

Eventi asincroni

Sistemi Operativi II - Corso di laurea in Ingegneria Informatica

Sirio Scipioni

scipioni@dis.uniroma1.it

<http://www.dis.uniroma1.it/~scipioni>

MIDLAB

<http://www.dis.uniroma1.it/~midlab>

Sommario

- Nozioni di base
- Gestione non sicura dei segnali
- Gestione sicura dei segnali

Nozioni di base

- I meccanismi di segnalazione sono un semplice mezzo tramite il quale l'accadimento di un evento può essere notificato ad un processo.
- Il processo può eventualmente eseguire specifiche azioni associate a tale accadimento.
- Una segnalazione è talvolta classificata come un semplice messaggio, tuttavia essa differisce da un messaggio in diversi modi quali:
 - ✗ Una segnalazione viene in genere inviata occasionalmente da un processo, molto più spesso dal sistema operativo come risultato di un evento eccezionale (es. errore di calcolo in virgola mobile).
 - ✗ Una segnalazione può non avere contenuto informativo, in particolare il ricevente può non arrivare a conoscere l'identità di chi emette la segnalazione.

Trattamento delle segnalazioni

- Trattare le segnalazioni è molto importante per i programmatori perché il sistema operativo invia in modo asincrono segnalazioni indipendentemente dalla "volontà" dei processi.
- Se i processi decidono di non "ignorarle esplicitamente" o di non "catturarle", essi possono venir automaticamente terminati.
- La gestione dei segnali prevede tre possibilità per un processo:
 - ✗ **Catturare il segnale:** verrà eseguita quindi una funzione specificata dal programmatore.
 - ✗ **Ignorare esplicitamente il segnale:** in questo caso il processo "scarta" semplicemente il segnale (non è possibile per tutti i segnali).
 - ✗ **Ignorare implicitamente il segnale:** il processo accetta il comportamento di default predefinito dal sistema operativo.

Segnali UNIX più frequenti

- **SIGHUP: Hangup**. Il processo riceve questo segnale quando il terminale a cui era associato viene chiuso (ad esempio nel caso di un xterm) oppure scollegato (ad esempio nel caso di una connessione via modem o via telnet).
- **SIGINT: Interrupt**. Ricevuto da un processo quando l'utente preme la combinazione di tasti di interrupt (solitamente Control + C).
- **SIGQUIT: Quit**. Simile a SIGINT, ma in più, in caso di terminazione del processo, il sistema genera un "core dump", ovvero un file che contiene lo stato della memoria al momento in cui il segnale SIGQUIT è stato ricevuto. Solitamente SIGQUIT viene generato premendo i tasti Control+|.
- **SIGILL: Illegal Instruction**. Il processo ha tentato di eseguire un'istruzione proibita (o inesistente).
- **SIGKILL: Kill**. Questo segnale non può essere catturato in nessun modo dal processo ricevente, che non può fare altro che terminare. Mandare questo segnale è il modo più sicuro e brutale per "uccidere" un processo.

Segnali UNIX più frequenti

- **SIGSEGV: Segmentation violation**. Generato quando il processo tenta di accedere ad un indirizzo di memoria al di fuori del proprio spazio.
- **SIGPIPE: Broker Pipe**. Generato quando il processo tenta di scrivere su una PIPE per la quale non esistono processi in lettura.
- **SIGTERM: Termination**. Inviato ad un processo come richiesta non forzata di terminazione.
- **SIGALRM: Alarm**. Inviato ad un processo allo scadere del conteggio dell'orologio di allarme.
- **SIGCHLD: Child death**. Inviato ad un processo quando uno dei suoi figli termina.
- **SIGUSR1, SIGUSR2: User defined**. Non hanno un significato preciso, e possono essere utilizzati dai processi utente per implementare un rudimentale protocollo di comunicazione e/o sincronizzazione.

Comportamenti di default

Il comportamento di default tenuto da un processo alla ricezione di un segnale dipende dal sistema operativo; molti SO si attengono però alle seguenti indicazioni:

- tutti i segnali, tranne SIGKILL e SIGCHLD, sono ignorati implicitamente ed al loro arrivo il processo destinatario termina.
- SIGCHLD è ignorato implicitamente, ma il suo arrivo non provoca la terminazione del processo destinatario.
- SIGKILL non è ignorabile, ed il suo arrivo provoca la terminazione del processo destinatario.

Generazione di segnali

```
int kill(int pid,  
         int segnale);
```

Descrizione: richiede l'invio di un segnale.

Argomenti:

1. *pid*: identificatore del processo destinatario del segnale;
2. *segnale*: specifica il numero del segnale da inviare.

