UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea in Informatica

Relazione Progetto Dropbox

Sistemi Operativi e Laboratorio

Prof. Maurizio Angelo Bonuccelli
Prof. Alessio Conte

Carmine Vitiello
578070- Corso A

Anno Accademico 2020/2021

Introduzione

Il Progetto prevedeva la creazione di un **file storage server** nel quale si potesse memorizzare vari file in una **memoria volatile**. Le impostazioni del server verranno recuperate in automatico al momento dell'avvio dell'eseguibile **./server**, che verrà affiancato dalla collocazione del file config.txt.
Il server dovrà gestire le connessioni con **vari client**, che potranno avere **una solo connessione**, che terminerà quando i **thread Worker** del server riusciranno a soddisfare le richieste.
A fine di ciò il client si disconneterà e il server attenderà altre connessioni oppure soddisferà altri client.

Dettagli implementativi

Il progetto richiedeva **varie strutture dati** e l'ausilio di **thread**, visto che la specifica descriveva un server con più thread per accellerare lo scambio di messaggi tra i client e il server.
Per gestire la mole delle richieste e lo scambio di file da disco a memoria principale, ho optato per una struttura dati chiamata **Queue** che attua una politica **FIFO**, anche se in alcuni casi specifici questa politica non viene rispettata. Oltre a questa tipologia di struttura ne ho utilizzate altre per ovviare ad altri problemi che ora discuteremo.

1 Tipo Queue

Il tipo di dato **Queue** rappresenta una **coda generica** nella quale andiamo a descrivere l'inizio e la fine della coda attraverso due puntatori di tipo **Node**. Infine vado a tenere traccia pure del numero di elementi appartenenti alla coda per verificare quando quest'ultima sia vuota.

2 Tipo Node

Il tipo **Node** rappresenta un elemento generico della coda **Queue**, dal momento che incamera un campo **void*** per salvare una qualsiasi tipologia di informazioni. Infatti, grazie a questo stratagemma ho evitato di definire altri elementi di coda inutili con campi di tipo differente. Oltre al campo data salvo pure il puntatore next obbligatorio per gestire la coda Queue.

3 Tipo Statistiche

Il tipo **Statistiche** rappresenta una struttura dati per tenere traccia di alcune informazioni sull'andamento del server. Infatti, controllo il numero Massimo di **file salvati** su server, il valor Massimo di **byte** che sono stati **occupati** e il numero di volte che il server ha dovuto **espellere** un serie di file per inserirne uno nuovo.

4 Tipo Nodo Comando

Il tipo Nodo **Comando** è una struttura dati dotata di 3 campi di cui fanno parte un carattere, chiamato cmd, per indicare il **flag** di un comando ricevuto dalle richieste dei client, poi una stringa chiamata **name** per indicare il **parametro**/i del comando e infine un **intero**, che andrà ad indicare il parametro numerico se fosse necessario, invece se questo non lo fosse sarà inizializzato a **zero**.

5 Tipo Comando Client

Il tipo **Comando Client** viene utilizzato al lato server per inserire le richieste dei client in una coda in maniera

semplificata, visto che sono state già analizzate dal parser al lato client. Infatti, la struttura Comando Client ha una campo char per identificare il flag del comando, una stringa per indicare il parametro del comando e infine un long per indicare l'id del client che ha fatto la richiesta.
Rispetto alla **struttura Nodo comando**, che viene utilizzata prettamente nei client per incampsulare le richiestesemplificate dei client attraverso il parser, il Comando Client serve prettamente lato server per gestire la coda di attesa dei comandi, che verranno eseguiti in concreto dai thread Worker.

6 Tipo Msg

Il tipo **Msg** è una struttura di **supporto** per la **lettura** dei socket dai client. Infatti, viene utilizzata per evitare di allocare un struttura più complessa, come Comando Client, nel caso in cui il socket sia vuoto oppure di errore durante la lettura.

7 Tipo File Ram

Il tipo **File Ram** è una struttura dati che rappresenta un file nella **memoria RAM del server**. Infatti, contiene:

- Una stringa nome per rappresentare il **nome** del file
- Una stringa buffer per immagazzinare il **contenuto** del file
- Un numero Long per rappresentare la **lunghezza** del file
- Un **Flag**, rappresentato da un intero, per indicare se il file fosse **aperto** e da quale client
- Un **mutex** per evitare che il file venga modificato da più thread in contemporanea

Questa tipologia di informazione verrà messa in una coda per rappresentare quali file contiene il server rispettando la politica della FIFO.

