Esercitazione [10]

Approfondimenti su Socket e Pipe

Leonardo Aniello - aniello@dis.uniroma1.it
Daniele Cono D'Elia - delia@dis.uniroma1.it
Federico Lombardi - lombardi@dis.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. 2015-2016

Sommario

- Soluzione esercizio Logger
- Soluzione esercizio EchoProcess su FIFO
- Approfondimenti
  - Scritture su descrittore
  - Letture da descrittore
- Esercizio: implementazione di una chat end-to-end
- Slides di riepilogo
  - Socket
  - Pipe
  - Gestione errori

Soluzione Esercizio Logger

- `main()`
  - Dopo le chiamate `pipe()` e `fork()` bisogna chiudere:
    - Il descrittore della socket nel Logger (processo figlio)
    - I descrittori del file di log e di lettura da pipe nel padre (Server)
  - Dopo il `while(1)`, va chiuso il descrittore di scrittura su pipe
    - Per «pulizia», anche se il codice non verrà raggiunto
- `startLogger()` - *processo Logger*
  - Chiudere il descrittore di scrittura su pipe
  - Lettura da pipe con gestione casi particolari
    - La pipe è stata chiusa → `break`
    - La lettura è stata interrotta da un segnale → `continue`
    - Si è verificato un altro errore → `exit`
  - All'uscita, chiudere il descrittore di lettura da pipe

Soluzione Esercizio EchoProcess su FIFO

- Creazione FIFO tramite `mkfifo()`
- Apertura FIFO tramite `open()`
  - Il server apre `fifo_client` in lettura ed `fifo_echo` in scrittura
  - Il client apre `fifo_client` in scrittura ed `fifo_echo` in lettura
  - Entrambi seguono lo stesso ordine di apertura, pena un deadlock!
- Scrittura su FIFO
  - La dimensione del messaggio è nota, quindi la `write()` va gestita finché non sono stati scritti tutti i byte
  - Gestione EINTR ed errori
- Lettura da FIFO
  - La dimensione del messaggio non è nota, quindi ha senso gestire letture parziali (cioè dei soli dati disponibili in quel momento)
  - Gestione EINTR ed errori

Scritture su descrittore

- Quando ci apprestiamo a scrivere su un descrittore file/socket/pipe, la dimensione dei dati n è nota
- Aspetti da tenere a mente
  - se `ret==-1 && errno==EINTR` l'operazione è stata interrotta senza che alcun byte sia stato scritto
  - se `ret>0 && ret<N` la scrittura è stata interrotta dopo soli ret bytes: ne restano ancora N-ret da scrivere!
    - usare un accumulatore `bytes_read` per i byte letti dall'inizio
  - [socket, pipe] il processo può terminare con un `SIGPIPE`
    - socket: l'endpoint ha chiuso la connessione
    - pipe: l'endpoint ha chiuso il descrittore usato per la lettura

Letture da descrittore

- Quando ci apprestiamo a leggere da un descrittore, due sono gli scenari ricorrenti :
  1. lunghezza dati nota
  2. lunghezza dati variabile + condizione di terminazione
- Quando la dimensione N è nota, usiamo un ciclo che "assicuri" che esattamente N bytes siano stati letti
  - ipotesi: assumiamo che i dati siano effettivamente disponibili (non gestiamo errori di alto livello!)
  - pertanto, quanto visto per letture da file si applica a socket e pipe analogamente in questo scenario

Letture a lunghezza variabile

- Quando la lunghezza dei dati da leggere è variabile, va verificata una condizione di terminazione
  1. lunghezza dati nota
  2. lunghezza dati variabile + condizione di terminazione
- Alcuni scenari ricorrenti
  - un simbolo (e.g., ritorno a capo `'\n'`) contrassegna la fine
    - leggere un byte per volta e controllare il valore scritto nel buffer
  - un header di lunghezza fissa H specifica la grandezza del payload N (il messaggio contiene quindi H+N bytes)
    - si può implementare come due letture di lunghezza nota: dopo aver letto l'header, si estrae N ed implementa un secondo ciclo per leggere N bytes (tenendo sempre conto di eventuali interruzioni)

