UNIVERSITÀ DI PISA

Università di Pisa -- Dipartimento di Informatica
Corso di Laurea in Informatica

Progetto di Laboratorio di Sistemi Operativi

Francesco Amodeo
Matricola: 560628

Git repository: https://github.com/fram112/file-storage-server

1. Introduzione

Il progetto consiste nell'implementazione di un'applicazione **Client-Server**, che realizza un *file storage server,* in cui la memorizzazione dei file avviene in **memoria principale**. Lo storage ha una capacità fissata, definita nel file di configurazione passato al momento dell'avvio del server. Ogni client, attraverso un protocollo richiesta-risposta, può inviare più richieste al server, in modo da ottenere o modificare lo stato dello storage. Lo storage implementa un meccanismo di espulsione dei file quando vengono superati i limiti di numero di file e di quantità di byte memorizzabili.

2. Istruzioni di avvio

Nella working directory del progetto è fornito un *Makefile* che permette la compilazione e l'aggiornamento automatico dei target. Per ottenere gli eseguibili del client e del server:

```
$ make
```

Per avviare il server: `$ bin/server -f <configfile>`

Per avviare il client: `$ bin/client -a <clientusername> -f <serversocket> [options]`

3. Scelte di progettazione

Di seguito vengono elencate tutte le scelte di progettazione effettuate per l'implementazione fornita.

3.1. Server

Il server è composto da un singolo processo multithreaded, implementato secondo lo schema "**master-worker**", nel quale il thread master/manager si occupa delle connessioni con i client e del dispatching delle richieste verso i thread worker, i quali le gestiscono, accedendo a tutte le operazioni che fornisce lo storage.
Il thread manager effettua il parsing del file di configurazione, nel quale sono indicati il nome del server socket, il percorso del file di log, la capacità massima dello storage in termini di bytes e di numero di file, la politica di rimpiazzamento dei file ed infine il numero di thread worker incaricati della gestione delle richieste.

3.1.1. Multiplexing delle richieste e terminazione

Il manager, utilizzando la `select`, effettua il multiplexing dei descrittori, in modo da gestire le nuove connessioni e le operazioni di I/O, selezionando i descrittori che sono pronti per una operazione di lettura. Quando arriva una nuova connessione, il server gli assegna un nuovo descrittore e lo inserisce nell'insieme dei file descriptor su cui lavora la `select`. Nel momento in cui il descrittore diventa pronto per un'operazione di lettura, viene assegnato ad un thread worker (se vi sono disponibili), il quale **gestirà interamente** la richiesta, leggendo il messaggio inviato dal client, processando la richiesta ed inviando la risposta.
Il descrittore assegnato al worker viene quindi rimosso dall'insieme dei descrittori della `select`, e verrà aggiunto nuovamente, solamente quando il thread worker **comunicherà** di aver processato correttamente la richiesta. Questo tipo di comunicazione tra thread manager e i thread worker, avviene per mezzo di una `pipe`, nella quale ogni worker scrive il descrittore del client appena servito, con **un segno positivo** se la richiesta è stata processata correttamente, mentre con **un segno negativo** se si è verificato un errore oppure se il client si è disconnesso in modo inatteso. Lo stesso tipo di comunicazione viene utilizzato tra il manager ed un thread che gestisce la ricezione dei segnali, il quale scrive in un'altra `pipe` per notificare al server che lo stato delle variabili `shutdown_` e `shutdown_now` è stato modificato e pertanto deve iniziare la procedura di terminazione. La prima variabile viene settata alla ricezione del segnale `SIGHUP`, e corrisponde al caso di terminazione in cui, non vengono più accettate nuove richieste da parte di nuovi client, ma vengono servite tutte le richieste dei client connessi al momento della ricezione del segnale; quindi, il server terminerà solo quando tutti i client connessi chiuderanno la connessione. La seconda invece, viene settata alla ricezione dei segnali `SIGINT` o `SIGQUIT`, e corrisponde al caso in cui il server termina il prima possibile, ossia non accetta più nuove richieste da parte dei client connessi, chiudendo tutte le connessioni attive.

3.1.2. Strutture dati utilizzate dal server

- `theadpool_t`: implementazione di terze parti di un thread pool fornita durante il corso, il cui codice è incluso nel progetto. In particolare, è stata utilizzata per definire lo schema master-worker, spiegato in

precedenza, dove i worker fanno parte di un **pool di thread** che si alternano nella gestione delle richieste, prelevandole da una coda di task pendenti. Fornisce operazioni per aggiungere un task in coda, e per la creazione e distruzione della struttura.

