Sistemi Operativi e Reti di Calcolatori (SOReCa)

Corso di Laurea in *Ingegneria Informatica e Automatica (BIAR)*

Terzo Anno | Primo Semestre

A.A. 2025/2026

Esercitazione [04] File Pipe FIFO

Riccardo Lazzeretti lazzeretti@diag.uniroma1.it
Paolo Ottolino paolo.ottolino@uniroma1.it
Matteo Cornacchia cornacchia@diag.uniroma1.it

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Sommario

- Soluzione esercizi precedenti (lab-02: Shared Memory)
- Descrittori in C
- Esercizio 1: Lettura/Scrittura su file
- Pipe
- Esercizio 2: IPC via pipe
- Named pipe (FIFO)
- Esercizio 3: EchoProcess su FIFO

Obiettivi

- Imparare ad effettuare operazioni di input e output usando i descrittori (fd) in UNIX
- Lettura/scrittura su file
- Invio/ricezioni messaggi su socket

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Soluzione: Applicazione Modulare

- Lab03, Esercizio 1

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Shared Memory

Lab03-es1: Applicazione Modulare

- L'applicazione è sviluppata in due componenti.
  - Il primo (requester) carica dati nella memoria condivisa
  - Il secondo (worker) li elabora
  - Il primo li stampa
- L'applicazione è composta da due processi generati tramite fork
- Completare il codice dell'applicazione request/worker
- Sorgenti
  - `makefile`
  - `req_wrk.c`
- Suggerimento: seguire i blocchi di commenti inseriti nel codice
- Suggerimento: inserire elementi per la sincronizzazione
- Test:
  - Lanciate l'applicazione, deve stampare alla fine i valori elaborati (il quadrato dei numeri interi da `0` a `NUM-1`)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Shared Memory

Lab03-es1: Applicazione Modulare – uso dei semafori

```
/ data array
/* bla bla bla Add any needed resource **/
int fd;
sem_t *sem_worker, *sem_request;
```

```
int request() {
// map the shared memory in the data array
if ((data = (int *) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
0))== MAP_FAILED)
        handle_error ("main: mmap");
int i;
  for (i = 0; i < NUM; ++i) {
    data[i] = i;
  }
  printf("request: data generated\n");
// Signal the worker that it can start the elaboration and wait it has
terminated
  if (sem_post(sem_worker) != 0)
    handle_error("request: cannot unlock the worker semaphore");

  if (sem_wait(sem_request) != 0)
    handle_error("request: cannot unlock the request semaphore");

  printf("request: acquire updated data\n");

  printf("request: updated data:\n");
  for (i = 0; i < NUM; ++i) {
    printf("%d\n", data[i]);
  }

//release resources
 if (munmap(data, SIZE) == -1)
        handle_error("main: munmap");

  return EXIT_SUCCESS;
}
```

```
int work() {

// map the shared memory in the data array
  if ((data = (int *) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0))==
MAP_FAILED)
        handle_error ("main: mmap");
  printf("worker: mapped address: %p\n", data);

 // Wait that the request() process generated data
  printf("worker: waiting initial data\n");
  if (sem_wait(sem_worker) != 0)
    handle_error("worker: cannot lock the worker semaphore");

  printf("worker: acquire initial data\n");

  printf("worker: update data\n");
  int i;
  for (i = 0; i < NUM; ++i) {
    data[i] = data[i] * data[i];
  }

  printf("worker: release updated data\n");

   // Signal the requester that elaboration terminated

  if (sem_post(sem_request) != 0)
    handle_error("worker: cannot lock the request semaphore");

  // Release resources
  if (munmap(data, SIZE) == -1)
      handle_error("main: munmap");

  return EXIT_SUCCESS;
}
```

