LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Segnali

Segnali in Unix

Ci sono vari eventi che possono avvenire in maniera asincrona al normale flusso di un programma, alcuni dei quali in maniera inaspettata e non predicibile. Per esempio, durante l'esecuzione di un programma ci può essere una richiesta di terminazione o di sospensione da parte di un utente, la terminazione di un processo figlio o un errore generico.

Unix prevede la gestione di questi eventi attraverso i **segnali**: quando il sistema operativo si accorge di un certo evento, genera un segnale da mandare al processo interessato il quale potrà *decidere* (nella maggior parte dei casi) come comportarsi.

Segnali in Unix

Il numero dei segnali disponibili cambia a seconda del sistema operativo, con Linux che ne definisce 32. Ad ogni segnale corrisponde sia un valore numerico che un'etichetta mnemonica (definita nella libraria "*signal.h*") nel formato `SIG`XXX. Alcuni esempi:

`SIGALRM` (alarm clock)
`SIGCHLD` (child terminated)
`SIGCONT` (continue, if stopped)
`SIGINT` (terminal interrupt, CTRL + C)
`SIGKILL` (kill process)
`SIGQUIT` (terminal quit)
`SIGSTOP` (stop)
`SIGTERM` (termination)
`SIGUSR1` (user signal)
`SIGUSR2` (user signal)

Segnali in Unix

Per ogni processo, all'interno della process table, vengono mantenute due liste:

- **Pending signals**: segnali emessi che il processo dovrà gestire.
- **Blocked signals**: segnali non comunicati al processo.

Ad ogni schedulazione del processo le due liste vengono controllate per consentire al processo di reagire nella maniera più adeguata.

Gestione dei segnali

I segnali sono anche detti "*software interrupts*" perchè sono, a tutti gli effetti, delle interruzioni del normale flusso del processo generate dal sistema operativo (invece che dall'hardware, come per gli *hardware interrupts*).

Come per gli interrupts, il programma può decidere come gestire l'arrivo di un segnale (presente nella lista *pending*):

- Eseguendo l'azione default.
- Ignorandolo (non sempre possibile) → programma prosegue normalmente.
- Eseguendo un handler personalizzato → programma si interrompe.

Default handler

Ogni segnale ha un suo handler di default che tipicamente può:

- Ignorare il segnale
- Terminare il processo
- Continuare l'esecuzione (se il processo era in stop)
- Stoppare il processo

Ogni processo può sostituire il gestore di default con una funzione "custom" (**a parte per SIGKILL e SIGSTOP**) e comportarsi di conseguenza. La sostituzione avviene tramite la system call `signal()` (definita in "*signal.h*").

signal() system call

```
sighandler_t signal(int signum, sighandler_t handler);
typedef void (*sighandler_t)(int);
```

```
#include <signal.h> <stdio.h> <stdlib.h>
void myHandler(int sigNum){
    printf("CTRL+Z\n");
}
int main(){
    signal(SIGINT,SIG_IGN);   //Ignore signal
    signal(SIGCHLD,SIG_DFL);  //Use default handler
    signal(SIGTSTP,myHandler);    //Use myHandler
    return 0;
}
```

Esempio:

```
#include <signal.h>    //sigCST.c
#include <stdio.h>
#include <stdlib.h>

void myHandler(int sigNum){
    printf("CTRL+Z\n");
    exit(2);
}

int main(void){
    signal(SIGTSTP,myHandler);
    while(1);
}
```

```
$ gcc sig[CST|DFL|IGN].c -o
sig.out
$ ./sig.out
$ <CTRL+Z>
```

```
#include <signal.h>    //sigDFL.c
int main(){
    signal(SIGTSTP,SIG_DFL);
    while(1);
}
```

```
#include <signal.h>    //sigIGN.c
int main(){
    signal(SIGTSTP,SIG_IGN);
    while(1);
}
```

Custom handler

Un handler personalizzato deve essere una funzione di tipo `void` che accetta come argomento un intero, il quale rappresenta il segnale catturato. Questo consente l'utilizzo di un solo handler per segnali differenti.

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

void myHandler(int sigNum){
    if(sigNum == SIGINT) printf("CTRL+C\n");
    else if(sigNum == SIGTSTP) printf("CTRL+Z\n");
}

signal(SIGINT,myHandler);
signal(SIGTSTP,myHandler);
```

signal() return

signal() restituisce un riferimento all'handler che era precedentemente assegnato al segnale:

- NULL: handler precedente era l'handler di default
- 1: l'handler precedente era SIG_IGN
- *address*: l'handler precedente era `*(address)`

