PROGETTO SISTEMI OPERATIVI E LABORATORIO

Davide Parrini Mat-599858

GitHub: https://github.com/davideparrini/ProgettoSol_2021_FileStorageServer

-SERVER

Il server prende come argv[1] il nome del file config.txt (solo il nome del file .txt, es. config) da dove prende le impostazioni necessarie per essere eseguito e le mette in una struttura dati "config"

```
typedef struct config_{
        int n_thread_workers;  // numero thread workers
        int max_n_file;  // numero massimo di file possibili da memorizzare nel server
        double memory_capacity; // capacità massima del server in Mb
        char socket_name[NAME_MAX]; // path assoluto del socket
} config;
```

implementata in utils.h, tramite il metodo void setUpServer(config *Server); .

Per l'implementazione del fileStorage è stata utilizzata come struttura dati una "hashtable" (implementata in myhashstoragefile.h/myhashstoragefile.c) a lista di trabocco bidirezionale, con l'aggiunta di una lista bidirezionale, sempre appartente alla struttura dati della hash, chiamata "cache" (di capacità massima proporzionale alla capacità massima della hash in rapporto 1 : 10, con minima capacità uguale a 1 indipendentemente dalla capacitò della hash) la cui funzionalità è quella di una cache con politica LRU, ma al suo interno avranno accesso solo file che sono stati modificati, considerando che i file possono essere espulsi solo se modificati. Quindi dal momento in cui un file è stato modificato, se è utilizzato per qualsiasi tipo di operazione (append, lettura, chiusura, apertura..) esso verrà spostato nel fileStorage in testa alla lista della cache, tramite il metodo void update_file(hashtable *table,file_t* file);. L'algoritmo di rimpiazzamento prenderà come target la cache come ultima risorsa, prendendo come vittima per primi i file nella coda della cache fino ad arrivare alla testa.

Il server quindi inizializza la hash con il metodo void init_hash(hashtable *table, config s);, prendendo come parametro le informazioni ottenute dal file config ed inizializza una lista di "files_removed" che conterrà tutti i file espulsi o rimossi dallo Storage.

Ed infine inizializza a 0 una struttura dati di interi "stats" che conta il numero volte che ogni operazione avuto con successo.

Il funzionamento del server è gestito da un thread manager, il quale rimane sempre in ascolto di connessioni, un thread sighandler_thread, che riceve e gestisce i segnali, mettendosi in attesa con sigwait dei segnali SIGINT SIGHUP e SIGQUIT indicati da una maschera apposita, e dei thread workers che hanno la funzione di analizzare le richieste in arrivo e soddisfarle.

Tramite l'utilizzo di due pipe avviene la comunicazione necessaria per coordinare il server durante il suo funzionamento. La prima pipe mette in comunicazione il thread manager e i thread workers in modo tale che quando un worker termina l'esecuzione di una richiesta, scrive sulla pipe (pipeWorker_Manager) al manager che il descrittore del client, in cui era scritta la richiesta che è stata soddisfatta, può essere rimesso a disposizione e sblocca la select. Per gestire la concorrenza sulla scrittura della pipe è stato utilizzato un "flag_semaforo" e l'ausilio di pthread_cond, flag_semaforo == VERDE -> si può scrivere sulla pipe.

La seconda pipe mette in comunicazione il thread manager e il thread sighandler. All'arrivo di un segnale SIGINT/SIGHUP/SIGQUIT il sighandler scrive sulla pipe (signal_pipe) il tipo di segnale al manager. E quest'ultimo analizza il segnale ricevuto :

if (sig == SIGINT || sig == SIGQUIT) sveglio tutti i thread worker che sono in stato di attesa (Ovvero non stanno eseguendo richieste) e chiudo il server aggiornando il "flag_closeServer" = 1 e "termina" = 1 (la condizione del while(!termina), ciclo del thread manager necessario per la sua continua esecuzione);

if (sig == SIGHUP) chiudo il descrittore d'ascolto per nuove connessione (server_fd) e aggiorno il "flag_SigHup" = 1 e semplicemente posticipo la chiusura istantanea del server fino a che ho connessioni attive. Quando non ho più connessioni attive, mi riconduco al caso SIGINT/SIGQUIT e chiudo il server.

Una volta chiuso il server creo un fileLog.txt con al suo interno tutte le statistiche del server, file nel server al momento di chiusura, e tutti i file che sono stati espulsi o rimossi dal server, stampando a schermo il suo contenuto. Tutto tramite il metodo void create_FileLog(); .