Prova al calcolatore - Indicazioni

Per la prova al calcolatore ci si aspetta che lo studente nelle operazioni su descrittori gestisca:

- errori restituiti dalle system calls
- interruzioni nelle letture/scritture
  - sia senza che alcun byte sia stato scritto/letto nel buffer, sia in presenza di letture/scritture parziali (da gestire quindi sempre!)
- casi speciali in fase di lettura (`ret==0`)
  - [file] viene raggiunta la fine del file
  - [socket] l'endpoint ha chiuso la comunicazione
  - [pipe] tutti i descrittori in scrittura sono stati chiusi

Per `SIGPIPE`, potrebbero eventualmente esservi fornite istruzioni per completare un signal handler già predisposto.

[Esercizio 1] Chat end-to-end via socket

- Una chat end-to-end permette a due utenti di scambiare messaggi tra loro senza passare per una terza entità
- Codice: due moduli di init + parte comune per chat session
  - modulo iniziale scelto a seconda degli argomenti da linea di comando
- Modulo `accept` (endpoint che configura il servizio)
  - Configura una listening socket su una porta data
  - Una volta accettata una connessione, inizia la chat session
- Modulo `connect` (endpoint che si connette al servizio)
  - Si connette all'indirizzo e alla porta dati in input
  - Una volta instaurata la connessione, inizia la chat session
- Durante una chat session, ciascun peer invia all'altro quanto letto da `stdin` e mostra invece su `stdout` quanto invece ricevuto

[Esercizio 1] Chat session

- All'inizio di una chat session il programma lancia due thread
  - *receiveMessage*: si mette in attesa di dati via `recv()`, e quando riceve un messaggio lo stampa a video
  - *sendMessage*: si mette in attesa di dati via `fgets()`, e quando l'utente inserisce un messaggio lo stampa a video e lo invia
  - Il descrittore di socket è in comune tra i due thread
- Quando un peer riceve il messaggio «BYE»
  - *receiveMessage* si sblocca e può terminare in maniera pulita
  - L'utente viene avvisato del termine della sessione ed invitato a premere «INVIO» per uscire
    - In questo modo anche *sendMessage* può sbloccarsi e terminare in maniera pulita

[Esercizio 1] Letture non bloccanti?

- Quando un peer invia il messaggio «BYE»
  - *sendMessage* può terminare in maniera pulita
  - *receiveMessage* si troverebbe però bloccato in `recv()`
    - PROBLEMA: non può terminare in maniera pulita!
      (l'altro endpoint deve chiudere la connessione esplicitamente... quindi per terminare dovrei attendere che l'altro utente prema «INVIO»!)

[Esercizio 1]: Funzione `select()`

- Consente di monitorare uno o più descrittori, in attesa che uno di essi diventi pronto per operazioni di I/O
- È possibile specificare un timeout
- `select()` si sblocca quando si verifica una di queste situazioni:
  - Uno dei descrittori monitorati diventa pronto
  - Il timeout impostato scade
  - Arriva un segnale

[Esercizio 1] Come è usata `select()`

- Prima di effettuare la `recv()`, abbiamo usato `select()` per monitorare il descrittore della socket con un timeout di 1.5 s
- In questo modo:
  - La `recv()` viene invocata solo quando c'è effettivamente un messaggio da leggere dalla socket
  - In caso di assenza di messaggi ricevuti, la `select()` si sblocca dopo 1.5 secondi all'interno di un ciclo
    - ad ogni iterazione del ciclo possiamo verificare se l'ultimo messaggio inviato dal programma è stato «BYE» e uscire di conseguenza!

