Relazione sul Progetto di Sistemi Operativi e Laboratorio

A.A. 2020/2021

Daniele Del Zompo

13 luglio 2021

1 Introduzione e informazioni generali

Il progetto è disponibile su Github alla repository pubblica con indirizzo `https://github.com/DigletDiz/SOLproject`.

2 Makefile

Per compilare il progetto è messo a disposizione un Makefile con diverse opzioni.

`make` L'opzione base è la regola `all`, che può essere invocata dando il comando `make`. Questa regola compila l'intero progetto.

`make cleanall` Per ripulire la directory dai vari file oggetto, eseguibili e librerie viene anche messa a disposizione una regola `make clean`.

`make test1` La regola `make test1` mostra le varie opzioni del client.

`make test2` La regola `make test2` esegue il secondo test, che mostra l'algoritmo di rimpiazzamento del server.

3 Includes

Tra i vari header creati, come `queue.h` e `list.h` che implementano delle semplici code e liste, degno di nota è sicuramente `rwmutex.h`, che contiene le strutture per implementare delle read-write lock. `commcs.h` contiene le strutture di comunicazione tra client e server.

Per implementere le hashtables utilizzate nel progetto, ho utilizzato un codice preso da Jakub Kurzak, maggiori dettagli in `icl_hash.c`.

4 API

Le funzioni API sono definite in `api.h`. Tra queste, `myWriteFile` permette di scrivere un file nel server

La comunicazione tra server e API sfrutta un preciso protocollo. In particolare le API inviano una struttura dati al server. Questa struttura dati è presente in `commcs.h` e si chiama request, che può contenere oltre al codice dell'operazione anche il pathname del file richiesto, contenuto del file se si sta facendo una write etc.

Ogni chiamata di API prima di concludersi aspetta un feedback da parte del server, per sapere se la richiesta è stata soddisfatta o meno.

Le funzioni `lockFile`, `unlockFile`, `removeFile` e `writeFile` non sono state realizzate. Per ovviare alla mancanza di scrittura nel server è stata comunque creata, come già menzionato, `myWriteFile`, la quale può prendere come argomento più file separati da una virgola (esempio -v ./cartellaFile/file1,./cartellaFile/file2).

5 Il client

Il client sfrutta le API fornite per inoltrare delle richieste al server. Le richieste vengono passate al client tramite linea di comando ed esso si occupa di farne il parsing e controllare che rispettino i vincoli dati dalla specifica. Questo controllo tuttavia viene ignorato se viene passata al client l'opzione `-h`, che stampa un messaggio che spiega le varie opzioni e fa terminare il client immediatamente.

Per fare il parsing il client sfrutta `getopt`. ATTENZIONE: ci sono opzioni con valori opzionali: `-R` e `-t`. Per chiamare queste opzioni con un valore esplicito NECESSARIO non lasciare spazi tra il comando e il valore scelto.

Rispetto a quanto descritto nella specifica, il client ha un'opzione `-a` che permette di fare l'append di un file presente nel server. Esempio di comando append -a ./fileTest1/ade.txt@" ciao". In pratica l'argomento passato viene parsato e diviso in: path del file (prima di @) e stringa da appendere a path(dopo la @). La stringa dopo la @, che è quella che viene effetivamente aggiunta alla fine del file richiesto, deve essere tra "" come nell'esempio.

il client ha un'altra opzione aggiuntiva `-v` che permette di scrivere un file nel server usando `myWriteFile`. Come detto anche in precedenza è possibile passare più path relativi separati da una virgola.

Il client si occupa di trasformare ogni path relativo dato dall'utente in un path assoluto, in modo che il server identifichi i file solamente attraverso il loro path assoluto. La directory passata alla opzione `-d` se non esiste viene creata.

6 Il server

Il server è l'applicazione multithreaded che gestisce lo storage di file. All'apertura legge un file di configurazione con il seguente formato:

```
SOCKNAME=<nome del sockname>
MAX_FILE= <numero massimo di file>
STORAGE_CAPACITY= <capacità massima del server in bytes>
CLIENTS_EXPECTED= <il numero di client attesi per la sessione>
CORES = <i core della macchina dove si esegue il server>
THREAD_WORKERS = <numero di thread worker nel thread pool>
```

Di default il server assume che il file di configurazione si trovi in `server_config.txt`; se si vuole cambiare basterebbe cambiare il valore della define presente in cima al server.

Il server crea due hashmaps: una per contenere i file, l'altra per tenere i client connessi con i rispettivi file aperti. Questo perché quasi tutte le operazioni(come la read o l'append) richiedono che il client che ha fatto richiesta abbia in precedenza aperto il file in questione.

Viene creata una coda dove vengono inseriti anche qui i file presenti nel server. Questa coda serve per decidere quale file deve essere espulso dall'algoritmo di espulsione se si raggiunge la capacità massima o se si supera il numero di file massimo.

Viene creata anche una serie di rwmutex, struttura presente in `rwmutex.h`. Il numero di mutex create è uguale al numero di core passati come argomento nel file di config (oppure sono uguali al MAXFILE se i core fossero di più del massimo di file). Queste read-write lock servono per rendere la struttura più efficiente: infatti ogni lock viene assegnata ad una porzione dell'hashtable così da aumentare la concorrenza.

Le richieste dei client sono gestite interamente dai thread workers: il main thread è bloccato sulla select e aspetta di ricevere richieste dai client per poi mandarle ai workers. I workers inviano la risposta a seconda della operazione richiesta tramite una struttura presente in `commcs.h` che si chiama reply, dove sono presenti i campi necessari a rispondere alla richiesta del client. Il worker sia in caso di errore che di successo manda un messaggio di risposta al client: se il messaggio è di errore gli manda il codice relativa all'errno generato dall'errore.

La select può svegliarsi anche tramite la chiusura della signalpipe, nel qual caso avvia il protocollo di terminazione a seconda del segnale ricevuto.

Un'altra pipe è stata utilizzata per rimandare dai workers i descrittori dei client che hanno concluso una richiesta per essere riinseriti nel readyset della select.

Per gestire la memoria limitata dai parametri definiti dal file di configurazione, il server implementa degli algoritmi di rimpiazzamento, che consentono di eliminare file in modo da liberare spazio. L'algoritmo espelle i file tramite una coda FIFO.

7 Test e scripts

Per testare le varie funzionalità del server e del client sono stati creati due file di test, `scripts/test1.sh` e `scripts/test2.sh`.

Il primo serve a mostrare le varie opzioni del client e il suo funzionamento, mentre il secondo serve a mostrare il funzionamento dell'algoritmo di rimpiazzamento dei file.

I due script di test possono anche essere lanciati scrivendo `make test1` e `make test2` del Makefile.