Esercitazione [06]

Pipe & FIFO

Riccardo Lazzeretti – lazzeretti@diag.uniroma1.it

Sistemi di Calcolo 2
Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2020-2021

Sommario

- Soluzione esercizi su shared memory
- Descrittori in C
- Esercizio: Lettura/Scrittura su file
- Pipe
- Esercizio: IPC via pipe
- Named pipe (FIFO)
- Esercizio: EchoProcess su FIFO

Obiettivi Esercitazione

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

Descrittori in UNIX

- 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…)

Letture con i descrittori

- 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

Letture con i descrittori
Gestione degli 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

Letture con i descrittori
Esempio in 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>
}
```

Scritture con i descrittori

- 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

Scritture con i descrittori
Esempio in 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>
}
```

Esercizio: 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`

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 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.

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)
```

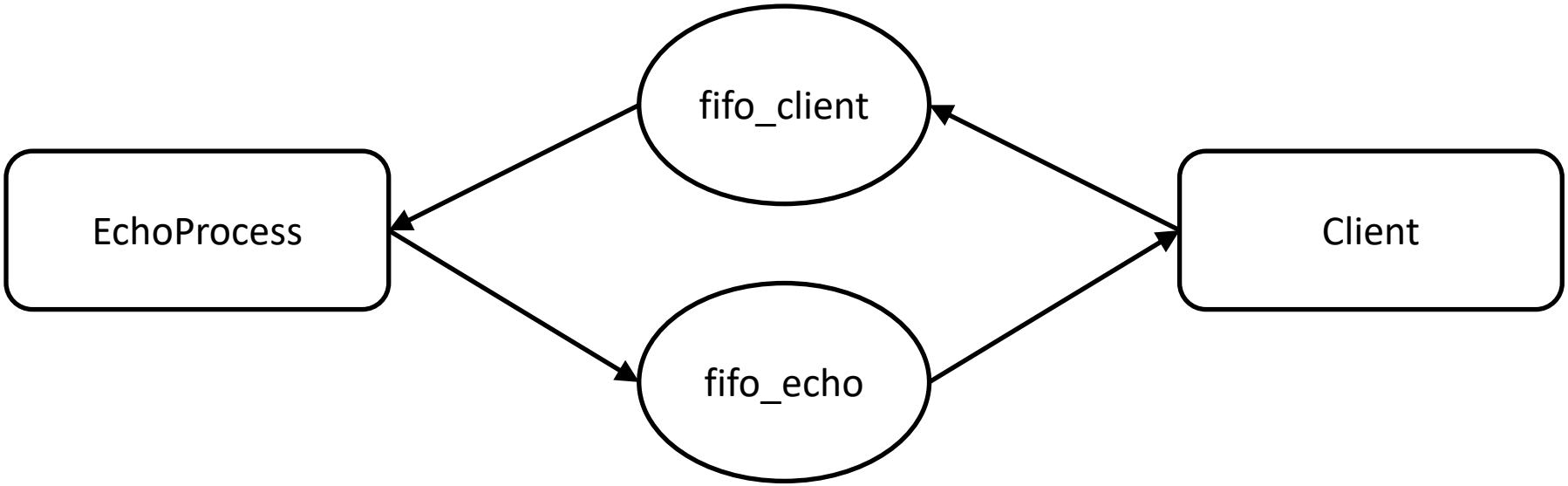
- `fd` è il descrittore della pipe
- `data` conterrà il messaggio letto
- `data_len` specifica quanti byte deve leggere
- La funzione restituisce il numero dei byte letti
- Suggerimenti:
  - Si legge dalla pipe come da un File
  - Controllare che tutti i byte siano stati letti (vedi esercitazione lettura/scrittura 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
- 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

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)