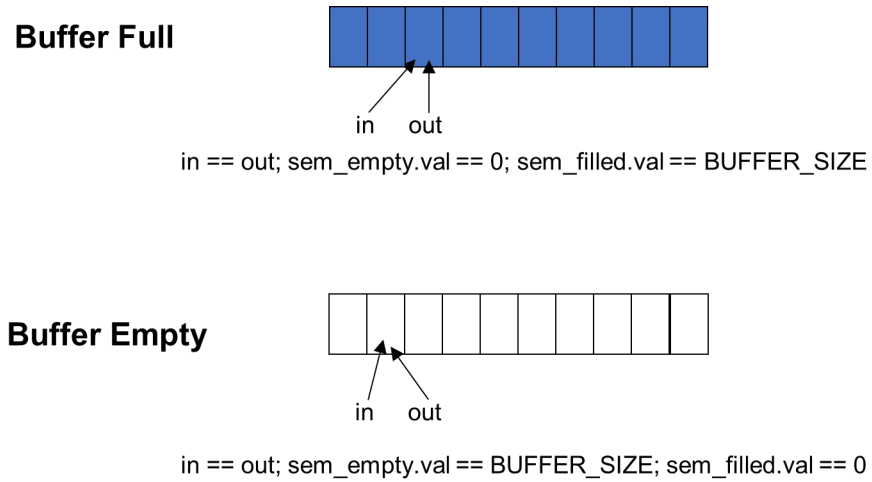
Shared Memory

Lab03-es1: Applicazione Modulare – creazione/distruzione semafori

```
int fd;
sem_t *sem_worker, *sem_request;
int main(int argc, char **argv);
```

```
// Create and open the needed resources
    sem_unlink(SEM_NAME_REQ);
    sem_unlink(SEM_NAME_WRK);

    sem_request = sem_open(SEM_NAME_REQ, O_CREAT | O_EXCL, 0600,
0);
    if (sem_request == SEM_FAILED) handle_error("sem_open
filled");

    sem_worker = sem_open(SEM_NAME_WRK, O_CREAT | O_EXCL, 0600,
0);
    if (sem_worker == SEM_FAILED) handle_error("sem_open filled");

    shm_unlink(SHM_NAME);

      fd = shm_open(SHM_NAME, O_CREAT | O_EXCL | O_RDWR, 0600);
      if (fd < 0)
        handle_error("main: error in shm_open");

      if(ftruncate(fd, SIZE) == -1)
           handle_error ("main: ftruncate");
```

```
// Close and release resources
    ret = sem_close(sem_worker);
    if (ret) handle_error("main: sem_close worker");
    ret = sem_close(sem_request);
    if (ret) handle_error("main: sem_close request");

    // then unlink
    ret = sem_unlink(SEM_NAME_REQ);
    if (ret) handle_error("main: sem_unlink request");
    ret = sem_unlink(SEM_NAME_WRK);
    if (ret) handle_error("main: sem_unlink worker");

    ret = close(fd);
         if (ret == -1)
        handle_error("main: cannot close the shared memory");

    ret = shm_unlink(SHM_NAME);
    if (ret) handle_error("main: shm_unlink");
```



Soluzione: Produttore/Consumatore

- Lab03, Esercizio 2

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Shared Memory

Lab03-es2: Produttore/Consumatore

- L'applicazione è sviluppata in due moduli separati.
- Si tiene conto della configurazione con `NUM_CONSUMERS` consumatori e `NUM_PRODUCERS` produttori
- Il buffer e le posizioni di in e out sono posizionati in memoria condivisa

- Completare il codice dell'applicazione produttore/consumatore
- Sorgenti
  - `makefile`
  - `producer.c`
  - `consumer.c`
- Suggerimento: seguire i blocchi di commenti inseriti nel codice
- Informazione: gli elementi per la sincronizzazione (vedi esercitazione 3 in lab) sono già inseriti
- Test:
  - Lanciate prima `producer` (crea semafori e memoria condivisa) e poi `consumer`

