Esercitazione [10]

Riepilogo su Socket e Pipe

Leonardo Aniello - aniello@dis.uniroma1.it
Daniele Cono D'Elia - delia@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. 2014-2015

Sommario

- Riepilogo socket
  - Mettersi in ascolto su una socket
  - Connettersi ad una socket
  - Send e receive su socket
  - Chiusura socket
- Riepilogo pipe/FIFO
  - Creazione pipe/FIFO
  - Read e write su pipe
  - Chiusura pipe/FIFO
- Gestione errori
- Esercizi
  - Chat su socket
  - Chat su FIFO

Riepilogo socket

- Socket per comunicazione su stack protocollare TCP/IP
- Canale di comunicazione bidirezionale inter-processo (e anche inter-macchina)
- Connessione tra due endpoint, ognuno nella forma IP:porta
  - IP e porta vanno 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 endpoint locale, funzione `bind()`
- Mettersi in ascolto, funzione `listen()`
- Attesa per accettare connessioni, funzione `accept()` bloccante
  - Vanno gestiti eventuali interrupt
  - Ritorna un descrittore per comunicare con l'endpoint remoto
  - *Negli scenari multi-thread, è possibile staccare un thread dedicato per gestire tale comunicazione e consentire così di rimettersi in attesa di altre connessioni*

Connettersi ad una socket
Chiusura socket

- Creazione socket, funzione `socket()`
  - Descrittore di socket per la comunicazione con l'altro endpoint
- Connessione al server, funzione `connect()` che richiede
  - Descrittore della socket
  - Struttura dati di tipo `struct sockaddr` impostata con le info sull'endpoint al quale connettersi
  - Dimensione di tale struttura dati
- Per entrambi gli endpoint, al termine della comunicazione la socket deve essere chiusa, funzione `close()`

Send e receive su socket

- Invio messaggi, funzione `send()`
  - Vanno gestiti eventuali interrupt
  - Non vanno gestiti invii parziali (semplificazione per il nostro corso)
  - Ultimo parametro sempre zero (semplificazione per il nostro corso)
  - Ritorna il numero di byte realmente scritti, `-1` in caso di errore
- Ricezione messaggi, funzione `recv()`
  - Vanno gestiti eventuali interrupt
  - Ultimo parametro sempre zero (semplificazione per il nostro corso)
  - 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`

Riepilogo pipe/FIFO

- Canale di comunicazione unidirezionale inter-processo (ma non inter-macchina)
- Da considerare come 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

Creazione pipe/FIFO

- Creazione pipe
  - Funzione `pipe(int fd[2])`
  - Fornisce due descrittori: lettura con `fd[0]`, scrittura con `fd[1]`
  - Utilizzo standard: un processo deve leggere, l'altro scrivere
    - Chi deve scrivere chiude il descrittore di lettura
    - Chi deve leggere chiude il descrittore di scrittura
- Creazione FIFO
  - Funzione `mkfifo()`, prende in input il nome della FIFO
  - Solo un processo crea 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 $\rightarrow$ possibilità di deadlock!!!

Read e write su pipe

- Invio messaggi, funzione `write()`
  - Vanno gestiti eventuali interrupt
  - Vanno gestiti invii parziali
  - Ritorna il numero di byte realmente scritti, `-1` in caso di errore
  - Se si cerca di scrivere su una pipe quando tutti i suoi descrittori di lettura sono stati chiusi, viene ricevuto il segnale `SIGPIPE` (broken pipe)
- Ricezione messaggi, funzione `read()`
  - Vanno gestiti eventuali interrupt
  - Ritorna il numero di byte realmente letti, `-1` in caso di errore
    - Ritorna `0` quando tutti i descrittori di scrittura della pipe sono stati chiusi → se un processo è bloccato in una `read()` e non ha prima chiuso un eventuale descrittore di scrittura della stessa pipe, si verifica un deadlock!!

Chiusura pipe/FIFO

- Chiusura pipe
  - Funzione `close()`
  - Dopo la `pipe()` e la `fork()`, padre e figlio hanno entrambi sia il descrittore di lettura che quello di scrittura aperti → ognuno deve chiudere il descrittore che non usa per evitare deadlock
- Chiusura FIFO
  - Oltre alla `close()`, bisogna gestire la rimozione della FIFO per evitare che nelle esecuzioni successive la `mkfifo()` fallisca
  - Quando tutti i descrittori della FIFO sono stati chiusi, la si può rimuovere con la funzione `unlink()`

