UNIVERSITÀ DI PISA

RELAZIONE PROGETTO SOL

==========

STUDENTE: ***LUCA RIZZO***
MATRICOLA: ***598992***

A.A 2020/2021

INTRODUZIONE

Lo scopo del progetto è la realizzazione di un file server storage per la memorizzazione dei file in memoria principale.

Lo sviluppo è stato guidato dalla realizzazione di 3 componenti interconnesse:

- Server: si occupa della memorizzazione dei file in memoria centrale e della gestione delle connessioni e delle richieste da parte dei client.
- API: libreria di interazione con il file server. Si occupa di gestire tutto il protocollo di comunicazione con il server. La chiamata di una funzione di tale libreria genererà una serie di interazioni, secondo specifici protocolli, con il server, al fine di soddisfare la specifica della suddetta funzione.
- Client: si occupa di parsare la linea di comando e generare così una lista di richieste che saranno tradotte in una serie di chiamate alle funzioni dell'API implementata.

Ho realizzato tutte le parti obbligatorie di consegna per l'appello di Luglio, aggiungendo le parti opzionali di produzione di un file di log e l'opzione –D per il salvataggio dei file in seguito ad un capacity miss relativo a una *writeFile()* o un *appendToFile()*.

Nel corso della relazione, quando citerò una particolare funzione, per brevità, mi limiterò a specificare solo il nome della funzione e non tutti i parametri e il valore di ritorno.

Repository Git: https://github.com/luca-rizzo/Progetto-SOL_2020-21
Librerie di terze parti: per la tabella hash e per il threadpool mi sono servito delle implementazioni fornite a lezione su didawiki.

SERVER

Configurazione file storage

Le configurazioni del file storage server vengono definite attraverso un file il cui nome va passato da linea di comando.

Il formato del file [config].txt deve essere il seguente (eventuali righe vuote o spazi non creano alcun problema e le righe possono essere in qualsiasi ordine):

```
n_workers: (intero che indica il numero di worker del server)
n_max_file: (intero che indica il numero massimo di file che posso memorizzare nel server)
max_storage: (intero che indica dimensione massima del server in bytes)
sockname: (stringa contente nome file associato al socket)
```

Il passaggio da linea di comando di un file di configurazione del server che non ha il formato richiesto (es. sono specificati solo alcuni parametri) provocherà l'avvio del server con configurazioni di default.

File Storage e file

Il file storage è stato implementato con la struct t_file_storage che contiene al suo interno una tabella hash per la memorizzazione dei file: in tal senso nella tabella hash la chiave sarà il path assoluto del file e il valore sarà una struct t_file. I file sono stati implementati con la struct t_file. Entrambe le definizioni sono presenti nel file header server.h

Strutture dati ausiliarie

La struttura dati *clientAttivi* contiene una coda che mantiene l'fd di tutti i client connessi con il server e una mutex per l'accesso a tale struttura dati in mutua esclusione. Tale struttura dati permetterà di chiudere tutte le connessioni attive in seguito alla recezione di un segnale SIGINT o SIGQUIT oppure di controllare se ho ancora client attivi in seguito ad un segnale di SIGHUP.

La struttura dati *logStr* viene utilizzata per gestire le operazioni di loggin sul file *./log/log.txt* (a partire dalla directory principale del progetto).

Struttura implementazione server

Il server è stato implementato, come richiesto, come un singolo processo multi-threaded secondo lo schema "**master-worker**", con l'aggiunta di un signal handler thread che si occupa della gestione dei segnali.

Il thread "main" sarà il master e avrà i seguenti compiti:

- accettare nuove connessioni

- affidare richieste da parte di un client ad un worker thread: il thread "main", quando si sblocca dalla select perché è pronto un fd che si riferisce ad un client, aggiunge all'interno del threadPool (tramite la funzione *addToThreadPool()*) una "richiesta" contenente la generica funzione che ciascun worker deve eseguire (*funcW()*) e gli argomenti della funzione. Questa è la modalità per avvertire un worker che un client ha una richiesta da fare.
- ascoltare endpoint di lettura della pipe di comunicazione con il signal handler thread, il quale mi comunica, una volta ricevuto un segnale SIGHUP/SIGINT/SIGQUIT, il segnale ricevuto: sulla base del segnale ricevuto agirò di conseguenza come specificato nel progetto.

