SOL - Progetto

Francesco Lorenzoni - 599427

July 13, 2021

Contents

1 Introduzione

Parti facoltative Sono state realizzate tutte parti facoltative tranne la compressione dei file, in particolare: file di log, script statistiche.sh, target test3, lockFile/unlockFile, flag -D, algoritmo di rimpiazzamento LRU.

Librerie di terze parti È stata usata la libreria `icl_hash` fornitaci sulla pagina Didawiki del corso e realizzata da *Jakub Kurzak*. Sono state utilizzate le implementazioni di `readn` e `writen` fornite durante il corso.

GitHub repo github.com/frenzis01/SOL-Progetto

2 Filesystem

Concorrenza È stata utilizzata una singola mutex (`store.lockStore`) per le operazioni legate allo stato "globale" dello storage, quali ricerca, modifica del contenuto, creazione e eliminazione dei file; tale mutex viene acquisita all'inizio e rilasciata solo al termine di una delle suddette operazioni. Per gestire la concorrenza sui singoli file sono state utilizzate due mutex (`ordering` e `mutex`) e a una variabile di condizione `go`, applicando sui singoli file una delle soluzioni al problema dei lettori/scrittori viste durante il corso.
C'è una funzione del filesystem per ogni funzione di `clientApi.h` (ad eccezione di `writeFile()`, che viene realizzata dal filesystem tramite `appendToFile()` con un flag settato). Le principali differenze fra tali funzioni si possono osservare fra quelle come `appendToFile()` o `readNFiles` - che necessitano di mantenere la mutex globale per tutta la durata dell'operazione - e quelle come `lockFile()` che invece hanno bisogno di acquisire la mutex globale solo per la ricerca del file.

O_LOCK Per ogni file è presente un campo `lockedBy` e una coda `lockersPending` che rappresentano il file descriptor dell'attuale owner del flag O_LOCK e la coda di chi è in attesa di acquisirlo. In questo modo non è necessario che un thread worker attenda che l'acquisizione di O_LOCK vada a buon fine, sarà libero di elaborare altre richieste; Non appena l'owner rilascerà O_LOCK (o il file verrà rimosso) allora verrà aggiornato l'owner e un worker potrà comunicare l'esito dell'acquisizione di O_LOCK al nuovo owner o a tutti i client che erano in attesa su quel file, in caso di rimozione di quest'ultimo.

Strutture Dati I file sono salvati in una struttura FIFO e viene utilizzato un dizionario per velocizzare le ricerche (il dizionario referenzia nodi della lista usando come chiave i path dei file). Le strutture sono mantenute sincronizzate.

Eviction Se il server è stato configurato per rimpiazzare i file con la politica LRU, allora ogni chiamata (con successo) a `storeSearch()` rimuove dalla lista `store.files` il file trovato e lo reinserisce in coda. In questo modo i file referenziati più recentemente si trovano in coda.
Sia in caso di FIFO che di LRU l'algoritmo può ottenere i possibili candidati per la rimozione scorrendo la lista dalla testa.
Se `storeEviction()` è stata chiamata da `appendToFile()`, evita di provare a rimuovere il file su cui era stata chiamata la suddetta `appendToFile()` ed evita i file vuoti; se invece è stata chiamata da una `openFile(O_CREATE)` non c'è il pericolo di rimuovere un file che non esiste nello storage e considera validi anche i file vuoti. `storeEviction()` viene sempre chiamata due volte, una prima volta facendole ignorare i file con O_LOCK settato, la seconda considerandoli validi.

Note sulla semantica `appendToFile()` e `readFile()` falliscono se il file non è stato precedentemente aperto con `openFile()` dal Client richiedente. Qualsiasi richiesta di un Client, ad eccezione di `closeFile()`, fallisce se il file ha il flag O_LOCK settato e il richiedente non ne è l'owner.

writeFile() A livello di implementazione `writeFile()` è realizzata usando `appendToFile` aggiungendo il controllo che l'operazione immediatamente precedente richiesta dal Client sia una `openFile(O_CREATE|O_LOCK)`. Per realizzare tale controllo è presente, per ogni file, il campo `fdCanWrite` che viene inizializzato solo da una chiamata con successo a `openFile(O_CREATE|O_LOCK)` e viene settato a 0 da tutte le altre funzioni quando terminano con successo. Questo ci garantisce che se il valore di `fdCanWrite` corrisponde al File Descriptor del Client richiedente la `writeFile()`, allora l'ultima operazione terminata con successo su tale file può essere solo una `openFile(O_CREATE|O_LOCK)` richiesta dallo stesso Client.