Gestione errori

- In caso di errore, funzioni diverse si comportano diversamente
  - Alcune ritornano `-1` ed assegnano alla variabile `errno` il codice dell'errore avvenuto
    - Esempio: `socket()`
    - In questi casi, usare la macro `ERROR_HELPER`
  - Altre ritornano direttamente il codice dell'errore avvenuto
    - Esempio: `pthread_create()`
    - In questi casi, la macro `GENERIC_ERROR_HELPER` permette di esplicitare la condizione da controllare (es. `ret != 0`)
      - Per le funzioni specifiche della libreria *pthread*, si può utilizzare anche la macro `PTHREAD_ERROR_HELPER` definita ad hoc
  - Il codice delle macro si trova nell'appendice della dispensa

Esercizio: chat su socket

- Processo chat_socket in modalità *accept* (1° argomento)
  - Si mette in ascolto su una certa porta (2° argomento)
  - Una volta accettata una connessione, inizia la sessione
- Processo chat_socket in modalità *connect* (1° argomento)
  - Si connette ad un certo indirizzo (2° argomento) su una certa porta (3° argomento)
  - Una volta instaurata la connessione, inizia la sessione
- Durante la sessione, i processi
  - Leggono messaggi da `stdin` e li inviano all'altro processo
  - Stampano su `stdout` i messaggi ricevuti dall'altro processo
- Quando un processo invia «BYE», chiude la socket e termina
- Quando un processo riceve «BYE», chiude la socket termina

Esercizio: chat su socket
Come funziona

- Una volta iniziata la sessione, il processo lancia due thread
  - *receiveMessage*: bloccato sulla `recv()`, quando riceve un messaggio lo stampa a video
  - *sendMessage*: bloccato sulla `fgets()`, quando l'utente ha inserito un messaggio lo stampa a video e lo invia
  - Entrambi i thread lavorano sullo stesso descrittore di socket
- Quando un processo riceve il messaggio «BYE»
  - *receiveMessage* si sblocca e può terminare in maniera «pulita»
  - L'utente viene avvisato del termina della sessione ed invitata a premere «INVIO» per uscire
    - In questo modo anche *sendMessage* può sbloccarsi e terminare in maniera «pulita»

Esercizio: chat su socket
Problema sulla terminazione

- Quando un processo invia il messaggio «BYE»
  - *sendMessage* si sblocca e può terminare in maniera «pulita»
  - *receiveMessage* rimane bloccato sulla `recv()`, anche se la socket è stata chiusa da *sendMessage* → non può terminare in maniera «pulita» fino a che l'altro endpoint non chiude la connessione esplicitamente

Esercizio: chat su socket
Funzione `select()`

- Consente di monitorare più descrittori, in attesa che almeno uno diventi pronto per qualche operazione di I/O
- È possibile specificare un timeout
- Si sblocca quando si verifica una di queste situazioni
  - Almeno uno dei descrittori monitorati diventa pronto
  - Il timeout impostato scade
  - Arriva un segnale

Esercizio: chat su socket
Come viene usata la `select()`

- Prima di effettuare la `recv()`, viene usata la `select()` per monitorare il descrittore della socket con un certo timeout (1.5 s)
- In questo modo
  - La `recv()` viene invocata solo quando c'è effettivamente un messaggio da leggere dalla socket → non è più bloccante!!
  - In caso di assenza di messaggi ricevuti, la `select()` si sblocca ogni 1.5 secondi → il thread *receiveMessage* può verificare periodicamente se la sessione sia conclusa e di conseguenza può terminare in maniera pulita!!
- **NB: la funzione `select()` e le macro `FD_ZERO` e `FD_SET` non fanno parte del programma del corso, quindi non è richiesto di doverle utilizzare ne' oggi ne' all'esame**

Esercizio: chat su socket
Cosa fare?

- Completare il codice in `chat_socket.c`
  - Seguire i suggerimenti nei commenti «COMPLETE CODE HERE»
  - Gestire gli errori usando le apposite macro
- Per la compilazione, usare il Makefile fornito
- Per l'esecuzione, usare due terminali
  - `./chat_socket accept <porta>`
  - `./chat_socket connect 127.0.0.1 <porta>`

Esercizio: chat su FIFO

- Scenario identico alla chat su socket
- La comunicazione sfrutta una coppia di FIFO invece di una socket
- Anche in questo caso
  - Ogni processo ha i due thread *sendMessage* e *receiveMessage* che scrivono e leggono messaggi
  - La terminazione del *receiveMessage* nel processo che invia il «BYE» è problematica e viene gestita con la `select()`
- Completare il codice in `chat_fifo.c`, seguendo le istruzioni fornite ed usando le apposite macro per la gestione degli errori
- Per l'esecuzione, usare due terminali (`prefix` viene usato per formare il nome da dare alla FIFO)
  - `./chat_fifo accept <prefix>`
  - `./chat_fifo connect <prefix>`