Shared Memory

Lab03-es2: Produttore/Consumatore – uso della shm (semafori già definiti)

```
struct shared_memory *myshm_ptr; //definizione shared memory
int fd_shm;
sem_t *sem_empty, *sem_filled, *sem_cs; //definizione semafori named
```

```
void produce(int id, int numOps) {
    int localSum = 0;
    while (numOps > 0) {
        // producer, just do your thing!
        int value = performRandomTransaction();

        int ret = sem_wait(sem_empty);
        if (ret) handle_error("sem_wait empty\n");

        ret = sem_wait(sem_cs);
        if (ret) handle_error("sem_wait cs");

        /** write value in the buffer inside the shared memory and update
the producer position */
        myshm_ptr->buf[myshm_ptr->write_index] = value;
        myshm_ptr->write_index++;
        if (myshm_ptr->write_index == BUFFER_SIZE)
            myshm_ptr->write_index = 0;
        /**/
ret = sem_post(sem_cs);
        if (ret) handle_error("sem_post cs");

        ret = sem_post(sem_filled);
        if (ret) handle_error("sem_post filled");

        localSum += value;
        numOps--;
    }
    printf("Producer %d ended. Local sum is %d\n", id, localSum);
}
}
```

```
void consume(int id, int numOps) {
    int localSum = 0;
    while (numOps > 0) {
        int ret = sem_wait(sem_filled);
        if (ret) handle_error("sem_wait filled");

        ret = sem_wait(sem_cs);
        if (ret) handle_error("sem_wait cs");

        /** write value in the buffer inside the shared memory and update
the producer position */

        int value = myshm_ptr->buf[myshm_ptr->read_index];
        myshm_ptr->read_index++;
        if (myshm_ptr->read_index == BUFFER_SIZE)
            myshm_ptr->read_index = 0;

        ret = sem_post(sem_cs);
        if (ret) handle_error("sem_post cs");

        ret = sem_post(sem_empty);
        if (ret) handle_error("sem_post empty");

        localSum += value;
        numOps--;
    }
    printf("Consumer %d ended. Local sum is %d\n", id, localSum);
}
```

Shared Memory

Lab03-es2: Produttore/Consumatore – apertura/chiusura memoria

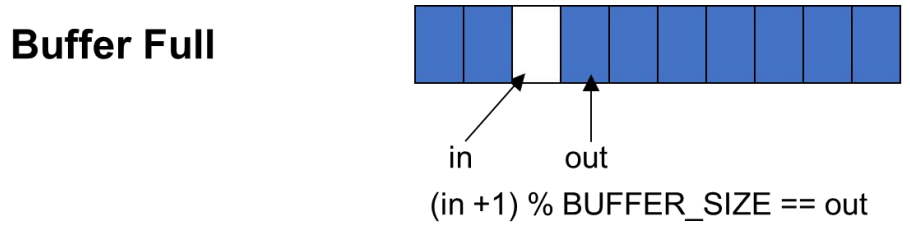
```
struct shared_memory *myshm_ptr; //definizione shared memory
int fd_shm;
sem_t *sem_empty, *sem_filled, *sem_cs; //definizione semafori named
```

