

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

Uso di PIPE e FIFO

Claudio Vicari

Definizioni

- ➔ Le *pipe* sono stati i primi strumenti per l'IPC in Unix
 - sono prive di nome, quindi possono essere usate solo da processi che abbiano una relazione “di parentela”
- ➔ Le *FIFO* sono state aggiunte in seguito, per scavalcarne i limiti. A volte sono chiamate anche “Named PIPEs”

Pipes

- ➔ una pipe è un flusso di dati unidirezionale
- ➔ si crea con la funzione
`int pipe(int fd[2]);`
- ➔ restituisce nel vettore fd due nuovi file descriptor
 - fd[0] viene aperto in lettura
 - fd[1] viene aperto in scrittura
- ➔ quel che viene scritto in fd[1] può essere letto in fd[0]

Pipes

- ➔ una coppia di file descriptor, quindi, viene creata all'interno di un singolo processo, con una singola chiamata
- ➔ tipicamente la funzione viene chiamata da un processo prima di una fork, quindi la pipe viene usata per la comunicazione fra due processi che discendono dal primo
- ➔ un processo tipicamente chiude almeno uno dei due estremi della pipe, per usare solo l'altro
- ➔ quando chiamiamo comandi in una shell come:
 - `cmd1 | cmd2 | cmd3 ....`
 - la shell crea una pipe per ogni segno |

Pipes: esempio client/server

- ➡ il main crea due pipe e un processo
 - ➡ il parent serve da client, il child da server
 - ➡ il client:
 - legge un pathname da tastiera
 - lo invia al server
 - riceve dal server il file e lo stampa
 - ➡ il server legge dal client il nome del file e ne reinvia il contenuto
- ```
$> gcc -o /tmp/prova mainpipe.c wrapunix.c
client.c server.c
```

Client

Server

main

# *Pipe: funzione popen, pclose*

```
#include <stdio.h>
FILE *popen(const char *cmd, const char *type);
int pclose(FILE *stream);
```

- ➔ popen crea una pipe, quindi fork e invoca cmd in una shell
- ➔ se type=="r", il processo chiamante può leggere lo standard output di cmd
- ➔ se type=="w", il processo chiamante può scrivere nello standard input di cmd
- ➔ pclose chiude lo stream, quindi fa wait

# *Esempi: funzione popen*

Visualizza (r)

Visualizza (w)

- ➔ esempio-popen-r.c
  - stampiamo la prima riga di risultato di una grep su /etc/services
- ➔ esempio-popen-w.c
  - in output appare il risultato di una grep su quello che le fprintf scrivono

# ***FIFO***

- ➔ le PIPE sono identificate solo dal fd. Solo processi con un 'avo' in comune possono usarle per comunicare
- ➔ FIFO significa *First In, First Out*. Il comportamento di una FIFO, comunque, è identico a quello di una pipe
- ➔ le FIFO sono identificate da un nome, che consente a più processi di riferirsi alla stessa struttura

# *Creare FIFO*

```
int mkfifo(const char *pathname, mode_t
mode);
```

- ➔ crea una FIFO **come file** nel pathname specificato
- ➔ questa funzione può creare una FIFO solo se non ce n'è già una con lo stesso pathname
- ➔ anche con un comando da shell si può creare una FIFO
  - \$a> mkfifo /tmp/prova\_fifo
  - \$a> file /tmp/prova\_fifo
  - \$a> echo "dati" > /tmp/prova\_fifo
  - \$b> cat /tmp/prova\_fifo (legge)
  - \$b> cat /tmp/prova\_fifo (si blocca!)

# Utilizzare FIFO

- ➔ quindi, con `mkfifo`, la struttura viene creata nel file system, da cui si può accedere
- ➔ con `find / -type p` possiamo vedere che alcune applicazioni usano proprio queste strutture
- ➔ per aprirne una già esistente, si usa `open`
- ➔ la FIFO può essere aperta (con `open`) o read-only oppure write-only – è *half-duplex*
- ➔ quando si fa write in una FIFO, i dati vengono scritti alla fine (append), mentre una read restituisce sempre quello che c'è all'inizio
- ➔ non è possibile fare `lseek`, né per le pipe né per le FIFO

# *FIFO: client/server indipendenti*

- ➔ `gcc -o /tmp/client client_main.c wrapunix.c client.c`
- ➔ `gcc -o /tmp/server server_main.c wrapunix.c server.c`
- ➔ lanciare prima il server e poi il client
- ➔ funziona come l'esempio con le PIPE, ma i due programmi vengono lanciati separatamente

Client

Server

# *Altre proprietà di PIPE e FIFO*

- ➔ se si cerca di leggere più dati di quelli disponibili, una read restituisce tutto quello che può leggere – attenzione a gestire questa situazione
- ➔ la write è atomica, se scriviamo un numero di byte minore di PIPE\_BUF (4096 in SuSE Linux 10.0)
- ➔ se si scrive in una pipe o FIFO che nessuno ha aperto in lettura, riceviamo un SIGPIPE
  - se il processo non lo gestisce, viene terminato
  - altrimenti write restituisce l'errore EPIPE
  - il modo più semplice è ignorare SIGPIPE e controllare l'errore di write

# *Usi tipici delle FIFO*

- ➔ poiché le FIFO sono file la cui posizione è nota, un demone può usarli per gestire richieste da parte di numerosi client
  - grazie all'atomicità delle write, possiamo garantire che le richieste dei client non saranno confuse
  - la comunicazione può comunque essere spostata su un'altra FIFO per uno specifico client
- ➔ le FIFO, pur risiedendo sul filesystem, possono essere usate solo su filesystem locali – non consentono alcuna comunicazione fra processi che risiedono in macchine differenti

# *Alternative nell'uso delle FIFO*

- ➔ per distinguere richieste differenti, possiamo usare le solite alternative:
  - 1 terminare ogni richiesta con una sequenza di caratteri nota
  - 2 imporre una lunghezza fissa alle richieste
  - 3 nei socket, possiamo chiudere la connessione al termine della richiesta
- ➔ tutti i file descriptor possono essere usati con la libreria `stdio.h`: basta usare `fdopen` o `fopen`
- ➔ `PIPE_BUF` per lo standard Posix dipende dalla path specificata: con `pathconf` possiamo scoprire questo limite al runtime

# *Esercizi*

- ➡ scrivere un server ed un client:
  - il server apre una FIFO, e vi attende richieste dai client
  - ogni client scrive al server il nome di un file
  - il server restituisce al client la seconda riga del file. Il client, quindi, la stampa ed esce
  - il server può gestire solo un client per volta
- ➡ scrivere un programma che scriva in una FIFO due valori interi, passatigli da riga di comando
- ➡ scrivere un altro programma che stampi a video la somma di questi valori