UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Laboratorio di Sistemi Operativi

IPC: FIFO

LEZIONE 13

prof. Antonino Staiano

Corso di Laurea in Informatica – Università di Napoli Parthenope

antonino.staiano@uniparthenope.it

Pipe con nome: FIFO

- Fin qui siamo stati in grado di scambiare i dati tra processi legati tra loro:
  - Tali processi sono avviati da un antenato comune
  - Questa è una limitazione poiché talvolta è necessario che processi tra loro non in relazione siano in grado di scambiarsi i dati
- Questo è possibile farlo con l'uso delle pipe con nome o FIFO
  - Le FIFO sono pipe che possono connettere due (o più) processi qualsiasi
  - Una FIFO è un tipo speciale di file che si comporta come le pipe senza nome
    - La creazione di una FIFO è simile alla creazione di un file
      - Il pathname di una FIFO esiste all'interno del file system

Pipe con nome: mkfifo()

- Possiamo creare una FIFO dalla linea di comando o da programma
  - Dalla linea di comando:
    - $ mkfifo filename
  - Da programma:

```
#include <sys/stat.h>
int mkfifo(const char *filename, mode_t mode);
/* restituisce 0 se OK, -1 in caso di errore */
```

Pipe con nome: mkfifo()

- La funzione mkfifo() implica O_CREAT|O_EXCL
  - Crea una nuova FIFO o restituisce un errore EEXIST se la FIFO già esiste
  - Se non è desiderata la creazione di una nuova FIFO, è necessario invocare open() anziché mkfifo()
  - Per aprire una FIFO esistente o creare una nuova FIFO se questa non esiste
    1. Si invoca mkfifo()
    2. Si controlla un eventuale errore EEXIST e, se questo si verifica, si invoca open()

Esempio: mkfifo()

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main(){
 int res = mkfifo("/tmp/my_fifo", 0777);
 if (res == 0)
 printf("FIFO creata\n");
  exit(0);
}
```

Esempio: mkfifo() (cont.)

- Il programma usa la funzione mkfifo() per creare un file speciale
- Sebbene impostiamo i permessi 0777 questi sono filtrati dalla maschera dell'utente (umask), così come avviene nella normale creazione di file
  - I permessi risultanti saranno 755 se, ad esempio, umask è 022
- Possiamo rimuovere la FIFO come un file convenzionale usando il comando rm o usando la system call unlink() da programma
- Dopo aver eseguito il programma:

```
$ ls -lF /tmp/my_fifo
```

```
prwxr-xr-x   1 staiano   staiano 0 April 3  14:55 /tmp/my_fifo|
```

Accedere ad una FIFO

- Una caratteristica utile delle pipe con nome è che risiedendo nel file system per cui le possiamo usare in comandi dove normalmente utilizziamo nomi di file
  - Proviamo a leggere dalla FIFO (vuota)

```
$ cat < /tmp/my_fifo
```

  - Ora scriviamo nella FIFO (usando un altro terminale, infatti il primo comando si sospende in attesa di dati scritti nella FIFO)

```
$ echo "sdsdfasd" > /tmp/my_fifo
```

    Si vedrà l'output apparire dal comando cat. Se non si invia alcun dato alla FIFO, il comando cat si sospenderà fino a che non lo si interrompe (Ctrl-C)
  - Possiamo fare entrambe le cose ponendo il primo comando in background