3 Server

Comunicazione Per la comunicazione fra i thread sono presenti due pipe e un buffer (FIFO) di Client richiedenti gestito con una mutex e una variabile di condizione. I canali di comunicazione sono dunque i seguenti:
`signalHandler` $\xrightarrow{pipe}$ `Dispatcher`, `Workers` $\xrightarrow{pipe}$ `Dispatcher`, `Dispatcher` $\xrightarrow{buffer}$ `Workers`.

Segnali Il server ignora `SIGPIPE`. I segnali vengono gestiti da un thread dedicato tramite `sigwait()`. In caso di `SIGINT`, `SIGQUIT` o `SIGHUP` il signalHandler setta opportunamente il flag `myShutdown`, chiude la sua pipe ('svegliando' la select del thread Dispatcher) e termina. Il Dispatcher, se la condizione di terminazione è verificata, termina. Dopo aver eseguito `pthread_join(Dispatcher)` il main invia *#workers* richieste `NULL` ai Worker, alcuni dei quali potrebbero essere ancora in attesa sulla variabile di condizione del buffer da prima che venisse settato `myShutdown` e dunque essere ignari della terminazione del thread Dispatcher che, chiaramente, non invierà loro altre richieste.
È stato gestito anche il `SIGUSR1`, che non comporta la terminazione del signalHandler e che rappresenta una richiesta di stampa su `stdout` delle statistiche del server, la quale, richiedendo l'acquisizione di `store.lockStore` (dunque essendo potenzialmente lenta), verrà gestita da uno dei worker.

Worker Un Worker riceve un Client che ha formulato una richiesta tramite il buffer condiviso con il Dispatcher (il quale chiama `pthread_cond_signal()` ogni volta che inserisce un client nel buffer). Se la richiesta è `NULL` allora è stata inviata dal `main()` e termina, altrimenti legge la richiesta.
Se durante una lettura o una scrittura un Worker si accorge che un Client si è disconnesso provvede a rimuovere il Client dallo storage, ad avvisare eventuali nuovi O_LOCK owner e a scrivere `fd_client * (-1)` sulla pipe con il thread Dispatcher, il quale, leggendo un valore negativo chiamerà `close(fd_client)` e aggiornerà il numero di connessioni attive.
Se la lettura della richiesta va a buon fine, il Worker chiama la funzione del Filesystem necessaria e informa il Client dell'esito. In caso ci siano file letti o evictati, questi vengono inviati al Client. In particolare, nel caso di file evictati, il Worker provvede ad informare tutti i client che stavano cercando di acquisire O_LOCK su tali file che il file desiderato non è più presente nello storage.
Al termine dell'elaborazione di una richiesta il Worker scrive il `fd` del Client richiedente sulla pipe con il Dispatcher. Due casi particolari che hanno una gestione leggermente diversa sono le richieste di `lockFile()` e `unlockFile()`.

Dispatcher Il thread Dispatcher aggiunge al set di ascolto della select i file descriptor dei client che sono pronti a scrivere (che si sono appena connessi, o che ha ricevuto dalla pipe con i worker) e rimuove quelli che hanno fatto una richiesta (che verrà "presto" presa in carico da un worker). Ascolta (in lettura) la pipe con i Workers e la pipe con il signalHandler, la quale verrà chiusa solo dopo l'avvenuta ricezione di un segnale di terminazione.

Logging L'accesso concorrente al logger è gestito tramite una mutex. Si tiene traccia delle operazioni eseguite sul Filesystem e del loro risultato, dell'invio di file (sia evictati che letti) ai Client, dell'aggiunta/rimozione dei Client e dell'avvio/terminazione del server. Al fine di non ridurre eccessivamente le prestazioni, le scritture sul file di log non avvengono direttamente su disco, ma su un'area in memoria di dimensione fissa. Quando tale area risulta troppo piena per permettere una nuova scrittura, ne viene riportato il contenuto su disco.