Restituzione: -1 in caso di fallimento.

UNIX permette di inviare segnali esclusivamente ad altri processi avviati dallo stesso utente. L'unica eccezione è data dall'utente ROOT (amministratore) che può inviare segnali a qualsiasi processo esistente.

La system call *kill* è asincrona (non bloccante): il processo mittente prosegue immediatamente nell'esecuzione del proprio codice, mentre al destinatario viene notificato il segnale.

Generazione di segnali

```
int raise(int segnale);
```

Descrizione: invia un segnale al processo corrente.

Argomenti:

1. *segnale*: specifica il numero del segnale da inviare.

Restituzione: -1 in caso di fallimento.

Generazione di segnali

E' possibile programmare la generazione temporizzata di uno specifico segnale:

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

Descrizione: programma un timer interno al sistema operativo in modo tale che questo invii un segnale specifico al processo stesso dopo un lasso di tempo definito.

Argomenti:

1. *seconds*: Il numero di secondi dopo il quale verrà notificato il segnale.

Restituzione: il numero di secondi rimanenti alla generazione di un evento impostato da una precedente chiamata ad *alarm*.

Il segnale generato è di tipo SIGALRM;

Effettuare una chiamata ad *alarm* quando ne era stata già effettuata un'altra non fa altro che impostare il valore del timer pari al numero di secondi specificato nell'ultima chiamata.

Catturare i segnali

```
void (* signal(int sig, void (*func)(int)))(int);
```

Descrizione: specifica il comportamento che il processo terrà alla ricezione di uno specifico segnale.

Argomenti:

1. *sig*: Il numero del segnale per il quale si vuole specificare il comportamento;
2. *func*: puntatore alla funzione di gestione del segnale oppure SIG_DFL / SIG_IGN.

Restituzione: SIG_ERR in caso di errore, altrimenti il valore della precedente funzione di gestione del segnale che viene sovrascritto con *func*.

SIG_DFL impone al processo il comportamento di default.

SIG_IGN impone al processo di ignorare esplicitamente il segnale.

Attendere un segnale

```
int pause(void);
```

Descrizione: blocca il processo in attesa di un qualsiasi segnale.




Restituzione: sempre -1.

pause ritorna sempre al completamento della funzione di handling del segnale.

Non è possibile sapere direttamente da *pause* quale segnale ha provocato lo sblocco; si può rimediare facendo sì che l'handler del segnale modifichi il valore di una variabile globale.




Eredità delle impostazioni

Come vengono gestite le impostazioni dei segnali quando si creano nuovi processi figli ?

-  il comportamento associato alla ricezione di un segnale viene "ereditato" dai processi figli.
-  per quanto riguarda le funzioni del tipo `execX()`, solo le impostazioni di `SIG_IGN` e `SIG_DFL` vengono mantenute, mentre per ogni segnale armato con una specifica funzione di gestione viene automaticamente impostato il comportamento di default.
-  ovviamente il nuovo programma può definire al suo interno una nuova gestione dei segnali.

Interruzione della computazione

Alla ricezione di un segnale, la computazione in un processo viene interrotta. In che modo riprende, una volta completata la fase di gestione del segnale ?

-  Se la computazione interrotta era l'esecuzione di una generica istruzione in modo utente, essa riprende da dove era stata interrotta (istruzione macchina puntata dal program counter prima della gestione del segnale). **Nozione di software interrupt.**
-  Se la computazione interrotta era una system call su cui il processo era bloccato, la system call viene abortita (fallisce) e la variabile globale *errno* (codice di errore) viene settata al valore EINTR (si veda l'header file "errno.h"). **ATTENZIONE:** la system call abortita non viene richiamata automaticamente !
-  System call che non bloccano il processo sono eseguite in modo atomico e non vengono mai interrotte da alcun segnale.

System call bloccanti

Alla luce di quanto detto la corretta invocazione di una system call bloccante è di questo tipo:

```
while ( systemCall() == -1 )
    if ( errno != EINTR ) {
        printf ("Errore!\n");
        exit(1);
    }
```

Lo schema seguente non basta:

```
if ( systemCall() == -1 ) {
    printf ("Errore!\n");
    exit(1);
}
```

Esempio 1

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

/* a questa funzione sarà demandata la gestione del segnale SIGALRM */
void gestione_timeout() {
    printf("*** I'm alive ! ***\n");
    /* ATTENZIONE ! Dobbiamo "riarmare" il segnale !!! */
    signal(SIGALRM, gestione_timeout);
    alarm(5);
}

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

    /* impostiamo la gestione del segnale SIGALRM */
    signal(SIGALRM, gestione_timeout);

