Relazione progetto SOL A.A. 2020/21
Leonardo Alfreducci – matr. 598010

Introduzione
La specifica richiede la realizzazione di un file storage server, ovvero la realizzazione di un server che memorizza in memoria principale file che vengono inviati dai client. Il server, attraverso il parsing di un file di configurazione che gli viene passato come argomento, può contenere un numero massimo di file al suo interno ed ha una capacità limitata: qualora un client cercasse di mandare dei file che vanno oltre alla capacità effettiva del server, quest'ultimo si comporta come una cache di file implementata con una coda FIFO: il primo elemento aggiunto sarà il primo ad essere rimosso, così da far spazio al file che deve essere aggiunto.

Tipo di dato *Queue*
Come già spiegato nell'introduzione, essendo il server implementato come una cache di file, ho deciso di utilizzare una coda FIFO. Il tipo di dato *Queue* contiene all'interno diversi campi:
- La testa della coda,
- L'ultimo elemento della coda,
- La lunghezza della coda, che viene incrementata e decrementata con operazioni di inserimento e rimozione

La testa e la coda della lista sono di tipo *Node*, descritto nel seguito.

Tipo di dato *Node*
Un *Node* è un nodo generico della lista. Contiene al suo interno l'elemento successivo nella lista e il dato, che ho scelto di implementare come *void**. Ho fatto questa scelta così che la lista sia il più generica possibile e possa essere utilizzata per diversi scopi. Infatti la coda FIFO, oltre ad essere utilizzata come già detto per memorizzare i file all'interno del server, è anche utilizzata in altri momenti:
- Nel server, i comandi che arrivano dai client vengono inseriti in una coda, e appena un thread worker è libero lo gestisce;
- Nel client, i comandi parsati dal *Parser* vengono inseriti in una coda, e vengono uno ad uno mandati al client, rispettando il protocollo richiesta-risposta.

Tipo di dato (e gestione) *Statistiche*
Questo tipo di dato contiene diverse variabili:
- Numero totale di file memorizzati che il server ha raggiunto;
- Lo spazio occupato al massimo dal server;
- Numero di volte che l'algoritmo di rimpiazzamento è stato eseguito.

Queste statistiche vengono aggiornate quando viene fatta una scrittura o una cancellazione di un file sul server.

Operazioni per la coda
Ho implementato alcune fra le più comuni operazioni che vengono normalmente svolte in una FIFO che erano utili ai miei scopi:
- *initQueue()*: svolge le operazioni per inizializzare una coda vuota;
- *push(Queue **q, void* el)*: inserisce l'elemento *el* come ultimo elemento nella coda;
- *pushTesta(Queue **q, void* el)*: inserisce l'elemento *el* in testa alla coda;
- *pop(Queue **q)*: ritorna e rimuove il primo elemento dalla coda;
- *pop2(Queue **q)*: simile alla *pop*, ritorna e rimuove il secondo elemento dalla coda (la scelta di implementare *pop2* sarà chiara quando spiegherò come ho trattato la rimozione di un file dal server perché pieno);
- *returnFirstEl(Queue *q)*: come è intuibile, ritorna (questa volta senza rimozione) il primo elemento dalla coda *q*;
- *removeFromQueue(Queue **q, Node* toDelete)*: rimuove il nodo *toDelete* dalla coda.

All'interno di queste operazioni vengono anche gestiti i possibili errori che si possono riscontrare.

Tipo di dato *ComandoClient*

Questo tipo di dato è utilizzato dal server per salvare le richieste che gli arrivano dai client e salvarle in una coda. Quando un thread worker è libero, farà la *pop* dalla coda dei comandi per avere un comando da eseguire.

