

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

Segnali

Claudio Vicari

Panoramica

- ➔ i segnali sono interruzioni (interrupt) software
 - servono a gestire il verificarsi di eventi
 - per esempio, lo scadere di un allarme previsto dal programmatore

- ➔ vengono gestiti in maniera asincrona:
 - cioè, non avvengono nel flusso del programma principale, ma in un flusso distinto

Concetti fondamentali

- ⇒ i segnali hanno nomi: `SIG<>`, che corrispondono a valori identificativi interi (vedi `kill -l`)
- ⇒ i numeri sono definiti nell'header `<signal.h>`
- ⇒ esempi di segnali:
 - CTRL+C: genera un `SIGINT`
 - eccezioni come il segmentation fault: `SIGSEGV`
 - rottura di una *pipe*: `SIGPIPE`
 - scadenza di un allarme: `SIGALARM`
- ⇒ modi per inviare segnali:
 - funzione `kill` in C
 - comando `kill` nella shell

Esempio per SIGSEGV

su una shell:

```
$> sleep 1h
```

su una seconda shell:

```
$> killall -SIGSEGV sleep
```

analogamente:

```
$> ark&
```

```
$> killall -SIGSEGV ark
```

Gestione dei segnali

- ➔ i segnali sono asincroni: non vengono chiamati dal programma come semplice risultato di un test
- ➔ i segnali sono gestiti dal sistema operativo: quindi, il programmatore in effetti istruisce il sistema a comportarsi in una determinata maniera
- ➔ i segnali possono essere:
 - ignorati (a parte SIGKILL, SIGSTOP)
 - gestiti con funzioni programmate
 - essere gestiti dal gestore di default

Esempio: nohup

```
shell a> sleep 10s
```

```
shell b> killall -SIGHUP sleep
```

```
shell a> nohup sleep 10s
```

```
shell b> killall -SIGHUP sleep
```

nohup ignora i segnali di hangup!

Gestione dei segnali: esempio

- ➔ il segnale `SIGHUP` chiede al processo di terminare (hangup) per la chiusura del terminale
- ➔ il comando `nohup` lancia il processo specificato, con un gestore di segnali che lo rende immune al segnale `SIGHUP`
- ➔ `nohup` è un comando molto utile! Per poter lanciare programmi in background, che continuino anche quando non siamo presenti al terminale... es.: `wget`

Segnali importanti

⇒ SIGALRM

- lanciato dal programma stesso, per essere usato come un timer, con le funzioni:
 - `setitimer`
 - `alarm`

⇒ SIGUSR1, SIGUSR2

- segnali che possono essere usati dai programmi per scopi interni (vedi `man fsck`)

Segnali importanti: controllo

- ⇒ SIGSTOP, SIGTSTP
 - SIGSTOP non può essere ignorato né catturato, per mantenere controllo sui processi
 - SIGTSTP viene inviato ai processi in foreground in corrispondenza dell'invio di Ctrl+Z da tastiera
- ⇒ SIGCONT
 - inviato ad un processo cui si chieda di riprendere l'esecuzione (azione di default)

Segnali di terminazione

- ➔ SIGABRT: generalmente lanciato dal programma stesso, per segnalare una condizione anomala

esempio: funzione `assert()`

```
#include <assert.h>
```

```
void assert(scalar expression);
```

serve a far terminare un programma, nel caso in cui l'espressione sia valutata falsa. Molto comoda per un debugging rapido!!

Segnali di terminazione

- ➔ SIGINT: generato per uscita normale, quando si digita Ctrl+C
- ➔ SIGILL: generato nel caso che un processo esegua una istruzione illegale
- ➔ SIGSEGV: per indicare che un processo ha tentato di accedere ad un'area di memoria non valida
- ➔ SIGTERM: segnale di terminazione semplice
- ➔ SIGKILL: segnale di terminazione, non ignorabile.
Un modo sicuro per uccidere i processi
- ➔ SIGHUP: la sessione di terminale è chiusa. Viene inviato anche ai processi in background, ma solo al *session leader* (il processo padre). Gli altri segnali di terminazione vengono inviati a tutti i processi in foreground!