```
$ cat < /tmp/my_fifo &
[1]  1316
$ echo "sdsdfasd" > /tmp/my_fifo
 sdsdfasd
```

Accedere ad una FIFO

- Poiché non ci sono dati nella FIFO, il programma cat si blocca, in attesa di qualche dato in arrivo
- Nell'ultimo caso (cat in background), il processo cat inizialmente è bloccato ed in background
- Quando echo rende disponibili dei dati, il comando cat legge i dati e li stampa sullo standard output
- Osserviamo che il programma cat, poi, esce senza aspettare altri dati
  - Esso non si blocca perché la FIFO sarà chiusa quando il secondo comando che immette i dati nella FIFO ha finito

Aprire una FIFO con open()

- Abbiamo appena visto come si comporta una FIFO quando vi accediamo usando la riga di comando
- Vediamo ora il comportamento da programma quando vi accediamo in lettura e scrittura
- La principale restrizione quando si apre una FIFO è che un programma non può aprirla per leggere e scrivere nella modalità O_RDWR
  - In tal caso il risultato è indefinito
  - Questa è una restrizione relativa, poiché usiamo le FIFO per passare i dati in una singola direzione e dunque non c'è necessità di aprirla in modalità O_RDWR
- Un'operazione di scrittura su una FIFO aggiunge sempre i dati in coda e un'operazione di lettura restituisce sempre ciò che si trova all'inizio della FIFO

Aprire una FIFO con open() (cont.)

- Un'altra differenza nell'aprire una FIFO rispetto ad un file regolare, è l'uso dell'argomento oflag (il secondo parametro di open) con l'opzione O_NONBLOCK
  - L'utilizzo di questa modalità di apertura non solo cambia il modo in cui la chiamata ad open viene elaborata, ma cambia anche il modo in cui sono elaborate le richieste di lettura e scrittura sul descrittore di file restituito
  - Ci sono 4 combinazioni consentite di O_RDONLY, O_WRONLY e O_NONBLOCK

Aprire una FIFO con open() (cont.)

- `open(const char *path, O_RDONLY);`

  in questo caso, la chiamata ad open si bloccherà; non ritorna fino a che un processo apre la stessa FIFO per scrittura

- `open(const char *path, O_RDONLY | O_NONBLOCK);`

  la chiamata ad open ha successo e ritorna immediatamente, anche se la FIFO non è stata aperta in scrittura da alcun processo

- `open(const char *path, O_WRONLY);`

  la chiamata ad open si bloccherà fino a che un processo apre la stessa FIFO in lettura

- `open(const char *path, O_WRONLY | O_NONBLOCK);`

  ritorna sempre immediatamente, ma se nessun processo ha la FIFO aperta in lettura, open ritornerà un errore, -1, e la FIFO non sarà aperta. Se un processo ha aperto la FIFO in lettura, il descrittore di file restituito può essere usato per scrivere al suo interno

Esempio

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
int main(int argc, char *argv[]){
  int res;
  int open_mode = 0;
  int i;
  if (argc < 2) {
       printf("Uso: %s <combinazioni di O_RDONLY O_WRONLY O_NONBLOCK>\n", argv[0]);
       exit(1);
  }
```

Esempio (cont.)

```
/* Impostiamo il valore di open_mode dagli argomenti. */
 for(i = 1; i < argc; i++) {
  if (strncmp(*++argv, "O_RDONLY", 8)==0) open_mode |= O_RDONLY;
  if (strncmp(*argv, "O_WRONLY", 8)==0)   open_mode |= O_WRONLY;
  if (strncmp(*argv, "O_NONBLOCK", 10)==0)open_mode |= O_NONBLOCK;
  }
/* Se la FIFO non esiste la creiamo. Poi viene aperta */
 if (access(FIFO_NAME, F_OK) == -1) {
  res = mkfifo(FIFO_NAME, 0777);
  if (res != 0) {
  printf("Non posso creare la FIFO %s\n", FIFO_NAME);
  exit(1);
  }
}
```

```
#include <unistd.h>
int access(const char *pathname, int mode);
/* Restituisce 0 se OK, -1 in caso di errore */
Verifica l'accesibilità del real user
Mode può essere R_OK, W_OK, X_OK, F_OK.
```

Esempio (cont.)

```
printf("Processo %d apre la FIFO\n", getpid());
res = open(FIFO_NAME, open_mode);
printf("Risultato processo %d: %d\n", getpid(), res);
sleep(5);
if (res != -1) close(res);
printf("Processo %d terminato\n", getpid());
exit(0);
}
```