  Se si riceve il segnale SIGHUP, il thread main chiuderà fd_skt (che è responsabile di accettare nuove connessioni) e il server terminerà non appena avrà servito tutte le richieste dei client che sono in quel momento connessi. Visto che le connessioni sono chiuse unicamente dai worker threads, fondamentale è il timer sulla select che si sveglia ogni 200ms per controllare se non ho più connessioni attive.

  Se si riceve il segnale SIGINT o SIGQUIT il thread main esce immediatamente dal loop principale e notifica i threads worker che devono uscire (indipendentemente dal fatto che ci siano ancora richieste da parte dei client connessi).

  In entrambi i casi (SIGHUP o SIGINT/SIGQUIT) la notifica di uscita per i worker threads verrà generata dalla funzione *destroyThreadPool(),* fornita nell'implementazione del thread pool, che sveglierà tutti i workers che erano in attesa sulla variabile di condizione, farà la join di quest'ultimi e deallocherà la struttura dati threadPool.

  I worker threads che erano in attesa, una volta svegliati, usciranno dal loop principale poiché è stata modificata la variabile interna pool->exiting.

  Un worker thread che non era in attesa ma stava servendo una richiesta, una volta servita tale richiesta, uscirà dal loop principale e terminerà (sempre perché è stata modificata la variabile interna pool->exiting).

  Una volta terminati tutti i worker thread, verranno chiuse tutte le connessioni con i client, verrà stampato un resoconto dell'attività del server e verrà liberata tutta la memoria allocata in modo da generare un'uscita pulita.
- ascoltare endpoint di lettura della pipe di comunicazione con i worker threads che mi comunicano quali fd_client sono da riascoltare.

Un generico worker thread resterà in attesa di richieste su una variabile di condizione: una volta notificato dell'arrivo di una "richiesta" (tramite una signal generata dalla funzione *addToThreadPool()*), esegue la funzione generica *void funcW(void* arg)* che gestirà una particolare richiesta da parte di un client.

arg conterrà fd del client da servire (da cui leggere o scrivere per comunicare col client), la pipe su cui scrivere per eventualmente notificare il main thread di riascoltare tale fd e l'id dell'thread.

Le richieste sono gestite attraverso uno specifico protocollo di comunicazione tra server e API descritto di seguito:

1. leggo il tipo dell'operazione definito in conn.h. Potrei anche leggere l'end_of_file dovuto alla chiusura del socket lato client: in questo caso chiuderò la connessione.
2. a seconda del tipo dell'operazione mi atterrò ad un protocollo specifico
   - **op** (openFile): leggo lunghezza path, leggo contenuto path, leggo flag, eseguo richiesta (creo o apro un file per client fd), scrivo valore errno in caso di errore o 0 in caso di successo;
   - **rd** (readFile): leggo lunghezza path, leggo contenuto path, eseguo richiesta (leggo un file dalla tabella hash), scrivo valore errno in caso di errore o 0 in caso di successo, [se scrivo 0] scrivo dimensione file, scrivo contenuto file (se dimensione file>0).
   - **wr** (writeFile): leggo lunghezza path, leggo contenuto path, effettuo controlli, scrivo errno in caso d'errore o 0 se devo leggere il file, [se scrivo 0] leggo dimensione file, leggo contenuto file (se dimensione file>0), eseguo richiesta(scrittura di un file), scrivo valore errno in caso di errore o 0 in caso di successo, [se scrivo 0] scrivo lunghezza path, scrivo contenuto path, scrivo dimensione file, scrivo contenuto file per ogni file espulso in seguito a capacity miss.
   - **ap** (appendToFile): leggo lunghezza path, leggo contenuto path, effettuo controlli, scrivo errno in caso d'errore o 0 se devo leggere contenuto da appendere al file, [se scrivo 0] leggo dimensione buffer, leggo contenuto buffer (se dimensione buffer>0), eseguo richiesta(appendere buffer su un file), scrivo valore errno in caso di errore o 0 in caso di successo, [se scrivo 0] scrivo lunghezza path, scrivo contenuto path, scrivo dimensione file, scrivo contenuto file per ogni file espulso in seguito a capacity miss.

- **re** (removeFile)/ **cl** (closeFile): leggo lunghezza path, leggo contenuto path, eseguo richiesta (rimuovo un file / chiudo il file per client fd), scrivo valore errno in caso di errore o 0 in caso di successo.
- **rn** (readNFiles): leggo numero file da leggere, determino numero file da inviare, scrivo errno in caso di errore o 0 in caso di successo fino a questo momento, [se scrivo 0] scrivo numero file letti, scrivo lunghezza path, scrivo contenuto path, scrivo dimensione file, scrivo contenuto file per ogni file letto.

