UNIVERSITÀ DI PISA

Relazione Progetto di Sistemi Operativi e Laboratorio

Chiara Maggi
Mat. 578517

A.A. 2020/21

INTRODUZIONE

Il progetto prevede la realizzazione di un file storage server, di un client e di una API, utilizzata per le comunicazioni. Le funzioni della API vanno a scrivere e leggere su un socket di comunicazione le varie informazioni, permettendo al client di fare determinate richieste al server, e al server di soddisfarle seguendo il protocollo "richiesta risposta".
Al seguente link è possibile trovare la mia repository github pubblica:
https://github.com/ChiaraMaggi/ProgettoSOL20-21

SERVER

Organizzazione server:

Il server è implementato come un singolo processo multi-threaded secondo lo schema "master-workers". Il server può essere configurato manualmente, passandogli con l'opzione "-f fileconfig" il file che si vuole usare come riferimento. Il file verrà parsato dallo specifico programma parsing.c e deve essere necessariamente nella seguente forma:

```
--File Storage Server Configuration--

-Number of Workers Thread:
NUM_WORKERS_THREAD = 4

-Max Number of Files:
MAX_NUM_FILE = 10

-Storage Size (in MByte):
STORAGE_SIZE = 1

-Socket Name:
SOCKET_NAME = ./SOLsocket.sk

-Number of buckets for hashtable:
NUM_BUCKETS = 1000
```

Se non viene passato nessun file o il parsing non va a buon fine il server verrà settato con valori di default.
Il thread master corrisponde al thread main, che sfrutta il multiplexing di canali per restare in ascolto di nuove connessioni da parte dei client e inviare le richieste del client ai worker. La comunicazione tra master e worker è implementata tramite una coda (formata da nodi del tipo queuenode_t) di file descriptor: il thread master, quando riceve una nuova richiesta da un client pronto in lettura, ne inserisce il file descriptor nella coda; in seguito uno dei thread worker libero lo preleverà.
Una volta prelevato il file descriptor di un client, il worker soddisfa la sua richiesta che è passata sotto forma di una struct request_t con due campi: la tipologia di richiesta (enum type_t) e un puntatore a delle informazioni. La tipologia della request andrà a richiamare la funzione corretta per la gestione di quel determinato tipo di richiesta, interpretando il comando e sfruttando il protocollo di comunicazione con l'API tramite il socket.
Il worker quindi ritorna una risposta che verrà correttamente decodificata dalla API al client. In particolare ho assunto che la risposta = 0 indica nessun errore mentre -1 = ENOENT, -2 = EPERM, -3 = EEXIST e -4 = EFBIG. Una volta soddisfatta la richiesta, il worker rispedisce il file descriptor del client, tramite pipe al master thread, almeno che la richiesta appena soddisfatta dal worker non sia

closeConnection. In quel caso siamo sicuri che il client non abbia più richieste da fare e quindi non è necessario che il master thread si rimetta in ascolto su quel file descriptor.
Ho gestito gli accessi concorrenti sulla coda dei file descriptor dei client con una mutex sulla coda e una variabile di condizione per evitare attesa attiva quando la coda è vuota.

Organizzazione cache
Per implementare la cache ho adottato come struttura dati di appoggio una hashtable (in icl_hash.c sono presenti funzioni di utilità per la gestione di una hashtable) con liste di trabocco, la cui grandezza è settabile nel file di configurazione. I nodi sono del tipo Node_t, che tra i vari campi ha: 'key' che identifica univocamente ogni nodo e contiene il path assoluto del file memorizzato nella cache, e 'data' che contiene una particolare struct file_t che permette di memorizzare informazioni utili relative al determinato file.
In particolare, file_t contiene:
- il file descriptor del creatore;
- un indice per gestire le politiche del rimpiazzo;
- un array di interi che conterrà in ogni momento i client che hanno il file aperto (possono essere al massimo MAX_OPEN macro definita da me);
- una mutex per gestire l'accesso concorrente al file;
- il contenuto del file:
- la dimensione;

Inoltre per gestire correttamente l'accesso concorrente alla cache ho utilizzato un'altra mutex che blocca l'intera hashtable.
La politica di rimpiazzo utilizzata è del tipo FIFO, e sfrutta il campo 'index' dei vari file memorizzati: ricerca il più piccolo indice, che indicherà il file inserito meno recentemente.
Durante tutto il periodo di attività del server vengono aggiornate due strutture aggiuntive: una tiene costantemente aggiornato lo stato del server (numero di file memorizzati, spazio libero e occupato), l'altra memorizza il cambiamento delle statistiche nel corso della vita del server, che verranno poi pubblicate alla chiusura (massimo numero di file memorizzati, massima quantità di Mbytes…).
Nota: nel codice sono presenti numerose fprintf commentate che possono essere utilizzate se si vogliono visionare, anche lato server, i vari errori che per scelta implementativa ho reso visibili solo lato client.

CLIENT
Il client può comunicare al server le richieste, esclusivamente da riga di comando, utilizzando i comandi disponibili.
Al momento dell'invio delle varie richieste la prima cosa che viene fatta è il controllo specifico sui singoli comandi che necessitano di determinate condizioni per funzionare.
In base al tipo di comando viene richiamata la specifica funzione che andrà a sfruttare le funzioni dell'API per comunicare al server cosa fare.

Per qualsiasi comando il client manderà al server esclusivamente i path assoluti dei file che si vogliono far interagire con la cache.
Ogni comando sarà diviso in più azioni a seconda se si opera su uno o più file. Alla fine di ogni azione viene pubblicato l'esito del comando sul singolo file, cosicché se un certo comando è andato a buon fine solo in parte è possibile capire cosa è fallito e cosa no.
Nel caso di -w, -W, -r le operazioni principali specifiche della singola istruzione (writeFile o readFile) sono sempre precedute da openFile e succedute da closeFile.

API

La API contiene le funzioni che il client può utilizzare per comunicare con il server. Tutte le funzioni presenti seguono uno schema standard, che consiste nello scrivere la richiesta (del tipo request_t) sul socket (e in alcuni casi anche altre informazioni) e leggere una risposta in base alla quale verrà settato l'errno corretto (vedi 'SERVER').

MAKEFILE

il Makefile presenta i target:
- **all** di default per costruire gli eseguibili server e client;
- **clean** per eliminare gli eseguibili;
- **cleanall** per eliminare tutti i file generati dal makefile quali eseguibili, file oggetto, temporanei, librerie…
- **test1** per lanciare il primo test: dopo aver avviato il server, con valgrind in background, e il file di configurazione specifico, lancia uno script bash test1.sh che a sua volta lancia vari client per testare tutte le opzioni.
- **test2** per lanciare il secondo test: dopo aver avviato il server, senza valgrind in background, lancia un singolo client che testa le sostituzioni in cache.

In entrambi i casi una volta che uno script termina viene inviato il segnale SIGHUP al server.