

NB: Siccome è una GNU extension, bisogna aggiungere la macro `_GNU_SOURCE` **prima** di ogni include oppure nel compilatore con `-D_GNU_SOURCE`

- `strerror(iut err)` `<string.h>`

Stampa una descrizione dell'errore (es: "no such process").

Per usare `errno` bisogna includere `<errno.h>`

KILL > DA CRED

Si può usare il system program `Kill <pid> [-signal]` per inviare segnali ad un processo.

- `<pid>`: Il PID del processo target.
- `[-signal]`: Il segnale da inviare. Può essere passato come numero o testo

Esempio: `Kill 100 -9` o `Kill 100 -SIGTERM`

Si può usare `Kill -l` per avere una lista dei segnali

PAUSE > `<un std.h>` `iut pause();` ma non è signal

Manda il processo corrente in sleep fino a quando non riceve un segnale

STAMPA DEI SEGNALI > `<string.h>`

- `strsignal(iut sig)`: restituisce la descrizione (es: terminazione)
- `sigdescr_np(iut sig)`: restituisce il nome simbolico `[_D_GNU_SOURCE]`