

RELAZIONE SISTEMI OPERATIVI LABORATORIO

Matricola 564117

Barberi Federico

2018/2019

Indice :

- **Introduzione al progetto;**
- **Moduli e loro implementazione/interazione;**
- **Comunicazione;**
- **Strutture dati condivise;**
- **Segnali;**
- **Gestione eventuali errori;**
- **Start;**
- **Test;**
- **Note finali.**

Introduzione al progetto:

Il progetto prevede la realizzazione di un object store per la memorizzazione di grandi quantità di dati. I client si connettono a un server utilizzando un preciso protocollo e possono registrarsi, memorizzare dati all'interno del loro spazio personale, richiedere dati e cancellare dati.

Il mio progetto si basa su un server sempre in ascolto di richieste di connessione da parte dei client, ogni qualvolta riceve una richiesta di connessione crea un thread, lanciato in modalità detached, per gestirla e, attraverso la funzione threadF, legge un header contenente una determinata operazione richiesta dal client e la gestisce.

Moduli e loro implementazione/interazione:

Il mio progetto si compone di 4 file.c:

1. Server.c;
2. Client.c;
3. Objstore.c con relativa interfaccia objstore.h;
4. Objlist.c con relativa interfaccia objlist.h;

Server.c :

Server.c è la parte centrale del progetto, implementa un socket sempre in ascolto di richieste da parte dei client, ogniqualvolta ne arriva una genera un thread per gestirla.

Dentro il threadF, il thread dispatcher che gestisce i messaggi dei client, sono implementate operazioni, tutte tramite chiamate di sistema(ex: open, read, write...) per la creazione di directory, file, scrittura all'interno di file, rimozione file, recupero contenuti da un file ecc..

Server.c inoltre si occupa della gestione dei segnali(SIGINT, SIGTERM, SIGUSR1) tramite un apposita funzione signalhandler che setta una o più variabili globali quando riceve uno di questi segnali, permettendo al server di terminare.

In caso si riceva SIGUSR1, server.c si occupa inoltre di formattare tutte le informazioni utili per le statistiche e poi scriverle su un file di nome testout.log, le informazioni contenute in questo file sono:

1. Numero client collegati al momento in cui si riceve il segnale;
2. Numero store andate a buon fine su 1000 tentate dal test;
3. Numero retrieve andate a buon fine su 600 tentate dal test;
4. Numero delete andate a buon fine su 400 tentate dal test;
5. Numero file contenuti nella directory data alla fine dei test;
6. Size totale (in byte) della directory data alla fine dei test.

Client.c:

Client.c rappresenta un semplice client fittizio utile per testare server e libreria objstore. Client.c prende a linea di comando un nome, che sarà poi il nome utente utilizzato da quel client per accedere all'objstore e richiedere le operazioni, e un intero nell'intervallo 1-3 corrispondente al tipo di test da eseguire(1->store, 2->retrieve, 3->delete). Prima di ogni test il client richiede la connessione all'objstore attraverso la funzione connect e alla fine richiama sempre la funzione leave per chiudere la connessione.

Objstore.c:

Objstore.c implementa le funzioni dell'interfaccia objstore.h, e' una libreria statica su cui i client si appoggiano per effettuare richieste al server.

Composta da 5 funzioni:

- Int os_connect(char * name);
- Int os_store(char * name, void * block, size_t len);
- Void * os_retrieve(char * name);
- Int os_delete(char * name);
- Int os_disconnect().

Nella mia implementazione objstore.c ha il compito di creare/chiudere il socket per la comunicazione tra client e server e di creare header che rispettino il protocollo definito dalle specifiche del progetto a seconda della chiamata di funzione effettuata dal client(es: REGISTER name\n se si richiama la funzione os_connect(name)), una volta creato l'header viene inviato al server che ha il compito di capire quale operazione effettuare e gestirla, una volta gestita in caso sia andato tutto a buon fine il server invia un messaggio di OK all'objstore, che prontamente fa sapere al client che l'operazione da lui richiesta e' andata a buon fine.

Objlist.c:

Objlist.c e' una libreria di appoggio, anch'essa statica, che ha il compito di implementare una struttura dati condivisa, precisamente una lista, la quale tiene traccia di quanti e quali client sono attivi in ogni momento nell'objstore.

Ho deciso di implementare questa struttura per la semplicita' nell'inserzione, cancellazione, stampa e conteggio elementi.

Lo scopo principale di questa struttura e' evitare che in un qualsiasi momento possano essere attivi piu di un client con lo stesso nome utente, quindi ogniqualvolta un client effettua una os_connect si va a controllare se in lista e' gia presente un client con lo stesso nome e in tal caso si invia un KO.

Essendo una struttura dati condivisa l'accesso per inserzione, cancellazione ed eventuali controlli viene "regolamentato" da una mutex, richiamata all'inizio di ogni funzione e rilasciata alla fine.

