

Processi – parte III

Sincronizzazione e comunicazione tra processi in Unix

- Segnali:
 - usati per trasferire ad un processo l'indicazione che un determinato evento si è verificato.
- Pipe:
 - struttura dinamica, creata ed usata dal processo che la genera.
- Fifo:
 - struttura simile alla Pipe, ma statica. Sopravvive ai processi che la usano.

Segnali

- Un segnale può essere inviato ad un processo al verificarsi di una determinata condizione.
- Normalmente i segnali provocano la **terminazione** del processo che li riceve:
 - terminazione normale (simile a quella ottenuta con exit)
 - terminazione anormale (core dumped).

Segnali

- Un processo può specificare una specifica azione da intraprendere all'arrivo di un segnale:
 - system call signal
- Ad ogni segnale è associato un nome simbolico usato dall'utente e un numero intero usato dal sistema.
- La corrispondenza è stabilita nel file **signal.h**:
 - /usr/lib/bcc/include/

Segnali: signal.h

- Esempi di segnali:

SIGINT	int. da terminale
SIGILL	istruzione non consentita
SIGALRM	compl. di un quanto di tempo
SIGKILL	terminazione di un processo
SIGTERM	terminazione di un processo (intercettabile)
SIGUSR1	
SIGUSR2	inviati da un processo per sincronizzazione

System call signal

- `int sig;`
`void(*func)();`
`signal(sig,func);`
- `sig`: specifica il nome del segnale
- `func`: è il puntatore ad una funzione che specifica come deve essere trattato il segnale

System call signal

- La signal deve essere chiamata prima che il segnale sia ricevuto, altrimenti, si ha il comportamento di default (terminazione).
- `signal(sig,SIG-IGN)` il segnale è ignorato dal processo.
- `signal(sig,SIG-DFL)` ristabilisce l'azione di default.

Esempio

```
#include<signal.h>
extern clean-exit();

main()
{
    signal(SIGINT,SIG-IGN);
    /*ignora interruzione da
    terminale*/
    signal(SIGHUP,SIG-DFL);
    /*termina allo spegnimento del
    terminale*/
    signal(SIGTERM,clean-exit);
    /*gestione del segnale di
    terminazione SIGTERM*/
    etc.
}

clean-exit()
{
    /*copia i buffer in memoria
    centrale su disco*/
    printf("salvataggio buffer
    completato\n");
    exit(1);
}
```

System call per la gestione dei segnali

- Unix fornisce altre primitive per la gestione dei segnali:
 - installazione di un allarme: **ALARM(sec)** (non è bloccante)
 - sospensione in attesa di un qualunque segnale: **PAUSE()**.
 - sospensione temporizzata: **SLEEP(sec)**
 - invio di segnali ad altri processi: **KILL(pid,sig)**

System call: alarm, pause e sleep

- La system call `alarm(sec)` setta un allarme per un segnale di tipo `SIGALRM` che dovrà essere inviato al processo dopo `sec` secondi.
- La system call `pause()` blocca il processo che la esegue in attesa di un qualunque segnale:
 - Il processo deve predisporre alla gestione del segnale che gli arriverà pena la terminazione
- La system call `sleep(sec)` blocca il processo che la esegue per `sec` secondi

System call kill

- La system call kill invia un messaggio specificato ad un processo specificato (indicando il pid):
 - Se non è specificato nessun segnale viene inviato SIGTERM
 - Il segnale inviato uccide il processo che lo riceve se non è predisposto alla sua gestione
 - Alcuni segnali come SIGKILL non possono essere "intercettati"

System call kill: esempio

```
void azione()
{
    printf("segnale ricevuto\n");
}
main()
{
    int pid;
    signal(SIGUSR1,azione);
    pid = fork();
    if(pid == 0)
    {
        pause();
        etc.
    }
    else{
        .....
        kill(pid,SIGUSR1);
        .....
    }
}
```

Pipe

- Canale di comunicazione unidirezionale: accessibile ad un estremo in lettura ed all'altro estremo in scrittura.
- Gestione Fifo, dimensione fissa (es. 4096 byte)
- Sincronizzazione tra processi:
 - un processo che legge si blocca se la pipe è vuota.
 - un processo che scrive si blocca se la pipe è piena.
- `int retval, fd[2];`
`retval = pipe(fd);`
- `fd[0]` identificatore del "file" aperto in lettura, `fd[1]` identificatore del file aperto in scrittura.

Pipe

- Comunicazione con pipe nella gerarchia di processi; ad esempio:
 - tra processi figli (ereditano il pipe dal padre)
 - tra un processo padre e un processo figlio.

Produttore e consumatore con pipe

```
main()
{
    int fd[2],pid,retval,status;
    .....
    retval = pipe(fd);
    .....
    if(pid=fork( ) == 0) /*produttore*/
        /*scrittura su fd[1]*/
    if(pid=fork( ) == 0) /*consumatore*/
        /*lettura su fd[0]*/
    pid=wait(&status);
    pid=wait(&status);
    /*attende entrambe le terminazioni*/
}
```

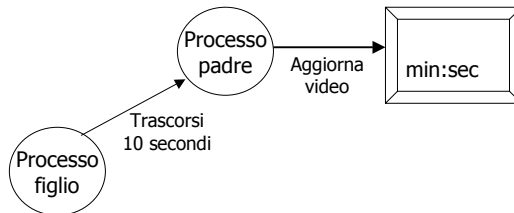
FIFO

- La pipe ha lo svantaggio di consentire la comunicazione solo tra processi in relazione di parentela, e di essere distrutta al termine del suo utilizzo.
- FIFO: canale unidirezionale del tipo first-in-first-out:
 - possiede un nome di file di sistema, permane nel sistema, ha un proprietario, un insieme di diritti e una lunghezza.
 - è creata dalla system call mknod:

```
int retval, mode;
char* path;
retval = mknod(path,mode);
```

Esercizio: orologio

- Realizzare un programma C in ambiente Unix che gestisca un contatore minuti-secondi così organizzato:
 - Esistono due processi uno padre e uno figlio
 - Il figlio avverte il padre con un segnale ogni dieci secondi
 - Quando riceve il segnale, il padre aggiorna il contatore a video



Soluzione

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

#define TRUE 1

int min=0,sec=0;

void stampa( )
{
    signal(SIGUSR1,stampa);
    sec +=10;
    if(sec == 60)
    {
        sec = 0;
        min ++;
        if (min == 60)
            min = 0;
    }
    printf("min: %d sec: %d\r",min,sec);
    fflush(stdout);
    return;
}

main( )
{
    int pid;

    signal(SIGUSR1,stampa);
    printf("min: %d sec: %d\r",min,sec);
    fflush(stdout);

    if((pid=fork())==0)
    {
        while(TRUE)
        {
            sleep(10);
            kill(getppid(),SIGUSR1);
        }
    }
    else
    {
        while(TRUE)
            pause();
    }
}
```