Universita' di Pisa - Dipartimento di Informatica

File Storage Server

Progetto di Laboratorio di Sistemi Operativi in C

Autore : Stefanelli Alessia
Matricola : 615430
Corso : A
Anno Accademico 2020 - 2021

1 Introduction

In questo progetto viene creato un file storage server che memorizza i file dati dal client in memoria principale e ne permette la lettura. L'architettura si basa su un approccio client-server. I client comunicano col server unicamente attraverso un API.
Il programma e' stato implementato e testato su una macchina Ubuntu 20.04.2 LTS 64-bit.
Il progetto e' archiviato nel seguente indirizzo di github :

`https://github.com/Lexy26/LSO-Project`

2 Client

Per gestire le richieste del client sono state create due strutture : "command_t" e "nb_ request", presenti nel file "util_client.h".

```
typedef struct {
    char option[2];
    char *param;
    char * dirname;
} command_t;
```

```
typedef struct {
    int tot_request;
    int char_t;
    int char_h;
    int char_p;
    char* char_f;
    command_t ** lst_char_abc;
}nb_request;
```

La struttura "command_t" fornisce le informazioni dell'operazione (ad es. -R n=3). La struttura "nb_ request" contiene l'insieme delle richieste totali fatte dal client, che si possono anche ripetere. Le opzioni -t, -h, -p, -f, non sono state inserite nella lista delle richieste, poiché devono essere trattate singolarmente.

Come primo passo viene inizializzata la strutttura "nb_ request" con tutti i parametri inizializzati a 0 dalla funzione initLstRequest(), che si trova nel file "util_client.c".
In seguito il parser degli argomenti delle richieste e' stato implementato attraverso la funzione getopt che ne estrapola i dati. Per le opzioni -r, -R, -w, -W viene usata la funzione createRequest() che inizializza la struttura dell'operazione e la inserisce nella lista delle richieste. Questa funzione e' presente nel file "util_client.c"
Il nome del socket, che permette la connessione col server, deve essere fornito dal client con l'insieme delle operazioni richieste, altrimenti si genera un errore. Per connettere il client al server il tempo massimo di attesa e' stato calcolato convertendo la variabile nsec in millisecondi presente nella struttura timespec ; mentre i millisecondi di attesa sono settatati con la macro MSEC.
Con **l'opzione -h** viene eseguita la stampa di tutte le operazioni che il client ha richiesto e infine termina l'esecuzione del programma.
Attraverso un istruzione 'while' le operazioni -r, -R, -w, -W, presenti in una lista, sono tutte trattate.
L'opzione -w e' stata implementata attraverso una funzione ricorsiva, chiamata recDirectory(), presente nel file "util_client.c". Nella funzione recDirectory viene eseguita la ricerca approfondita degli 'n' file da scrivere nello storage. Quando un file e' trovato, ne viene immediatamente ricavato il suo path assoluto con la funzione find_absolute_path() e successivamente eseguita la funzione openAppendClose(), contenente le seguenti API : openFile(), appendToFile(), closeFile(). Le funzioni find_absolute_path() e openAppendClose() sono presenti nel file "util_client.c"
L'opzione -W, per ogni file iterato, ricava il path assoluto ed esegue la funzione openAppendClose(), che invia le richieste al server per scrivere i file iterati.
L'opzione -d funziona se preceduta dalle opzioni -r e -R, in modo che ogni opzione abbia la

propria directory nella quale mettere i file letti. In caso di errore, viene ricordata la corretta sintassi.
L'opzione -r utilizza la funzione 'basename()', presente nell'include <libgen.h>, estrae dal path assoluto il nome del file letto nel storage, per creare cosi il path assoluto con la directory data e infine scrive il contenuto al suo interno.
L'opzione -R e' implementata unicamente nella funzione readNFiles() dell'API. Il client verifica soltanto la correttezza del parametro 'n', che indica il numero di file da leggere. Tutte le varianti di 'n' sono state implementate ad eccezione dell'operazione in cui non viene specificato 'n' dopo l'opzione.
L'opzione -t implementa un timer, la cui funzione 'timer()', presente nel file "util.c", consente di ritardare l'esecuzione di una richiesta consecutiva ad un'altra. L'attesa e' stata creata con l'uso della 'select()' e della struttura 'timeval'. L'implementazione della funzione e' stata presa dal seguente sito `https://qnaplus.com/c-program-to-sleep-in-milliseconds/`
L'opzione -p stampa l'operazione che sarà eseguita con il seguente formato :