L'interfaccia objlist.h offre le seguenti funzioni:

- Void Delete_List(ListaElementi * l) : per la deallocazione della memoria;

- `Void DeleteConnection(ListaElementi *l, char * user_name)` : cancella l'elemento di nome `user_name` dalla lista;
- `Int Is_Already_Conn(ListaElementi l, char * user_name)` : ricerca all'interno della lista l'elemento `user_name` per vedere se `user_name` e' attualmente connesso al server;
- `Int Number_Elem(ListaElementi l)` : funzione di appoggio che conta il numero di elementi all'interno della lista, utile solo per un output piu "elegante" evitando di stampare la lista nel caso essa fosse vuota.
- `Void Print_List(ListaElementi l)` : stampa la lista;
- `Void Set_Connection(ListaElementi * l, char * user_name)` : nel caso in cui un utente abbia ricevuto l'ok per registrarsi al objstore, viene aggiunto il suo `user_name` alla lista attraverso un inserimento in testa.

Comunicazione :

Client e server si scambiano messaggi attraverso un socket dedicato. Il client si interfaccia con objstore che ha il compito di formulare la richiesta in modo corretto attenendosi al protocollo. Il server una volta che riceve un header da parte di un client analizza i primi caratteri per capire a quale specifica operazione fa riferimento (store, leave ecc..), successivamente attraverso la funzione di tokenizzazione rientrando seziona il messaggio salvandosi i vari campi in apposite variabili per poi andare a operare per gestire la richiesta arrivata. Ogni qualvolta viene riscontrato un errore il server si occupa di interrompersi, mandare un messaggio di KO al client, uscire dal ciclo, liberare la memoria e chiudere la connessione con quel client. In caso sia andato tutto per il meglio il server prepara ed invia un messaggio di OK al client, aggiorna i rispettivi contatori per le statistiche, libera la memoria utilizzata e torna in ascolto di eventuali altre richieste.

Strutture dati condivise:

Il mio progetto fa uso di due strutture dati condivise:

1. Lista condivisa -> `objlist.h objlist.c`;
2. Un array di interi di 5 posizioni.

La lista condivisa, `Cond_List`, ha lo scopo di tenere traccia in ogni momento di quanti e quali sono i client attivi che operano sull'objstore.

L'array invece ha lo scopo di tenere traccia di quante operazioni sono andate a buon fine, per poter poi costruire il file `testout.log` che mostra le statistiche del server.

I 5 campi dell'array sono 5 contatori, ognuno con un significato diverso:

1. `Stats_Array[0]` -> Numero client connessi;
2. `Stats_Array[1]` -> Numero store andate a buon fine;
3. `Stats_Array[2]` -> Numero retrieve andate a buon fine;
4. `Stats_Array[3]` -> Numero delete andate a buon fine;
5. `Stats_Array[4]` -> Numero file nella directory data.

Essendo anche questa una struttura dati condivisa, l'incremento dei contatori viene regolamentato da una lock (`mutex_stat`).

Segnali:

All'interno del server viene installato anche un signalhandler per la trattazione dei seguenti segnali :

1. `SIGUSR1`;

2. SIGINT;
3. SIGTERM.

Faccio uso di due variabili globali, definite static volatile sig_atomic_t, quali stampa e termina. Entrambe inizializzate a 0, se viene catturato uno dei tre segnali elencati sopra il signalhandler ha il compito di settare le variabili nel modo seguente:

- SIGUSR1 -> stampa = 1, termina = 1;
- SIGINT e SIGTERM -> termina = 1.

Di conseguenza quando riceve uno di questi tre segnali il server esce dal ciclo while(1) e nel caso abbia ricevuto SIGUSR1 prepara tutte le informazioni riguardanti le statistiche per poi scriverle nel file testout.log, solo successivamente resetta la variabile di stampa a 0.

Gestione eventuali errori:

Il mio progetto ogni volta che viene effettuata una chiamata di funzione che puo fallire quale calloc, open, read, write ecc.. controlla e nel caso non fosse andata a buon fine manda un messaggio di KO al client attraverso objstore, stampa sullo standard error un messaggio di errore, elimina il nome del client dalla lista degli utenti attivi e poi chiude la connessione con il client.

Di conseguenza se l'objstore riceve un messaggio di KO chiude anche esso il sockfd e invia il valore FALSE al client che stampera il KO e il nome della funzione che ha fallito e terminera' con fallimento (exit(EXIT_FAILURE)).

Start:

All'interno dell'archivio si trovano i seguenti contenuti:

1. 4 file.c che sviluppano il codice (server.c, client.c, objstore.c, objlist.c);
2. 3 file.h che rappresentano le varie interfacce (conn.h, objstore.h, objlist.h);
3. La working directory "data";
4. Lo script bash testsum.sh per l'esecuzione dei test;
5. Makefile.

Attraverso il Makefile e' possibile generare gli eseguibili e lanciare le batterie di test.

Con il target all e' possibile creare gli eseguibili per client e server, compresi linking di librerie, il server deve essere lanciato da linea di comando (./server) e successivamente richiamando il target test viene lanciato uno script bash (testsum.sh) che, inizialmente testa client e server secondo le specifiche, poi lancia il segnale SIGUSR1 al server e stampa a schermo il contenuto di testout.log mostrando l'esito dei test.

Il Makefile e' dotato anche di un target cleandata che permette di "pulire" la working directory data.