Esempio

- Il programma ci consente di specificare da linea di comando la combinazione di O_RDONLY, O_WRONLY e O_NONBLOCK che vogliamo usare

- Il programma usa la chiamata di sistema access() per verificare se il file FIFO esiste già, creandolo se necessario

- Non cancelliamo la FIFO, poiché non abbiamo modo di dire se un altro programma ha già la FIFO in uso

O_RDONLY e O_WRONLY senza O_NONBLOCK

```
$ ./a.out O_RDONLY &
[1] 152
Processo 152 apre la FIFO
$ ./a.out O_WRONLY
Processo 153 apre la FIFO
Risultato processo 152: 3
Risultato processo 153: 3
Processo 152 terminato
Processo 153 terminato
```

- Consente al processo lettore di iniziare e di aspettare la chiamata ad open dello scrittore
  - poi consente ad ambo i programmi di continuare quando il secondo programma apre la FIFO
- Osserviamo che entrambi i processi lettore e scrittore si sono sincronizzati alla chiamata di open()

O_RDONLY e O_WRONLY con O_NONBLOCK

```
$ ./a.out O_RDONLY O_NONBLOCK &
[1] 160
Processo 160 apre la FIFO
Risultato processo 160: 3
$ ./a.out O_WRONLY
Processo 161 apre la FIFO
Risultato processo 161: 3
Processo 160 terminato
Processo 161 terminato
```

- In questo caso, il processo lettore esegue la chiamata ad open e continua immediatamente, anche se non è presente alcun processo scrittore
- Lo scrittore anch'esso continua immediatamente dopo la chiamata ad open() poiché la FIFO è già aperta in lettura

Leggere e scrivere in una FIFO bloccante

- Se la FIFO è aperta in modalità bloccante
  - Una read() su una FIFO vuota
    - aspetterà fino a che è disponibile qualche dato da leggere, se la FIFO è aperta in scrittura
    - Restituisce 0, se la FIFO non è aperta in scrittura
  - Una write() su una FIFO
    - Aspetterà fino a che i dati possono essere scritti, se la FIFO è aperta in lettura
    - Genera un segnale SIGPIPE se la FIFO non è aperta in lettura