8 Operazioni della Queue

Le operazioni descritte in seguito servono per gestire tutte le code del Progetto. Infatti, grazie ad esse la gestione delle code è eseguita con una metodologia corretta senza lasciare puntatori e varibili allo sbaraglio.
Le seguenti operazioni comprendono, come viene modificata la coda, che tipo di coda va a modificare e in che ambito vengono utilizzate.

- Queue* initQueue() -> va a inizializzare una coda q
- Void* pop(Queue** q) -> rilascia l'elemento Node che è in testa alla coda q
- Void* pop2(Queue** q) -> rilascia il secondo elemento Node che è nella coda q, perchè quando andiamo a eliminare delle vittime, dalla coda dei file del server, non vogliamo eleminare il file che stiamo inserendo se quest'ultimo fosse il primo della coda.
- Void printQueueNodoComando(Queue* q) -> Stampa la coda con Nodo Comando come data, però non è utilizzata, perchè veniva utilizzata prettamente per debug.
- Void printQueueFiles(Queue* q) -> stampa la code dei files contenuti nel server. Viene impiegata alla fine dell'esecuzione del server e durante alcune operazioni di rimozione file.
- Int push(Queue** q, void* el) -> inserisce in coda alla struttura Queue l'elemento el
- Int pushTesta(Queue** q, void* el) -> inserisce in testa alla struttura Queue l'elemento el e non rispetta la politica FIFO, perchè viene utilizzata per un comando specifico, cioè il w. Dal momento che, bisogna fare tutte le scritture subito dopo aver recepito il comando e non rimandarle in fondo,

perchè potrebbero esserci delle operazioni di lettura o rimozione sui file inseriti.

- Int insert(Queue** q, char cmd, char* name, int n) -> crea un NodoComando per incapsulare la richiesta del client, che è stata analizzata dal parser, per poi inserirla nella coda designata.
- Int removeFromQueue(Queue** q, Node* toDelete) -> elimina un nodo dalla coda
- Node* fileExistsServer(Queue* q, char* nomefile) -> verifica se un file è presente nella coda dei file e in caso affermativo lo restituisce, invece in caso negative restituisce NULL.
- Void* returnFirstEl(Queue* q) -> ritorno il puntatore al primo elemento della coda per evitare di ritornare interamente il nodo per dei controlli superficiale, come per esempio controllare se il primo elemento è lo stesso elemento che sto provando ad inserire ed eliminare in contemporanea. Infatti, quando l'elemento ha una size minore della capienza totale del server, ma momentanemente non c'è abbastanza spazio, allora provo a fare spazio espellendo dei file, però evitando di eliminare il file su cui vorrei scrivere.

9 Gestione dei segnali

I segnali vengono gestiti, come gia citato da un **thread specifico**, che andrà a catturare **i segnali mascherati** SIGINT, SIGQUIT e SIGHUP. I primi due segnali hanno un comportamento similare, visto che il server deve terminare il prima possibile, invece nel caso del segnale **SIGHUP** bisogna interrompere **l'accettazione** delle nuove connessioni e terminare il prima possibile quelle in esecuzione. Infatti, quando si riceverà il segnale SIGHUP, il server dovrà fare una **clear** del bit nella set che era dedicato al **listener**, così in modo tale da evitare l'accettazione di nuove connessioni. Dopo di che si aspetterà che il numero delle connessioni attive scenda a 0, in modo tale da chiudere il processo in sicurezza con tutte le accortezze del caso.
Per gli altri due segnali si ha una metodologia meno tollerante, dal momento che appena tutte le richieste in corso terminano, dovrà terminare anche il server pure se la coda dei comani da eseguire è ancora piena.