I valori di errno sono stati scelti tra quelli definiti in POSIX.1 e notificano il client dell'errore che è avvenuto: es. la scrittura di un file troppo grande restituisce un valore di errno = EFBIG oppure la creazione di un file già esistente restituisce un valore di errno = EEXIST.

Errori di protocolli interni al server durante una richiesta (es. errore inserimento valore in una coda o errore ricerca in tabella hash) restituiranno un valore di errno = EPROTO.

Soltanto errori di tipo EBADMSG (errore di comunicazione fatali) chiuderanno la connessione con il client poiché si tratta di errori sul protocollo di comunicazione con API (es. il client ha chiuso la connessione oppure API malevola che non rispetta protocollo di comunicazione ecc).

Di rilevanza è il caso in cui il server notifica il client dell'invio di n file espulsi ma si verifica un errore nell'invio dell'i-esimo file (con i<=n): in tal senso il server chiuderà la connessione con il client (che sarà quindi avvertito dell'errore tramite la lettura dell' EOF) poiché non sto rispettando il protocollo di comunicazione stabilito. Il client non potrà, in tal senso , rimanere bloccato su una readn in seguito ad errori in protocolli di comunicazione.

In tutti gli altri casi notificherò comunque il main thread che deve riascoltare fd per una successiva richiesta.

L'operazione **wr** può essere vista come una sorta di inizializzazione di un file appena creato sul server: in tal senso andrà a buon fine solo se l'ultima operazione di modifica sul file è stata la sua creazione.

La **chiusura di una connessione** con un client sarà compito di un worker: è importante non soltanto chiudere l'fd associato a quel client e rimuoverlo dalla lista dei client connessi, ma bisogna anche rimuovere l'fd riferito ad un client da ciascuna lista interna di fd di ogni file (chiudere tutti i file che il client aveva aperto), visto che lo stesso fd potrà essere associato ad un client diverso successivamente.

In seguito alla chiusura di una connessione il worker thread ovviamente non notificherà il main thread che deve riascoltare il fd relativo a quel client.

Dopo aver gestito una particolare richiesta un worker thread si mette in attesa di gestire una nuova richiesta da parte di un qualsiasi client connesso.

Accesso concorrente al file storage

Per "rispondere" alle richieste dei client, i worker thread accedono e modificano il file storage e i file che esso contiene: essi rappresentano dati condivisi e perciò dobbiamo garantire che ciascun worker thread abbia accesso esclusivo a tali dati.

Per garantire mutua esclusione per ciascuna operazione ho introdotto una mutex globale per tutto il file storage e due mutex e una variabile di condizione per ciascun file secondo il protocollo "fair" lettori-scrittori visto a lezione.

Per operazioni che modificano/rimuovono i file devo prima acquisire mutua esclusione in scrittura sul file attraverso una *startWrite()* e rilasciare muta esclusione al termine della modifica attraverso una *doneWrite()*.

Le operazioni di lettura dei file sono non mutuamente esclusive tra loro ma devono comunque acquisire mutua esclusione in lettura attraverso una *startRead()* e rilasciare mutua esclusione in lettura attraverso una *doneRead()*. In tal modo potrò avere al più uno scrittore(e nessun lettore) o più lettori(e nessuno scrittore) che accedono al file contemporaneamente.

Le richieste dei client si dividono in due gruppi: richieste creative/distruttive che provocano la modifica del file storage e richieste non-creative/non-distruttive che non modificano il file storage.

Per le richieste creative/distruttive (es. creazione file o rimozione file o scrittura che provoca capacity-miss e quindi avvio dell'algoritmo di sostituzione) l'iter di accesso al file sarà il seguente:

- acquire globale, cerca file, acquire file, operazione sul file, [release file], release globale

La lock globale sul file storage posso rilasciarla solamente alla fine dell'operazione perché altrimenti, altri worker thread potrebbero leggere degli stati non consistenti.

Per le richieste non-creative/non-distruttive (es. apertura di un file esistente, lettura di un file, scrittura di un file che non provoca capacity miss) l'iter di accesso al file sarà il seguente:

- acquire globale, cerca file, acquire file, release globale, operazione sul file, release file

Questa distinzione permette un accesso concorrente più rapido in quanto, ad esempio, la lettura di N file non bloccherà l'accesso a tutto il file storage ma solo l'accesso a tali file da parte di eventuali scrittori.

