Corso di Laboratorio di Sistemi Operativi
A.A. 2016–2017

Lezione 16

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

A.A. 2016–2017 - Primo Semestre - 02/12/2016

Comunicazione tra processi: segnali

Segnali

I segnali in Unix sono un meccanismo semplice per inviare degli interrupt software ai processi.

- Solitamente sono utilizzati per gestire errori e condizioni anomale, piuttosto che per trasmettere dati.
- Un processo può ricevere segnali inviati da altri processi, o provenienti sistema operativo, e può:
  1. eseguire una opportuna funzione per trattare l'errore (signal handling);
  2. bloccare il segnale;
  3. inviare il segnale ad un altro processo.
- Il comportamento di default alla ricezione della maggior parte dei segnali, è quello di terminare il processo.

Segnali

Nella tabella a fianco sono elencati i diversi tipi di segnale.

- In rosso quelli che causano una **terminazione anomala** del processo.
- Gli altri, per la maggior parte, causano una **terminazione normale**.

La differenza è che i primi non possono essere ignorati, mentre gli altri possono essere ignorati o gestiti in modo particolare.

N°	Nome	Causa
1	SIGHUP	terminal line hangup
2	SIGINT	interrupt program
3	**SIGQUIT**	quit program
4	**SIGILL**	illegal instruction
5	**SIGTRAP**	trace trap
6	**SIGABRT**	abort program (formerly SIGIOT)
7	**SIGEMT**	emulate instruction executed
8	**SIGFPE**	floating-point exception
9	SIGKILL	kill program
10	**SIGBUS**	bus error
11	**SIGSEGV**	segmentation violation
12	**SIGSYS**	non-existent system call invoked
13	SIGPIPE	write on a pipe with no reader
14	SIGALRM	real-time timer expired
15	SIGTERM	software termination signal
16	SIGURG	urgent condition present on socket
17	SIGSTOP	stop (cannot be caught or ignored)
18	SIGTSTP	stop signal generated from keyboard
19	SIGCONT	continue after stop
20	SIGCHLD	child status has changed
21	SIGTTIN	background read from control terminal
22	SIGTTOU	background write to control terminal
23	SIGIO	I/O is possible on a descriptor
24	SIGXCPU	cpu time limit exceeded
25	SIGXFSZ	file size limit exceeded
26	SIGVTALRM	virtual time alarm
27	SIGPROF	profiling timer alarm
28	SIGWINCH	Window size change
29	SIGINFO	status request from keyboard
30	SIGUSR1	User defined signal 1
31	SIGUSR2	User defined signal 2

Inviare segnali

La system call principale per inviare segnali è `kill()`:

```
#include <sys/types.h>
#include <signal.h>

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

Invia il segnale `sig` al processo con PID `pid`.

Nota: Un segnale può essere lanciato solo a processi dello stesso utente, a meno che non si abbiano privilegi di **root**.

Inviare segnali

Alternative

Altrimenti si può inviare un segnale a se stessi con `raise()`:

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

La funzione `alarm()` causa la ricezione di un segnale `SIGALRM` dopo l'intervallo di tempo specificato:

```
unsigned int alarm(unsigned int secs);
```

Inviare segnali

Il comando `kill`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

int main(int argc, char **argv)
{
  if(argc < 2) {
    fprintf(stderr, "Specificare il PID di un processo\n");
    return 1;
  }

  char *endptr = NULL;
  pid_t pid = strtoll(argv[1], &endptr, 10);
  if(*endptr != 0) {
    fprintf(stderr, "Specificare il PID di un processo\n");
    return 1;
  }

  if(kill(pid, SIGKILL) == -1) {
    fprintf(stderr, "Impossibile uccidere il processo %d: %s\n", pid,
            strerror(errno));
    return 2;
  }

  return 0;
}
```

Signal handling

Un segnale si può **gestire**, eseguendo una funzione ogni volta che viene ricevuto, in modo simile agli interrupt handler hardware.

```
typedef void (*sig_t)(int); // typedef per un puntatore a funzione
sig_t signal(int sig, sig_t handler);
```

La funzione `signal()` registra la funzione puntata da `handler` come gestore del segnale `sig`.

La funzione `signal()` è una versione semplificata di `sigaction()`, funzione più flessibile che permette di:

- Impostare delle maschere per bloccare determinati segnali
- Passare informazioni aggiuntive all'handler
- Impostare varie opzioni sull'interazione del segnale con le system call di I/O.

