Progetto di laboratorio di sistemi operativi

Samuele Bonini

Indice

1 Premessa

Parti opzionali. Sono state svolte *tutte* le parti facoltative, in particolare: flag `-D` del client, algoritmi di rimpiazzamento LRU e LFU, compressione dei file con algoritmo RLE, operazioni di lock/unlock dei file, file di log in JSON, script `statistiche.sh` e target `test3` del Makefile.

Librerie di terze parti. Per la tabella hash, è stata usata l'implementazione di Jakub Kurzak fornitaci su Didawiki.

Repository Git. https://github.com/samul-1/sol-project

2 Filesystem

Il modulo. Il filesystem è incarnato dalla `struct CacheStorage_t`, e i file dalla `struct FileNode_t`. Di seguito, con "filesystem" e "file" ci si riferisce a queste due struct. Oltre alle strutture dati (descritte al prossimo punto), il filesystem contiene i parametri per monitorare la capacità massima e corrente, una mutex e un buffer per il logging – entrambi descritti in seguito. I file contengono, oltre ai campi ovvi: un intero che rappresenta l'fd del client che attualmente ha la lock sul file, una lista di file descriptor corrispondenti ai client che hanno aperto quel file, una lista di file descriptor di client che sono in attesa di acquisire la lock, i campi necessari a gestire le capacity miss coi vari algoritmi e i tool per la mutua esclusione.

Infine, il modulo `filesystemApi.h` espone una serie di funzioni handler che implementano tutte le funzionalità del filesystem. Queste funzioni sono chiamate all'occorrenza dai thread worker per gestire le richieste dei client.

Strutture dati. Si è scelto di utilizzare una doppia struttura dati per lo storage dei file: una lista linkata con inserimento in coda e una tabella hash. Lo scopo è quello di rendere efficienti tutte le operazioni del filesystem. In particolare, la tabella hash viene usata per le operazioni di inserimento, ricerca e cancellazione con complessità $O(1)$ al caso medio, mentre la lista viene utilizzata per il rimpiazzamento dei file, e richiede $O(n)$ operazioni per trovare la vittima.

Mutua esclusione. È presente una mutex a livello "globale", su tutto lo storage, e due mutex per ciascun file. L'accesso ai file avviene secondo un protocollo di tipo lettore/scrittore,

impiegando una variabile di condizione e garantendo che ci sia al più un solo scrittore – e nessun lettore –, dove con scrittura s'intende qualsiasi operazione che modifica la struttura del file (anche solo un lock); possono invece esserci più lettori sullo stesso file. Ogni operazione su file inizia con l'acquisizione della mutex globale; si effettua poi una ricerca e si acquisisce la mutex del file, se esso viene trovato. Dopodiché, le operazioni che non hanno effetto sullo storage globale (ovvero non creano né cancellano file) rilasciano la mutex globale, mentre le operazioni che richiedono sincronizzazione a livello globale (cancellazione, creazione e scrittura) mantengono anche la mutex globale per tutta la loro durata. Così facendo non può verificarsi lo scenario in cui un file viene cancellato mentre un thread è in attesa di acquisire la mutex su di esso.

Semantica di open, close, lock e unlock. Le operazioni di open e close sono disaccoppiate da quelle di lock e unlock. Il lock/unlock è stato interpretato come una meccanica di "permessi", che quindi possono essere settati anche mentre il file non è in utilizzo. Se un client effettua il lock su un file, gli altri client che lo avevano aperto prima lo manterranno aperto, ma non potranno effettuare operazioni su di esso fino allo sblocco. Se un client tenta di effettuare lock su un file già bloccato, il thread che gestisce la richiesta lo inserirà in una lista di file descriptor in attesa, e non gli darà alcuna risposta, né reinserirà il suo fd nel readset del server (*v. sezione Server*). Questo approccio evita che un worker si blocchi fintanto che non può soddisfare la richiesta di lock da parte di un client. Quando il detentore del lock sbloccherà il file in questione, il thread che gestirà la richiesta riceverà in uscita da `unlockFileHandler()` l'fd del primo client in lista di attesa e gli invierà il codice `OK`, concludendo la richiesta che lo aveva posto in stato di wait. Qualora invece il file venisse cancellato, il thread gestore della richiesta invierà a tutti i client in attesa su quel file il codice `FILE_NOT_FOUND`.

Logging. Il logging delle operazioni viene effettuato mediante una chiamata a `logEvent()` presente dentro tutte le funzioni handler del filesystem. Gli eventi vengono formattati in JSON e inseriti in un apposito buffer concorrente. Un thread lanciato dal server esegue in un loop la `dequeue` dal buffer e scrive i nuovi eventi sul file indicato nel `config`.