Tipo di dato *fileRAM*

Un tipo *fileRAM* è un file che viene memorizzato all'interno della memoria principale. Contiene al suo interno informazioni necessarie a questo scopo, come il nome, il buffer, la lunghezza (in byte); ma anche un lock, così che due o più thread nel server non possano eseguire le stesse operazioni contemporaneamente nello stesso file (ad esempio, un thread ci scrive sopra e un altro deve eliminarlo per far posto ad un altro file), e un flag is_*locked*: infatti un client deve aprire un file prima di eseguire una qualsiasi operazione su di esso, settando così il flag al *connfd* del client, e deve chiuderlo quando ha finito, resettando così is_*locked* al valore di default *-1*. Quando il server esegue una qualsiasi operazione su un file richiesta dal client, verifica prima che il client abbia aperto il file (cioè che il suo *connfd* corrisponda a quello della *is_locked* del file). Se il client non ha aperto il file, non può eseguirci nessuna operazione sopra.

Configurazione del server

Il server come primo argomento richiede il file *config.txt* che deve poi parsare. Esempi di struttura di file di configurazione possono essere trovati all'interno della cartella *configs*. Il parser del client, per ogni riga di questo file di configurazione, verifica se la prima parola equivale ai parametri che vengono richiesti, e nel caso prende l'elemento dopo lo spazio come argomento.

Thread worker del server

Ogni thread worker del server ha un ciclo infinito, da cui esce solo se il server incontra un segnale. All'interno di questo ciclo, si mette in attesa che la coda dei comandi da eseguire abbia qualcosa al suo interno (svegliata poi da una *pthread_cond_signal* mandata dal *main*). Quando trova un comando da eseguire, controlla che tipo di comando è attraverso uno switch e si comporta di conseguenza:

- Apertura di un file: sono possibili due flag che vengono passati come prima cosa tra il client e il server: *flag di creazione* (in quel caso il file viene creato e viene settato il flag *is_locked*) oppure *flag di apertura* (viene solo settato il file *is_locked*, se il file non esiste viene restituito un errore). Viene anche eseguito un controllo se il server è pieno (come numero di file), e nel caso viene espulso il primo file all'interno della coda dei file.
  In caso di successo, restituisce al client *0*, in caso di fallimento *-1* (sto cercando di aprire un file già aperto / sto cercando di aprire un file che non esiste) o *-2* (sto cercando di creare ed aprire un file che esiste già).
- Chiusura di un file: controlla se il client che sta cercando di chiudere il file è lo stesso che ha settato il flag *is_locked*.
  In caso di successo, restituisce al client *0* e resetta il flag *is_locked*, in caso di fallimento restituisce *-1*.
- Scrittura di un file: controlla per prima cosa se il file esiste (dovrebbe già essere stato creato dalla *openFile*) e, ottenuta la lunghezza in byte di quel file, controlla che non sia maggiore di tutto lo spazio che il server ha. Non ha infatti senso eliminare tutti i file presenti nel server perché pieno (di spazio) per poi scoprire che il file è comunque troppo grande, quindi viene fatta subito questa verifica. In ultimis, vengono eventualmente eliminati dei file per far spazio al nuovo elemento, attraverso la *pop* o la *pop2*. Ho implementato anche la *pop2* perché se ci fosse una scrittura in append di un file già esistente, ma questo fosse proprio il primo della coda, la write non deve eliminare il primo ma il secondo elemento nella FIFO.
  Una volta superate queste "prove", il client manda al server il buffer che contiene il file e quest'ultimo memorizza il dato scrivendo in append (nel caso il file avesse già un contenuto, ovvero fossero già state fatte delle write su quel file), o da 0 (nel caso il buffer fosse vuoto).
  In caso di successo, restituisce al client *0*, in caso di fallimento restituisce *-1*.
- Cancellazione di un file: se il file esiste e il flag *is_locked* corrisponde al *connfd* del client, il file viene cancellato dalla coda attraverso la funzione *removeFromQueue* già analizzata.
  In caso di successo, restituisce al client *0*, in caso di fallimento restituisce *-1*.