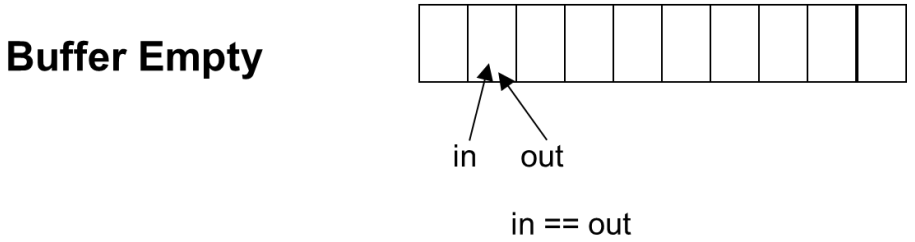
```
// Producer
void initMemory() {
    /** Request the kernel to creare a shared memory, set its size to the size of      *
struct shared_memory, and map the shared memory in the shared_mem_ptr variable. Initialize
the shared memory to 0. **/
    if ((fd_shm = shm_open (SH_MEM_NAME, O_RDWR | O_CREAT | O_EXCL, 0660)) == -1)
        handle_error("shm_open");

    if (ftruncate (fd_shm, sizeof (struct shared_memory)) == -1)
       handle_error ("ftruncate");

    if ((myshm_ptr = mmap (NULL, sizeof(struct shared_memory), PROT_READ | PROT_WRITE,
MAP_SHARED,
            fd_shm, 0)) == MAP_FAILED)
       handle_error ("mmap");

    // Initialize the shared memory
    // myshm_ptr -> read_index = myshm_ptr -> write_index = 0;
    memset(myshm_ptr, 0, sizeof(struct shared_memory));
}
void closeMemory() {
    /** unmap the shared memory, unlink the shared memory and close its descriptor **/
       // mmap cleanup
    int ret;
       ret = munmap(myshm_ptr, sizeof(struct shared_memory));
       if (ret == -1)
        handle_error("munmap");

    //close descriptor
    close(fd_shm);

       // shm_open cleanup
       ret = shm_unlink(SH_MEM_NAME);
       if (ret == -1)
        handle_error("unlink");
}
```

```
// Consumer
void openMemory() {
    /** Request shared memory to the kernel and map the shared
memory in the shared_mem_ptr variable.  **/

    if ((fd_shm = shm_open (SH_MEM_NAME, O_RDWR, 0660)) == -1)
        handle_error("shm_open");

    if ((myshm_ptr = mmap (NULL, sizeof(struct shared_memory),
PROT_READ | PROT_WRITE, MAP_SHARED,
            fd_shm, 0)) == MAP_FAILED)
       handle_error ("mmap");
}

void closeMemory() {
    /** unmap the shared memory and close its descriptor **/
     int ret;
      // mmap cleanup
      ret = munmap(myshm_ptr, sizeof(struct shared_memory));
      if (ret == -1)
        handle_error("munmap");

    //close descriptor
    close(fd_shm);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA





Soluzione: Producer/Consumer senza semafori

- Lab03, Esercizio 3

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Shared Memory

Lab03-es3: Producer/Consumer senza semafori

- L'applicazione è sviluppata in due moduli separati.
- Si tiene conto della configurazione con `1` consumatore e `1` produttore
- Il buffer e le posizioni di in e out sono posizionati in memoria condivisa

- Completare il codice dell'applicazione produttore/consumatore
- Sorgenti
  - `makefile`
  - `producer.c`
  - `consumer.c`
- Suggerimento: seguire i blocchi di commenti inseriti nel codice
- Informazione: presenta il vantaggio di non ricorrere a chiamate al kernel per la sincronizzazione, ma sacrifica una posizione del buffer e introduce busy waiting
- Test:
  - Lanciate prima `producer` (memoria condivisa) e poi `consumer`

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Shared Memory

Lab03-es3: Producer/Consumer senza semafori – read/write shm

```
struct shared_memory *myshm_ptr; //definizione shared memory
int fd_shm;
BUFFER_SIZE
```

```
void produce(int id, int numOps) {
    int localSum, next_pos = 0;
    while (numOps > 0) {
        // producer, just do your thing!
        int value = performRandomTransaction();

        // int ret = sem_wait(sem_empty);
        // if (ret) handle_error("sem_wait empty\n");
        // ret = sem_wait(sem_cs);
        // if (ret) handle_error("sem_wait cs");

        /** check that we can write
            write value in the buffer inside the shared memory and update the
producer position */
        while ((myshm_ptr->write_index +1) % BUFFER_SIZE ==
myshm_ptr->read_index); //busy waiting
        myshm_ptr->buf[myshm_ptr->write_index] = value;
        next_pos = (myshm_ptr->write_index +1) % BUFFER_SIZE;
        myshm_ptr->write_index = next_pos;
        /**/
        // ret = sem_post(sem_cs);
        // if (ret) handle_error("sem_post cs");
        // ret = sem_post(sem_filled);
        // if (ret) handle_error("sem_post filled");

        localSum += value;
        numOps--;
    }
    printf("Producer %d ended. Local sum is %d\n", id, localSum);
}
}
```