Rimpiazzamento. In base all'algoritmo di rimpiazzamento impostato nel file di configurazione, il filesystem utilizzerà la politica FIFO, LRU o LFU per selezionare le vittime. I file hanno due campi, `refCount` (resettato dopo la gestione di ogni capacity miss) e `lastRef`, che vengono aggiornati a ogni operazione che li riguarda, e un campo `insertionTime` per tenere traccia dell'ordine di creazione (per la politica FIFO); nel file `cacheFns.c` è definita una funzione di confronto file per ciascuno degli algoritmi implementati. Le funzioni effettuano il confronto su uno dei tre campi sopra citati in accordo con l'algoitmo in uso. La scelta della vittima diventa quindi un problema di trovare il "massimo" di una lista – la lista di file sul server –, dove in questo caso il massimo è definito in base all'ordine definito dalla funzione di rimpiazzamento, la quale viene passata come argomento a `getVictim()`. I file vittima devono essere mandati al client, pertanto quando vengono passati a `destroyFile()` l'argomento `deallocMemory` è impostato a false. Questo provoca l'unlinking del file dallo storage, ma la struct vera e propria non viene deallocata e viene passata tramite un parametro in uscita al chiamante, il quale la invierà poi al client (*v. sezione API client e protocollo di comunicazione*) e deallocherà il file.

Solo i file espulsi in seguito a write/append verranno inviati al client: quelli espulsi in seguito alla creazione di un nuovo file che eccede il limite sul numero di file non verranno inviati. Questa scelta deriva dal fatto che la funzione `openFile()` sul client non accetta un parametro `dirname` per salvare i file ricevuti dal server, a differenza di write e append.

Write e append. Il filesystem prevede una sola funzione per gestire sia write che append: `writeToFileHandler()`. Questo deriva dal fatto che si è osservato che la write è un caso particolare di append in cui il file è vuoto prima della scrittura. Per garantire la semantica della funzione client `writeFile()`, il filesystem mette a disposizione la funzione `testFirstWrite()`, la quale restituisce un booleano che indica se il client ha aperto il file con `O_CREATE|O_LOCK` dal client e non vi è ancora stato scritto niente. La funzione internamente controlla il campo `canDoFirstWrite`, che contiene l'fd del client che ha il diritto di fare la write. Ogni operazione su file setta questo campo a zero, garantendo che la funzione fallirà sempre se sono state fatte altre operazioni sul file. Il thread che si occupa di gestire una richiesta di write chiamerà come

prima cosa questa funzione: in caso di successo, procederà con la scrittura, altrimenti risponderà al client con codice `FORBIDDEN`.

Compressione. I file sono salvati sul server in formato compresso con l'algoritmo *run-length encoding* (RLE), sostanzialmente sostituendo a sequenze di byte identici la coppia `<byte, numero occorrenze>`. La variante utilizzata non aggiunge il numero di occorrenze per "sequenze" di un singolo byte; per le sequenze di due o più byte identici pone però due volte lo stesso byte prima del numero di occorrenze per eliminare ambiguità (altrimenti non sapremmo se un numero è il dato originale oppure il numero di occorrenze del byte precedente), e spezza run più lunghe di 9 occorrenze in run da 9, per avere sempre il numero di occorrenze su una singola cifra. Per esempio, `aaaaaaaaaaaab` diventa `aa9aa3b`.

3 Server

Implementazione. La comunicazione tra thread manager e thread worker avviene utilizzando un buffer concorrente. Il manager ascolta il readset con `select()` e, una volta ricevuta una richiesta da un client, inserisce nel buffer l'fd del client e lo rimuove dal readset. Il worker che lo leggerà facendo la `dequeue` gestirà la richiesta del client. Al termine della richiesta, il worker scriverà l'fd su una pipe (anch'essa ascoltata dal manager tramite `select()`), il quale lo reinserirà nel readset una volta letto. Se un worker legge `EOF`, scriverà `0` sulla pipe, segnalando l'uscita di un client (necessario per la gestione della terminazione). Se il client aveva richiesto una lock che non può essere soddisfatta al momento, il thread non scrive nulla sulla pipe, non dà alcuna risposta al client, e prosegue gestendo le prossime richieste.

File di config. Il parser del file di configurazione cerca, su ogni riga, una sola coppia `chiave:valore` in base al delimitatore passato come parametro. È possibile inserire linee vuote o commenti che iniziano con `#`. I commenti devono trovarsi su una riga a sé. L'insieme dei setting supportati dal server è esemplificato nel file `completeConfig.txt`, nella root directory del progetto.

Terminazione. Per la gestione di `SIGHUP`, viene impostato il flag `softExit`. All'arrivo di un nuovo client, il manager controlla sempre se tale flag è impostato e, se lo è, chiude immediatamente la connessione col nuovo client. Ogni volta che legge `0` dalla pipe, il manager controlla se il numero di client connessi (mantenuto in una variabile aggiornata a ogni connessione e disconnessione) è uguale a zero. In tal caso, il flusso del programma esce dal ciclo `while` principale. Nel caso di `SIGINT` o `SIGQUIT`, il flag `hardExit` causa l'uscita dal ciclo alla successiva iterazione (che, se il manager era bloccato su `select()`, avviene immediatamente dopo il ritorno con codice `-1`). Il resto della procedura di uscita è identica per i due casi: viene inviato un messaggo di terminazione per ogni worker e uno al thread del logger, si effettua la join su tutti i thread, e infine si rilasciano le strutture dati, stampando le statistiche. Quest'ultima parte può essere fatta senza l'acquisizione esplicita di mutua esclusione poiché il manager è l'unico thread rimanente a questo punto.

