UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

SISTEMI OPERATIVI LABORATORIO

Relazione:

File storage Server

Francesco Caprari 580154

29/01/2022

1. Indice

2. Specifica Progetto

Lo scopo del Progetto è la realizzazione di un ***file storage server*** che permette la scrittura e la lettura di file sulla memoria principale, per poter leggere e scrivere sul server il client utilizza un **API,** il server è implementato come un singolo processo multi-threaded.

Codice Esterno: Usando un **Hash map** per modellare il file system si utilizza come funzione Hash l'algoritmo **djb2,** per la lettura e la scrittura sulle socket sono state utilizzare le funzioni *writen* e *readn* mostrate a lezione

3. Server/ Strutture Dati

Per la Realizzazione del file system, quindi la gestione della memorizzazione dei file, si utilizza un Hash map che gestisce le collisioni con concatenamento, per rappresentare un file all'interno dell'Hash map si utilizza una struct(**file.h)**, infine c'è un ulteriore lista (**list.h**) per realizzare la politica di rimpiazzamento FIFO per la gestione dei file da espellere dal server in caso di "capacity misses".

Struct file

```
{
  int isopen;
  char path_name[MAXSIZE];
  void* cont;
  int dim;
  int fd;
  int lock;
  pthread_mutex_t mtxf;
};
```

isopen indica se quel file è aperto, quindi se l'ultima operazione effettuata è una open, path_name, cont*, dim, rappresentano rispettivamente il nome del file, il contenuto e la dimensione, fd indica il file descriptor del client che ha effettuato la OpenFile() o la LockFile(), la lock assumerà solo valori 0 e 1 a seconda se è stato effettuata un operazione di LockFile () o se il file è stato creato con il flag O_LOCK, l'ultima variabile è utilizzata per accedere in muta esclusione al file. Viene utilizzata un ulteriore struct per mantenere le specifiche del server indicate nel file di configurazione:

typedef struct serverconf

```
{
    int nwork;   numero worker
    int space;   spazio massimo
    int maxf;   numero massimo di file
    char* socketname;   nome della socket
    char* filelog;          nome del file di log
}ServerConf;
```

Viene utilizzata anche una lista per la gestione delle richieste (*queueric)*

Per gestire le situazioni di race condition si utilizzano 4 mutex e 2 variabili condizione rispettivamente per la gestione concorrente della coda delle richieste, dell' hash map, del file di log, della coda fifo e delle connessioni attive.

```
pthread_mutex_t mtxrequest = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condrequest = PTHREAD_COND_INITIALIZER;

pthread_mutex_t mtxhash = PTHREAD_MUTEX_INITIALIZER;


pthread_mutex_t mtxlog = PTHREAD_MUTEX_INITIALIZER;



pthread_mutex_t mtxfifo= PTHREAD_MUTEX_INITIALIZER;


pthread_mutex_t mtxcon= PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condcon=PTHREAD_COND_INITIALIZER;
```

4. Comunicazione Client Server

Client e Server si scambiano informazioni utilizzando una struct: (**Protcol.h**)

typedef struct request

{

char pathname[MAXS];	*nome del file da inviare*
int size;	*dimensione del file da inviare*
int flags;	*flag richiesti*
int fd;	*file descriptor del client(assegnato dal server)*
int OP;	*operazione richiesta*
void* contenuto;	*contenuto del file*

}request;

CLIENT

Il client invia le proprie richieste al server grazie ad una API, per ogni operazione presente il client assegna il codice al campo request.OP e il nome del file al campo pathname, nel caso della OPEN viene anche riempito il campo flags invece per le operazioni di scrittura viene assegnata anche la dimensione del file nel campo size.

Infine, il client invierà l'intera struct sulla socket e rimarrà in attesa della risposta del server.

SERVER

Il server manterrà una coda di *struct request* ad ogni richiesta dei client il server invocherà la funzione readMessage che leggerà l'intera struct e assegnerà il campo request.fd e in caso size sia diverso 0, quindi per le operazioni di scrittura, attenderà che il client invii anche il contenuto del file, una volta ottenute tutte le informazioni la richiesta verrà inserita in testa alla lista.

Le operazioni appena citate sono gestite da un **thread dispatcher** che, in più, si occupa di accettare le richieste di connessione dai vari client.