```
void consume(int id, int numOps) {
    int localSum = 0;
    int next_pos;
    while (numOps > 0) {
        // int ret = sem_wait(sem_filled);
        // if (ret) handle_error("sem_wait filled");
        // ret = sem_wait(sem_cs);
        // if (ret) handle_error("sem_wait cs");

        /** write value in the buffer inside the shared memory and update the
producer position */
        while (myshm_ptr->read_index == myshm_ptr->write_index); //busy waiting
        int value = myshm_ptr->buf[myshm_ptr->read_index];
        next_pos = (myshm_ptr->read_index +1)% BUFFER_SIZE;
        (myshm_ptr->read_index = next_pos;
        // ret = sem_post(sem_cs);
        // if (ret) handle_error("sem_post cs");
        // ret = sem_post(sem_empty);
        // if (ret) handle_error("sem_post empty");

        localSum += value;
        numOps--;
    }
    printf("Consumer %d ended. Local sum is %d\n", id, localSum);
}
```

Shared Memory

Lab03-es3: Producer/Consumer senza semafori – open/close shm == lab03-es02

```
struct shared_memory *myshm_ptr; //definizione shared memory
int fd_shm;
sem_t *sem_empty, *sem_filled, *sem_cs; //definizione semafori named
```

```
// Producer
void initMemory() {
    /** Request the kernel to creare a shared memory, set its size to the size of      *
struct shared_memory, and map the shared memory in the shared_mem_ptr variable. Initialize
the shared memory to 0. **/
    shm_unlink(SH_MEM_NAME);
    if ((fd_shm = shm_open (SH_MEM_NAME, O_RDWR | O_CREAT | O_EXCL, 0660)) == -1)
        handle_error("shm_open");

    if (ftruncate (fd_shm, sizeof (struct shared_memory)) == -1)
       handle_error ("ftruncate");

    if ((myshm_ptr = mmap (NULL, sizeof(struct shared_memory), PROT_READ | PROT_WRITE,
MAP_SHARED,
            fd_shm, 0)) == MAP_FAILED)
       handle_error ("mmap");

    // Initialize the shared memory
    // myshm_ptr -> read_index = myshm_ptr -> write_index = 0;
    memset(myshm_ptr, 0, sizeof(struct shared_memory));
}
void closeMemory() {
    /** unmap the shared memory, unlink the shared memory and close its descriptor **/
       // mmap cleanup
    int ret;
       ret = munmap(myshm_ptr, sizeof(struct shared_memory));
       if (ret == -1)
        handle_error("munmap");

    //close descriptor
    close(fd_shm);

       // shm_open cleanup
       ret = shm_unlink(SH_MEM_NAME);
       if (ret == -1)
        handle_error("unlink");
}
```

```
// Consumer
void openMemory() {
    /** Request shared memory to the kernel and map the shared
memory in the shared_mem_ptr variable.  **/

    if ((fd_shm = shm_open (SH_MEM_NAME, O_RDWR, 0660)) == -1)
        handle_error("shm_open");

    if ((myshm_ptr = mmap (NULL, sizeof(struct shared_memory),
PROT_READ | PROT_WRITE, MAP_SHARED,
            fd_shm, 0)) == MAP_FAILED)
       handle_error ("mmap");
}

void closeMemory() {
    /** unmap the shared memory and close its descriptor **/
     int ret;
      // mmap cleanup
      ret = munmap(myshm_ptr, sizeof(struct shared_memory));
      if (ret == -1)
        handle_error("munmap");

    //close descriptor
    close(fd_shm);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Descrittori in Unix

- File Descriptor (fd)