Segnali di condizioni

- ➔ SIGPIPE: in scrittura, segnala che il lettore di una PIPE è terminato. E' bene gestirlo nei server
- ➔ SIGIO: segnala un evento di I/O asincrono
- ➔ SIGURG: dati urgenti nei socket
- ➔ SIGCHLD: inviato al processo padre, al termine di ogni processo figlio. Deve essere intercettato, per liberare risorse mediante una chiamata a `wait`

Gestione dei segnali: la funzione `signal`

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

- ⇒ definita in `signal.h`
- ⇒ definisce il gestore per il segnale `signum`
- ⇒ lo rimpiazza con la funzione `handler`, definita dall'utente, oppure con:
 - `SIG_IGN` per ignorare il segnale
 - `SIG_DFL` per impostare il default
- ⇒ restituisce il valore del gestore precedente, oppure `SIG_ERR` in caso di errore

Funzione signal: esempio per SIGUSR1, SIGUSR2

Esempio: esempio3-signal_handling.c

```
$> gcc -o /tmp/e3 esempio3-signal_handling.c  
$> /tmp/e3
```

➔ quindi, proviamo a fare:

```
$> ps aux | grep e3
```

per scoprire il PID del processo

```
$> kill -SIGUSR1 <pid>
```

```
$> kill -SIGUSR2 <pid>
```

Funzione signal e pause

- ➔ Nota: la definizione secondo gli standard è

```
void (*signal(int signo,void (*func) (int))) (int);
```
- ➔ ma con

```
typedef void (*sighandler_t)(int);
```
- ➔ diventa del tutto equivalente a quella fornita su linux (anche lo Stevens definisce in modo simile)
- ➔ la funzione `int pause()`, serve ad interrompere finché non occorre un segnale, nel qual caso restituisce -1

Segnali all'avvio dei programmi

- ➔ quando un programma viene eseguito (exec)
 - i segnali ignorati dal chiamante rimangono ignorati
 - gli altri segnali sono portati all'azione di default (i puntatori a funzione non sono più validi dopo la exec!)

System call interrotte

- ➔ se il segnale avviene durante una *slow system call*, questa potrebbe essere interrotta!
 - in questo caso, la chiamata restituisce un errore e la variabile `errno` viene posta a `EINTR`
- ➔ le *slow system call* sono:
 - letture e scritture che possono bloccarsi potenzialmente per sempre (letture di PIPE, terminali, rete)
 - aperture di file speciali (per esempio, l'apertura di una connessione, che attenda la risposta di un modem)
 - pause e `wait`,
 - alcune chiamate di `ioctl`
 - alcune funzioni di comunicazione interprocesso

System call interrotte

- ➔ l'I/O da disco non viene mai interrotto, anche se può essere lento
- ➔ dobbiamo gestire queste condizioni! Possiamo:
 - gestirle esplicitamente
 - chiedere il restart automatico della chiamata
- ➔ gestione esplicita:
 - `if(errno==EINTR) go_back;`
- ➔ attenzione a cosa accade nelle varie implementazioni! Lo Stevens fornisce due versioni di `signal` per gestire correttamente il restart
- ➔ in seguito vedremo questa versione di `signal`

Funzioni rientranti

- ➔ alcune funzioni non possono essere usate contemporaneamente da due task, perché:
 - usano variabili statiche per i loro scopi
 - usano puntatori a dati statici
 - usano dati globali (senza sincronizzazione)
 - chiamano altre funzioni non rientranti
- ➔ quindi, se la funzione viene interrotta, il valore di queste variabili potrebbe cambiare in maniera non prevedibile
- ➔ questo è anche il problema che ci impedisce di usare funzioni come `printf` direttamente

Funzioni rientranti

- ➔ possiamo usare le versioni rientranti delle stesse funzioni, oppure evitarle
- ➔ esempio:
 - `struct passwd *getpwnam(const char *name);`
- ➔ il puntatore punta ad un'area statica! Questa è la versione rientrante:
 - `int getpwnam_r(const char *name, struct passwd *pwbuf, char *buf, size_t buflen, struct passwd **pwbufp);`
- ➔ questo problema avviene con tutti i tipi di programmazione asincrona, quindi anche con il multithreading
- ➔ attenzione anche a ripristinare il valore di `errno` quando siamo in un signal handler

Segnali concorrenti

- ➔ Se siamo in un signal handler e arriva un altro segnale, si interrompe anche il signal handler!
- ➔ Alcuni UNIX impediscono che un signal handler possa essere interrotto da un altro arrivo dello stesso segnale
- ➔ Un processo può scegliere di bloccare esplicitamente alcuni segnali, che rimangono in attesa
- ➔ Alcuni sistemi accodano i segnali, ma questo non è obbligatorio per lo standard
- ➔ L'ordine di arrivo dei segnali, in caso di contemporaneità, non è specificato