

Sviluppo di software di rete con l'utilizzo di sistemi UNIX

Segnali

Claudio Vicari

Funzioni per l'invio di segnali

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

- ➔ *pid* permette di selezionare il processo cui mandare il segnale: ad un processo specifico, a tutti i processi in un gruppo

```
int raise(int sig);
```

- ➔ invia un segnale al processo stesso
- ➔ entrambi restituiscono 0 in caso di successo, -1 in caso di errore
- ➔ si possono mandare segnali solo avendo i diritti appropriati: la regola principale, è che i processi devono avere gli stessi user id (real o effective)

Esempio: sleep2.c

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

static jmp_buf  env_alm;

static void
sig_alm(int signo)
{
    longjmp(env_alm, 1);
}

unsigned int
sleep2(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alm) == SIG_ERR)
        return(nsecs);
    if (setjmp(env_alm) == 0) {
        alarm(nsecs);          /* start the timer */
        pause();               /* next caught signal wakes us up */
    }
    return( alarm(0) );        /* turn off timer, return unslept time */
}
```

Esempio: sleep2.c

- ➔ La funzione `setjmp` salva il contesto della funzione chiamante
- ➔ Il problema è che fra `alarm` e pause potrebbe passare un periodo di tempo troppo lungo, e l'allarme sarebbe già scaduto
- ➔ In questo caso, pause si bloccherebbe per sempre
- ➔ Invece, con `setjmp` salva il contesto della funzione, e `longjmp` effettua un *goto non locale*
- ➔ `setjmp` restituisce 0 quando si imposta il punto di ritorno, restituisce non-zero quando invece è il risultato del *goto*

sigsetjmp, siglongjmp

- ➔ `sigsetjmp, siglongjmp` salvano e recuperano anche il contesto dei segnali che sono bloccati
- ➔ queste funzioni possono essere usate anche per interrompere un'operazione bloccante, se non viene completato in un periodo di tempo prefissato (esempi sullo Stevens)
- ➔ attenzione all'uso di queste funzioni! Rendono i programmi difficili da leggere e scrivere, e le interazioni fra segnali rendono anche difficile prevedere correttamente il comportamento...

Definire un signal set

➡ funzioni per definire insiemi di segnali:

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

```
int sigemptyset(sigset_t *set);  
int sigfillset(sigset_t *set);  
int sigaddset(sigset_t *set, int signum);  
int sigdelset(sigset_t *set, int signum);  
int sigismember(const sigset_t *set, int  
    signum);
```

Utilizzare i signal set

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

- può bloccare (how=SIG_BLOCK) il set di segnali
- sblocca il set se how=SIG_UNBLOCK
- imposta la maschera del processo se how=SIG_SETMASK

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

- pone in set il set di segnali che sono bloccati e in attesa, per il processo corrente

La funzione sigaction

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

- ➔ serve a impostare od esaminare i signal handler, è più flessibile della funzione signal
- ➔ consente di conoscere il signal handler corrente, senza cambiarlo (diversamente da signal)
- ➔ la struct permette di specificare:
 - il set di segnali da gestire
 - la funzione di gestione
 - opzioni per la gestione del segnale

Struct sigaction: dettagli

```
struct sigaction {  
    void (*sa_handler)(int);  
    sigset_t sa_mask;  
    int sa_flags;  
}
```

- ➔ in particolare, `sa_flags` consente di specificare, fra gli altri:
 - `SA_RESTART` per il restart automatico delle system call
 - `SA_NOCLDWAIT` per evitare processi zombie
 - `SA_RESETHAND` per impostare il segnale a `SIG_DFL` dopo l'entrata nel signal handler

Signal con restart, usando sigaction

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t
my_signal(int signo, sighandler_t func)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if(signo == SIGALRM) {
#ifdef SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT; /* SunOS */
#endif
    } else {
#ifdef SA_RESTART
        act.sa_flags |= SA_RESTART; /* SVR4, 4.3+BSD */
#endif
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

impostazione della
struct sigaction

Eccezioni per diversi sistemi

Signal senza restart, usando sigaction

```
#include <signal.h>

typedef void (*sighandler_t) (int);

sighandler_t
my_signal_intr(int signo, sighandler_t func)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
#ifdef SA_INTERRUPT                /* SunOS */
    act.sa_flags |= SA_INTERRUPT;
#endif
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Uso di siginfo

Visualizza esempio

- ➔ `siginfo` ci permette di scoprire molte cose:
 - il *pid* di chi ha inviato il segnale: `si_pid`
 - lo *uid* di chi ha inviato il segnale: `si_uid`
 - un codice per il *motivo* dell'invio: `si_code`

Uso di siginfo

Codice

- ⇒ proviamo il programma con:
 - `$> gcc -o /tmp/prova esempio-siginfo.c`
 - `$> /tmp/prova`
- ⇒ e quindi mandando segnali da un'altra shell dello stesso utente:
 - `$> killall -SIGUSR1 /tmp/prova`
- ⇒ oppure eseguendo `kill` con una shell di root, per vedere lo *UID* che varia
- ⇒ Inoltre, cambiando il valore di `DEBUG_SIGNALS` possiamo modificare facilmente il comportamento

sigsuspend per proteggere dai segnali

- ➔ possiamo proteggere una sezione di codice, in modo che non possa essere interrotta da un segnale
- ➔ un modo per farlo è usare sigprocmask, all'inizio della porzione critica blocca, alla fine sblocca
- ➔ un modo atomico (affidabile anche sui vecchi sistemi), è usare `sigsuspend`:
 - annulla la signal mask
 - e pone il processo in sleep, fino all'occorrenza di un segnale

Visualizza esempio

Esercizi possibili

- ➔ scrivere un programma in cui, una volta al secondo, un signal handler per SIGALRM stampi il numero di volte che è stato invocato, e il main immediatamente dopo stampi lo stesso numero. Il programma non deve terminare (a meno di altri segnali)
- ➔ scrivere un programma che, usando alarm(), scriva sullo standard output il numero di secondi (approssimativo) trascorsi da quando è stato lanciato
- ➔ scrivere due programmi, a e b:
 - a si pone in attesa di un segnale SIGUSR1. Quando lo riceve, invia un segnale SIGKILL al processo che lo ha inviato
 - b dopo essere stato avviato attende un secondo, quindi invia un segnale SIGUSR1, infine si pone in attesa