UNIVERSITÀ DI PISA

Relazione Progetto

Laboratorio di Sistemi Operartivi A.A. 2020/2021

Orsucci Gianluca

Matricola 565943, Corso B

1. INTRODUZIONE

Il progetto di Sistemi Operativi e Laboratorio dell'A.A. 2020/2021 consiste nella realizzazione di un sistema di *filestorage* gestito da un server multithreaded che comunica con i vari potenziali clienti tramite una API.

2. SERVER

Il server prende per la configurazione un file di testo chiamato "config.txt" in base al tipo di test che andremo a fare, composto in questo modo:

```
File Massimi=<num_of_file>
Dimensione Massima=<num_max_dim>
Numero thread=<num_of_thread>
Nome Socket=<name_of_socket>
File Log=<path_to_log>
```

Da questo file estrarrà i valori che andrà ad inserire nella struttura *config*.

Struttura interna. Il server maschera i segnali di SIGINT, SIGQUIT e SIGHUP e li utilizza per uscire dal ciclo while in cui accetta e chiude connessioni con i client. I primi due terminano il processo appena possibile, mentre il terzo smette di accettare nuovi client, esegue le richieste rimanenti e termina le connessioni già esistenti.
Il server è del tipo master-worker, dove in questo caso *Workers* è un thread che prende dalla coda un client ed esegue le sue richieste chiamando la funzione *execute*.

Thread Worker. Il thread worker estrae il *cfd* del client tramite la chiamata removeNode, tramite una read-write ottiene la richiesta da parte del client e ad eseguirla, come detto precedentemente, è la funzione *execute*. In base alla stringa che viene passata verrà eseguita una determinata operazione che porterà ad una *SYSCALL_WRITE* da parte del client, con il responso della funzione invocata precedentemente. Nel caso in cui la funzione non vada a buon fine verrà inviato al client l'esito negativo con il tipo di errore che è stato commesso.

Memoria Cache. Per scrivere su un file viene eseguita sempre la procedura open-write-close, più in particolare, in base al flag assegnato alla *openFile*, vado ad inserire il file nella cache settando le sue strutture dati ed eventualmente acquisendo anche la lock. Con la *writeFile* vado ad aggiungere il contenuto del file e aggiungo quale client l'ha aperto, infine eseguo la *closeFile* dove vado ad eliminare il cliente che aveva eseguito la *openFile*.
La cache utilizza una politica di rimpiazzamento di tipo FIFO, e ad ogni chiamata di *openFile* vado a vedere se il numero di file presenti in cache permette di aggiungere il file, altrimenti vado ad eliminare il primo file inserito. Lo stesso vale anche quando utilizzo la *writeFile*, dove però guardo il numero di bytes che vado ad inserire, e finche non c'è abbastanza spazio nella memoria per inserire il file, elimino in ordine di arrivo i file presenti su di essa.

Quando vado ad eliminare un file, per far posto ad altre richieste di scrittura, il server invierà al client il file che andrà ad eliminare, sarà poi questo ultimo a scegliere se salvarlo in qualche directory oppure no. Per leggere, scrivere o bloccare un file controllo in ogni funzione che nella struttura del file ci sia "`file->lock_flag`" uguale a -1 oppure sia uguale al *cfd* del client che cerca di utilizzarlo. In caso contrario, nella read e nella write viene generato un errore del tipo *ENOLCK*, mentre nel caso della lock il cliente dovrà attendere che un altro client rilasci la lock.
Al termine dell'esecuzione viene stampato a schermo un piccolo sunto dei file all'interno della cache, la loro size e il loro valore di lock. Inoltre sono presenti anche la memoria massima raggiunta, il numero massimo di file raggiunti e il numero di rimpiazzamenti effettuati.

3. CLIENT

Il client analizza tutti gli argomenti passati da linea di comando e li va ad inserire in una lista, per poi eseguirli in ordine di arrivo, tranne i comandi -h, -p, e -f che va a toglierli prima di entrare nel ciclo. Come detto precedentemente, ogni file viene aperto, viene eseguita l'operazione, e poi viene chiuso. Controllo che i valori di ritorno di tutte e tre le operazioni siano positivi, altrimenti, anche se solo un esito è negativo, l'esito globale risulta negativo.
Anche l'opzione -D può essere specificata una sola volta, e rappresenta la directory in cui il client andrà a salvare i file che sono stati rimpiazzati nella memoria cache del server.

4. INTERFACCIA

L'API è una libreria utilizzata dal client, con la quale esegue le richieste passate da linea di comando. L'interfaccia comunica sul canale socket con il server ed ogni volta che manda qualcosa al server stesso, controlla che il valore che gli viene poi

mandato sia un valore corretto, altrimenti genera un errore che comporta l'esito negativo dell'operazione.

5. MAKEFILE

Viene messo a disposizione un makefile che fornisce i target, i quali svolgono diverse operazioni: "all", "clean" e "cleanall". Il primo per generare gli eseguibili, il secondo ed il terzo per pulire le directory. Vengono messi a disposizione anche i target "test1", "test2" e "test3" che possono essere lanciati ed ognuno di loro svolge differenti operazioni.
Alla fine della chiamata ognuno dei test viene generato un sunto delle statistiche ricavato tramite il file di log: vengono quindi stampate il numero di read, write, lock e unlock, oltre al numero di bytes scritti e letti durante l'utilizzo.

6. FILE DI LOG

Al suo interno vengono specificate tutte le operazioni effettuate durante l'esecuzione del test, compreso l'esito dell'operazione, su quale path ha lavorato, ed eventualmente quanti bytes sono stati letti oppure scritti.
Vengono anche specificati il numero di richieste eseguite da ciascun server, il massimo numero di connessioni contemporanee, la dimensione massima della memoria e il numero massimo di file che sono stati presenti.