Relazione progetto sistemi operativi e laboratorio

Object Store

Selene Gerali (Matricola 546091)

Anno Accademico 2018/2019

Indice

1. Strutturazione del progetto

1.1 Strutturazione del codice
1.2 Protocollo di comunicazione
1.3 Strutture dati di appoggio

2. Funzionamento generale

3. Gestione dei segnali

4. Gestione degli errori e Testing

4.1 Gestione degli errori
4.2 Testing

1. STRUTTURAZIONE DEL PROGETTO

Il progetto è strutturato in diversi moduli, ognuno dei quali svolge una determinata funzionalità.
Di seguito sono brevemente spiegati i vari moduli utilizzati e il loro uso.

1.1 Strutturazione del codice

- server.c:
  Codice relativo al main del server e ai meccanismi di gestione delle interruzioni. Accetta nuove connessioni di client e gestisce la registrazione di essi. Contiene i metodi che si occupano di creare un nuovo thread per ogni nuova connessione, il quale contiene meccanismi di ricezione e parsing dei messaggi inviati dal client in ascolto.

- op_server.c:
  Codice relativo alle funzioni che si interfacciano con il file system e con la tabella hash

- socket.c:
  Codice relativo alle funzioni che lavorano sul socket, in particolare sulla ricezione e sull'invio di un messaggio.

- hash_table.c:
  Codice relativo all'implementazione di una Hash Table concorrente.

- client.c:
  Codice relativo al main del client. Contiene le funzioni per effettuare le 3 tipologie di test richiesti. Il client si registra attraverso il nome utente passato come primo parametro e successivamente esegue uno dei 3 test in base al numero passato come secondo parametro. Il Test 1 esegue 20 STORE, il Test 2 esegue 20 RETRIEVE mentre il test 3 esegue 20 DELETE.

- objectstore.c:
  Codice relativo alla definizione di procedure per la gestione dei comandi

• util.c:

Contiene il codice relativo alle due funzioni: “readn” e “writen”.

Il file macros.h contiene varie macro utilizzate per la gestione di eventuali errori lanciati da parte di chiamate di sistema.

1.2 Protocollo di comunicazione

Il client invia un header al server contenente uno specifico comando. Questo header ha una dimensione fissa di 267 byte calcolati in base a: massima dimensione di un file POSIX (255) + massima dimensione della parola che costituisce il comando da eseguire (lunghezza della RETRIEVE = 8) + due spazi (2) + ‘\n\0’ (2).
Successivamente il server invia un messaggio di risposta, anch’esso di dimensione fissa di 8 byte, calcolati in base a: dimensione di KO (2) + codice di errore (2) + 2 spazi (2) + ‘\n\0’ (2).
Inoltre per leggere o scrivere un messaggio su un socket, utilizzo le funzioni fornite: “readn” e “writen”.

1.3 Strutture dati di appoggio

Come struttura dati di appoggio ho utilizzato una hash table, in cui ogni elemento della tabella è costituito da una chiave che è rappresentata dal nome del client (dato che è univoco) e dal file descriptor relativo. In questo modo la tabella rappresenta tutti i client connessi al server in un dato istante in modo da evitare così connessioni multiple da parte di client con lo stesso nome.
Essendo presenti più thread che agiscono sulla stessa struttura dati condivisa, è stato necessario introdurre un meccanismo di sincronizzazione attraverso l’utilizzo delle mutex offerte dalla libreria pthread.
Per inserire, ricercare o eliminare un client viene applicata la funzione hash al suo nome_utente, si acquisisce la mutex relativa alla partizione a cui appartiene ed infine si opera in lettura o in scrittura.

2. FUNZIONAMENTO GENERALE

La componente server è un eseguibile multi_threaded che rimane in attesa di richieste di connessione da parte di client i quali fanno uso di funzioni della libreria "objectstore" per connettersi, disconnettersi e fare richieste specifiche come la memorizzazione di un generico file, la lettura e la cancellazione di esso.
All'avvio dell'eseguibile viene inizializzata una hash table che rimane consistente per tutta la durata dell'esecuzione e viene creata una directory DATA dove verrano memorizzate tutte le directory relative ai client connessi. Inoltre il server mantiene delle informazioni riguardanti il suo stato interno come: numero di thread attivi, numero di file memorizzati e dimensione complessiva della directory.
Per ogni nuova connessione, il server ha il compito di creare un thread che si occuperà della gestione della comunicazione tra client-server.

A seguito di un segnale di terminazione, il server attende che i thread attivi in quell'istante terminino correttamente e dopodiché si occupa di deallocare la hash table inizializzata, rimuovere la directory DATA e infine eliminare il file name del socket dal workspace.

3. GESTIONE DEI SEGNALI

Il server dispone di un thread dedicato alla gestione dei segnali che vengono ricevuti durante l'esecuzione. Tale thread è non detached e le sue risorse vengono liberate tramite la funzione pthread_join eseguita dal main del server.
Il server per prima cosa effettua l'operazione adibita al mascheramento di tutti i segnali in modo che il thread "Signalhandler" non fa altro che mettersi in attesa di un segnale mediante la chiamata di sistema "sigwait". Quando il segnale viene catturato, se è di tipo SIGUSR1 il thread ha il compito di stampare sullo standard output le informazioni descrittive dello stato del

server, altrimenti setta ad 1 il contenuto della variabile puntata da "OS_STOPPED", il che determina la chiusura del server.

4. GESTIONE DEGLI ERRORI E TESTING

4.1 Gestione degli errori

Al fine del superamento dei test ho dato particolare importanza alla gestione degli errori. Oltre ai messaggi di fallimento delle operazione dei client sono stati gestiti i vari valori di ritorno di chiamate di sistema e di funzioni di libreria.

4.1 Testing

I vari test sono stati effettuati sulla macchina virtuale *Xubuntu* fornita durante il corso, su una macchina con sistema operativo *Ubuntu versione 18.04* e su una macchina con sistema operativo *Xubuntu versione 18.04.*