- `icl_hash_t`: implementazione di terze parti di una hash table fornita durante il corso, il cui codice è incluso nel progetto. Questa struttura dati è stata utilizzata come **lo storage** che effettivamente andrà a contenere i file. Oltre alle operazioni per la creazione e cancellazione della struttura, fornisce operazioni per l'inserimento, la cancellazione e la ricerca di elementi nella stessa.
- `list_t`: implementazione di una lista. Utilizzata in **tre diverse istanze**, nello storage, per tenere traccia dei file salvati e del relativo ordine di inserimento, nei campi di un file per tenere traccia dei client che hanno aperto il file ed infine per mantenere le richieste di lock in attesa che la lock su quel file venga rilasciata (vedere al paragrafo successivo). Fornisce operazioni per l'aggiunta e la rimozione di elementi in testa ed in coda, per il recupero dei riferimenti agli elementi in testa ed in coda, per la ricerca del riferimento o rimozione di un elemento specifico, oltre alle operazioni per la creazione e distruzione della struttura.

3.1.3. Mutua esclusione sullo storage e sui file

Lo storage è definito nel file `storage.c` e le operazioni che fornisce, vengono eseguite dai thread worker che devono processare le richieste dei client. Quando necessario, in base al tipo di operazione, i thread accedono allo storage in mutua esclusione grazie alle **read-write lock** `pthread_rwlock_t`. In particolare, l'accesso allo storage è protetto dalla lock `mutex` contenuta nel tipo `storage_t`, la quale permette l'accesso concorrente in lettura a tutti i thread che utilizzano operazioni che non modificano lo stato (ad es. apertura, lettura, lock e unlock di file) e l'accesso esclusivo in scrittura, a quei thread che utilizzano operazioni che invece, modificano lo stato dello storage (ad es. modificando il numero di file o la capacità, scrivendo o rimuovendo file). Lo stesso discorso vale per l'accesso ai file memorizzati nello storage; infatti, anche quest'ultimi sono protetti da una read-write lock contenuta nel tipo `file_t`.

3.1.4. Operazioni e vincoli sui file

Un file è identificato univocamente dal suo **path assoluto** ed i client che richiedono operazioni su di esso sono identificati da una stringa che rappresenta l'**username**. Se l'utente richiede operazioni accedono o manipolano il contenuto del file, come la lettura o la scrittura, è necessario prima aver aperto il file. Altre operazioni invece, come la lock, unlock e la cancellazione di file non richiedono che il file sia stato aperto prima di essere richieste. Degna di nota, è l'operazione che permette di leggere un numero qualsiasi file memorizzati, la quale anch'essa **non richiede** che tutti i file da leggere siano stati aperti in precedenza.

Come già accennato, un client può acquisire la lock su un file, diventando l'unico che lo può leggere e scrivere. Se su un file è già stata acquisita la lock, le successive richieste di lock da client diversi da quello che la detiene, **non vengono completate** fin quando non viene rilasciata. Per questo motivo il server mantiene le richieste incomplete in una lista, in modo da recuperarle quando la lock viene rilasciata. Il thread worker che ha processato la richiesta di lock che non si è potuta completare, inserisce la richiesta nella lista e **si libera** del task, lasciandolo al thread che processerà con successo la richiesta di unlock sullo stesso file e completerà conseguentemente la richiesta di lock incompleta. Infine, questo thread scriverà sulla `pipe` dei descrittori, in modo da **comunicare** al manager che la richiesta è stata completata ed il descrittore può essere aggiunto nuovamente all'insieme della select.
Il caso appena descritto riguarda la richiesta di un'operazione di lock esplicita attraverso la relativa chiamata all'API, ma un client può acquisire la lock su un file, anche per mezzo dei **flag** passati durante una richiesta di apertura di un file. In particolare, può essere utilizzato il flag `O_LOCK`, ma a differenza del caso precedente, se la lock è già stata acquisita da un altro client, l'operazione **fallisce immediatamente**, senza bloccarsi nell'attesa di un rilascio.

