GIORGIO DELL'IMMAGINE
601843
g.dellimmagine@studenti.unipi.it

1 Introduzione

Il progetto è stato sviluppato nella repository pubblica https://github.com/gio54321/os-project.
Sono state sviluppate tutte le parti opzionali, ad eccezione della memorizzazione dei file in formato compresso. In particolare è stata implementata la produzione del file di log, il test3, le operazioni di `lockFile` e `unlockFile`, l'opzione `-D` del client (i file espulsi vengono quindi mandati al client) e le politiche di rimpiazzamento LRU e LFU.
Non sono state utilizzate librerie esterne. Sono state utilizzate le implementazioni delle funzioni `readn` e `writen` fornite durante il corso.

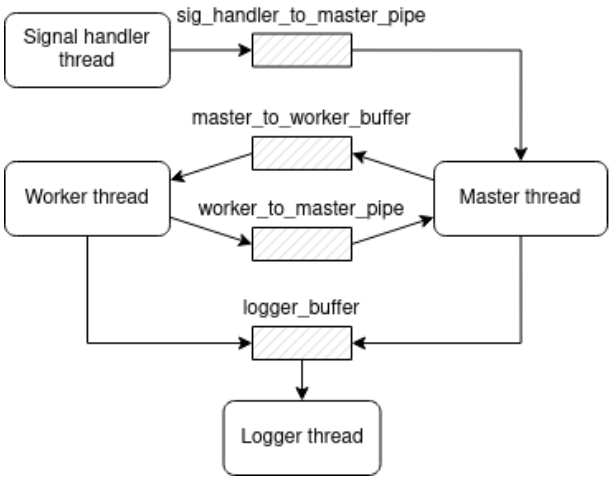
2 Protocollo di comunicazione

In generale ogni pacchetto inizia con un byte, che indica il tipo del pacchetto. I byte che seguono sono dipendenti da questo byte. Ad esempio il pacchetto di tipo `COMP` non contiene altre informazioni, mentre il pacchetto di tipo `FILE_P` contiene informazioni su un filename e dei dati. Stringhe e dati vengono sempre preposti dalla loro lunghezza. All'interno del file `docs/protocol_specification.txt` è definita la specifica completa di tutti i pacchetti.
Il protocollo di counicazione è basato su un modello di richiesta-risposta. Tutte le operazioni iniziano con una richiesta al server, da parte del client, e terminano con una risposta dal server, che può essere del tipo `COMP`, che indica che l'operazione è stata completata con successo, oppure `ERROR`, che indica che l'operazione non è andata a buon fine. Il pacchetto `ERROR` contiene anche un error code, che indica il tipo di fallimento.
All'interno del file `docs/protocol_specification.txt` è definita la specifica completa del protocollo di comunicazione.

3 Il server

3.1 Architettura e comunicazione fra thread

I vari thread comunicano tramite degli unbounded shared buffer e delle pipe. Ci sono due thread di supporto: uno per il logging e uno per il signal handling. Le relazioni di comunicazione sono illustrate in figura (è stato riportato per comodità un solo thread worker).



3.2 Implementazione delle operazioni di lock

Le operazioni di `lock` e `unlock` sono implementate tramite una coda di attesa, che è presente per ogni file nel server. Quando una operazione di `lock` viene richiesta su un file libero, allora ha successo, mentre se viene eseguita su un file dove è stata precedentemente richiesta la lock da un altro client, allora il server mette il client nella coda di attesa di quel file, e non manda la risposta al client (per rispettare la semantica

che l'operazione di lock non termina fino a che non viene garantita la mutua esclusione). Quando poi il client detentore della `lock` eseguirà l'operazione di `unlock` sul file, un client viene estratto dalla coda e viene completata l'operazione di `lock` messa in sospeso precedentemente.
Da notare è il fatto che se un file viene rimosso, sia per una richiesta di rimozione, sia per una espulsione, il server fa fallire tutte le operazioni di `lock` che erano in attesa su quel file.

3.3 Strutture dati

La struttura dati `vfile_t` rappresenta un file virtuale. Essa contiene informazioni quali il nome del file, dimensioni e un puntatore ai dati. Inoltre sono presenti due `fd_set`, uno per indicare quali client hanno aperto il file, e il secondo per indicare quali client sono attualmente nella coda di attesa per ricevere la lock su quel file. Una conseguenza di questo fatto è che il client che viene estratto dalla coda di attesa è sempre quello con file descriptor più alto, il che potrebbe portare a starvation di client con file descriptor più bassi, se si verificasse che molti client richiedono la mutua esclusione su un file. Questa implementazione è stata comunque scelta perché è estremamente semplice e in ogni caso rispetta la specifica fornita (non è indicato un ordine particolare di acquisizione della mutua esclusione).
I `vfile_t` sono contenuti all'interno di un `file_storage`, che è una collezione di vfile implementata tramite una lista doppiamente concatenata.

3.4 Gestione della concorrenza

La concorrenza è stata gestita da una singola reader-writer lock che controlla l'accesso all'intero file storage. In particolare i thread worker richiedono l'accesso in lettura solamente per servire le richieste di read e readN. Per servire tutte le altre richieste i worker richiedono l'accesso in scrittura in quanto, data la possibilità di espellere file, le operazioni devono essere eseguite in mutua esclusione.