4 Client-Server

Protocollo comunicazione La comunicazione fra Client e Server si basa su richieste di lunghezza e campi fissi (ad eccezione degli ultimi tre, la cui dimensione è determinata da quelli precedenti), in modo da sapere sempre quanti byte devono essere scritti/letti. In base all'operazione richiesta verranno utilizzati alcuni campi e altri no. Una richiesta è così strutturata:

Operazione	OpenFlags	nFiles	pathLen	dirLen	appendLen	path	dir	append
`char`	`char`	`int`	`u_short`	`u_short`	`size_t`	`pathLen`	`dirLen`	`appendLen`

L'invio dei file letti/rimossi da server a client sfrutta lo stesso principio. Un file inviato è così strutturato:

pathLen	path	size	content
`u_short`	`pathLen`	`size_t`	`size`

Client Api La formulazione delle richieste richieste e la ricezione dei file sono basate sul protocollo sopra descritto. La semantica delle funzioni è inalterata rispetto a quella descritta nel testo del progetto, con l'eccezione di due aggiunte: la possibilità di salvare su disco anche i file ricevuti in seguito a una `removeFile()` o `openFile()` in una directory (`char*` globale di default `NULL`) e l'espansione automatica di eventuali path relativi passati come argomento in path assoluti, dato che il Filesystem del server assume di utilizzare solo path assoluti.

5 Client

Command-Line parsing È stato realizzato un parser che sfrutta `getopt` per generare una coda di `Option` su cui il Client potrà iterare per formulare le richieste al server. In caso di uso improprio di una option, il parser non fallisce, si limita a stampare su stdout un messaggio di errore e a non inserire nella coda da restituire la suddetta option, garantendo la consistenza della coda restistuita.
Una `Option` è composta dal flag che identifica l'operazione e da un argomento, che può essere un singolo path, una lista oppure `NULL` a seconda dei casi.

Semantica Flag Nel testo non erano espressamente discusse quali chiamate di funzioni dell'Api del Client corrispondessero ai flag; è stata scelta un'implementazione che sembrasse ragionevole e fosse in linea con il resto del progetto, tuttavia altre scelte erano possibili e valide. In particolare, i flag -W e -w vengono implementati con `openFile(O_CREATE|O_LOCK)` seguita da `writeFile()` in caso di successo, mentre in caso di file già esistente viene richiesta `openFile()` senza flag seguita da `appendToFile()`.

Flag -d e -D Affinché questi flag siano considerati validi devono essere usati immediatamente dopo il corrispondente flag di lettura/scrittura, e la directory specificata viene tenuta in considerazione solo per la precedente operazione di lettura/scrittura.
Quando un file viene salvato in una directory, viene ricostruito il path assoluto del file all'interno della suddetta directory, creando tutte le subdir necessarie, onde evitare che un file venga sovrascritto da un altro con lo stesso nome ma con path assoluto diverso. Un file può essere sovrascritto solo in caso di ripetuto salvataggio dello stesso. (Esempio: `-r /dir1/dir2/f -d evicted` ⇒ File salvato in `./evicted/dir1/dir2/f`)

Flag -E È stato aggiunto il flag `-E` per settare il percorso della directory in cui salvare i file ricevuti in seguito a una `openFile()` o a una `removeFile()`. Se non viene passato nessun argomento, allora disabilita il salvataggio dei file su disco.

6 Note aggiuntive

Gestione errori In caso di un errore durante l'esecuzione di una funzione, si esegue una sezione di codice di cleanup, in cui viene liberata tutta la memoria allocata durante l'esecuzione e infine si ritorna un valore (generalmente -1 o `NULL`) che indica che la funzione non è terminata correttamente. Rispettando questo approccio e controllando il valore di ritorno della chiamate a funzione è possibile propagare l'errore a ritroso, liberando la memoria allocata ad ogni 'step', dunque permettendo di poter terminare l'esecuzione del processo senza memory leak anche in caso di errore.
Sia per il server che per il client è stata applicata l'idea sopra descritta, per quanto possibile. Fa eccezione il `main()` di `server.c` che gestisce errori in fase di inizializzazione e terminazione del server con `exit(EXIT_FAILURE)`.

statistiche.sh Lo script statistiche.sh viene eseguito al termine di ciascuno dei tre target test.

O_LOCK Deadlock Il server non gestisce eventuali deadlock dei Client legati al flag O_LOCK

Test personali Sono presenti anche alcuni file di test personali utilizzati in fase di sviluppo.

Argomenti server Il server richiede come argomento da linea di comando il percorso del file di configurazione. È possibile passare un secondo argomento, dal contenuto irrilevante, che abiliterà le stampe su stdout del Server.
Nei tre test realizzati il server viene eseguito con un solo argomento.