3.1.5. Politica di rimpiazzamento file

Quando a causa di una richiesta di scrittura di un file vengono superati i limiti di numero di file e byte totali memorizzati nello storage, viene avviato un algoritmo di rimpiazzamento dei file, che utilizza una politica FIFO. L'algoritmo inizia ad espellere in ordine, partendo dalla testa della lista che tiene traccia dei file contenuti nello storage, fintantoché lo spazio liberato o il numero di file memorizzati, permettono di accogliere il file della richiesta. Tutti i file espulsi **verranno spediti** al client, la cui richiesta ha provocato l'espulsione.
È importante notare, che un file **contribuisce** al totale del numero di file e alla capacità occupata, solamente dopo essere stato **effettivamente scritto**, pertanto, un file vuoto, creato con l'operazione di apertura con flag

`O_CREATE`, non verrà considerato ai fini del controllo dei limiti che effettua l'algoritmo di rimpiazzamento, perché vengono considerati solo i file nella lista `filenames_queue`, i quali vengono aggiunti solamente dopo essere stati scritti. Inoltre, se sono presenti dei file sui quali è stata acquisita la lock, questi **vengono comunque espulsi** se necessario, al fine di evitare il caso in cui un client riempia lo storage di file locked che non verrebbero mai espulsi, impedendo ad altri client l'inserimento di nuovi file.

3.1.6. Logging delle operazioni e statistiche

Durante l'esecuzione, i thread worker ed il thread manager effettuano, in mutua esclusione, il logging, sul file di log specificato nel file di configurazione, di tutte **le operazioni** che vengono richieste dai client o di **gestione interna** del server (ad esempio, l'arrivo di una nuova connessione, il nome del file letto e l'ammontare dei byte restituiti al client, la quantità di dati scritti, se è stata richiesta una operazione di lock su un file, se è partito l'algoritmo di rimpiazzamento dei file della cache e quale vittima è stata selezionata, etc.). Prima della terminazione il server stampa sullo standard output le statistiche generali dello storage ed effettua il logging del numero massimo di file raggiunto e della capacità (in termini di bytes) massima raggiunta. Il file di log successivamente viene utilizzato dallo script `statistiche.sh` che lo parsa e stampa un sunto dettagliato di tutte le statistiche dello storage.

3.2. Protocollo comunicazione Client-Server

Nell'header `protocol.h` viene definito il formato del messaggio che viene scambiato tra il client ed il server e le funzioni che permettono di costruire, inviare e ricevere un messaggio. Inoltre, vengono definiti i codici dell'operazione a cui si riferisce il messaggio e i flag dell'operazione di apertura di un file. Un messaggio è composto da due parti, un campo header e uno data. L'header contiene sempre il codice dell'operazione/esito operazione, l'username del client che ha fatto la richiesta, il path del file relativo alla richiesta. Il campo che indica la dimensione del campo data del messaggio potrebbe essere uguale a 0 quando non ci sono dati inclusi nel messaggio ed il campo `arg` potrebbe essere uguale a -1 se il messaggio non necessita di argomenti aggiuntivi. Lo schema di comunicazione segue il protocollo **richiesta-risposta**; pertanto, quando il client invia la richiesta al server, si blocca e si mette in attesa di ricevere la risposta.

3.3. Client

Il client è un processo single thread costruito per inviare richieste per creare/rimuovere/aprire/scrivere/... file nel/dal file storage server esclusivamente attraverso l'API descritta nella sezione successiva. Il programma client invia richieste al server sulla base degli argomenti specificati sulla riga di comando, dei quali si può ottenere una descrizione utilizzando il comando `-h`.

3.3.1. File Storage API

L'Interfaccia per interagire con il file server (API), implementata nel file `filestorage.c`, definisce tutte le richieste che possono essere inviate al server. Gestisce creazione, invio e ricezione dei messaggi per come sono definiti nel protocollo ed infine restituisce l'esito della richiesta, che corrisponderà a 0 in caso di successo, mentre a -1 in caso di fallimento. Se l'API viene utilizzata in modalità *verbose* stampa l'esito formattato sullo standard output. Nel caso di fallimento della richiesta, il server invia nel campo codice del messaggio, un valore appropriato di `errno` che rappresenta il tipo di errore e questo viene utilizzato dall'API per **settare** la variabile `errno`.
Di seguito sono elencati i valori di errno che utilizza il server per comunicare un errore:

- `EINVAL`: argomento non valido
- `EEXIST`: il file esiste già
- `ECANCELED`: operazione annullata
- `ENOENT`: file non trovato
- `EACCES`: il client non è autorizzato ad accedere al file
- `EPERM`: operazione non permessa, ad esempio il client non ha aperto il file
- `EFBIG`: contenuto del file troppo grande
- `EBUSY`: non è possibile completare l'operazione al momento, ad esempio lock file
- `EALREADY`: operazione già richiesta, ad esempio file già aperto

3.3.2. Strutture dati utilizzate dal client

- `queue_t`: implementazione di una coda FIFO. Utilizzata per mettere in coda le richieste ricevute da linea di comando, in modo da inviarle **singolarmente**, chiamando la relativa funzione dell'API. Inoltre, questa struttura viene utilizzata dalla funzione `lsR` (funzione ausiliaria per il comando `-w`), che esplora ricorsivamente una directory data e mette in coda tutti i file trovati. La coda fornisce operazioni per effettuare di `push` in fondo alla coda, `push` in testa (utilizzata per dare precedenza alla richiesta di connessione rispetto alle altre), `pop` dalla testa, oltre a quelle per la creazione e distruzione della struttura.

