Signal

Meccanismo per notificare i processi

- l'utente preme ctrl-C al terminale: SIGINT
- accesso a un indirizzo illegale: SIGSEGV
- kill(2) system call, kill(1) utility

Sequenza di eventi:

1. il segnale è *generato*
2. ... il segnale è *pendente* ...
3. il segnale è *consegnato* (handled) quando il processo destinatario reagisce in maniera appropriata

I segnali sono identificati da *numeri* o *costanti simboliche*
es. SIGINT o 2: interrupt from keyboard

Unix supporta 31 diversi segnali

Gestione dei segnali

L'azione associata a un segnale può essere:

- Ignore the signal
- Catch the signal (invoca una nostra funzione)-
- Lasciare l'azione di default

```
$ sleep 10 &
[1] 1664
$ kill 1664
[1]+  Terminated              sleep 10
$ sleep 10
^C
$
```

Gestione dei segnali

Ogni segnale ha una *default action*
- abort: termina il processo con *core dump*
- exit: termina il processo
- ignore: ignora il segnale
- stop: sospende il processo
- continue: riavvia il processo, se sospeso

Un processo può *cambiare l'azione di default*:
- ignorare il segnale
- eseguire una subroutine definita dall'utente detta *signal handler*

Un processo può bloccare temporaneamente un segnale

per SIGKILL e SIGSTOP non si può cambiare l'azione di default

Default actions

SIGABRT	abnormal termination	terminate+core
SIGCHLD	change in status of a child	ignore
SIGSEGV	invalid memory reference	terminate+core
SIGINT	terminal interrupt character	terminate
SIGHUP	hangup	terminate
SIGALRM	timer expired (alarm)	terminate
⋮	⋮	⋮

Quando vengono generati?

- Eccezioni
  Es. tentativo di eseguire un istruzione illegale
- Altri processi
  Possono eseguire la kill(2) per mandare un segnale
- Interrupt da terminale
  I tasti ctrl-C o ctrl-\ mandano un interrupt al processo in *foreground*
- Job control
  Un processo in *background* che cerca di leggere o scrivere da terminale riceve un interrupt.
- Quote
  Es. quota di cpu ecceduta
- Notifiche
  I\O non bloccante: un processo riceve un signal quando l'operazione è terminata
- Alarm
  Un processo può richiedere un SIGALRM a una certa ora

Quando vengono gestiti?

Solo il processo destinatario può gestire il segnale

Il kernel controlla se ci sono segnali pendenti
- prima di ritornare in modo utente dopo un interrupt o syscall
- prima che il processo chiami wait_interruptible()
- all'uscita da wait_interruptible()

Se ci sono segnali pendenti, e il processo è in user-mode:
- il kernel salva il contesto corrente del processo ed esegue l'azione specificata

Se il processo è in kernel-mode:
- la syscall termina con errore EINTR (interrupted system call)

Molte chiamate di sistema vengono fatte *ripartire automaticamente* dopo la gestione dell'interrrupt

Esempio: ctrl-C da terminale

L'utente preme ctrl-C

Viene generato un interrupt (come per qualsiasi pressione di tasto)

Il driver del terminale riconosce che è un carattere speciale e manda SIGINT al processo in foreground

Quando il processo viene schedulato, gestirà il segnale al ritorno dal context switch

Se invece il processo è corrente, gestirà il segnale al ritorno dall'interrupt handler

Esempio: eccezione hardware

Il programma fa accesso a indirizzo di memoria illegale

La CPU genera un interrupt sincrono

L'interrupt handler manda SIGSEGV al processo

Quando l'interrupt handler ritorna dal kernel mode, il processo gestisce l'interrupt

E se il processo è bloccato?