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Descrittori in Unix

File Descriptor - presentazione

- I *file descriptor* (FD) sono un'astrazione per accedere a file o altre risorse di input/output come pipe e socket
- Ogni processo ha una *tabella dei descrittori* associata
  - Standard input 0
  - Standard output 1
  - Standard error 2
- Le operazioni di apertura (o creazione) di una risorsa di input/output sono legate al tipo della risorsa stessa
  - `open()` per i file
  - `pipe()` per le pipe
  - `socket()` o `accept()` per le socket (prossimamente…)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Descrittori in Unix

File Descriptor - lettura

- La funzione `read()` è definita in `unistd.h`

```
ssize_t read(int fd, void *buf, size_t nbyte);
```

  - `fd`: descrittore della risorsa
  - `buf`: puntatore al buffer dove scrivere il messaggio letto
  - `nbyte`: numero massimo di byte da leggere

  Ritorna il numero di byte realmente letti, o `-1` in caso di errore

  - Per i file, ritorna `0` in caso di end-of-file
  - Per le socket, ritorna `0` in caso di connessione chiusa

- Il buffer deve essere dimensionato per contenere `nbyte` byte
  - Tale dimensione dipende dall'applicazione
    - Formato del file da leggere
    - Messaggi da scambiare in un protocollo di comunicazione

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Descrittori in Unix

File Descriptor – lettura: interrupt

- L'esecuzione della funzione `read()` ha una certa durata
  - Per i file, richiede il tempo necessario per leggere fino a `nbytes` byte
  - Per le pipe è piuttosto breve perché legge in memoria
- Nel tempo tra l'invocazione ed il termine della `read()`, la chiamata può essere interrotta da un segnale
  - Se l'interruzione avviene prima di riuscire a leggere qualsiasi dato (zero byte letti), la `read()` ritorna `-1` ed `errno` viene settato a `EINTR`
- può capitare che la read sia interrotta prima che siano ricevuti `nbytes` bytes
  - la `read()` ritorna il numero di byte letti fino a quel momento (byte letti > 0)
- Una corretta implementazione deve riconoscere queste situazioni ed invocare di nuovo la `read()` per ritentare/completare la lettura

Descrittori in Unix

File Descriptor – lettura: esempio (pseudo-C)

```
while(<not all bytes have been read>) {
    // read from fd up to n bytes and store into buf
    int ret=read(fd, buf, n);

    // no more bytes to read, quit
    if (ret==0) break;

    if (ret==-1) {
        if (errno==EINTR) continue; /* interrupted before reading
any byte, retry */
        /*error_handler */ // an error occurred...
    }

    /* if interrupted when less than n bytes were read, pay attention to where to
write on buf on resume! */
    <do something with read bytes>
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Descrittori in Unix

File Descriptor – scrittura

- La funzione `write()` è definita in `unistd.h`

```
ssize_t write(int fd,const void *buf,size_t nbyte);
```

  - `fd`: descrittore della risorsa
  - `buf`: puntatore al buffer contenente il messaggio da scrivere
  - `nbyte`: numero massimo di byte da scrivere

  Ritorna il numero di byte realmente scritti, o `-1` in caso di errore

- Gestione degli interrupt analoga alla `read()`
  - In caso di interrupt prima di aver scritto il primo byte, viene ritornato `-1` e settato `errno` a `EINTR`
  - In caso di interrupt dopo aver scritto almeno un byte, viene ritornato il numero di byte realmente scritti

Descrittori in Unix

File Descriptor – scrittura: esempio (pseudo-C)

```
while(<not all bytes have been written>) {
   // write to fd up to n bytes from buf
   int ret=write(fd, buf, n);

   if (ret==-1) {
       // interrupted before writing any byte, retry
       if (errno==EINTR) continue;

       // an error occurred...
       exit(EXIT_FAILURE);
   }

   /* if interrupted when less than n bytes were written, pay attention to
where you start      reading from in the buffer on resume */
   <do something>
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Descrittori in Unix

Lab04 es1: copiare un file in C

- Sorgente da completare: `copy.c`
- Argomenti
  - File sorgente S
  - File destinazione D
  - Dimensione B del batch di lettura/scrittura (opzionale, default 128 byte)
- Semantica