3.3.3. Interfaccia a linea di comando ed invio delle richieste

Il client fornisce una interfaccia a linea di comando, attraverso cui possono essere inviate al server **più richieste**, concatenando, uno dopo l'altro i comandi disponibili. È obbligatorio usare almeno i due comandi `-a` e `-f`, che servono rispettivamente per indicare, l'**username** che si vuole utilizzare ed il **socket** del server a cui ci si vuole connettere. Gli argomenti a linea di comando del client possono essere ripetuti più volte (ad eccezione di `-f`, `-h`, `-p`). Il client prende la lista dei comandi e la elabora **suddividendola** in una o più richieste da inviare al server tramite le relative chiamate all'API. Pertanto, il client costruisce una coda nella quale si trovano le singole richieste, pronte per essere inviate dalla funzione `sendrequests`, che **intervallerà** l'invio del tempo passato come argomento del comando `-t`, se incluso. Se è stato incluso il comando `-p` verranno stampati in modo formattato il tipo e l'esito delle richieste, file di riferimento e dove è rilevante, i byte letti o scritti. È importante notare che, nel caso in cui la chiamata all'API della funzione `openFile` con i flag `O_CREATE|O_LOCK` (utilizzata per i comandi `-w` e `-W`) dovesse fallire con errno settato al valore `EEXIST`, il client effettuerà una **nuova chiamata** all'API per la stessa richiesta, ma questa volta utilizzando la `appendToFile`, dato che lo storage permette solamente **scritture in modalità append** sui file già esistenti.

4. Test

Nella cartella `tests` del progetto si possono trovare dei file di test e dei file di configurazione, che utilizzati con i relativi script di test, permettono di testare tutte le operazioni del file storage server.

Il primo test permette di avere una panoramica del funzionamento di ogni tipo di richiesta intervallandole di 200 ms. Degno di nota, è il caso in cui un client prova ad acquisire la lock su un file già bloccato da un altro client e pertanto si ferma ad attendere che venga rilasciato. Nel test si può vedere come, non appena il file viene rilasciato, la richiesta di lock incompleta viene subito completata. Il server viene configurato con un thread worker e viene avviato con il programma `valgrind` con l'argomento `–leak-check=full`, in modo da dimostrare **l'assenza di memory leak ed errori**. In questo caso lo storage è configurato con il limite di 10000 file e una capacità di 128 MB. Il server infine viene terminato con il segnale `SIGHUP`, in modo da **completare tutte le richieste** dei client connessi alla ricezione del segnale. Per avviare il test 1:

```
$ make test1
```

Il secondo test ha l'obiettivo di dimostrare il corretto funzionamento dell'algoritmo di rimpiazzamento, in questo caso il server viene configurato con un limite di 10 file, una capacità di 1 MB e un numero di thread worker pari a 4. Un client invia i file contenuti nella directory `tests/test2/send` e successivamente un secondo client prova ad inviare il **file trigger** che azionerà l'algoritmo di rimpiazzamento, il quale effettua l'espulsione del primo file inserito nello storage. Questo file **verrà inviato** al client, la cui richiesta ha determinato l'espulsione, che **lo memorizzerà** nella directory `tests/test2/ejected`.
Il server viene terminato con un segnale `SIGHUP`. Per avviare il test 2:

```
$ make test2
```

Il terzo ed ultimo test ha l'obiettivo di stressare, **per 30 secondi**, il server e lo storage, configurati con un limite di 100 file, una capacità di 32 MB ed un numero di thread pari a 8. Viene lanciato uno script Bash che esegue **ininterrottamente** un numero di processi client, in modo tale che ce ne siano sempre almeno 10 in esecuzione contemporaneamente. Il server viene terminato con un segnale `SIGINT`, cioè **termina immediatamente** senza aspettare di completare tutte le richieste dei client connessi. Il test vuole dimostrare **l'assenza di errori a runtime** e che il sunto delle statistiche produce valori "ragionevoli". Per avviare il test 3:

```
$ make test3
```