Vengono anche mandati in esecuzione una serie di **thread Workers** che si occupano di prelevare una richiesta dalla coda e di completare la funzione richiesta grazie al campo OP, una volta completata la richiesta invierà un codice al client che corrisponderà all'esito dell'operazione.

I codici cambiano a seconda dell'esito dell'operazione (**Protcol.h**)

5. Gestione delle interruzioni

Il server all'avvio maschera i segnali e manda in esecuzione un thread (**signalHandler**) che si occuperà di gestire le interruzioni, il signal handler nel caso della ricezione di un segnale di tipo **SIGINT** O **SIGQUIT** agisce su due variabili globali che faranno terminare immediatamente i thread worker e il thread dispatcher, Mentre nel caso della ricezione di un segnale di tipo **SIGHUP** il thread aspetta che non ci siano più connessioni attive, agendo sulla variabile globale **activecon** che conta il numero di connessioni, poi successivamente opererà alla stessa maniera del caso precedente, in entrambi i casi viene mandato un messaggio fittizio sulla pipe collegata alla select del dispatcher, al termine della gestione dei segnali vengono liberate le strutture dati e viene mandato in esecuzione uno script (**scriptstat.sh**) che agendo sul file di log stamperà un sunto delle statistiche sulle prestazioni del server.

6. Log File

Nel log file vengono specificate tutte le operazioni richieste del client o di gestione interna, il server per memorizzare le informazioni sul file utilizza due procedure **LOG** e **LOG_file**, la prima è sfruttata per scrivere tutte le informazioni relative ad ogni operazione:

NC : <activeconn>	*connessioni attive in quel momento*
NF : <n°file>	*numero di file presenti in quel momento sul server*
WR : <n° byte scritti>	*operazione di write e byte letti*
RD : <n° byte letti>	*operazione di read e byte letti*
TH : <thread>	*thread che ha effettuato quell'operazione*
NB : <n°byte occupati>	*nbyte presenti in quel momento sul server*
CL	*operazione di close*
LCK	*operazione di lock*
AR	*l'algoritmo di rimpiazzamento è stato eseguito*

La procedura **LOG_file** invece viene invocata alla terminazione del server per scrivere il pathname dei file rimanenti sul LogFile

7. Scelte progettuali

Open File: nel caso in cui ci sia un segnale di capacity miss all'interno dello storage causato dal superamento del numero massimo dei file, specificato nel file di config, il server inserirà comunque il file nello storage e rispedirà al client la testa della coda fifo.

Lock File: se la lock sul file specificato non può essere acquisita perché posseduta da un altro client, viene creata una nuova richiesta uguale a quella precedente e inserita nella coda delle richieste finché la lock non viene rilasciata da un altro thread, ogni volta che il client termina la connessione viene rilasciata la lock automaticamente su tutti i file di quel client.

Politica di Rimpiazzamento: nel caso di capacity miss viene invocata la procedura Handlercapacity che sfruttando la coda FIFO preleverà i file dalla testa della coda fino a quando non ci sarà abbastanza spazio, quindi il server invierà il numero di file da spedire al client e poi tutto il loro contenuto.

8. Make file e Test

Il make file mette a disposizione i comandi per creare l'eseguibile del client e del server e per l'esecuzione di *test1 test2* e *test3,* è presente un ulteriore script *statistiche.sh* che viene invocato automaticamente alla terminazione del server che agendo sul log file va stampare sullo standart output le statistiche del server.

Test1: manda in esecuzione dei client che provano tutte le operazioni dell'API e manda un segnale di SIGHUP al server una volta che l'ultimo client ha terminato.

Test2: similmente al test precedente manda in esecuzione dei client, ma questa volta il fine è quello di testare la politica di rimpiazzamento del server.

Test3: Per effettuare lo stress test, lo script memorizza il tempo corrente e il momento in cui dovrò interrompere l'esecuzione dei client, manda in esecuzione i client per il periodo indicato cercando di avere contemporaneamente dieci client attivi, una volta terminato manda un segnale di SIGINT al server.

Per compilare ed eseguire:

chmod +x Test/test1

chmod +x Test/test2

chmod +x Test/test3

chmod +x Test/scriptclient

chmod +x script/scriptstat

make test1

make test2

make test3