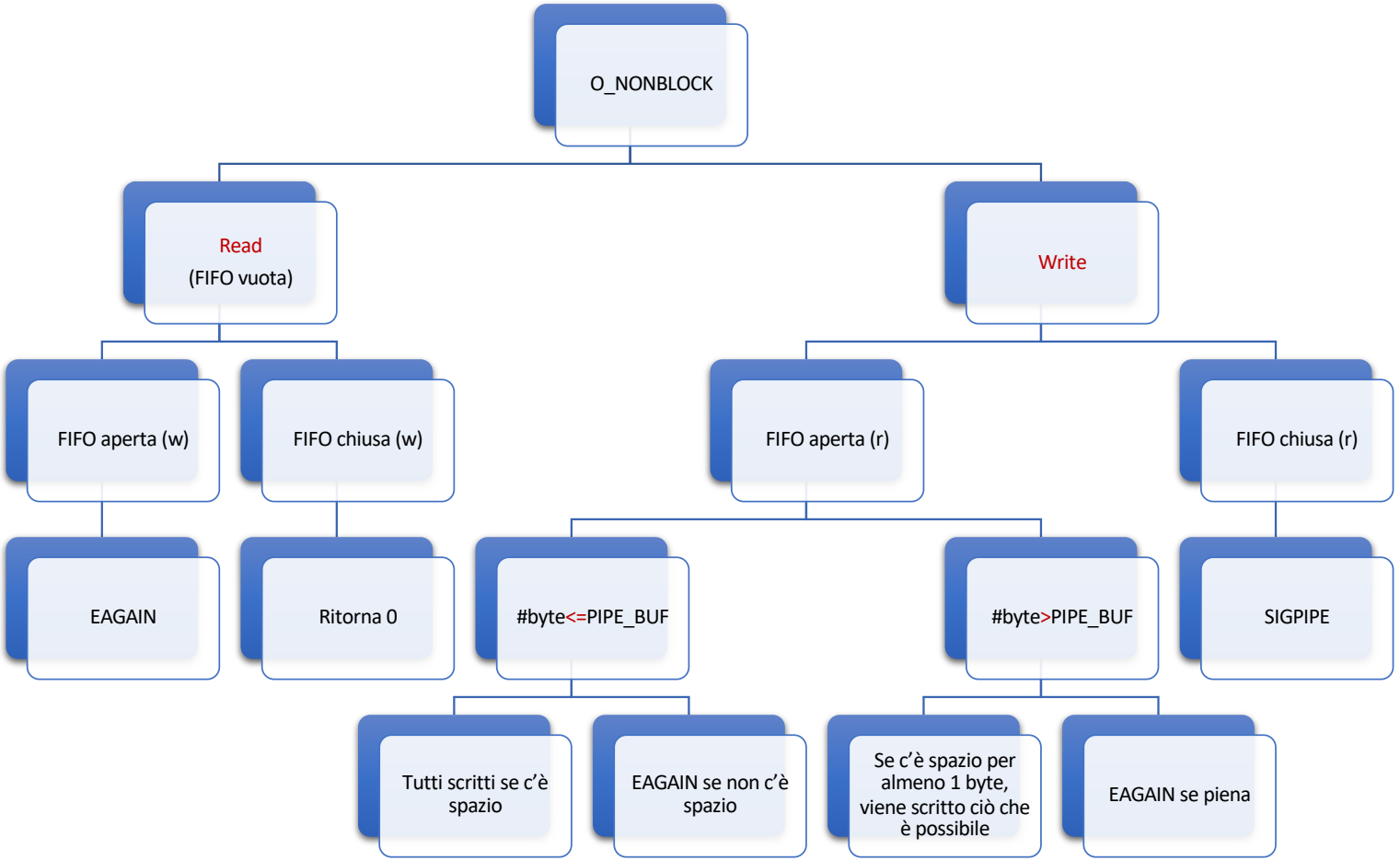
Leggere dalle FIFO non bloccanti

- L'utilizzo della modalità O_NONBLOCK influisce sul comportamento delle chiamate read() sulle FIFO
  - Una read() su una FIFO vuota non bloccante
    - Restituisce un errore (EAGAIN) se la FIFO è aperta in scrittura
    - Restituisce 0, se la FIFO non è aperta in scrittura

Scrivere sulle FIFO non bloccanti

- L'utilizzo della modalità O_NONBLOCK influisce sul comportamento delle chiamate write() sulle FIFO
  - Una write() su una FIFO non bloccante
    - Genera un segnale SIGPIPE se la FIFO non è aperta in lettura
    - Se la FIFO è aperta in lettura
      - Se il numero di byte da scrivere è <= di PIPE_BUF
        - Se c'è spazio per il numero di byte specificato, sono trasferiti tutti i byte
        - Se non c'è spazio per tutti i byte specificati la write() ritorna immediatamente con un errore (EAGAIN)
      - Se il numero di byte è > PIPE_BUF
        - Se c'è spazio per almeno 1 byte nella FIFO, il kernel trasferisce tanti byte quanto spazio c'è nella FIFO e la write() restituisce il numero di byte scritti
        - Se la FIFO è piena, ritorna immediatamente con l'errore EAGAIN

Lettura e scrittura con O_NONBLOCK impostato



Esempio: comunicazione tra due processi con FIFO (produttore-consumatore)

```
#include <unistd.h>
…
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1000 * 10)
int main(){
 int pipe_fd;
 int res;
 int open_mode = O_WRONLY;
 int bytes_sent = 0;
 char buffer[BUFFER_SIZE];
 if (access(FIFO_NAME, F_OK) == -1) {
   res = mkfifo(FIFO_NAME, 0777);
   if (res != 0) {
     printf("Could not create fifo %s\n", FIFO_NAME);
     exit(-1);}
     }
```

Produttore (fifo1.c): crea una FIFO se richiesto, poi vi scrive i dati quanto prima.