"REQUEST ==== " + il tipo di operazione + i file o directory o n. + " ===="

Alla precedente riga viene successivamente stampato un elenco costituito dal path assoluto del file e dal numero di bytes letti o scritti, con il seguente formato "File : %s Size : %ld".
Ogni volta che un file viene inserito nello storage con l'opzione o -w o -W viene generata una stampa contenente i file espulsi a seguito di un "capacity misses". (la freccia verso l'alto indica il file che ha causato le espulsioni elencate di seguito)
Sempre con questa opzione si attiva sia la stampa di apertura che di chiusura della connessione col server.

In caso di possibili errori creati con la lettura o scrittura dei file, il client non termina l'esecuzione del programma, ma esegue una stampa specificando l'errore e quale file o directory lo ha provocato, per poi proseguire con l'esecuzione dell'operazione successiva.

3 API

Premessa

Per implementare l'API sono state utilizzate le seguenti funzioni integrate nel file "util.c" :

- **sendMsg()** : invia il messaggio al server con le informazioni di base necessarie alla specifica funzione dell'API che ne fa uso (vedi schema qui di seguito).
- **sendMsg_File_Content()** : invia il messaggio con le info + il contenuto del file che deve essere inserito nello storage dal server oppure letto dal client.
- **recievedMsg()** : funzione utilizzata per la ricezione del messaggio sia dal lato server che dal lato client
- **receivedMsg_File_Content()** : funzione per la lettura del contenuto del file.
- **timer()** : funzione che determina il tempo di attesa in millisecondi.

Per inviare e ricevere i messaggi sono state utilizzate due funzioni presentate durante le esercitazioni : **writen()** e **readn()** che permettono di non perdere le informazione contenute nei messaggi. Queste due funzioni sono presenti nel file "conn.c"
Tutte le funzioni dell'API rispecchiano uno schema ricorrente che si basa sul protocollo di comunicazione richiesta-risposta.
La struttura del messaggio e' formata da un intero 'size_t' che rappresenta la lunghezza del messaggio e da un 'unsigned char *' che ne indica il contenuto. Il messaggio e' stato implementato attraverso il tipo 'unsigned char*' dal momento che e' un tipo di dato che rappresenta

perfettamente 1 byte, e quindi risulta il più adeguato nella rappresentazione dei file binari. La struttura generale dei messaggi e' descritta brevemente nello schema qui sotto riportato.

```
// ------ GENERAL STRUCTURE OF MESSAGE ------ //
/* Send from CLIENT to SERVER ( sendMsg(),       sendMsg\_File\_Content())
 * SERVER Received from CLIENT( recievedMsg() + readn())
 * message structure (Request):
 *                    API_ID    ARG1     ARG2         ARG3
 * - OpenFile      -> api_id + pathname + flag
 * - AppendToFile  -> api_id + pathname + size_byte + content_file
 * - CloseFile     -> api_id + pathname
 * - CloseConnection -> api_id + sockname
 * - readFile      -> api_id + pathname
 * - readNFile     -> api_id +   N
 * */
/* Send from SERVER to CLIENT( writen(len)+writen(str), sendMsg\_File\_Content())
 * CLIENT Received from SERVER( recievedMsg(),     receivedMsg\_File\_Content())
 * message structure (Answer):
 *                       ARG1         ARG2       ARG3      ARG4
 * - OpenFile      -> -1 ko| 0  ok + file removed
 * - AppendToFile  -> -1 ko| 0,1 ok + file removed
 * - CloseFile     -> -1 ko| 0  ok
 * - CloseConnection -> -1 ko| 0 ok
 * - readFile      -> -1 ko| 0  ok + size_buf + content_buf
 * - readNFile     -> -1 ko| 0  ok + pathname + size_buf + content_buf
 * */
```