N.B.: `select()` e le relative macro `FD_ZERO` e `FD_SET` **non fanno parte del programma del corso,** quindi non è richiesto di doverle utilizzare (né oggi né nelle prove di esame al calcolatore)

[Esercizio 1] Passi da fare

- Completare il codice in `chat-socket.c`
  - Seguire i suggerimenti presenti nei blocchi di commenti
  - Si noti che in ricezione si dovrà leggere 1 byte per volta!
- Per la compilazione, usare il `Makefile` fornito
  - `make chat-socket`
- Per l'esecuzione, usare due terminali ed un numero di porta >1024
  - `./chat-socket accept <porta>`
  - `./chat-socket connect 127.0.0.1 <porta>`

[Esercizio 2] Chat end-to-end via FIFO

- Simile alla chat end-to-end via socket, con la differenza che per la comunicazione usa una **coppia** di FIFO e non un singolo descrittore
  - si veda l'esercizio "EchoProcess su FIFO" per il paradigma delle coppie
- Analogamente all'implementazione via socket:
  - Ogni peer esegue i thread *sendMessage* e *receiveMessage*
  - La terminazione in *receiveMessage* per l'utente che invia il messaggio «BYE» viene gestita impiegando `select()`
- Completare il codice in `chat-fifo.c`, seguendo le istruzioni nei blocchi di commenti e compilando con `make chat-fifo`
- Per l'esecuzione, usare due terminali (`prefix` viene usato per formare il nome da dare alle due FIFO – si veda la funzione `main`)
  - `./chat_fifo accept <prefix>`
  - `./chat_fifo connect <prefix>`

Slides di riepilogo

- Socket
  - Mettersi in ascolto su una socket
  - Connettersi ad una socket
  - send() e recv()
  - Chiusura
- Pipe/FIFO
  - Creazione
  - read() e write()
  - Chiusura
  - [FIFO] Rimozione
- Gestione meccanizzata degli errori

Riepilogo su Socket

- Socket per comunicazione su stack protocollare TCP/IP
- Canale di comunicazione bidirezionale inter-processo (e anche inter-macchina naturalmente)
- Connessione tra due endpoint, ognuno con identità <IP, porta>
  - IP e porta rappresentati in network byte order
  - Per gli IP usare le funzioni `inet_addr()` e `inet_ntop()`
  - Per la porta usare le funzioni `htons()` e `ntohs()`
- Una volta instaurata la connessione, è possibile usare il descrittore della socket per inviare (`send()`) e ricevere (`recv()`) messaggi

Mettersi in ascolto su una socket

- Creazione socket: funzione `socket()`
  - Descrittore di socket dedicato per accettare nuove connessioni
- Binding della socket su un indirizzo locale: funzione `bind()`
- Attivare una listening socket: funzione `listen()`
- Attesa per accettare connessioni: funzione `accept()` bloccante
  - Vanno gestite eventuali interruzioni
  - Ritorna un descrittore ad una nuova socket per poter comunicare con l'endpoint remoto
  - *Negli scenari multi-thread, è possibile creare un thread dedicato per gestire la comunicazione e consentire così di mettersi in attesa di altre connessioni (senza necessità di join!)*

Connettersi ad una socket
Chiusura socket

- Creazione socket via funzione `socket()`
  - Descrittore di socket per la comunicazione con l'altro endpoint
- Connessione al server via funzione `connect()` che richiede:
  - Descrittore della socket da usare
  - Struttura dati di tipo `struct sockaddr` configurata con i dati dell'endpoint al quale connettersi
  - Dimensione di tale struttura dati
- Per entrambi gli endpoint, al termine della comunicazione la socket deve essere chiusa tramite `close()`

send() e recv()

- Semplificazione: ultimo parametro sempre uguale a `0`
- Invio messaggi: funzione `send()`
  - Vanno gestite eventuali interruzioni
  - Ricezione segnale `SIGPIPE` per scrittura su socket chiusa!
  - Ritorna il numero di byte realmente scritti, `-1` in caso di errore
    - Gestire esplicitamente situazioni di invii parziali
- Ricezione messaggi: funzione `recv()`
  - Vanno gestite eventuali interruzioni
  - Bisogna specifica il numero (massimo) di byte da leggere
  - Ritorna il numero di byte realmente letti, `-1` in caso di errore
    - Se la connessione viene chiusa dall'altro endpoint, ritorna `0`
    - Gestire ricezioni parziali se la dimensione dei dati è nota!