10 Server

Il server prende in input da riga di comando il file config attraveso l'insermento del percorso assoluto del file impostazioni. Nelle impostazioni sono compresi vari campi tra i quali:

- La **campienza** massima del server
- Il numero **Massimo** di file che si possono memorizzare in contemporanea
- Il nome del **socket**
- Il **numero di thread Worker** che possono essere disposti all'esecuzione

Il server dopo aver ottenuto le impostazioni di base inizia la creazione delle statistiche con il tipo dato **Statistiche**, poi inizializza il thread **tGestoreSegnali**, che andrà a gestire i segnali **SIGINT**, **SIGQUIT** e **SIGHUP**, visto che senza un thread prescelto e una adeguata Maschera, i segnali potrebbero essere catturati da qualsiasi thread e non è una cosa corretta nella situazione attuale. Dopo di che, si andrà ad inizializzare i thread worker con le loro pipe associate e tutte le strutture dati di supporto come le code. Infine, andremo ad adibire le set e la select per gestire le connessioni ai vari client. Inoltre, per tenere traccia delle connessioni attive andremo ad incrementare un semplice contatore chiamato NumConnessioniAttive, così in modo tale da capire, quando terminare il server in modo completo e sicuro. Durante le varie richieste il server leggerà i vari socket, per inserire le richieste in una coda, che verranno soddifatte dai Thread Worker.

11 Thread Worker

I **Thread Worker** eseguono come gia detto le richieste dei client per conto del server, fino a quando non si riceve un **segnale**. Dopo aver fatto la pop dalla coda dei comani il thread gestisce la richiesta attraverso un swicht, che a seconda del **flag** del comando, rappresentato da una lettera come per esempio: 'W', si comporterà in maniera adeguta alla richiesta del client. Infatti, ho ricreato un gestione dei comandi dove il thread possa gestire i file nel seguente modo:

1. aprire/creare il file
2. scriverlo/leggerlo/eleminarlo
3. chiudere il file se non è stato eliminato

Questa gestione da la possibilità ad un solo client di modificare un determinato file alla volta, dal momento che fino a quando il medesimo file non è stato chiudo, dal client che l'ha aperto, non potra esser alterato. Infatti, grazie ad un flag chiamato **is_opened** possiamo verificare se il file è aperto e da quale client attraverso l'Id, che andiamo a memorizzare nella stessa variabile. Invece, in caso che il file fosse chiuso troveremmo il valore a -1. Inoltre, attraverso la funzione openFile indicata dal flag 'o' possiamo gestire anche la creazione dei file con il flag **O_CREATE**, così in modo tale da spezzare la scrittura di un file in 3 fasi, come scritte in precedenza, visto che andremo a creare un File Ram vuoto nel server, poi modificheremo il contenuto attraverso delle scritture in append e infine lo chiuderemo.

12 Client

Il client inizia chiamando il parser che analizza l'input da shell per poi collegarsi al server ed iniziare a mandare le richieste semplificate al server. In seguito appoggiandosi alle **API** si riesce a passare o ricevere le varie **informazioni** necessarie all'adempimento delle richieste come **nome** file, **size** del nome o del **contenuto** oppure il contenuto stesso.

13 Miscellanea

In più rispetto al client e al server ci sono altri file di supporto, come l'util.c/h che vanno ad assicurare un controllo sicuro di alcuni paramentri, come i numeri che vengono passati per stringa e bisogna accertarsi che siano realmente numeri. Inoltre, tutte le scritture(write) o letture(read) hanno una gestione completa attraverso le funzioni writen e readn, e il controllo su tutte le "System Call", attraverso alcune macro, che rendono più veloce ed immediata la gestione dei dati e il debug in caso di errori.

Organizzazione del Progetto e GitHub

Il Progetto ha una cartella sorgente (src) per dividere la parte di codice da quella testuale, poi all'interno della src si trovano vari cartelle:

- FileTest: contiene dei file di test "vuoti", ma con una varie capienze per provare l'algoritmo di rimpiazzamento che sono stati creati con il seguente comando: `dd if=/dev/zero of=file.txt count=1024 bs=1024`
- Test: contiene i file TestScript
- Configs: contiene I file config.txt
- Includes: indica i file .h

Per avviare il Progetto bisogna utilizzare i vari comandi quandi si è nella cartella src:

- “make all” per compilare il progetto
- “make test1” per avviare il primo test
- “make test2” per avviare il secondo test.

Alla fine dell’esecuzione di un test si troverranno i risultati dei client nella cartella Output e poi nelle rispettive cartelle a seconda del numero del test.
Durante lo sviluppo del Progetto ho utilizzato una repository pubblica su GitHub con il seguente link: https://github.com/itakartor/ProgettoSol2021.git