  Effettuare una copia di S in D tramite una sequenza di letture da S e scritture in D a blocchi di B byte per volta
- Esercizio: completare il codice dove indicato
  - Per testare la propria soluzione è disponibile lo script `test.sh`

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

IPC tramite pipe

- pipe semplici tra processi «relazionati»
- FIFO tra processi non «relazionati»

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

IPC tramite pipe

Overview

- Meccanismo di comunicazione inter-processo
- Canale di comunicazione unidirezionale
- `int pipe(int fd[2])`
  - `fd[0]` descrittore di lettura
  - `fd[1]` descrittore di scrittura
  - ritorna `0` in caso di successo, `-1` altrimenti
- Chiamate a `read()` su pipe ritornano `0` quando tutti i descrittori di scrittura sono stati chiusi
- Chiamate a `write()` su pipe causano `SIGPIPE` («broken pipe») quando tutti i descrittori di lettura sono stati chiusi
  - Nota: vale anche per scritture su socket ormai chiuse!

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

IPC tramite pipe

Lab04 es. 2: Comunicazione unidirezionale di processi via pipe con sincronizzazione

- Il processo padre crea CHILDREN_COUNT processi figlio che condividono una pipe unica:
  - nella quale WRITERS_COUNT figli scrivono (*writers*)
  - e dalla quale READERS_COUNT figli leggono (*readers*)
- I writers scrivono nella pipe in mutua esclusione tramite un semaforo il cui nome è definito nella macro WRITE_MUTEX
- I readers leggono dalla pipe in mutua esclusione grazie ad un altro semaforo, il cui nome è specificato nella macro READ_MUTEX
- All'avvio, il padre crea i semafori named assicurandosi che non esistano già, e passa come argomento ai processi figlio il puntatore all'oggetto sem_t su cui ciascun reader o writer dovrà operare
- Una volta avviati:
  - i writers devono scrivere nella pipe MSG_COUNT messaggi in totale (ognuno dovrà quindi scriverne MSG_COUNT/WRITERS_COUNT).
  - ogni reader deve leggere dalla pipe MSG_COUNT/READERS_COUNT messaggi e verificarne l'integrità
  - ogni messaggio è un array di MSG_ELEMS interi, che viene considerato integro se tutti i suoi elementi hanno lo stesso valore.
- Infine, il padre deve attendere esplicitamente la terminazione dei figli e liberare le risorse.

Named Pipe (FIFO)

- Processi «non relazionati» (no fork())

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Named Pipe (FIFO)

Overview

- Simili alle pipe, consentono comunicazione tra processi non «relazionati» (nessun legame padre-figlio via fork)
- Una FIFO è un **file speciale** per comunicazione unidirezionale
- Creazione: `int mkfifo(const char *path, mode_t mode)`
  - `path`: nome della FIFO
  - `mode`: permessi da associare alla FIFO (es. `0666`)
  - Ritorna `0` in caso di successo, `-1` altrimenti
- Apertura: `int open(const char *path, int oflag)`
  - Nome FIFO e modalità di apertura (`O_RDONLY`, `O_WRONLY`, etc)
  - Ritorna il descrittore della FIFO, `-1` altrimenti
- Chiusura: `int close(int fd)`
- Rimozione: `int unlink(const char *path)`