Libero la memoria dello storage e della lista di file rimossi dallo storage.

-CLIENT

Il client per avviare la sua esecuzione argc deve essere >= 2, prende come argv[1] il path della cartella dove eseguire i test (es. test/test1) e deve essere passato almeno un comando.

Inizialmente vengono inizializzate delle stringhe dove sono scritti i path assoluti delle cartelle per i test ("testDirPath" cartella dove sono presenti i file per eseguire i test, "testToSave" cartella dove vengono salvati i file tramite le opzioni -d e -D, al suo interno sono presenti 2 cartelle specifiche per differenziare i file salvati da quale opzione "test-d" "test_D").

Vengono poi inizializzate delle variabili necessarie al countdown, dopo il quale il client cesserà di provare a connettersi al server.

Viene fatta una passata di getopt per acquisire i comandi passati da terminale. In particolare vengono eseguiti subito i comandi per la configurazione del client, ovvero l'opzioni -f -t -p e -h (anche se quest'ultima non serve per configurare il client, ma dopo la stampa delle istruzioni termina l'esecuzione del client) , le quali non necessitano di una connessione al server. Le altre opzioni consentite vengono invece messe in una "char_queue" (una coda di "opzioni", che memorizza per ogni nodo il carattere dell'opzione e l'argomento di tale opzione ottenuto dalla getopt. La coda è implementata in myqueueopt.h/.c). Le richieste all'interno della char_queue vengono scandite dopo la effettiva connessione al server.

Ogni richiesta ha un suo metodo specifico (es. arg_W) che esegue il comando specifico di ogni opzione.

Per quanto riguarda le opzioni -d e -D, essendo funzioni ausiliarie delle opzioni -r -R (per l 'opzione -d) e -w -W -a (per l'opzione -D) è necessario che siano eseguite dopo queste opzioni, senno abbiamo un errore. (es. corretto funzionamento -> ./client test/test1 -f /tmp/server_sock -r pippo.txt -d test-d ; funzionamento errato -> ./client test/test1 -f /tmp/server_sock -d test-d -r pippo.txt)

Ed essendo funzioni ausiliarie ho scelto di fare eseguire le richieste -d -D direttamente dalle funzioni -r -R -w -W -a, quindi non esistono metodi arg_d o arg_D. E il tempo di attesa relativo all'opzione '-t' non avrà effetto su di esse.

Ho implementato delle opzioni aggiuntive:

‘-a’ : (-a nomefile:TestoToAppend) che permette di fare una append di un testo specificato dopo i ‘:’ ad un file specifico. E’ possibile utilizzarla insieme all’opzione -D.

‘-o’: (-o O_FLAG:file1[,file2]) che permette di fare una openFile con i O_FLAG (O_CREATE,O_LOCK,O_NOFLAGS con possibilità di metterli in or bit a bit con ‘-‘ es. O_CREATE-O_LOCK:pippo.txt) specificati su una lista di file, separati da ‘,’ .

‘-O’: (-O O_FLAG:dirname[,n=0]) che permette di fare una openFile con i O_FLAG (modalità identica all’opzione -o) a *n* file a partire dalla directory specificata dopo ‘:’ *dirname* visitando le sue sotto directory ricorsivamente, se *n* == 0 o non specificata apre tutti i file della cartella.

‘-C’: (-C file1[,file2]) che permette di fare una closeFile sulla lista di file, separati da ‘,’ .

‘-G’: (-G dirname[,n=0]) che permette di fare una closeFile su *n* file della cartella specificata *dirname* visitando le sue sotto directory ricorsivamente, *n* == 0 o non specificata chiude tutti i file della cartella.

La visita ricorsiva delle sotto directory delle opzioni -w -O -G sono implementate in 3 metodi differenti:

int writeFileDir(char* dirpath,char* dir_rejectedFile,int n,int flag_end,size_t* written_bytes,int tutto_ok);

int openFileDir(char* dirpath,int n,int flag_end,int tutto_ok);

int closeFileDir(char* dirpath,int n,int flag_end,int tutto_ok);

O meglio, il funzionamento è il medesimo, ma si distinguono per le funzioni writeFile, openFile e closeFile e la writeFileDir ha come parametro size_t* written_bytes per salvarsi il numero di bytes scritti.