```
#include <signal.h> <stdio.h>                                    //return.c
void myHandler(int sigNum){}
int main(){
    printf("DFL: %p\n",signal(SIGINT,SIG_IGN));
    printf("IGN: %p\n",signal(SIGINT,myHandler));
    printf("Custom: %p == %p\n",signal(SIGINT,SIG_DFL),myHandler);
}
```

Alcuni segnali

SIGXXX	**description**	**default**
SIGALRM	(alarm clock)	quit
SIGCHLD	(child terminated)	ignore
SIGCONT	(continue, if stopped)	ignore
SIGINT	(terminal interrupt, CTRL + C)	quit
SIGKILL	(kill process)	quit
SIGSYS	(bad argument to syscall)	quit with dump
SIGTERM	(software termination)	quit
SIGUSR1/2	(user signal 1/2)	quit
SIGSTOP	(stopped)	quit
SIGTSTP	(terminal stop, CTRL + Z)	quit

Esempio

```
#include <signal.h> <stdio.h> <unistd.h> <sys/wait.h>   //child.c
void myHandler(int sigNum){
    printf("Child terminated! Received %d\n",sigNum);
}
int main(){
    signal(SIGCHLD,myHandler);
    int child = fork();
    if(!child){
        return 0; //terminate child
    }
    while(wait(NULL)>0);
}
```

Inviare i segnali: kill()

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

Invia un segnale ad uno o più processi a seconda dell'argomento *pid:*

- pid $>$ 0: segnale al processo con PID=pid
- pid $=$ 0: segnale ad ogni processo dello stesso gruppo di chi esegue "kill"
- pid $=$ -1: segnale ad ogni processo possibile (stesso UID/RUID)
- pid $<$ -1: segnale ad ogni processo del gruppo |pid|

Restituisce 0 se il segnale viene inviato, -1 in caso di errore.

Ogni tipo di segnale può essere inviato, non deve essere necessariamente un segnale corrispondente ad un evento effettivamente avvenuto!

```c
#include <signal.h> <stdio.h> <stdlib.h> <sys/wait.h> <unistd.h>
//kill.c
void myHandler(int sigNum){printf("[%d]ALARM!\n",getpid());}
int main(void){
    signal(SIGALRM,myHandler);
    int child = fork();
    if (!child) while(1); // block the child
    printf("[%d]sending alarm to %d in 3 s\n",getpid(),child);
    sleep(3);
    kill(child,SIGALRM); // send ALARM, child's handler reacts
    printf("[%d]sending SIGTERM to %d in 3 s\n",getpid(),child);
    sleep(3);
    kill(child,SIGTERM); // send TERM: default is to terminate
    while(wait(NULL)>0);
}
```

Kill da bash

kill è anche un programma in bash che accetta come primo argomento il tipo di segnale (`kill -l` per la lista) e come secondo argomento il PID del processo.

```c
#include <signal.h> <stdio.h> <stdlib.h> <unistd.h>  //bash.c
void myHandler(int sigNum){
    printf("[%d]ALARM!\n",getpid());
    exit(0);
}
int main(){
    signal(SIGALRM,myHandler);
    printf("I am %d\n",getpid());
    while(1);
}
```

```
$ gcc bash.c -o bash.out
$ ./bash.out
# On new window/terminal
$ kill -14 <PID>
```

Programmare un alarm: alarm()

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

```
#include <signal.h> <stdio.h> <stdlib.h> <unistd.h>  //alarm.c
short cnt = 0;
void myHandler(int sigNum){printf("ALARM!\n"); cnt++;}
int main(){
    signal(SIGALRM,myHandler);
    alarm(5); //Set alarm in 5 seconds
    //Set new alarm (cancelling previous one)
    printf("Seconds remaining to previous alarm %d\n",alarm(2));
    while(cnt<1);
}
```

Mettere in pausa: pause()

```
int pause();
```

```
#include <signal.h> <unistd.h> <stdio.h>                    //pause.c

void myHandler(int sigNum){
    printf("Continue!\n");
}
int main(){
    signal(SIGCONT,myHandler);
    signal(SIGUSR1,myHandler);
    pause();
}
```

```
$ gcc pause.c -o pause.out
$ ./pause.out
# On new window/terminal
$ kill -18/-10 <PID>
```

Bloccare i segnali

Oltre alla lista dei “pending signal” esiste la lista dei “blocked signals”, ovvero dei segnali ricevuti dal processo ma volutamente non gestiti. Mentre i segnali ignorati non saranno mai gestiti, i segnali bloccati sono solo *temporaneamente* non gestiti.

Un segnale bloccato rimane nello stato *pending* fino a quando esso non viene gestito oppure il suo handler tramutato in *ignore*.

L’insieme dei segnali che vanno bloccati è detto “signal mask”, una maschera dei segnali che è modificabile attraverso la system call `sigprocmask()`.

Bloccare i segnali: sigset_t

Una signal mask può essere gestita con un `sigset_t`, ovvero una lista di segnali modificabile con alcune funzioni. Queste funzioni modificano il sigset_t, non la maschera dei segnali del processo!

`int sigemptyset(sigset_t *set);` Svuota
`int sigfillset(sigset_t *set);` Riempie
`int sigaddset(sigset_t *set, int signo);` Aggiunge singolo
`int sigdelset(sigset_t *set, int signo);` Rimuove singolo
`int sigismember(const sigset_t *set, int signo);` Interpella

Bloccare i segnali: sigprocmask()

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

A seconda del valore di `how` e di `set`, la maschera dei segnali del processo viene cambiata. Nello specifico:

- `how = SIG_BLOCK`: i segnali in `set` sono aggiunti alla maschera;
- `how = SIG_UNBLOCK`: i segnali in `set` sono rimossi dalla maschera;
- `how = SIG_SETMASK`: `set` diventa la maschera.

Se `oldset` non è nullo, in esso verrà salvata la vecchia maschera (anche se `set` è nullo).

Esempio