Operazioni critiche come la rimozione di un file da parte di un altro thread mentre sono in attesa di acquisire mutua esclusione su tale file non sono concesse in quanto, mentre sono in attesa di acquisire la acquisire mutua esclusione su un file, mantengo la lock globale e blocco qualsiasi altra operazione.

Ogni operazione terminata **correttamente** verrà salvata nel **file di log** presente nella directory ./log(a partire dalla directory principale del progetto) attraverso la funzione *scriviSulFileLog().*

Rimpiazzamento dei file nel file storage

Quando si raggiunge il numero massimo di file memorizzabili nel file storage, verrà avviata la politica di sostituzione che selezionerà una vittima secondo la politica FIFO: tale file verrà eliminato e non sarà in nessun modo più accessibile.

In seguito ad un capacity-miss (quando si raggiunge la massima capacità in byte del server) dovuto ad una scrittura/append su un file, verrà avviata la politica di sostituzione e individuata una vittima (non deve ovviamente essere il file su cui voglio scrivere) ma il file rimosso dal file storage non viene eliminato, ma inserito in una coda e inviato successivamente al client (una volta inviato al client, il file verrà ovviamente deallocato).

Per adottare una politica FIFO alla creazione del file verrà settato il valore struct timespec tempoCreazione contenuto in struct t_file che permetterà di individuare il file "più vecchio" presente nel file server.

API

L'API permette la comunicazione tra client e server tramite un insieme di funzioni definite in *api.h*.

L'API è realizzata in maniera del tutto indipendente dal client e pertanto verrà compilata in una libreria dinamica libapi.so che successivamente sarà linkata per generare l'eseguibile bin/client.

Ciascun client mantiene una sola connessione con il server: in tal senso una variabile globale *fd_skt* mantiene il valore dell'fd del socket su cui scrivere richieste e leggere risposte. Tale variabile viene settata in seguito a una *openConnection()* e resettata in seguito a una *closeConnection().*

L'API soddisfa la specifica di ogni funzione comunicando secondo un protocollo ben definito con il server (ovviamente speculare rispetto a quello del server):

- openFile(): scrive operazione op, scrive lunghezza path, scrive contenuto path, scrive flag, legge risultato operazione (errno o 0)
- readFile(): scrive operazione rd, scrive lunghezza path, scrive contenuto path, legge risultato operazione (errno o 0), [se risultato=0] legge dimensione file, legge contenuto file.
- writeFile(): scrive operazione wr, scrive lunghezza path, scrive contenuto path, legge risultato operazione fino a questo punto (errno o 0), [se risultato=0] scrive dimensione file, scrive contenuto file, legge risultato operazione fino a questo punto (errno o 0), [se risultato=0] legge numero file espulsi, legge lunghezza path, legge contenuto path, legge dimensione file, legge contenuto file per ogni file espulso
- appendToFile(): scrive operazione ap, scrive lunghezza path, scrive contenuto path, legge risultato operazione fino a questo punto (errno o 0), [se risultato=0] scrive dimensione buffer, scrive contenuto buffer, legge risultato operazione fino a questo punto (errno o 0), [se risultato=0] legge numero file espulsi, legge lunghezza path, legge contenuto path, legge dimensione file, legge contenuto file per ogni file espulso
- removeFile()/closeFile(): scrive operazione re/cl, scrive lunghezza path, scrive contenuto path, legge risultato operazione (errno o 0)
- readNFiles(): scrive operazione rn, scrive numero di files da leggere, legge risultato operazione fino a questo punto (errno o 0), [se risultato=0] legge numero file inviati, legge lunghezza path, legge contenuto path, legge dimensione file, legge contenuto file per ogni file inviato

Tutte le funzioni dell'API settano errno tramite il valore di errore ritornato dal server oppure errno viene settato opportunamente dalle chiamate di sistema per le operazioni interne (es salvare file in directory ecc) oppure errno è settato in maniera specifica dall'API in seguito a particolari errori (es eseguire due *OpenConnection()* sullo stesso socket). La lista dei possibili errori è documentata nel file api.h.

E' stata aggiunta all'interfaccia anche un'ulteriore funzione *scriviContenutoInDirectory*() per i salvataggio in directory locali di file letti/espulsi inviati dal server.