Dopo l’esecuzione di ogni richiesta presa dalla char_queue viene liberata la memoria del nodo della lista e utilizzo il metodo msleep (implementato in utils.h/.c) per dormire msec settati con l’opzione -t.
Quando la char_queue è svuotata completamente, mi disconnetto dal client e termino l’esecuzione del client.

-SERVER API

Interfaccia per interagire con il file server (API) prende come variabili extern dal client:

*char *socket_path* , il socket path del client;

int client_fd, il descrittore del client per interagire col server;

size_t print_bytes_readNFiles, una variabile utile per tenere il conto dei bytes letti durante la readNfiles, in caso in cui le stampe per operazione sono attive (tramite l'opzione -p).

E' stato implementato un metodo aggiuntivo per ricevere i file dal server da memorizzare su disco in una determinata cartella, utilizzato nei metodi writeFile e appendToFile.

int getlistFiletoReject_createfileInDir(const char *dirname,int fd_receptor);

Durante l'implementazione dei metodi readFile, readNFiles, writeFile e appendToFile sono stati utilizzate dei flag ("*flag_ok" "flag_dirname")* utili per "sincronizzare"/ far combaciare il funzionamento delle varie readn/writen tra server e serverAPI, in caso ,generalmente, di fallimento dell'operazione (ad esempio la readFile di un file non esistente/file chiuso, c' ho comporta che non si deve fare una readn del contenuto del file e senza la verifica di tali flag il client o il server si bloccherebbe su una readn/writen) o nel caso in cui avviene una richiesta di scrittura di file su disco su una determinata directory.

- STRUTTURE DATI UTILIZZATE

request: implementata in request.h/.c, è la struttura dati che ottiene il server quando ascolta il descrittore mandato dal client.

```
typedef struct f{
        int socket_fd; //variabile dove è scritto il descrittore del client da lato server
        int flags; //variabile che indica i flag di una determinata task/richiesta
        int c; //contatore generico
        char pathfile[NAME_MAX]; //path del file della richiesta
        size_t request_size; //dimensione della richiesta
        req_type type; // req_type è un enum utile per identificare il tipo di richiesta
} request;
```

response: implementata in response.h, è la struttura dati inviata dal server al client per identificare l'esito della richiesta in precedenza esaudita, ed eventuali valori di ritorno.

```
typedef struct s{
        response_type type; // response_type è un enum utile per identificare l'esito
                            specifico della richiesta
        int c; //contatore generico, utile come valore di ritorno
        size_t size; //size della risposta
} response;
```

file_t: implementata in myhashstoragefile.h/.c ,è la struttura dati che rappresenta un determinato file nello storage file del server.

```
typedef struct node{
    int fd; //descrittore specifico del file, inizializzato a -2 se non utilizzato o chiuso
    char* abs_path; // path assoluto del file
    void* content; // contenuto del file
    size_t dim_bytes; // dimensione del file
    int modified_flag; // flag che indica se il file è stato modifcato o meno
    int open_flag; // flag che indica se il file è apero o meno
    int o_create_flag; //flag che indica se il file è stato aperto con O_CREATE
    int locked_flag; // flag che indica se il file è lockato
    int inCache_flag; //flag che indica se il file è nella cache ( == 1)
    struct node *next; // puntatore al nodo successivo
    struct node *prec; // puntatore al nodo precedente
}file_t;
```

list_file: implementata in myhashstoragefile.h/.c ,è una lista specifica per file_t, ha come argomenti rifermenti alla testa e alla coda della lista, dimensione in bytes della lista e la cardinalità della lista.

hashtable: implementata in myhashstoragefile.h/.c ,è una hashtable utilizzata per implementare lo storage file del server.

E' una hash a lista di trabocco, più una lista esclusiva che rappresenta la cache.

-**POLITICA DI RIMPIAZZAMENTO LRU:**

L' algoritmo di rimpiazzamento è implementato direttamente in myhashstoragefile.h/.c ,

nei metodi:

```
int init_file_inServer(hashtable* table,file_t *f,list_file* list_reject);
int ins_file_inServer(hashtable* table,file_t *f,list_file* list_reject);
int modifying_file(hashtable* table,file_t* f,size_t size_inplus,list_file* list_reject);
```

-**PARTI OPZIONALI SVOLTE**

1. File Log.
2. Opzione -D
3. LRU
4. Opzione -a
5. Opzione -o
6. Opzione -O
7. Opzione -C
8. Opzione -G