```
#include <signal.h>

int main(){
    sigset_t mod,old;
    sigfillset(&mod); // Add all signals to the blocked list
    sigemptyset(&mod); // Remove all signals from blocked list
    sigaddset(&mod,SIGALRM); // Add SIGALRM to blocked list
    sigismember(&mod,SIGALRM); // is SIGALRM in blocked list?
    sigdelset(&mod,SIGALRM); // Remove SIGALRM from blocked list

    // Update the current mask with the signals in ‘mod’
    sigprocmask(SIG_BLOCK,&mod,&old);
}
```

Esempio 2

```
$ kill -10 <PID> # ok
$ kill -10 <PID> # blocked
```

```c
#include <signal.h> <unistd.h> <stdio.h>          //sigprocmask.c
sigset_t mod, old;
int i = 0;
void myHandler(int signo){
    printf("signal received\n");
    i++;
}
int main(){
    printf("my id = %d\n",getpid());
    signal(SIGUSR1,myHandler);
    sigemptyset(&mod); //Initialise set
    sigaddset(&mod,SIGUSR1);
    while(1) if(i==1) sigprocmask(SIG_BLOCK,&mod,&old);
}
```

Verificare pending signals: sigpending()

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

```
//sigpending.c
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

sigset_t mod,pen;
void handler(int signo){
    printf("SIGUSR1 received\n");
    sigpending(&pen);
    if(!sigismember(&pen,SIGUSR1))
      printf("SIGUSR1 not pending\n");
    exit(0);
}
```

```
int main(){
    signal(SIGUSR1,handler);
    sigemptyset(&mod);
    sigaddset(&mod,SIGUSR1);
    sigprocmask(SIG_BLOCK,&mod,NULL);
    kill(getpid(),SIGUSR1);
    // sent but it's blocked...
    sigpending(&pen);
    if(sigismember(&pen,SIGUSR1))
        printf("SIGUSR1 pending\n");
    sigprocmask(SIG_UNBLOCK,&mod,NULL);
    while(1);
}
```

sigaction() system call

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

struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; //Signals blocked during handler
    int sa_flags; //modify behaviour of signal
    void (*sa_restorer)(void); //Deprecated, not POSIX
};
```

https://linux.die.net/man/2/sigaction

Esempio

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>   //sigaction.c
void handler(int signo){
    printf("signal received\n");
}
int main(){
    struct sigaction sa; //Define sigaction struct
    sa.sa_handler = handler; //Assign handler to struct field
    sigemptyset(&sa.sa_mask); //Define an empty mask
    sigaction(SIGUSR1,&sa,NULL);
    kill(getpid(),SIGUSR1);
}
```

Esempio: blocking signal

```
$ kill -10 <PID> ; sleep 1
  && kill -12 <PID>
```

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>  //sigaction2.c
void handler(int signo){
    printf("signal %d received\n",signo);
    sleep(2);
    printf("Signal done\n");
}
int main(){
    printf("Process id: %d\n",getpid());
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask); //Use an empty mask → block no signal
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}
```

Esempio: blocking signal

```
$ kill -10 <PID> ; sleep 1
  && kill -12 <PID>
```

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>  //sigaction3.c
void handler(int signo){
    printf("signal %d received\n",signo);
    sleep(2);
    printf("Signal done\n");
}
int main(){
    printf("Process id: %d\n",getpid());
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask,SIGUSR2); // Block SIGUSR2 in handler
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}
```

Esempio: sa_sigaction

```
$ echo $$ ; kill -10 <PID> # custom
$ kill -10 <PID> # default
```

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>  //sigaction4.c
void handler(int signo, siginfo_t * info, void * empty){
    //print id of process issuing the signal
    printf("Signal received from %d\n",info->si_pid);
}
int main(){
    struct sigaction sa;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags |= SA_SIGINFO; // Use sa_sigaction
    sa.sa_flags |= SA_RESETHAND; // Restore def handler afterward
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}
```

CONCLUSIONI

I segnali sono uno strumento di comunicazione tra processi molto semplice ma efficace: essendo un metodo molto “antico” non è particolarmente flessibile (essendo nato quando le risorse erano più limitate dei tempi attuali), ma nella sua forma base è comodo e multipiattaforma. L’uso di “signal” è standard, ma alcuni comportamenti possono essere indefiniti, mentre “sigaction” è più affidabile ma meno portabile tra sistemi operativi.