

# Esercitazione [08]

## Pipe & FIFO

Daniele Cono D'Elia – [delia@diag.uniroma1.it](mailto:delia@diag.uniroma1.it)

Riccardo Lazzeretti – [lazzeretti@diag.uniroma1.it](mailto:lazzeretti@diag.uniroma1.it)

Luca Massarelli – [massarelli@diag.uniroma1.it](mailto:massarelli@diag.uniroma1.it)

Sistemi di Calcolo - Secondo modulo (SC2)

Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2017-2018

# Sommario

- Soluzione esercizio su EchoServer multi-thread
- Obiettivi dell'esercitazione
- Pipe
- Esercizio: IPC via pipe
- Named pipe (FIFO)
- Esercizio: EchoProcess su FIFO

# Soluzione EchoServer multi-thread

- Esercizio: completare l'EchoServer in modalità multi-thread
- Soluzione
  - La struct `handler_args_t` richiede:
    - Un campo `int` per il descrittore della socket
    - Un campo `struct sockaddr_in*` per i dati di rete del client
  - In `thread_connection_handler()`, va fatta la `free` della struct `handler_args_t`
    - Era stata allocata nel main thread tramite `malloc()`
    - Richiede prima la `free` della struttura dati con le info sul client, allocata anch'essa nel main thread tramite `calloc()`
  - Creazione thread per gestione connessione
    - Assegnare valori all'oggetto di tipo struct `handler_args_t`
    - `pthread_create() + pthread_detach()` con gestione errori
    - "Preparare" la memoria per le info sul client per la connessione da accettare all'iterazione successiva (vedere «posizione» `calloc`)

# Obiettivi Esercitazione

- Implementare comunicazione inter-processo tramite pipe
  - Usando pipe semplici tra processi «relazionati»
  - Usando FIFO tra processi non «relazionati»

# Overview sulle pipe

- 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!

# Esercizio: Comunicazione unidirezionale di processi via pipe con sincronizzazione (1)

- 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

# Esercizio: Comunicazione unidirezionale di processi via pipe con sincronizzazione (2)

- Una volta avviati:
  - i writers devono scrivere nella pipe MSG\_COUNT messaggi in totale (ognuno dovrà quindi scriverne  $\text{MSG\_COUNT} / \text{WRITERS\_COUNT}$ ).
  - ogni reader deve leggere dalla pipe  $\text{MSG\_COUNT} / \text{READERS\_COUNT}$  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.

# Obiettivi principali

- Gestione processi figlio: creazione/attesa terminazione processi figlio (implementata)
- Mutua esclusione inter-processo: creazione/utilizzo/rimozione di semafori (implementata)
- Comunicazione su pipe: invio e ricezione dati di lunghezza fissa (da implementare)



# Write to PIPE

- Implementare la funzione

```
int write_to_pipe(    int fd,  
                     const void *data,  
                     size_t data_len)
```

- `fd` è il descrittore della pipe
- `data` contiene il messaggio da scrivere
- `data_len` specifica quanti byte deve scrivere
- La funzione restituisce il numero dei byte scritti
- Suggerimenti:
  - Si scrive nella pipe 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 pipe)
  - Scrittura parziale
  - Altri errori

# Read from PIPE

- Implementare la funzione

```
int read_from_pipe(int fd,  
                  void *data,  
                  size_t data_len)
```

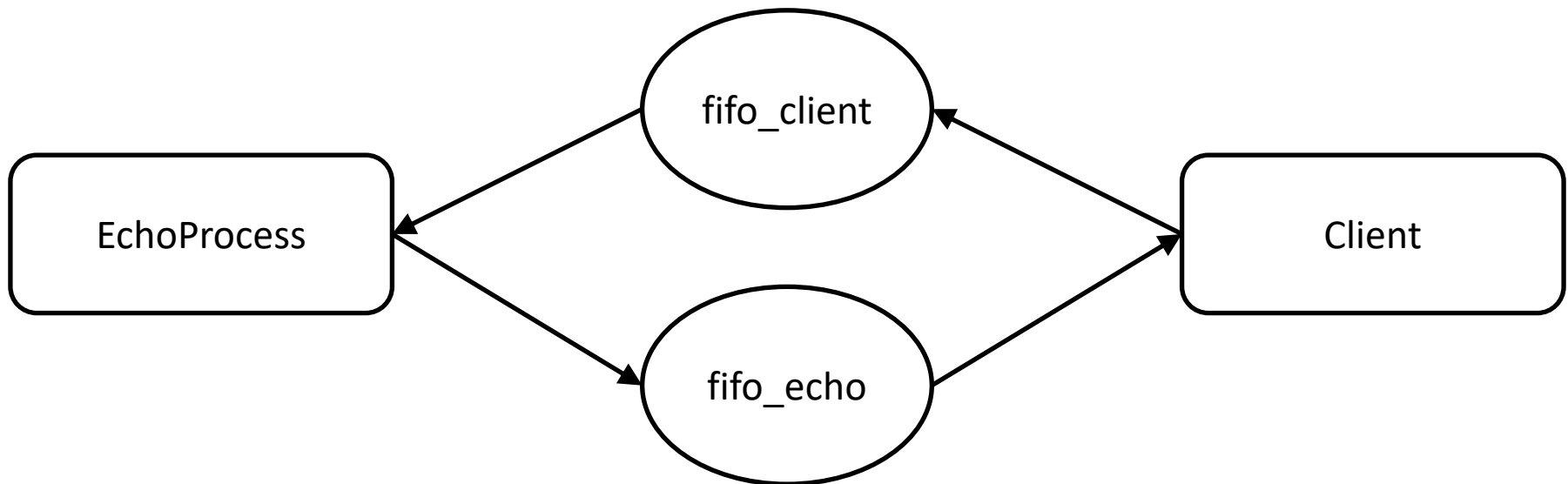
- `fd` è il descrittore della pipe
- `data` conterrà il messaggio letto
- `data_len` specifica quanti byte deve leggere
- La funzione restituisce il numero dei byte letti
- Suggerimenti:
  - Si legge dalla pipe come da un File
  - Controllare che tutti i byte siano stati letti (vedi esercitazione lettura/scrittura su file)
  - Gestione errori dovuti a interruzioni (non è stato letto dalla pipe)
  - Lettura parziale
  - Altri errori (gestire esplicitamente chiusura inaspettata in endpoint)

# Overview sulle named pipe (FIFO)

- 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)`

# Esercizio: 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`)



# 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
- Suggestimenti:
  - 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

# 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 conoscere 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)