Dipende da che cosa sta aspettando

- attese brevi (I/O completion): non vale la pena di interrompere l'attesa
- attese lunghe (input da terminale): l'attesa viene interrotta

Due categorie di wait:

- interruptible
- uninterruptible

Kill

```
NAME
     kill - send signal to a process

SYNOPSIS
     #include <signal.h>

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

Segnali inaffidabili

La prima implementazione di signal in Unix System V R3 era inaffidabile

```
void handler(int sig) {
  signal(SIGINT, handler);  // reinstalla lo handler
  ...                       // gestisci il segnale
}

main() {
  signal(SIGINT, handler); // installa lo handler
  ...
}
```

Gli handler non sono persistenti, e devono essere reinstallati ogni volta $\Rightarrow$ race condition!

Segnali affidabili

- Handler persistenti
- Masking
- Atomic “unblock and wait”

```
SYNOPSIS
       #include <signal.h>

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

DESCRIPTION
       The sigaction system call is used to change the action
       taken by a process on receipt of a specific signal.

       signum specifies the signal and can be any valid signal
       except SIGKILL and SIGSTOP.

       If act is non-null, the new action for signal signum is
       installed from act. If oldact is non-null, the previous
       action is saved in oldact.
```

```
The sigaction structure is defined as something like
       struct sigaction {
           void (*sa_handler)(int);
           void (*sa_sigaction)(int, siginfo_t *, void *);
           sigset_t sa_mask;
           int sa_flags;
       }
sa_handler specifies the action to be associated with
signum and may be SIG_DFL for the default action, SIG_IGN
to ignore this signal, or a pointer to a signal handling
function. This function receives the signal number as its
only argument.

sa_sigaction also specifies the action to be associated
with signum. This function receives the signal number as
its first argument, a pointer to a siginfo_t as its second
argument and a pointer to a ucontext_t (cast to void *) as
its third argument.

sa_mask gives a mask of signals which should be blocked
during execution of the signal handler. In addition, the
signal which triggered the handler will be blocked, unless
the SA_NODEFER or SA_NOMASK flags are used.
```

Scrivere codice di rete scalabile

Basato su “Scalable Network Programming” di Felix von Leitner

http://bulk.fefe.de/scalable-networking.pdf

Il protocollo HTTP

Richiesta HTTP: dammi il documento /pippo.html

```
GET /pippo.html HTTP/1.1
Host: www.example.com
(una riga vuota indica la fine della richiesta)
```

Risposta HTTP: eccolo

```
HTTP/1.1 200 OK
Date: Wed, 12 Sep 2007 20:25:50 GMT
Server: Apache/1.3.37 (Unix) PHP/4.4.7
Content-Length: 2673
Content-Type: text/html
Content-Language: en

<html>
 <head>
  <title>Pippo!</title>
... seguono N righe di codice HTML
```

A simple web server

Pseudocodice:

```
crea una socket s in attesa sulla porta 80
aspetta una connessione
quando arriva una connessione
  leggi la richiesta
  rispondi
```

A simple web server

```
int cfd,fd=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP);
struct sockaddr_in si;
si.sin_family=PF_INET;
inet_aton("127.0.0.1",&si.sin_addr);
si.sin_port=htons(80);
bind(fd,(struct sockaddr*)si,sizeof si);
listen(fd);
while ((cfd=accept(fd,(struct sockaddr*)si,sizeof si)) != -1) {
  read_request(cfd);
  write(cfd,"200 OK HTTP/1.0\r\n\r\nCiao ciao.", 19+10);
  close(cfd);
}
```

A simple web server

```
int cfd; // fd della socket che comunica con il client
while ((cfd = accept(...)) != -1) {
  read_request(cfd);
  write(cfd, "200 OK HTTP/1.0\r\n\r\nCiao ciao.", 19+10);
  close(cfd);
}
```

Questo server fa schifo:

- non implementa davvero il protocollo
- può accettare una sola connessione per volta :-(

Un web server migliore

```
int cfd; // fd della socket che comunica con il client
while ((cfd = accept(...)) != -1) {
  if (fork() == 0) {
    /* handle connection in a child process */
    read_request(cfd);
    write(cfd, "200 OK HTTP/1.0\r\n\r\nCiao ciao.", 19+10);
    close(cfd);
    exit(0);
  }
}
```

Un processo per connessione

Un processo per connessione

Vecchia scuola: si fa così fin dagli anni '70

Faceva così il primo web server al CERN

Problema: il costo della fork(2)

Un benchmark per misurare il costo di una fork(2)

