**Relazione Progetto Sistemi Operativi**

***Amelia Sava MAT:602208 Corso A 2021/2022***

**Istruzioni per la compilazione**

(cartelle necessarie per test e funzionamanto)
mkdir libs
mkdir savedfiles
mkdir expelled

chmod 777 ./scripts/*.sh
make cleanall
make all

In alternativa si può compilare con:
make test1
make test2
make test3

## *1. Introduzione e struttura dei file*

Il progetto implementa un file storage server, si hanno due file principali “server.c” e “client.c”, che implementano rispettivamente server e client, entrambi fanno appoggio su “ops.h” e “conn.h”, due file che riprendono funzioni viste, implementate o fornite dal docente durante il corso.

In particolare “ops.h” riprende “util.h” utilizzato a lezione, aggiungendovi una enum per rendere più leggibili le operazioni richieste dal server, una funzione per allocare memoria in maniera sicura e una funzione che stampa in maniera leggibile le operazioni definite dalla enum.

Mentre il file “conn.h”, oltre alle funzioni viste a lezione, implementa una struttura lista che contiene dei nodi messaggio, i quali incapsulano le informazioni necessarie per la comunicazione tra client e server.

Il file “filelist.h” invece definisce un’altra lista che contiene operazioni di gestione di nodi file, la quale viene anche utilizzata dalla libreria HashLFU come appoggio alle sue funzionalità principali.

Vengono anche definite due librerie “coms” e “HashLFU”, la prima implementa le API che permettono la comunicazione tra client e server, la seconda è una tabella hash che rappresenta la memoria cache del server.

## *2. Server*

### *2.1 File di Configurazione*

Nel main si fa il parsing del file config.txt, il formato del file config.txt prevede, oltre alla possibilità di commentare con #, di scrivere gli argomenti uno per riga, supponendo che l'ordine venga rispettato.

Le righe devono essere terminate da ; e il nome della variabile separato dall'uguale, possono esserci spazi.

### *2.2 Memoria Cache*

Per rappresentare la memoria cache si è utilizzata una struttura dati hash, con politica di rimozione LFU.

La rimozione LFU viene implementata incrementando la frequenza dei nodi ogni volta che vengono utilizzati: per ridurre i tempi di ricerca quando si deve rimuovere un nodo in seguito a una capacity miss, il nodo utilizzato viene spostato in cima alla propria lista di trabocco. In questo modo si ha la certezza che i nodi meno utilizzati saranno sempre gli ultimi e per trovare il minimo basta quindi controllare la fine di ogni lista della tabella ed eliminare il minimo, al posto di fare una ricerca esaustiva di ogni singolo file salvato in memoria.

### *2.3 Funzioni lato server*

Lato server esistono funzioni che ricevono i dati dalle API e completano l’operazione richiesta sulla memoria cache, restituendo vari risultati, i quali verranno poi inviati al client.

### *2.4 Threads*

Il thread main parte inizializzando tutte le risorse necessarie, parsando il file di configurazione e creando il file di log. Vengono anche fatti partire n worker thread per la gestione delle richieste del client e un thread signal handler per la gestione della ricezione dei segnali.

Dopo l'inizializzazione delle risorse il main utilizza una select per accettare connessioni e gestire le operazioni di lettura e scrittura.

Il server fa anche uso di una lista di richieste. Ricevuta una connessione il client invia la richiesta che il server aggiunge alla lista, viene dunque segnalato ai worker, che sono in attesa sulla variabile di condizione della lista, che è stato inserito un elemento. Il worker, se non è stato ricevuto un segnale, esegue le operazioni richieste dal client. Una volta processata l'operazione, il worker, a seconda della necessità, reinserisce il client nel set perché possa inviare altre richieste, altrimenti gestisce la chiusura della connessione.

Quando viene ricevuto un segnale il main termina deallocando tutte le risorse.

### *2.5 Segnali*

Ci sono implementazioni riguardati tre tipi di segnali, SIGINT, SIGQUIT e SIGHUP, che portano a due possibili interruzioni del server, definite fast stop e slow stop.

Nel caso di un'interruzione più brusca, fast stop, il worker interrompe il suo ciclo ed esce immediatamente, altrimenti, nello slow stop, continua a prelevare richieste dalla coda finche' ce ne sono poi si interrompe.

A gestire le interruzioni si ha un thread dedicato pronto a ricever i segnali e ad applicare le necessarie modifiche alle variabili globali per interrompere le altre operazioni in maniera adeguata.

## 3. Client

Il client opera tramite richieste da riga di comando. Gli argomenti da linea di comando, ad esclusione di quelli a effetto immediato, come "-h", "-p" e "-f", sono inseriti in coda a una lista per controllare che vengano usati correttamente. Cioè che rispettino le dipendenze stabilite tra loro e mantengano l'ordine in cui sono stati richiesti dall'utente.

Ogni operazione, dopo essere stata controllata, va poi a chiamare la rispettiva API per comunicare con il server.

Quando vengono passati i file si suppone che venga utilizzato il path assoluto del file per identificarli. Però, anche per comodità di esecuzione e testing, si prevedono casi in cui l'utente stia lavorando nella directory corrente, si è quindi implementato un metodo per sostituire "./" con il path assoluto della directory. Quindi l'utente su riga di comando può usare "./".

Si suppone che il socket sia sempre nella directory corrente.

Le funzioni isdot, cwd e write_from_dir_find sono basate su funzioni viste a lezione e servono come supporto alle operazioni del client.

Di particolare nota è la write_from_dir_find , che scorre un'intera cartella e, ricorsivamente, tutte le sue sub directory, leggendo tutti o un dato numero di file. Utilizzata per implementare l'opzione -w, quando trova un file lo scrive nel server tramite API e decrementa il numero di file da scrivere, nel caso fallisca, la funzione passa al prossimo file per completare la quantità richiesta dall'utente.

## 4. API

In aggiunta alla specifica richiesta nella traccia del progetto per la libreria di API, si è scelto di aggiungere alla funzione OpenFile un risultato di ritorno aggiuntivo 1. In questo modo si implementa il controllo sulla Writefile successiva. Si ha quindi uno stato di "ready for write" in modo che si possa determinare che l'operazione precedente ad una write sia stata una open file con il flag O_CREATE settato ed andata a buon fine.