Open Connection : L'attesa e' stata implementata con la funzione timer(). Mentre la fine del tempo assoluto e' stato implementato con un decremento della variabile nsec presente in timespec, fino al raggiungimento del valore <= a zero.
Close Connection : L'implementazione di close connection usa le funzioni sendMsg() e recievedMsg() e ne controlla la coerenza del sockname del client con quello del server.
Open File : La verifica dell'apertura del file nello storage si basa su un flag aggiuntivo, chiamato O_OPEN. Sia O_OPEN che O_OCREATE sono stati implementati come macro.
Read N File : Il while della funzione continua a iterare finché il segnale inviato dal server e' uguale a 1, e procederà a leggere i file inviatigli. Il while termina quando il server invia sia un segnale uguale a 0, corrispondente alla terminazione della lettura dei file, che in caso di errore (segnale uguale a -1).
Append to File : Se l'inserimento del file nel storage e' andato a buon fine, la funzione controlla se ci sono file espulsi. In caso positivo viene stampata una tabella contenente il pathname di tutti i file che sono stati espulsi dallo storage.
Funzioni Opzionali : Le funzioni della parte opzionale, removeFile(), writeFile(), lockFile(), lockFile(), sono state scritte, ma non implementate.

4 Server

La lettura del **File di Configurazione**, viene eseguita attraverso il comando con il seguente formato :

"Usage : ./server -F <fileConfig.txt>"

I parametri presenti nel file includono : numero di thread, numero di file, dimensione della memoria in byte, nome del socket e il file di log. Il file .txt deve avere il seguente formato :

"<identificatore> <numero/nome_file> #<commento descritto>"

Il carattere '#' indica l'inizio di un commento. La lettura del file di configurazione ("configurazione.c/.h") avviene riga per riga. Ogni riga e' tokenizzata per estrarre i parametri, i quali vengono salvati nella struttura creata appositamente "config_t". Nel caso in cui la sintassi non e' corretta il server viene avvertito e sono utilizzati dal programma valori di default.

Il processo principale del server e' costituito dal **Thread Manager**.Questo processo gestisce i seguenti casi :

- l'arrivo dei segnali (attraverso le due pipe 'pipe_hup' 'pipe_int_quit') ;
- l'accettazione di nuovi client ;
- l'inserimento (attraverso la pipe 'pipe_fd') dei client appena la richiesta e' terminata ;
- l'inserimento delle richieste nella coda.

Il messaggio con le informazioni utili per l'esecuzione della richiesta sono inseriti nella struttura "sms_request".
Gli N_TREAD **Thread Worker** vengono attivati nella parte di inizializzazione del server. Questi thread restano in stato di ascolto fino a che almeno una richiesta viene inserita nella coda. Questo viene implementato attraverso la funzione "threadF()".

Le **Signal** sono state implementate seguendo la struttura e la logica esposta nelle esercitazioni. In particolare questo implica che esiste un thread worker (threadSignal()) che lavora esclusivamente sulla base di segnali e pipe.
A questo scopo e' stata definita la struttura "signalHandler_t" che passa al thread le info necessarie per il suo funzionamento. I segnali sono rappresentati da variabili globali, come anche il numero di clienti 'nclient'. 'nclient' e' utilizzato con la variabile 'sigHUP' per consentire la terminazione controllata del server. Non appena il segnale 'sigHup' e' attivato, il server attende la fine dell'esecuzione di tutte richieste dei client già connessi, ma non consente l'arrivo di nuove richieste.

La **Queue** (coda) e' implementata utilizzando la struttura dati "Linked List Lineare" doppiamente linkata con la politica di rimpiazzamento FIFO. Infatti nella struttura "sms_request" sono presenti due puntatori 'son' e 'father'. La gestione della coda si basa sulle seguenti funzioni :

- **createQueue()** : inizializza la struttura della coda ;
- **push()** : inserisce la richiesta dalla head (quindi in cima) ;
- **pop()** : toglie, dalla coda, la richiesta che si trova nel last (quindi in fondo) ;
- **printQueue()** : stampa la coda (quest'ultima e' stata utilizzata unicamente per debuggare).