Esempio (cont.)

```
printf("Process %d opening FIFO O_WRONLY\n", getpid());
pipe_fd = open(FIFO_NAME, open_mode);
printf("Process %d result %d\n", getpid(), pipe_fd);
if (pipe_fd != -1) {
   while(bytes_sent < TEN_MEG) {
            // supponiamo che buffer sia riempito altrove
               res = write(pipe_fd, buffer, BUFFER_SIZE);
             if (res == -1) {
              printf("Write error on pipe\n");
                  exit(1);
                 }
                 bytes_sent += res;
   }
close(pipe_fd);
}
else
{
 exit(1);
}
printf("Process %d finished\n", getpid());
exit(1);
}
```

Esempio (cont.)

Consumatore (fifo2.c): legge ed elimina i dati dalla FIFO

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
int main(){
  int pipe_fd;
  int res;
  int open_mode = O_RDONLY;
  char buffer[BUFFER_SIZE];
  int bytes_read = 0;
  printf("Process %d opening FIFO O_RDONLY\n", getpid());
  pipe_fd = open(FIFO_NAME, open_mode);
  printf("Process %d result %d\n", getpid(), pipe_fd);
```

Esempio (cont.)

```
if (pipe_fd != -1) {
  do {
       res = read(pipe_fd, buffer, BUFFER_SIZE);
       bytes_read += res;
  } while (res > 0);
    close(pipe_fd);
}
else {
  exit(-1);
}
printf("Process %d finished,%d bytes read\n",getpid(),bytes_read);
exit(0);
}
```

Esempio (cont.)

```
$ ./fifo1 &
[1] 375
Process 375 opening FIFO O_WRONLY
$ ./fifo2
Process 377 opening FIFO O_RDONLY
Process 375 result 3
Process 377 result 3
Process 375 finished
Process 377 finished, 10485760 byte read
```

Esempio (cont.)

- Entrambi i programmi usano la FIFO in modo bloccante
- Iniziamo *fifo1* (lo scrittore/produttore) per primo, che si blocca, in attesa di un lettore che apra la FIFO
- Quando *fifo2* (il consumatore) è avviato, lo scrittore è sbloccato ed avvia la scrittura dei dati nella pipe
  - Nello stesso tempo, il lettore inizia a leggere i dati dalla pipe

Comunicazione Client-Server con FIFO

- Un utilizzo delle FIFO consiste nell'inviare i dati tra un client ed un server
  - Se abbiamo un server che è contattato da numerosi client, ogni client può scrivere la sua richiesta ad una FIFO che il server crea
  - Poiché possono esserci multipli scrittori per la FIFO, le richieste inviate dai client al server devono essere minori di PIPE_BUF byte di dimensione
    - Previene intrecci delle scritture dei client

Comunicazione Client-Server con FIFO

- Vogliamo un singolo processo server che accetta richieste, le elabora, e restituisce i dati risultanti alla parte richiedente (il client)
- Vogliamo consentire a processi client multipli di inviare dati al server
  - Per semplicità assumiamo che i dati da elaborare siano spezzati in blocchi, ciascuno più piccolo di PIPE_BUF byte
- Poiché il server elaborerà solo un blocco di informazione per volta, consideriamo un'unica FIFO che è letta dal server e scritta da ciascun client
  - Aprendo la FIFO in modo bloccante, il server ed i client saranno automaticamente sincronizzati come richiesto
- Restituire i dati ai client è leggermente più difficile
  - Abbiamo bisogno di organizzare una seconda pipe, una per client, per i dati ritornati
  - Passando l'identificatore di processo (PID) del client nei dati originali inviati al server, entrambe le parti possono usarlo per generare il nome unico per la FIFO di ritorno

Esempio: client-server

- In primis, abbiamo bisogno di definire un file header **client.h** che definisce i dati comuni ai programmi client e server