```
pipe(pfd);
for (i=0; i<4000; ++i) {
  gettimeofday(&a, 0);
  if (fork() > 0) {
    write(pfd[1], "+",1); block(); exit(0);
  }
  read(pfd[0], buf, 1);
  gettimeofday(&b, 0);
  printf(%llu\n, difference(&a,&b));
}
```

Each child will write a single character into a pipe, and the parent will read that single character out of the pipe. Then the child will block and wait for SIGTERM.

Pre-forking

Pseudocodice:

```
Crea N processi lavoratori
Crea una socket s in attesa sulla porta 80
aspetta una connessione
quando arriva una connessione
  passa la richiesta a un processo lavoratore
```

Usato da Apache http server

Il modello una-connessione-per-processo

Pro:

- Facile da implementare
- Sicuro: ogni processo è isolato dagli altri
- Ragionevolmente performante
  - Tempo di fork in Linux 2.6: circa 200 microsecondi => 5000 fork/s

Contro:

- Il sistema operativo è tunato per pochi processi (< 4000)
- In particolare, per pochi processi *runnable* (< 3 per CPU)

Usare thread invece che processi

```
Crea N processi lavoratori
Crea una socket s in attesa sulla porta 80
aspetta una connessione
quando arriva una connessione
  passa la richiesta a un processo lavoratore
```

FIFO

A pipe with a name and a place in the filesystem

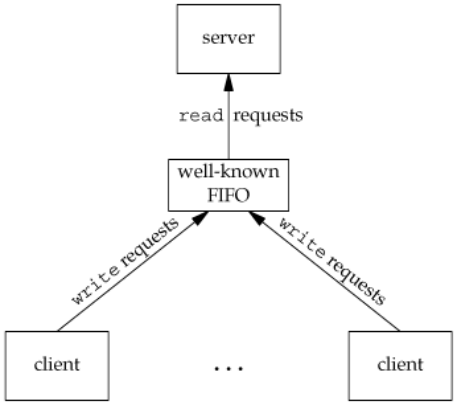
Can be used for inter-process communication even by processes that are not in a parent-child relationship

La chiamata mkfifo(2) è simile alla open(2)

FIFO

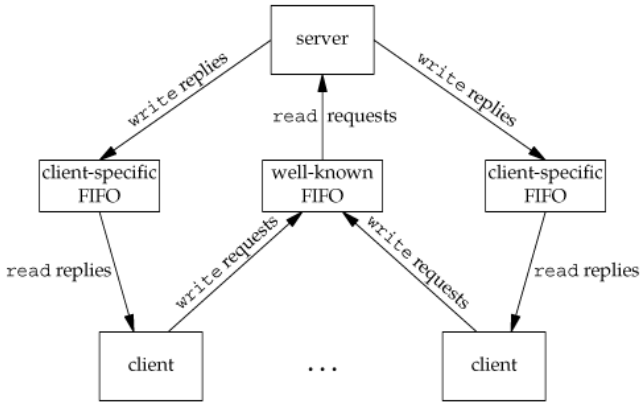
```
NAME
     mkfifo - make a fifo file

SYNOPSIS
     #include <sys/types.h>
     #include <sys/stat.h>

     int mkfifo(const char *path, mode_t mode);

DESCRIPTION
     Mkfifo() creates a new fifo file with name path.  The access permissions
     are specified by mode and restricted by the umask(2) of the calling
     process.

     The fifo's owner ID is set to the process's effective user ID.  The
     fifo's group ID is set to that of the parent directory in which it is
     created.

RETURN VALUES
     A 0 return value indicates success.  A -1 return value indicates an
     error, and an error code is stored in errno.
```

Usare i FIFO per sistemi client-server



Problema: come restituiamo le risposte ai client?

Usare i FIFO per sistemi client-server



Soluzione: usiamo un pathname canonico basato sul pid del client.