3.5 Gestione delle politiche di rimpiazzamento

Limplementazione della politica FIFO è banale in quanto essendo i file memorizzati tramite una lista, basta sceglierne iprimo elemento. Per l'implementazione della politica di rimpiazzamento LFU ogni file ha un contatore di "accessi", che viene incrementato ad ogni operazione che riguarda il file. L'implementazione della politica di rimpiazzamento LRU è analoga, semplicemente ogni file ha un campo che indica l'ultimo timestamp di utilizzo, che viene aggiornato ad ogni operazione che riguarda il file.
Siccome questi campi devono poter essere aggiornati anche durante le operazioni di read, l'accesso è protetto da una mutex.
La funzione `atomic_update_replacement_info()` si occupa di agiornare atomicamente questi campi.

3.6 Produzione del file di log

Per la produzione del file di log è presente un thread dedicato che riceve delle stringhe da uno shared buffer e le scrive sul file di log. Questo garantisce che non ci sia concorrenza nelle scritture del file su disco. Il formato generale dei messaggi di log che vengono scritti dai worker quando servono delle richieste è il seguente:
`[timestamp] [W: numero worker] [C: id client] [op] (REQUEST|SUCCESS|INFO) contenuto`

3.7 Chiusura della connessione da parte del client

La chiusura della connessione da parte del client è segnalata semplicemente dalla chiusura del file descriptor del socket. Dato che i client vengono identificati dal loro file descriptor, e dato che una volta chiuso il file descriptor del client da parte del server, lo stesso file descriptor sarà utilizzato in futuro per identificare un nuovo client, si rende necessario eseguire una procedura di cleanup dell'intero file storage. Questo vuol dire chiudere tutti i file che aveva aperto, eseguire la unlock di tutti i file di cui deteneva la lock e pulirlo da eventuali code di lock. Al termine della procedura di cleanup è garantito che può connettersi un altro client con lo stesso file descriptor e non si avranno effetti indesiderati, quindi al termine (e solo al termine) è possibile chiudere il file descriptor del client.
La procedura di cleanup risulta particolarmente importante se il client, per qualche motivo, termina bruscamente senza chiudere o eseguire la unlock dei file sul server.

3.8 Parsing del file di configurazione

Il file di configurazione ha una sintassi molto semplice del tipo `chiave=valore`. Il file viene letto all'avvio del server e contiene le seguenti informazioni: numero di thread worker, massimo numero di file e massima dimensione dello storage, il nome del socket e la politica di rimpiazzamento.

3.9 Gestione dei segnali e terminazione

La gestione dei segnali è stata effettuata tramite l'ausilio di un thread deticato. Il signal handler thread comunica con il server tramite una pipe, la cui parte di lettura viene inserita nel set della select. Il thread rimane in attesa tramite la chiamata `sigwait`, e quando riceve un segnale invia tramite la pipe il codice `HARD_EXIT`, oppure `SOFT_EXIT`, in modo tale che il thread master possa gestire i due tipi di terminazione. A tale scopo il server mantiene un contatore dei client attualmente connessi. I thread worker e logger vengono terminati tramite una chiamata `close` ai rispettivi buffer, che fa fallire ogni scrittura sul buffer, svuota il buffer e quando è vuoto fa fallire ogni lettura dal buffer.

3.10 Note di implementazione

Quando un client vuole creare un file, (quindi richiede una `openFile(_,O_CREATE)`) e il numero massimo di file nel server verrebbe superato, un file viene eliminato dal server e non viene mandato al client, in quanto la chiamata `openFile` non ha campi che indichino dove scrivere il file rimosso.

4 Il client

4.1 Gestione delle operazioni e dei path

Ogni operazione è pensata come a sé stante, per cui ogni richiesta è preceduta da una `openFile` ed è susseguita da una `closeFile`.
Il client è pensato per funzionare sia per lavorare con path relativi, che con path assoluti, in particolare il filename mandato al server sarà lo stesso che viene fornito dall'utente tramite la riga di comando. I file espulsi e letti sono memorizzati dentro le cartelle specificate con le opzioni `-D` e `-d`, e sono inseriti automaticamente all'interno delle loro rispettive sottocartelle. Se le sottocartelle non esistono vengono create ricorsivamente dal client.
Ad esempio se la cartella specificata per i file letti è `/A/B`, il file letto `/C/D/E.xyz` verrà memorizzato in `/A/B/C/D/E.xyz`, mentre il file letto `F/G.xyz` verrà memorizzato in `/A/B/C/F/G.xyz`.

5 Testing e script bash

5.1 Unit tests

Durante lo sviluppo è stato molto utile avere test di unità per testare sigolarmente i vari moduli. Si possono eseguire tutti i test di unità tramite il target `make run-all-tests`. I sorgenti per i test di unità sono dentro la cartella `tests`.

5.2 Target di test

I target di test usano alcuni file di varia natura, che si trovano dentro `test_data`. I file `rand-*.bin` sono stati generati con dei byte casuali.
I file letti dal client sono scritti dentro `test_out`, in particolare dentro `test_out/read` per i file letti e dentro `test_out/ejected` per i file espulsi.

5.3 Script per le statistiche

Tutti gli script sviluppati si trovano nella cartella `scripts`.
Lo script delle statistiche prende in ingresso il filename del file di log. Un esmpio di utilizzo potrebbe essere `./scripts/statistiche.sh log.txt`.
Lo script esegue il parsing del file di log per ricavare le informazioni richieste. Le informazioni sono ricavate tramite l'utilizzo di comandi standard unix, come `wc`, `awk`, `sed`, `grep` e `bc`.