```
/* client.h */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define SERVER_FIFO_NAME "/tmp/serv_fifo"
#define CLIENT_FIFO_NAME "/tmp/cli_%d_fifo"
#define BUFFER_SIZE 20
struct data_to_pass_st {
  pid_t  client_pid;
  char   some_data[BUFFER_SIZE];};
```

/* server.c */

```
#include "client.h"
#include <ctype.h>
int main(){
  int server_fifo_fd, client_fifo_fd;
  struct data_to_pass_st my_data; // struttura da leggere e restituire
  int read_res;
  char client_fifo[256]; // per il nome della FIFO del client
  char *tmp_char_ptr;
  mkfifo(SERVER_FIFO_NAME, 0777);
  server_fifo_fd = open(SERVER_FIFO_NAME, O_RDONLY);
  if (server_fifo_fd == -1) {
        printf("Server fifo failure\n");
        exit(1);
  }
  sleep(10); // accodiamo i client per scopi dimostrativi
  do {
  read_res = read(server_fifo_fd, &my_data, sizeof(my_data));
  if (read_res > 0) {
/* In questa prossima fase, eseguiamo qualche operazione sui dati appena letti dal client.
  Convertiamo tutti i caratteri in lettere maiuscole e combiniamo il CLIENT_FIFO_NAME con il pid del
  client ricevuto. */
```

/* server.c */ (cont.)

```
tmp_char_ptr = my_data.some_data;
while (*tmp_char_ptr) {
  *tmp_char_ptr = toupper(*tmp_char_ptr);
  tmp_char_ptr++;
}
 sprintf(client_fifo,CLIENT_FIFO_NAME,my_data.client_pid);
```

/* server.c */ (cont.)

```
/* Poi restituiamo i dati elaborati aprendo la pipe del client in sola
  scrittura e modalità bloccante. Infine arrestiamo la FIFO del server
  chiudendo il file e facendo l'unlink della FIFO. */
   client_fifo_fd = open(client_fifo,O_WRONLY);
   if (client_fifo_fd != -1) {
        write(client_fifo_fd, &my_data, sizeof(my_data));
        close(client_fifo_fd);
   }
  }
} while (read_res > 0);   /* chiude il do */
   close(server_fifo_fd);
   unlink(SERVER_FIFO_NAME);
   exit(0);
}
```

/* client.c */

```
#include "client.h"
#include <ctype.h>
int main(){
int server_fifo_fd, client_fifo_fd;
struct data_to_pass_st my_data;
char client_fifo[256];
server_fifo_fd = open(SERVER_FIFO_NAME, O_WRONLY);
if (server_fifo_fd == -1)
{  fprintf(stderr, "Sorry, no server\n");
   exit(1);
}
my_data.client_pid = getpid();
sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
if (mkfifo(client_fifo, 0777) == -1)
{
fprintf(stderr, "Sorry, can't make %s\n", client_fifo);
exit(1);
}
/* I dati dei client sono inviati al server. Successivamente è aperta la FIFO client (sola lettura,
  modalità bloccante) ed i dati sono letti dal client. Infine, la FIFO del server è chiusa e la FIFO
  client rimossa dalla memoria. */
```

/* client.c */ (cont.)