Esempio di signal handling

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

void ahah(int x) {
  printf("ahah you cannot terminate me!\n");
}

int main()
{
  signal(SIGINT, ahah);
  signal(SIGTERM, ahah);
  signal(SIGKILL, ahah); // Funzionera'?

  while(1) {
    printf("Hey apple!\n");
    sleep(1);
  }

  return 0;
}
```

Comunicazione tra processi: socket

Socket

I **socket** (presa/spinotto) sono un meccanismo di comunicazione interprocesso **bidirezionale**, a differenza delle pipe.

- Come le pipe, una volta configurato il meccanismo si ottengono dei **file descriptor** su cui si può scrivere e leggere normalmente.
- I socket adottano una filosofia **client**/**server**.
- Il processo server **ascolta** su un indirizzo, mentre il/i processo/i client si **connettono** a tale indirizzo.
- È lo stesso modello di funzionamento delle **comunicazioni in rete**. Infatti, oltre che la comunicazione tra processi in esecuzione sullo stesso sistema, l'interfaccia a socket è quella usata per la comunicazione via rete.
- Esistono quindi più **domini**:
  - i socket UNIX-domain, per la comunicazione locale.
  - i socket Internet-domain, per la comunicazione IPv4/IPv6.
  - altri (Novell, AppleTalk, . . .) caduti in disuso.

Utilizzo dei socket

L'utilizzo dell'interfaccia a socket è più complesso di quello delle pipe.
Molte funzioni e chiamate di sistema entrano in gioco.

Funzione	Scopo	Usato da
`socket()`	Crea il file descriptor di un capo della connessione	entrambi
`bind()`	Lega il socket ad un indirizzo specifico	server
`listen()`	Blocca il processo in ascolto sul socket	server
`accept()`	Accetta una connessione in arrivo	server
`connect()`	Connette un socket ad un altro socket in ascolto	client

Utilizzo dei socket

Funzione `socket()`

La funzione `socket()` va chiamata sia dal client che dal server per aprire un file descriptor che verrà usato nelle operazioni successive.

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

I valori degli argomenti specificano il dominio (locale, internet, ...) e il protocollo di comunicazione utilizzato (nel caso di internet).

Per i socket di dominio UNIX, l'uso della funzione è il seguente:

```
int fd = socket(AF_LOCAL, SOCK_STREAM, 0);
```

Utilizzo dei socket

Funzione `bind()` e `listen()`

La funzione `bind()` lega un socket ad un indirizzo, per il successivo ascolto, che viene abilitato dalla funzione `listen()`:

```
#include <sys/types.h>
#include <sys/socket.h>

struct sockaddr_un {
    short sa_family;     // = AF_LOCAL
    char  sun_path[108]; // Indirizzo del socket
};

int bind(int sockfd, const struct sockaddr *addr, size_t addr_len);
int listen(int sockfd, int queue_size);
```

Il parametro `queue_size` di `listen()` è il numero massimo di client che possono restare in attesa di una connessione.

Nei socket locali l'indirizzo è il nome di un **file**. Il file viene creato da `bind()`, ma non si tratta di un file regolare, bensì di un **socket file**.

Uso dei socket

Funzioni `connect()` e `accept()`

Dopo la chiamata a `socket()`, un processo può connettersi come client ad un socket su cui esista un processo in ascolto.

```
int connect(int sockfd, const struct sockaddr *address, size_t addr_len);
```

La comunicazione si instaura effettivamente quando il server chiama `accept()`:

```
int accept(int sockfd, struct sockaddr *address, size_t *addr_len);
```

`accept()` blocca il processo finchè un client non si connette:

- restituisce un **nuovo** file descriptor collegato all'altro capo della comunicazione.
- Il vecchio file descriptor può contemporaneamente essere utilizzato per accettare un'altra connessione (solitamente forkando il processo)
- Da questo momento i due processi possono comunicare leggendo e scrivendo dati sui file descriptor a loro disposizione.

Leggere e scrivere da una socket

Una volta ottenuti i file descriptor dei due capi del socket, ci si può leggere e scrivere come su qualsiasi altro file. Esistono però delle funzioni più flessibili, specializzate ad operare su socket:

```
ssize_t send(int fd, const void *buffer, size_t length, int flags);
ssize_t recv(int fd, void *buffer, size_t length, int flags);
```

Le due funzioni operano come rispettivamente `write()` e `read()`, ma forniscono il parametro aggiuntivo `flags`, con il quale si possono specificare opzioni aggiuntive.

Esempio

Un programma di esempio di uso dei socket è allegato alle slide nell'archivio `upper.zip`:

- Il progetto è costituito da due programmi: un client e un server
- Il server resta in ascolto di connessioni, e rispedisce al client qualsiasi dato ricevuto, ma trasformato in maiuscolo.
- Il client legge righe di testo dallo standard input e le invia al server, stampando la risposta.
- Il server gestisce connessioni multiple creando un processo per ogni client, in modo che il processo padre possa tornare ad ascoltare mentre il figlio gestisce il singolo client.
- In allegato trovate il server, il client è per esercizio.

Esercizi

Esercizio

Per venerdì 9 dicembre

1. Scrivere il programma client per connettersi al programma `upperserver`. Il programma deve:
   - Connettersi al server e stampare il messaggio di benvenuto che viene ricevuto. Il server spedisce un dato di tipo `int` con la lunghezza del messaggio (compreso il terminatore nullo), seguito dal messaggio stesso.
   - Leggere righe di testo dallo standard input, inviarle al server, e ricevere la risposta, che sarà della stessa lunghezza, stampandola in output.