    /* impostiamo la generazione del segnale (5 secondi) */
    alarm(5);

    /* nel frattempo continuiamo a fare le nostre operazioni... */
    while(1) read(0, &c, 1);
}
```

Esempio 2

```
/*      TCP Server multiprocesso      */

/* dichiarazioni ed include */

/* dichiariamo il gestore del segnale */
void handler_sigint();

int main(int argc, char *argv[]) {
    /* definizione variabili */
    /* interpretazione linea di comando */
    /* creazione socket */
    /* bind e listen sul socket */

    /* armiamo il segnale */
    signal(SIGINT, handler_sigint);

    /* loop infinito */
}

/* Definizione del gestore del segnale */
void handler_sigint(){
    /* Chiudiamo i socket */
    close(list_s);
    close(conn_s);
    exit(0);
}
```

Problemi con segnali inaffidabili

La gestione dei segnali vista fino ad ora presenta due problemi:

1. Reset del gestore dopo una notifica del segnale:

```
...
signal(SIGINT, sig_int);
...

void sig_int() {
    ...
    signal(SIGINT, sig_int);
    ...
}
```

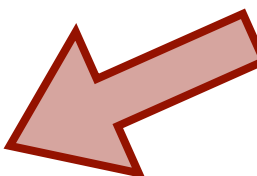
Esiste una finestra temporale all'interno della quale la notifica di un secondo segnale può far terminare il processo (prima della nuova *signal()* è attiva la gestione di default!).

Problemi con segnali inaffidabili

La gestione dei segnali vista fino ad ora presenta due problemi:

1. Reset del gestore dopo una notifica del segnale:

```
...  
signal(SIGINT, sig_int);  
...  
  
void sig_int() {  
    ...  
    signal(SIGINT, sig_int);  
    ...  
}
```



Esiste una finestra temporale all'interno della quale la notifica di un secondo segnale può far terminare il processo (prima della nuova *signal()* è attiva la gestione di default!).

Problemi con segnali inaffidabili

2. Attesa di un segnale:

```
...
int segnale_arrivato=0;
...
signal(SIGINT, sig_int);
...
if (! segnale_arrivato) {
    ...
    pause(); /*Il processo attenderà un ulteriore segnale */
    ...
}

void sig_int() {
    signal(SIGINT, sig_int);
    segnale_arrivato = 1;
}
```


Esiste una finestra temporale all'interno della quale la notifica di un segnale può

Problemi con segnali inaffidabili

2. Attesa di un segnale:

```
...
int segnale_arrivato=0;
...
signal(SIGINT, sig_int);
...
if (!segnale_arrivato) {
    pause(); /*Il processo attenderà un ulteriore segnale */
    ...
}

void sig_int() {
    signal(SIGINT, sig_int);
    segnale_arrivato = 1;
}
```



Esiste una finestra temporale all'interno della quale la notifica di un segnale può

Gestione affidabile dei segnali

Un processo può bloccare temporaneamente la consegna di un segnale (escluso il SIGKILL).

Un eventuale segnale bloccato non viene perso, ma rimane "pendente" fino a che:

- il processo lo sblocca (e quindi il segnale viene consegnato);
- il processo sceglie di ignorare il segnale.

Un processo può modificare l'azione da eseguire prima della consegna del segnale.

Il blocco dei segnali viene realizzato impostando opportunamente i bit costituenti la cosiddetta "signal mask" del processo.

La "signal mask" rappresenta l'insieme dei segnali che il processo sta bloccando.

La modifica avviene per mezzo di un'apposita struttura dati: `sigset_t`.

Signal Set

Il tipo `sigset_t` rappresenta un insieme di segnali (signal set).

Il SO ci mette a disposizione una serie di funzioni (definite in `signal.h`) per la gestione dei signal set:

```
int sigemptyset (sigset_t *set);
```

Svuota *set*.

```
int sigfillset (sigset_t *set);
```

Mette tutti i segnali in *set*.

```
int sigaddset (sigset_t *set, int signo);
```

Aggiunge il segnale *signo* a *set*.

```
int sigdelset (sigset_t *set, int signo);
```

Toglie il segnale *signo* da *set*.