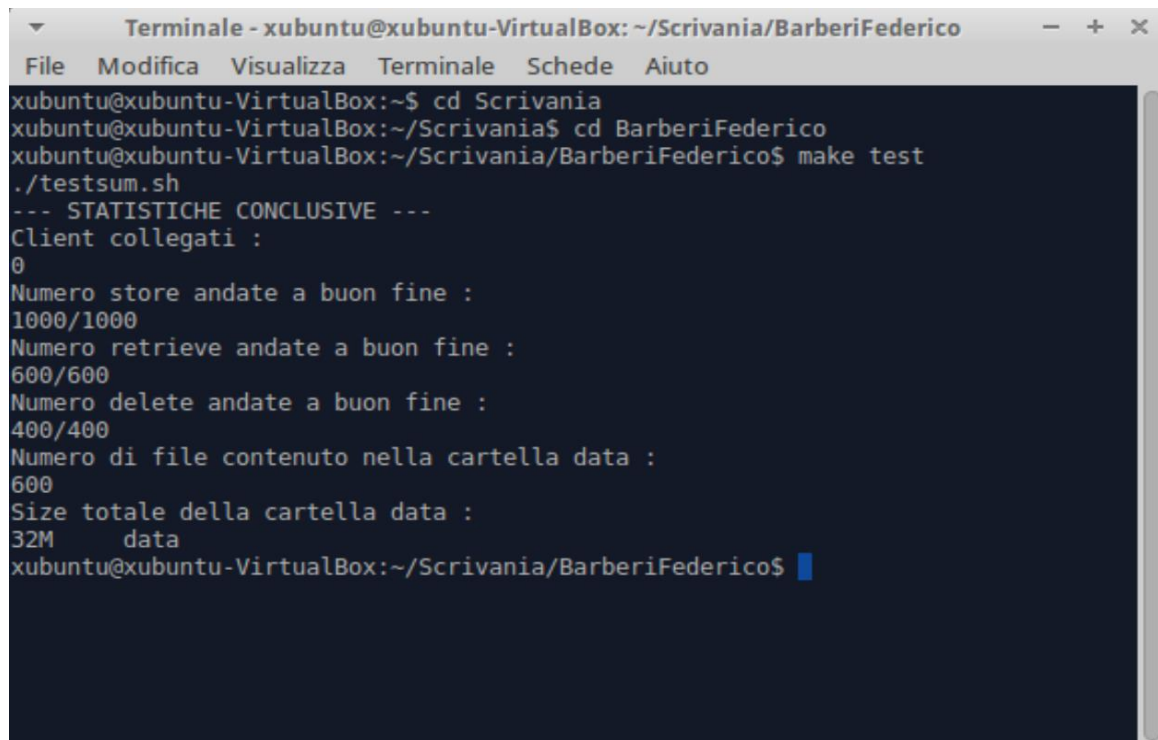
Test:

Quando viene richiamato il target test dal makefile viene eseguito un semplice script bash, testsum.sh, che va a eseguire la batteria di test definita nelle specifiche del progetto.

Alla fine della batteria viene lanciato il segnale SIGUSR1, attraverso la funzione kill, viene scritta sul file delle statistiche la dimensione della directory data al momento del segnale e viene fatto un cat del file testout.log in modo tale da vedere a schermo l'esito dei test. Una volta eseguito il cat, lo script si occupa di eliminare il file testout.log e termina.

Il progetto e' stato testato su due S.O. : xubuntu e macOS, la batteria di test e' compatibile solo con xubuntu dato che lo script fa uso della funzione "pidof" per la ricerca del pid del server, funzione che su mac non e' compatibile (ricerca del pid tramite : ps | grep server).

Il risultato di una batteria di test eseguita con successo e' il seguente :

A screenshot of a terminal window titled "Terminale - xubuntu@xubuntu-VirtualBox: ~/Scrivania/BarberiFederico". The terminal shows the following commands and output:

```
xubuntu@xubuntu-VirtualBox:~$ cd Scrivania
xubuntu@xubuntu-VirtualBox:~/Scrivania$ cd BarberiFederico
xubuntu@xubuntu-VirtualBox:~/Scrivania/BarberiFederico$ make test
./testsum.sh
--- STATISTICHE CONCLUSIVE ---
Client collegati :
0
Numero store andate a buon fine :
1000/1000
Numero retrieve andate a buon fine :
600/600
Numero delete andate a buon fine :
400/400
Numero di file contenuto nella cartella data :
600
Size totale della cartella data :
32M      data
xubuntu@xubuntu-VirtualBox:~/Scrivania/BarberiFederico$
```

Note finali:

Il codice e' stato scritto rispettando librerie conformi POSIX e testato con valgrind con attivi i flag per vedere i memory leak (--leak-check=full).

I test fanno uso di una stringa fittizia di 100 byte ("Lorem ipsum dolor..."), ripetuta inizialmente una sola volta e poi 100 volte, 150 volte ecc... per raggiungere i 100000byte la scaletta utilizzata e' stata la seguente : 100 – 10000 – 15000 -20000 - ... - 100000 andando poi ad aumentare circa 5Kbyte ogni volta.