4 API client e protocollo di comunicazione

Messaggi. Lo scambio di messaggi tra client e server utilizza il seguente pattern: prima di ogni segmento del payload, se ne antepone la lunghezza in byte, normalizzata su un numero di cifre noto sia al client che al server (dieci). Così facendo, il destinatario del messaggio può effettuare una prima `read` di lunghezza pre-determinata, allocare un buffer in ingresso di dimensioni pari al valore letto, ed effettuare una seconda lettura di quella dimensione.

In aggiunta a quanto detto, le richieste del client e le relative risposte del server sono precedute da un singolo byte che identifica la richiesta/risposta (per esempio, `REMOVE_FILE` nella richiesta e `FORBIDDEN` nella risposta). La `openFile()` aggiunge inoltre un byte in fondo alla richiesta, dopo il path del file da aprire, che corrisponde ai flag utilizzati.

Per quanto riguarda `readNFiles()`, la richiesta è formata dal byte del codice richiesta seguito dal numero di file che si vuole leggere.

Autenticazione e protocollo uscita. I client sono identificati univocamente dal server mediante il loro fd. Dato che un processo può riciclare gli fd dopo averli chiusi, occorre garantire che un successivo client non venga "scambiato" per il precedente nel caso in cui condivida il numero di file descriptor. Per impedire ciò, all'uscita di ogni client il server esegue la funzione `clientExitHandler()` che: sblocca tutti i file sui quali il client aveva fatto la lock (eventualmente notificando il primo client in lista di attesa sul file), rimuove il client dalla lista di attesa di un file se questo era bloccato in seguito a una richiesta di lock, e chiude tutti i file che il client aveva aperto sul server (ovvero rimuove il suo fd dalla lista che tiene traccia di quali client hanno aperto il file).

5 Client

Path dei file. I comandi `-w` e `-W` del client accettano path relativi. Tuttavia, i file vengono inviati al server denominandoli col loro path assoluto. Il path dato in input al client viene passato come argomento a `realpath()` per ottenere il relativo path assoluto. Questa decisione è stata presa per evitare collisioni nel caso in cui due client inviino al server file che si trovano in path assoluti diversi, ma che coincidono nel path relativo rispetto a dove si trova l'eseguibile del client. Questo significa che, per i comandi `-r`, `-l`, `-u` e `-c`, il path che si passa come argomento potrebbe essere in generale diverso da quello che si è passato per scrivere il file. Per esempio, se mi trovo in `/home/Desktop/progetto` e dentro alla directory ho sia l'eseguibile del client che un file `file1.txt`, potrò dare al client il comando `-W file1.txt`, ma per leggerlo dovrò usare `-r /home/Desktop/progetto/file1.txt`.

Flag -D e -d. In accordo con l'utilizzo del path assoluto per l'invio dei file, i file salvati con l'opzione `-d`/`-D` manterranno il path che avevano sul server: tutte le directory presenti nel loro pathname verranno create ricorsivamente all'interno della directory specificata come argomento.

6 Gestione degli errori

Approccio generale. Si è optato, ove possibile, per la propagazione degli errori, demandandone la gestione al chiamante della funzione che ha generato l'errore. Le funzioni del filesystem e quelle dell'API del client restituiscono `-1` e settano `errno` in caso di errori non fatali. Alcuni errori, come quelli generati da chiamate quali `p_thread_mutex_lock()` e simili, sono trattati come errori fatali e causano l'uscita immediata dal programma (con la famiglia di macro `DIE_ON_*`). Se possibile, viene fatto un cleanup minimo prima dell'uscita, ma in caso di errori di questo tipo non vi è comunque alcuna garanzia di successo.

Sul client. Quando una richiesta del client genera un errore sul server, il server risponde con un codice di errore tra quelli definiti in `responseCode.h`. L'idea è di segregare il funzionamento interno di `errno` ed esporre un'interfaccia uniforme e indipendente dall'architettura sottostante (si pensi a un client non sviluppato in C che non comprenderebbe gli errori in `errno` eventualmente presenti in una risposta del server). Dal punto di vista del client, una risposta dal server contenente un codice di errore causa un ritorno dalla funzione con `errno` settato a `EBADE`. In particolare, il client ignora questo tipo di errore nella maggior parte dei casi – salvo stampare un messaggio di errore se le stampe sono abilitate – e prosegue con la richiesta successiva. Questo permette al client di distinguere errori endogeni (che *devono* essere gestiti, eventualmente anche uscendo dal programma) da errori esogeni, che non pregiudicano in alcun modo il flusso di esecuzione. L'unico caso in cui `EBADE` ha un effetto è quello in cui le richieste vengono eseguite in "transazioni": per esempio, il comando `-W` corrisponde a una open, write, close. Se la open fallisce e l'errore che il client registra è `EBADE`, le successive write e close vengono e il client prosegue alla prossima richiesta da inviare.