Riepilogo su pipe/FIFO

- Canale di comunicazione unidirezionale inter-processo (ma non inter-macchina)
- Simile ad un buffer su cui effettuare letture/scritture
- Per processi «relazionati» tramite `fork()` si usano le pipe semplici
  - Descrittori per lettura/scrittura ereditati dal processo padre
- Per processi non «relazionati» si usano le FIFO (named pipe)
  - Le FIFO vengono identificate tramite nome
  - Sono a tutti gli effetti dei file speciali
    - Provare `ls -al` nella directory corrente mentre un codice esegue!

Creazione di pipe/FIFO

- Creazione pipe
  - Funzione `pipe(int fd[2])`
  - Fornisce due descrittori: lettura con `fd[0]`, scrittura con `fd[1]`
  - Tipico utilizzo: `fork` con un processo lettore e l'altro scrittore
    - Lo scrittore chiude il descrittore di lettura
    - Il lettore chiude il descrittore di scrittura
- Creazione FIFO
  - Funzione `mkfifo()` prende in input il nome della FIFO
  - Solo un processo crea (e poi distrugge) la FIFO
  - Tutti i processi che usano la FIFO devono aprirla (incluso quello che l'aveva creata) con la funzione `open()`
    - In lettura: macro `O_RDONLY`
    - In scrittura: macro `O_WRONLY`
  - L'apertura di una FIFO è **bloccante** fino a quando non viene aperta l'altra estremità della FIFO stessa → possibilità di deadlock!

read() e write() su pipe/FIFO

- Invio messaggi: funzione `write()`
  - Vanno gestite eventuali interruzioni
  - Vanno gestiti invii parziali!
  - Ritorna il numero di byte realmente scritti, `-1` in caso di errore
  - Ricezione del segnale `SIGPIPE` quando si prova a scrivere su una pipe/FIFO ma tutti i descrittori in lettura sono stati chiusi
- Ricezione messaggi, funzione `read()`
  - Vanno gestite eventuali interruzioni
  - Ritorna il numero di byte realmente letti, `-1` in caso di errore
    - Ritorna `0` quando tutti i descrittori di scrittura della pipe/FIFO sono stati chiusi → [pipe] se un processo esegue `read()` ma non ha chiuso il suo descrittore di scrittura ho un deadlock!

Chiusura pipe/FIFO

- Chiusura pipe
  - Funzione `close()`
    - Dopo una `fork()`, padre e figlio hanno i descrittori di lettura e di scrittura entrambi aperti → ognuno deve chiudere il descrittore che non usa per evitare deadlock!
- Chiusura FIFO
  - Dopo `close()`, bisogna gestire la rimozione della FIFO
    - se non rimossa, la FIFO continua ad esistere sul filesystem e questo renderebbe problematica la `mkfifo()` nelle esecuzioni successive
  - Una volta chiusi tutti i descrittori associati alla FIFO, si può procedere alla rimozione tramite la funzione `unlink()`

Gestione meccanizzata degli errori

- Il comportamento delle funzioni in presenza di errori può variare
  - Gran parte delle system calls restituiscono `-1` e impostano la variabile `errno` con il codice dell'errore avvenuto
    - Esempio: `read()`
    - In questi casi, la macro `ERROR_HELPER` può stampare a video una descrizione dell'errore e terminare il programma
  - Altre funzioni restituiscono direttamente il codice dell'errore
    - Esempio: `pthread_create()`
    - La macro `GENERIC_ERROR_HELPER` permette di adattare la gestione dell'errore alle caratteristiche della funzione
      - Per le funzioni della libreria pthread, si può utilizzare la macro `PTHREAD_ERROR_HELPER` definita ad hoc
- Fare riferimento alla dispensa e al materiale dell'Esercitazione 4