- Lettura di un file: se il file esiste e il flag *is_locked* corrisponde al *connfd* del client, il file viene letto e viene mandata al client prima la lunghezza (in byte) del file, poi il buffer che contiene il dato da leggere.
  In caso di successo, restituisce il file letto al client, in caso di fallimento restituisce *-1*.
- Lettura di 'N' file: il server manda al client il numero di file che può effettivamente leggere (nel caso in cui 'N' sia minore del numero di file effettivamente memorizzati nel server), e in seguito manda uno ad uno i nomi dei file da leggere.
  Restituisce il numero di file che può leggere, con i relativi nomi.

Una volta terminata l'operazione, il thread worker si occupa di reinserire nella *set* il *connfd* del client di cui ha appena eseguito il comando, e manda un segnale al *main* scrivendo nella sua pipe per risvegliare la *select* (i dettagli di questo saranno chiari nel paragrafo del *main* del server).

Main del server

Il *main* del server per prima cosa esegue diverse operazioni preliminari:

- Esegue le operazioni per gestire i segnali;
- Inizializza le statistiche, i thread, il socket.

Entra dunque in un ciclo da cui esce solo se ha ricevuto un segnale *SIGINT*, in cui si blocca fin da subito nella *select* in attesa di nuove connessioni o di nuovi comandi da parte dei client (o di essere svegliato dalle pipe dei thread).
Una volta che la *select* si è "svegliata", viene controllata se la connessione arriva dal socket (è una connessione da un nuovo client) o da una connessione già attiva.

- Nel primo caso, vengono svolte le operazioni di accettazione di una nuova connessione;
- Nel secondo caso, controlla se la richiesta arriva da una pipe (dei segnali, descritta nel paragrafo sui segnali, oppure di un thread, che ha avuto il solo scopo di svegliare la *select*) oppure da un client connesso (nuova operazione da svolgere).
  - Se il segnale arriva da un client, il suo *connfd* viene rimosso dalla *set* della *select* così che quest'ultima non si svegli ogni volta che il server ed il client si scambiano un messaggio. Infine il server riceve il comando dal client con il parametro associato (ad esempio, comando '*r*', parametro '*foto.png'*) e inserisce questo comando nella coda dei comandi, insieme al *connfd* del client che ha richiesto l'operazione;
  - Se il segnale arriva dalla pipe associata ad un thread, l'unico scopo era svegliare la *select* così che all'iterazione successiva possa prendere la *set* aggiornata. Se ci immaginiamo infatti un solo client che ha molti comandi da eseguire, mentre il thread svolge una operazione la *select* nel *main* si blocca sulla *set* vecchia, e non vede quando il thread riaggiunge il *connfd* del client alle richieste che il *main* deve accettare. L'utilizzo delle pipe che interagiscono in questo modo con il client è un modo per ovviare a questo problema.

Quando il *main* riceve un segnale *SIGINT*, esce dal ciclo "infinito" in cui si trovava e stampa le statistiche e libera la memoria che aveva allocato (eventuali comandi rimasti, coda dei file, array, …).

Ricezione dei segnali nel server

Per gestire i segnali, il server crea un thread che si occupi della ricezione. Questo thread, chiamato *tSegnali*, si mette in attesa con una *sigwait*, e una volta ricevuto un segnale setta una variabile globale su ciò che ha visto e scrive su una pipe che è all'interno della *set* della *select*, così da svegliarla. Quando la *select* del *main* viene svegliata, controlla come già specificato se si tratta della pipe dei segnali. In tal caso controlla quale segnale è arrivato e agisce di conseguenza, come specificato sotto.

Gestione dei segnali nel server

I segnali che vengono accettati dal server sono *SIGINT* (e *SIGQUIT*, che si comporta nel medesimo modo) e *SIGHUP*. Nel caso di ricezione di un segnale *SIGINT*/*SIGQUIT*, il server non deve più accettare nuovi comandi, quindi il *main* esce subito dal suo ciclo infinito. Nel caso di ricezione di un segnale SIGHUP il server deve finire di gestire e rispondere ai client già connessi, ma non deve più accettare nuove connessioni da parte di nuovi client. Chiude quindi il socket che accetta le connessioni, e ogni volta che un client si disconnette controlla se ci sono altre connessioni attive. Se non ci sono altre connessioni il server può finalmente terminare, e si comporta come se avesse ricevuto un segnale *SIGINT*/*SIGQUIT*.