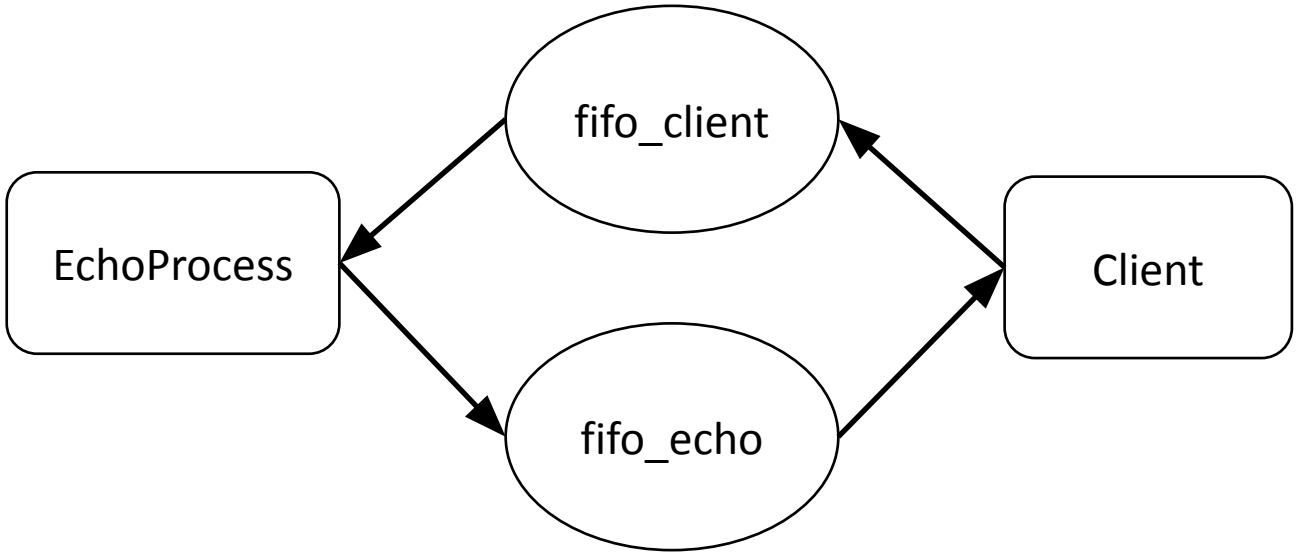
Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Named Pipe (FIFO)

Lab04 es3: EchoProcess su FIFO

- Il server prepara (crea) due FIFO
  - `echo_fifo` per inviare messaggi al client
  - `client_fifo` per ricevere messaggi dal client
- La comunicazione client-server avviene tramite queste due FIFO
- Esercizio: completare codici di client (`client.c`) e server (`echo.c`) e lettura/scrittura (rw.c)



Named Pipe (FIFO)

Lab04 es3: write to FIFO

- Implementare la funzione

```
void writeMsg(int fd, char* buf, int size)
```

- `fd` è il descrittore della FIFO
- `buf` contiene il messaggio da scrivere
- `size` specifica quanti byte deve scrivere
- Suggerimenti:
  - Si scrive nella FIFO come in un File
  - Controllare che tutti i byte siano stati scritti (vedi esercitazione lettura/scrittura su file)
    - Gestione errori dovuti a interruzioni (non è stato scritto nella FIFO)
    - Scrittura parziale
    - Altri errori

Named Pipe (FIFO)

Lab04 es3: read from FIFO

- Implementare la funzione

```
int readOneByOne(int fd, char* buf, char separator)
```

- `fd` è il descrittore della FIFO
- `buf` contiene il messaggio da scrivere
- `separator` è il carattere utilizzato per terminare il messaggio ('\n')
- Suggerimenti:
  - Puoi leggere dalla FIFO come da un normale FILE
  - Non puoi conscere la dimensione del messaggio!!!
    - Leggi un byte per volta
    - Esci dal ciclo quando trovi il carattere `separator` ('\n')
  - Ripeti la read quando interrotta prima della lettura del dato
  - Restituisci il numero totale di byte letti
  - Se sono stati letti 0 bytes, allora l'altro processo ha chiuso la FIFO senza preavviso (errore da gestire)

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Autovalutazione

https://forms.gle/E94c87VU1WUwebWK8

Named Pipe (FIFO)

Lab04 es4, riepilogo

- Modificare la soluzione dell'esercizio di riepilogo come si ritiene più opportuno introducendo pipe o fifo
  - Le scelte dovrebbero introdurre un qualche vantaggio

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma