Progetto di Sistemi Operativi e Laboratorio (a.a. 2018-2019)

Object Store

Francesco D'Izzia

Matricola 544107
Corso B

Indice

1. Struttura del progetto

Il progetto è stato strutturato in più file al fine di garantire maggiore flessibilità, modularità e organizzazione del codice stesso. Di seguito è possibile trovare la lista dei file con una sintetica descrizione del loro ruolo.

1.1. File sorgenti

- *server.c* : si occupa di gestire il server, mettendosi in attesa dei client e servendoli creando un relativo thread worker.
- *lib.c* : il vero "cuore" dell'Object Store, ovvero la libreria contenente le funzioni richieste (os_connect, os_store, etc...)
- *common.c* : contiene una serie di funzioni, variabili e macro ausiliarie necessarie sia al server, che alla libreria.
- *thread_worker.c* : descrive il loop eseguito dai vari thread worker generati dal server, legge l'header inviato dal client e lo manda al parser.
- *parser.c* : effettua il parsing delle richieste, una volta decodificata la request procede con l'esecuzione dell'operazione richiesta dal client.
- *client.c* : è il client che adopera le funzioni implementate nella libreria per l'invio delle varie richieste. Verrà richiamato dallo script bash per l'esecuzione dei test.
- *hashtable.c* : implementazione delle tabelle hash e delle relative funzioni.

1.2. Script bash

- *test_base.sh* : si occuperà di eseguire i test base (viene lanciato dal comando `make test`).
- *testsum.sh* : stampa a schermo un resoconto dei test eseguiti e manda il segnale SIGUSR1 al server (anch'esso viene lanciato da make test, subito dopo l'esecuzione dei test).
- *test_files.sh* : esegue un piccolo test extra (quest'ultimo viene lanciato dal comando `make test2`), vedere la sezione **Test e note** per maggiori informazioni.
- *monitor.sh* : script usato durante lo sviluppo e il debugging del progetto, per verificare la corretta chiusura dei thread legati al server.
- *usr1.sh* : altro piccolo script usato durante il debugging, invia semplicemente un segnale di tipo SIGUSR1 al server, che provvederà a stampare le statistiche a schermo.

2. Overview e scelte progettuali

Data l'impostazione del progetto, aldilà di alcuni vincoli legati alla rigidità del protocollo, è stato dato molto spazio riguardo alle possibili scelte implementative e concettuali personali.
In questa sezione cercherò di presentarle e di motivarle brevemente.

2.1. Gestione delle richieste dei client

Il server comprende un ciclo all'interno del quale esso si mette in attesa delle connessioni da parte dei client e attraverso una `select` con timer decide quando è il momento di servire un determinato client, creando un thread worker apposito che si occupa di prendersene cura.

Nelle fasi iniziali del progetto, al posto della select avevo optato per un file descriptor `non bloccante`, tuttavia ho deciso in seguito di cambiare perché da alcuni test effettuati, osservando attentamente l'utilizzo delle risorse e in particolare della CPU, quest'ultima, anche in assenza di connessioni da parte dei client, continuava a iterare nel ciclo, consumando continuamente il processore e avvicinandosi ad uno stato di *busy waiting*.

La select, scelta con un timer opportuno (in questo caso pari a `10 ms`), permette di alleggerire il carico di lavoro della CPU, senza dover rinunciare alla necessità da parte del server di non fossilizzarsi in attesa che un client si connetta: voglio infatti evitare che il server possa diventare "ostaggio" dei client.

2.2. Thread Worker

Ognuno dei thread worker si occupa di leggere l'header, inviato dal rispettivo client e di richiamare il parser per effettuare la successiva decodifica ed esecuzione della richiesta data.
In questo caso invece ho ritenuto ragionevole utilizzare un file descriptor **non** bloccante: difficilmente avrò dei client che rimangono connessi per diverso tempo senza fare assolutamente nulla, ma devo comunque fare in modo che il thread (e quindi il server, che lo aspetta) non si blocchi e diventi schiavo del client, basti pensare ad un client del tipo:

```
os_connect("user");
sleep(5);
os_disconnect();
```

Se il fd non fosse di tipo unblocking, non sarebbe possibile in questo caso far terminare (in modo pulito e ordinato, maggiori dettagli poco più avanti) il server e i thread nel bel mezzo dei 5 secondi di pausa del client, una cosa sicuramente poco gradita, che però grazie al fd non bloccante riusciamo a evitare.

2.3. Terminazione "gentile" del server e dei thread

Il server e i thread condividono una variabile booleana `running`, di tipo volatile sig_atomic_t: il continuo operare del server e dei thread è dettato da tale variabile, infatti finché è settata a true il loop andrà avanti, sia quello di accettazione dei client, che quello di decodifica delle richieste presente nel thread worker (notare che se il client ha finito e si è già disconnesso il corrispettivo thread worker terminerà ugualmente, a prescindere dalla variabile running).

Il server, come già specificato nella traccia, avvia la procedura di terminazione quando riceve un segnale (diverso da SIGUSR1 e SIGPIPE): tutto ciò che farà l'handler dei segnali in quel caso sarà semplicemente settare a false la variabile running, in modo tale da interrompere le continue iterazioni dei cicli. Infine si andranno dunque a eseguire determinate istruzioni per assicurarsi di uscire in modo sicuro e corretto.

Nel caso del server, esso si metterà in attesa della terminazione di tutti gli altri thread worker, sospendendosi sulla variabile di condizione `empty`, mentre nel caso dei thread si andrà prima a decrementare la variabile `n_clients`, che rappresenta il numero di client attualmente connessi al server, dopodiché si controllerà nuovamente il valore: nel caso sia uguale a zero, verrà mandata una signal al server per svegliarlo, che fatto ciò potrà deallocare le risorse e poi terminare.

2.4. Signal Handler Thread

La gestione dei segnali, considerando l'ambiente multi-threaded, ho deciso di gestirla utilizzando un thread apposito. Tale thread, finché la variabile running è settata a true, rimane in attesa dei segnali attraverso la `sigwait`, per poi andare a verificare la tipologia di segnale ricevuto: nel caso sia un SIGUSR1 allora procederò verso la stampa delle statistiche, in ogni altro caso (fatta eccezione per SIGPIPE, che viene ignorato) setto il flag running a false e preparo il necessario per la terminazione.
Prima di generare il thread avrò già mascherato i vari segnali, eseguendo `sigfillset` per comunicare di voler bloccare tutti i segnali e settando la maschera con `pthread_sigmask`.

2.5. Stampa delle statistiche

La stampa delle statistiche, essendo richiamata direttamente dal signal handler, necessita di utilizzare funzioni *async-signal-safe*: per questo motivo avevo pensato di utilizzare una write piuttosto che una printf per stampare le informazioni a schermo, tuttavia, dato che sarebbe risultato scomodo convertire alcune tipologie di dati in stringhe (così da stamparli con write), ho quindi cambiato approccio, facendo in modo di stampare le statistiche al di fuori dell'handler, dal quale invece verrà settato un flag nel caso di SIGUSR1, il quale verrà controllato nel loop del server e, nel caso sia settato a true, procederà con la stampa delle statistiche (che essendo fuori dall'handler potrà utilizzare printf senza problemi). Il calcolo delle statistiche viene effettuato mediante la funzione ricorsiva `ftw` (*file-tree-walk*).

Le seguenti informazioni vengono stampate: size totale dello store (in byte e MB), numero di oggetti, numero di sottocartelle all'interno della cartella `data` e numero di client attualmente connessi.

2.6. Parsing delle richieste

Per quanto riguarda il parser: si occupa semplicemente, sfruttando il formato definito dal protocollo, di spezzettare la richiesta in più campi (tipo di operazione, nome dell'utente/dell'oggetto, lunghezza dell'eventuale dato inviato e newline). Il comportamento è abbastanza semplice per quanto riguarda la maggior parte delle richieste: di default leggo uno standard di 512 byte, che saranno sicuramente abbastanza per leggere fino al newline, dato che al caso più impegnativo oltre all'header di base e spazi (una decina di byte) dovrò leggere il nome utente (che avrà un massimo di 255 byte, limite dettato da UNIX stesso) e un certo numero di cifre numeriche che descrive la lunghezza (che sicuramente entrerà nei restanti byte).
Un po' più interessanti da analizzare sono la STORE e la RETRIEVE (essenzialmente simmetriche nell'implementazione).

La STORE calcola, in base alla dimensione di ciò che ha letto nell'header, se è necessaria una ulteriore read di `x` byte per concludere la lettura, dove x è dato dalla differenza tra la lunghezza dei dati e la quantità di dati già letti nella prima read. Se invece ha già letto tutti i dati con la prima read può procedere direttamente alla creazione dell'oggetto.

La RETRIEVE essenzialmente funziona in modo quasi identico, la differenza consiste nel fatto che la lettura dei dati viene fatta dal client (dato che riceverà l'header di "risposta" con i dati dal server), andrà a tokenizzare l'header ricevuto e farà lo stesso calcolo della STORE per capire se ha finito di leggere o meno i dati.

2.7. Gestione di più client dello stesso utente

Dalla traccia veniva detto che i nomi erano garantiti essere distinti tra i vari client, non era però specificato se era possibile una situazione in cui uno stesso utente poteva avere o meno più client registrati e operanti nello stesso momento.
Ho scelto quindi di gestire il suddetto caso facendo in modo di registrare gli utenti in una struttura dati condivisa, così da poter verificare, prima di ogni registrazione, che l'utente non sia già connesso attraverso un altro client.
Nel caso in cui sia già presente un utente collegato con quel nome, il secondo utente fallirà l'operazione di registrazione, restituendo `KO Multiple clients with the same username \n`.

Inizialmente la struttura dati che avevo adoperato era una semplice linked list, tuttavia in seguito ad alcuni test ed esperienze dirette ho deciso di passare ad una tabella hash, principalmente per questioni di velocità.

Dato che la tabella hash è condivisa, ho piazzato una mutex per ogni cella della tabella (quindi una per ogni linked list). La tabella hash gestisce essenzialmente delle stringhe (i nomi degli utenti). Dopo una ricerca relativa a delle funzioni hash operanti su stringhe, ho pensato di scegliere la funzione hash **djb2** di Dan Bernstein (`http://www.cse.yorku.ca/~oz/hash.html`), che si è distinta tra le altre candidate per le ottime performance registrate.

3. Test e note

3.1. Esecuzione dei test

Per eseguire i test sarà prima necessario, dopo aver compilato con il comando `make` o `make all`, aver avviato il server (avviandolo normalmente da shell con `./server` o eseguendo `make dserver` qualora si voglia avviarlo con valgrind e relativi flags).
Avviato il server, basterà dare il comando `make test` per eseguire i test e vederne i risultati.

3.2. Test supplementare

Oltre ai test base, per assicurarmi del corretto funzionamento durante lo sviluppo, ho voluto testare il comportamento nel caso in cui i client andavano a inviare dati binari di diverse tipologie, magari di uso quotidiano, come immagini e file pdf. In questo micro-test, ci sono quattro utenti che vanno a caricare quattro tipologie di file (due immagini, un pdf e una gif animata) e richiedono lo storing dei suddetti oggetti tramite libreria. I file originali sono presenti nella directory `testFiles`: al termine di questo test (eseguibile tramite il comando `make test2`) se il tutto è andato a buon fine sarà possibile verificare manualmente la presenza dei vari file all'interno dei determinati spazi utente nella directory `data`.

È possibile provare ad aggiungere ulteriori tipologie di file nella cartella testFiles, per poi avviare da shell il client digitando una cosa del tipo:
`./client "nome_utente" 4 "./testFiles/nome_file" "nome_oggetto"`.

3.3. Ulteriori info

Il progetto è stato testato sui seguenti OS:

- Ubuntu 18.10
- Xubuntu 14.10 (macchina virtuale ufficiale del corso)