Le strutture e funzioni sopra riportate della coda sono in "util_server.h/.c"

Anche lo **Storage**, come la Queue, e' implementata utilizzando la struttura dati "Linked List Lineare" doppiamente linkata. I file inseriti nello storage utilizzano la struttura 'node_t' che contiene le info necessarie dei file.
La funzione createStorage() permette l'inizializzazione di tutto lo storage e di conseguenza della struttura 'info_storage_t'.
Le due strutture e le funzioni si trovano nel file "file_storage.c/.h".

L'**esecuzione delle richieste** del thread worker sono indicate attraverso la variabile api_id che identifica il tipo di operazione che dovrà essere processata :

- **api_id=1** : il thread eseguirà la richiesta 'close connection', quindi verificherà che i socketname coincidano, invierà' la risposta e chiuderà la connessione.

- **api_id=2** : il thread eseguirà la richiesta 'open file'. Nel caso del flag O_CREATE il thread crea la struttura file con createFileNode() e la inserisce nello storage insertCreateFile() facendo attenzione che il numero di file sia disponibile, in caso contrario viene effettuata la rimozione dei file con removeFile(), che saranno spediti al client. Invece, nel caso del flag O_OPEN viene settata la variabile del file, 'fdClient_id', con il numero del file descriptor del client, in modo da indicare che il file e' aperto da questo client e che quindi nessun altro può accedervi.
  Qualora il file fosse aperto da un altro client, si attende attraverso un 'while' che l'altro client abbia terminato le sue operazioni.Nel caso di time out viene inviato un messaggio di errore al client.
- **api_id=3** : il thread eseguirà la richiesta 'read file', quindi eseguirà una ricerca del file con la funzione searchFileNode(). Qualora il file fosse presente, ma non ancora aperto, si permette al thread che si occupa della openFile di completare l'operazione. In caso di time out il server invierà un messaggio di errore.
- **api_id=4** : il thread eseguirà la richiesta 'read n file', quindi controlla il valore di n ed esegue la lettura di 'n_read' file.
- **api_id=5** il thread eseguirà la richiesta 'append to file' per aggiungere il contenuto del file attraverso la funzione UpdateFile(). In primo luogo questa funzione controlla che la size del buffer sia inferiore alla dimensione totale dello storage. Se cosi non fosse, il file stesso viene eliminato (se non e' stato ancora modificato). In secondo luogo la funzione controlla la memoria disponibile per verificare la possibilità dell'inserimento del file, in caso contrario viene eseguito l'algoritmo di rimpiazzamento finché non si libera spazio sufficiente per inserire il file. Anche in questo caso e' stato previsto un time out per evitare che il thread rimanga in uno stato di wait permanente.
- **api_id=6** il thread eseguirà la richiesta 'close file', nel caso il file sia trovato attende che le relative operazioni siano state concluse per chiuderlo.

Per elaborare le **Statistiche** finali e' stata creata la struttura 'statistics_t'. E' una variabile globale in modo che sia il file "server.c" che quello contente i thread worker possano accedervi.
Per stampare lo storage e' stata implementata la funzione printStorage(), che stampa in una tabella il path e la size in bytes del file.
La stampa delle operazioni viene memorizzata nel file di log dato, il cui nome e' fornito dal file di configurazione.

5 Makefile

Sono stati implementati il test1 e test2. Per eseguire i test e' necessario lanciare "make cleanall" e successivamente "make test1" o "make test2".
La directory dei test contenente i file da inserire nello storage e' 'SupportTestDirectory'. I file letti dal client saranno inseriti nella directory 'ReadFileFolder'.
L'output dei client nel test2 saranno presenti nei file clientX.txt, dove X e' il numero del client.
L'output del valgrind del test1 si trova nel file output_server.txt
L'output del server si trova nel file di log logfile.txt per il test1 e logfile2.txt per il test2. Nel test2 e' stato rilevato che la stampa del log file non arriva alla stampa della statistica finale.
Nel test2 e' stato testato anche quando la size del file e' maggiore della dimensione dello storage (con il file "SupportTestDirectory/DecimaSiggraph2017.pdf"). Questo file non e' stato possibile caricarlo su github a causa della sua dimensione di oltre 40 MB.