API client/server

- *openConnection*: apre una connessione *AF_UNIX* con il server. Prova a connettersi un numero limitato di volte e nell'arco di un numero massimo di tempo.
  Restituisce *0* in caso di successo, *-1* in caso di fallimento. *errno* viene settato opportunamente a seconda dell'errore riscontrato.
- *closeConnection*: chiude la connessione aperta con il server.
  Restituisce *0* in caso di successo, *-1* in caso di fallimento. *errno* viene settato opportunamente a seconda dell'errore riscontrato.
- *writeCMD*: scrive sul socket il comando che il server dovrà eseguire seguito dal parametro.
  Restituisce *0* in caso di successo, *-1* in caso di fallimento. *errno* viene settato opportunamente a seconda dell'errore riscontrato.
- *openFile*: setta il flag *is_locked* di un *fileRAM* del server al *connfd* del client, ed eventualmente crea il file. Può avere come flag *O_CREATE* (crea e setta il flag *is_locked* del file), oppure *O_LOCK* (setta solo il flag *is_locked* del file). Riceve dal client una risposta se tutto è andato come previsto. Se il server ritorna *-2* vuol dire che il server ha provato a creare il file ma esisteva già, quindi l'errore può essere risolto richiamando la *openFile* ma cambiando flag.
  Restituisce *0* in caso di successo, *-1* in caso di fallimento. *errno* viene settato opportunamente a seconda dell'errore riscontrato.
- *closeFile*: resetta il flag *is_locked* di un *fileRAM* del server a *-1*. Per eseguire la *closeFile* è necessario aver prima eseguito una *openFile*.
  Restituisce *0* in caso di successo, *-1* in caso di fallimento. *errno* viene settato opportunamente a seconda dell'errore riscontrato.
- *removeFile*: rimuove un *fileRAM* dal server. Per eseguire questo comando è necessario aver prima eseguito una *openFile*.
  Restituisce *0* in caso di successo, *-1* in caso di fallimento. *errno* viene settato opportunamente a seconda dell'errore riscontrato.
- *readFile*: legge un *fileRAM* dal server. Per eseguire questo comando è necessario aver prima eseguito una *openFile*. All'interno di *buf*, passato come argomento alla funzione, inserisce il buffer del file letto, e all'interno di *size* i bytes di quel file.
  Restituisce *0* in caso di successo, *-1* in caso di fallimento. *errno* viene settato opportunamente a seconda dell'errore riscontrato.
- *readNFiles*: legge '*N*' file dal server. Riceve da questo un intero che corrisponde al numero dei file che può effettivamente leggere (che è minore o uguale ad '*N*'). Successivamente entra in un ciclo in cui ad ogni iterazione riceve il nome di un file, che va ad inserire all'interno di un array. Una volta ricevuti i nomi di tutti i file, per ognuno di essi chiama la funzione *readFile* sopra descritta. Scrive infine il buffer ottenuto da questa funzione sul disco, attraverso la funzione *writeBufToDisk*.
  Restituisce *n* in caso di successo (il numero di file letti), *-1* in caso di fallimento. *errno* viene settato opportunamente a seconda dell'errore riscontrato.
- *writeFile*: Manda al server il nome del file e la sua lunghezza in byte; apre il file in locale e successivamente chiama l'*appendToFile*, che manda effettivamente il buffer del file al server. Prima di eseguire la *writeFile* deve ovviamente essere eseguita la *openFile*. Particolarmente degno di nota è il caso in cui stiamo provando a scrivere (in append) su un file che esiste già all'interno del server. La *openFile* viene in questo caso chiamata due volte: la prima volta con il flag *O_CREATE* (che restituirà *-2*, errore), e la seconda volta con il flag *O_LOCK* (che avrà successo). Verrà quindi mostrato un errore per la *openFile* con il primo flag, ma la seconda volta avrà successo.
  Restituisce il risultato dell'*appendToFile*.
- *appendToFile*: da eseguire dopo aver eseguito la *writeFile*. Richiede al server la scrittura in append di un buffer *buf* di lunghezza *size* byte. Riceve dal server il valore di ritorno della funzione.
  Restituisce *0* in caso di successo, *-1* in caso di fallimento. *errno* viene settato opportunamente a seconda dell'errore riscontrato.