```
sprintf(my_data.some_data,"Hello from %d",my_data.client_pid);
printf("%d sent %s, ", my_data.client_pid, my_data.some_data);
  write(server_fifo_fd, &my_data, sizeof(my_data));
  client_fifo_fd = open(client_fifo, O_RDONLY);
   if (client_fifo_fd != -1) {
    if(read(client_fifo_fd, &my_data, sizeof(my_data)) > 0)
       printf("received: %s\n", my_data.some_data);
    close(client_fifo_fd);
   }
close(server_fifo_fd);
unlink(client_fifo);
exit(0);
}
/* sprintf identica a printf tranne per il fatto che l'output viene inserito nell'array
  puntato da my_data.some_data invece che su stdout */
```

Client-Server in esecuzione

- Per testare l'applicazione, eseguiamo una copia singola del server e più copie dei client
- Per avviarli e chiuderli tutti nello stesso tempo usiamo i seguenti comandi di shell

```
$ server &
$ for i in 1 2 3 4 5
do
client &
done
$
```

Client-server in esecuzione (2)

- Partono 1 processo server e 5 processi client

```
531 sent Hello from 531, received: HELLO FROM 531
532 sent Hello from 532, received: HELLO FROM 532
529 sent Hello from 529, received: HELLO FROM 529
530 sent Hello from 530, received: HELLO FROM 530
533 sent Hello from 533, received: HELLO FROM 533
```

- Le diverse richieste dei client sono intrecciate (interleaved) ed ogni client riceve gli opportuni dati ad esso restituiti

Client-server

- Il server crea la sua FIFO in modalità a sola lettura e si blocca
  - Ciò avviene fino a che il primo client si connette aprendo la stessa FIFO in scrittura
  - A quel punto, il processo del server si sblocca ed è eseguita una sleep in modo che le scritture dei client si accodano (la sleep non è usata nelle applicazioni reali)
  - Nel frattempo, dopo che il client ha aperto la FIFO del server, esso crea la propria FIFO identificata univocamente per leggere i dati provenienti dal server
    - Solo allora il client scrive i dati al server (bloccandosi se la pipe è piena) e dopo si blocca su una lettura della propria FIFO, in attesa della risposta

Client-server (cont.)

- Ricevendo i dati dal client, il server li elabora, apre la FIFO del client per la scrittura e scrive i dati
  - Ciò sblocca il client
  - Quando il client è sbloccato, esso può leggere dalla sua FIFO i dati scritti in essa dal server
- L'intero processo si ripete fino a che l'ultimo client chiude la pipe del server, provocando un fallimento della lettura del server (restituisce 0) poiché nessun processo ha la pipe del server aperta in scrittura
- Se questo fosse un processo server reale che necessita di aspettare altri client, dovremmo modificarlo per:
  - Aprire la propria FIFO in scrittura, in modo che la lettura si blocca sempre piuttosto che ritornare 0, oppure
  - Chiudere e riaprire la FIFO del server quando read() restituisce 0 byte, così il processo server si blocca con la open in attesa di un client

Esercizio 1

- Scrivere un programma che gestisce il comportamento di un padre e due figli (fpari e fdispari); il padre legge numeri positivi da tastiera fino a che non arriva un numero negativo
  - se il numero è pari lo manda al figlio pari
  - altrimenti lo manda al figlio dispari
- I figli effettuano le somme parziali, quindi rimandano le somme al padre che calcola la somma totale e la stampa a video

Esercizio 2

- Un processo padre crea N (N numero pari) processi figli. Ciascun processo figlio P_i e' identificato da una variabile intera i (i=0,1,2,3...,N-1).
- E' necessario gestire due casi, sulla base di un argomento da riga di comando:
  1. Se argv[1] è uguale ad 'a' ogni processo figlio P_i, con i pari, manda un segnale (SIGUSR1) al processo P_{i+1};
  2. Se argv[1] è uguale a 'b' ogni processo figlio P_i con $i < N/2$ manda un segnale (SIGUSR1) al processo $P_{i + N/2}$

Esercizio 1

Realizzare un programma C e Posix sotto Linux che con l'uso dei segnali sincronizzi un processo padre ed un processo figlio che scrivono e leggono, rispettivamente, un numero intero alla volta

(da 1 a n, dove n è passato da riga di comando) nella prima posizione di un file temporaneo opportunamente creato