CLIENT

Compito del client è quello di parsare la linea di comando al fine di determinare una lista di richieste da fare al server. Il parsing avviene tramite la funzione getopt e in base all'option character rilevato si aggiungerà una richiesta nella coda richieste.

In caso di richieste di scrittura o lettura di liste di file (denotate con la virgola: es –r file1.txt,file2.txt) queste verranno scomposte in richieste di lettura o scrittura singole e indipendenti.

In caso di richieste di scritture di intere directory con opzione –W, queste richieste verranno tradotte in tante richieste di scrittura indipendenti(una per ciascun file) visitando ricorsivamente la directory e tutte le subdirectory.

Le opzioni –d e –D vanno usate congiuntamente alle opzioni –r/-R e –w/-W e si riferiscono all'insieme di operazioni di lettura/scrittura che precedono immediatamente: ad esempio –r file1.txt,file3.txt –d fileLetti1 –r file2.txt –d fileLetti2 verrà tradotto nella lettura del file file1.txt e file3.txt nella directory fileLetti1 e nella lettura del file2.txt nella directory fileLetti2.

In seguito ciascuna richiesta verrà tradotta in una serie di chiamate alle funzioni dell'API: ad esempio una richiesta –W file1.txt verrà tradotto in una sequenza *openFile(), writeFile(), closeFile()*.

Il client stamperà esito dell'operazione solo se è stato settato il parametro –p: in caso di errore stamperà anche la motivazione del fallimento tramite la funzione perror (l'API setta errno).

Path

Le richieste di scrittura di un file tramite le opzioni –w e –W accettano path relativi che verranno convertiti in path assoluti tramite la funzione *realpath()* poiché nel file server un file è identificato univocamente dal suo pathname assoluto.

Le richieste di lettura o rimozione di un file devono specificare il pathname assoluto del file da rimuovere perché solo esso lo identifica univocamente.

Le opzioni –r o –c, tuttavia, accettano anche path relativi a partire dalla working directory del processo client: in tal caso non possiamo usare la funzione *realpath()* poiché un client potrebbe voler leggere un file che non è esistente nel suo file system.

Un pathname relativo passato all'opzione –r o –c viene tradotto in un pathname assoluto ottenuto concatenando il pathname della working directory con il pathname relativo.

Ho implementato così una semplice funzione *myrealpath()* che prende come argomento un pathname e interpreta tale pathname come relativo (e lo converte in un ipotetico path assoluto) se questo non comincia con / oppure lo interpreta come assoluto se comincia con /.

La scrittura dei file letti/espulsi dal file storage avviene tramite la funzione *int scriviContenutoInDirectory(void* buffer, size_t size, char* pathFile, char* dirToSave)* che ha il compito di scrivere il contenuto del *buffer* sul file *pathFile* nella directory *dirToSave*: siccome *pathFile* è la stringa che identifica univocamente un file, i file manterranno in *dirToSave* il path che avevano sul server.

In tal senso verranno create ricorsivamente delle directory.

Questa decisione è legata al fatto che ciascun file nel file storage è identificato univocamente dal suo path assoluto e quindi bisogna mantenere il path che il file aveva sul server per evitare che due file differenti inviati dal server abbiano lo stesso nome nel nostro file system locale (ciò causerebbe la possibilità di mantenere solo uno dei due file).

MAKEFILE

Per compilare il progetto e eseguire i test è stato realizzato un makefile con diverse opzioni.

- *make*: esegue il target di default *all* per compilare l'intero progetto generando file oggetto, librerie e eseguibili
- *make test1*: permette di eseguire il test1. Dopo aver avviato in backgroud il server, fa partire lo script bash script/test1.sh che fa partire 4 client che testano tutte le opzioni messe a disposizione dei client
- *make test2*: permette di eseguire il test2. Dopo aver avviato in backgroud il server, fa partire lo script bash script/test2.sh, il cui scopo principale è quello di mostrare la funzionalità dell'algoritmo di rimpiazzamento del file storage e il salvataggio(lato client) dei corrispondenti file espulsi dal file storage.
- *make cleanAll*: permette di ripulire tutte le directory da i vari file oggetto, librerie, eseguibili e file prodotti dai test
- *make clearTest1*: permette di ripulire ciò che viene prodotto dal test1 per poter rieseguire il test1 partendo da uno stato pulito.
- *make clearTest2*: permette di ripulire ciò che viene prodotto dal test2 per poter rieseguire il test2 partendo da uno stato pulito.