Main, parser e altre funzioni del client

Il *main* del client richiama innanzitutto la funzione *parser*, che inizializza la coda dei comandi. Su questa c'è poco da dire: attraverso una *getopt* va a prendere gli argomenti passati come argomento con cui il client si

dovrà interfacciare col server e li va ad inserire in una lista inizializzata appositamente, che verrà poi restituita al *main*. Rimando comunque ai commenti nel codice per una maggiore comprensione del *parser*.
Passa dunque a creare una maschera per il segnale *SIGPIPE*: questo perché se il server interrompesse inaspettatamente la connessione (ad esempio per un segnale che ha ricevuto, vedi *SIGINT*/*SIGQUIT*/*SIGHUP*) il client si bloccherebbe alla *readn*/*writen* successiva, che escono in automatico restituendo un segnale *SIGPIPE*. Per gestire gli errori di questo tipo con la *SYSCALL_EXIT* che ho definito è stato necessario quindi mascherare questo segnale.
Una volta aperta la connessione con il server, il client entra in un ciclo in cui a mano a mano estrae tutti gli elementi dalla coda dei comandi. Per ognuno di questi comandi viene chiamata la funzione *EseguiComandoClientServer*, che si occupa di chiamare le API sopra descritte e di gestirne i possibili errori.
Per quanto riguarda il comando '*w*' il client deve invece visitare ricorsivamente le directory, deve quindi chiamare la funzione *visitaRicorsiva*, su cui vale la pena spendere qualche parola.
In quest'ultima vengono analizzati tutti i file o directory contenute all'interno della cartella passata come argomento alla funzione. Se ciò che stiamo analizzando è un file e se devo ancora leggere dei file, questo viene aggiunto alla coda degli elementi da scrivere nel server con il comando '*W*'. Se invece è una directory, viene chiamata ricorsivamente la funzione con l'argomento la cartella che stiamo analizzando correntemente. Alla fine quindi questa funzione avrà quindi inserito in testa all'interno della coda comandi '*n*' file con il comando '*W*'.
Una volta che i comandi sono terminati, il client chiude la connessione con il server ed esce, facendo le apposite *free*.

Gestione del progetto & link GitHub

La cartella *src* contiene i file sorgente del codice da me sviluppato. I file di include sono all'interno della cartella *includes*, mentre all'interno della cartella *configs* si trovano i file di configurazione per i test sviluppati. All'interno di *testscripts* si trovano gli script bash che eseguono *test1* e *test2*, mentre all'interno di *upload_tests* dei file ("vuoti") di dimensione variabile creati ad hoc per eseguire i test che venivano richiesti. Questi ultimi file sono utili in particolare per testare l'algoritmo di rimpiazzamento di file all'interno del server che ho implementato.
Per eseguire il codice, trovandosi all'interno della cartella *src*:

- *make cleanall*: pulisce la cartella da tutti i file generati durante i test, dagli eseguibili creati e dal socket;
- *make all*: compila client e server, altrimenti separatamente compilabili lanciando
  - *make server*;
  - *make client*;
- *make test1*: esegue il *test1*;
- *make test2*: esegue il *test2*.

Eseguendo i test, l'output del server sarà mostrato all'interno della shell, mentre quelli dei singoli client si troveranno, una volta terminata l'esecuzione, all'interno delle directory *outputTest1* o *outputTest2*, a seconda del test eseguito (queste cartelle saranno create durante l'esecuzione).

Durante lo sviluppo di questo progetto ho utilizzato una repository pubblica creata appositamente su GitHub di cui lascio il link: https://github.com/Sn0wCooder/SOL-unipi.