```
int sigismember (sigset_t *set, int signo);
```

Controlla se *signo* è in *set*.

Gestione della signal mask

```
int sigprocmask(int how,  
                const sigset_t *set,  
                sigset_t *oset);
```

Descrizione: modifica la signal mask.

Argomenti:

1. *how*: indica in che modo intervenire sulla signal mask e può valere:
SIG_BLOCK: i segnali indicati in *set* sono aggiunti alla signal mask;
SIG_UNBLOCK: i segnali indicati in *set* sono rimossi dalla signal mask;
SIG_SETMASK: La nuova signal mask diventa quella specificata da *set*;
2. *set*: il signal set sulla base del quale verranno effettuate le modifiche.
3. *oset*: se non è NULL, in questa variabile verrà scritto il valore della signal mask PRIMA di effettuare le modifiche richieste.

Restituzione: -1 in caso di errore.

Segnali pendenti

```
int sigpending(sigset_t *set);
```

Descrizione: restituisce l'insieme dei segnali bloccati che sono attualmente pendenti.

Argomenti:

1. *set*: il signal set in cui verrà scritto l'insieme dei segnali pendenti.

Restituzione: -1 in caso di errore.

Gestione dei segnali

```
int sigaction(int sig,  
             const struct sigaction * restrict act,  
             struct sigaction * restrict oact);
```

Descrizione: Permette di esaminare e/o modificare l'azione associata ad un segnale.

Argomenti:

1. *sig*: il segnale interessato dalla modifica;
2. *act*: indica come modificare la gestione del segnale;
3. *oact*: in questa struttura vengono memorizzate le impostazioni precedenti per la gestione del segnale.

Restituzione: -1 in caso di errore.

La struttura sigaction

```
struct sigaction {  
    void (*sa_handler) (); /* indirizzo del gestore o SIG_IGN o SIG_DFL*/  
    void (*sa_sigaction) (int, siginfo_t *, void*); /* indirizzo del gestore  
                                                    che riceve informazioni  
                                                    aggiuntive sul segnale  
                                                    ricevuto (utilizzato al  
                                                    posto di sa_handler se  
                                                    sa_flags contiene  
                                                    SA_SIGINFO ) */  
    sigset_t sa_mask; /* segnali aggiuntivi da bloccare prima dell'esecuzione  
                      del gestore */  
    int sa_flags; /* opzioni aggiuntive */  
};
```

Notifiche (eventualmente multiple) dello stesso segnale durante l'esecuzione del gestore sono bloccate fino al termine del gestore stesso (a meno che *sa_flags* valga *NO_DEFER*).

Con l'uso di *sigaction* l'azione specificata per il trattamento di un segnale rimane attiva fino ad una successiva modifica.

Attesa di un segnale

```
int sigsuspend(const sigset_t *sigmask);
```

Descrizione: attende l'arrivo di un segnale.

Argomenti:

1. *sigmask*: prima di mettersi in attesa la funzione rende attiva questa signal mask;

Restituzione: sempre -1 con *errno* che vale EINTR.

sigsuspend, una volta terminata, ripristina la signal mask precedente alla sua chiamata.

Quando viene attivata, se esistono segnali pendenti che vengono sbloccati da *sigmask*, questi vengono immediatamente consegnati.

Se non esistono segnali pendenti, oppure questi non vengono sbloccati, la funzione mette il processo in attesa dell'arrivo di un qualsiasi segnale non bloccato.

Esempio 3 (1)

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

/* Il gestore del segnale SIGUSR1 */
void catcher(int signo) {
    static int ntimes = 0;
    printf("Processo %d: SIGUSR1 ricevuto #%d volte\n",getpid(),++ntimes);
}

int main() {
    int pid, ppid;
    struct sigaction sig, osig;
    sigset_t sigmask, oldmask, zeromask;

    sigemptyset(&zeromask);

    /* blocchiamo i segnali di tipo SIGUSR1 */
    sigemptyset(&sigmask);
    sigaddset(&sigmask, SIGUSR1);
    sigprocmask(SIG_BLOCK, &sigmask, &oldmask);

    /* installiamo il gestore del segnale */
    sig.sa_handler = catcher;
    sigemptyset(&sig.sa_mask);
    sig.sa_flags = 0;

    sigaction(SIGUSR1, &sig, &osig);
```

Esempio 3 (2)

```
if((pid=fork()) < 0) {
    perror("fork error");
    exit(1);
} else if(pid == 0) {
    /* Processo figlio */
    ppid = getppid();
    printf("figlio: mio padre e' %d\n", ppid);
    while(1) {
        sleep(1);
        kill(ppid, SIGUSR1);
        /* Sblocca il segnale SIGUSR1 e lo attende */
        sigsuspend(&zeromask);
    }
} else {
    /* Processo padre */
    printf("padre: mio figlio e' %d\n", pid);
    while(1) {
        /* Sblocca il segnale SIGUSR1 e lo attende */
        sigsuspend(&zeromask);
        sleep(1);
        kill(pid, SIGUSR1);
    }
}
}
```