Relazione Progetto Laboratorio Sistemi Operativi

Salvatore Guastella 598631

Indice

1 Premessa

Repository GitHub: https://github.com/salvogs/progettoSOL
Codice terze parti: tra le strutture dati è stata utilizzata una hash table implementata nel file **icl_hash.c** di *Jakub Kurzak* fornitaci durante il corso.
Parti implementate: sono state implementate tutte le parti descritte dalla specifica ad eccezione della compressione (file di log, script statistiche.sh, test3, lockFile e unlockFile, flag -D, LRU ed LFU).

2 Protocollo Comunicazione client-server

Ho deciso di optare per un protocollo di tipo testuale al fine di leggere e allocare i dati strettamente necessari. Alcuni campi hanno una dimensione prefissata, quindi conosciuta sia da client che da server. Il primo campo del payload è sempre l'*operazione*.
La tabella seguente mostra l'associazione dei dati inviati da ogni operazione.

Operazione	Payload
openFile	(operazione)(lunghezza pathname)(pathname)(flags)
closeFile	(operazione)(lunghezza pathname)(pathname)
readFile	(operazione)(lunghezza pathname)(pathname)
readNFile	(operazione)(N file da leggere)
writeFile	(operazione)(lunghezza pathname)(pathname)(dimensione file)(file)
appendToFile	(operazione)(lunghezza pathname)(pathname)(dimensione file)(file)
lockFile	(operazione)(lunghezza pathname)(pathname)
unlockFile	(operazione)(lunghezza pathname)(pathname)
removeFile	(operazione)(lunghezza pathname)(pathname)

Sul file **comPrt.h** sono presenti ulteriori dettagli riguardanti il numero di byte corrispondente a ogni campo del payload. Il messaggio di risposta da parte del server è un byte, in base all'esito dell'operazione (in comPrt.h sono definite le macro delle risposte). Se la risposta è *SENDING_FILE* allora vi sarà associato anche un file (con size ed eventuale pathname).

3 Server

3.1 File Configurazione

All'avvio deve essere passato come paramentro il file di configurazione, nel quale devono essere specificati **tutti** i parametri di configurazione.
Il parsing è implementato dal modulo **configParser.c** e prevede che vengano specificati i seguenti parametri di configurazione:

MAXCAPACITY capacità massima in byte del server

MAXFILENUM numero massimo di file memorizzabili

WORKERNUM numero di thread worker

SOCKETPATH nome da associare al socket

EVICTIONPOLICY politica di rimpiazzamento (0:FIFO,1:LRU,2:LFU)

I valori ai parametri sono assegnati tramite la sintassi "parametro**:**valore". Vi è la possibilià di lasciare righe vuote e commentare inserendo a inizio riga il carattere '#'.

3.2 Struttura

Seguendo la specifica ho implementato il server come un singolo processo multi-threaded secondo lo schema master-worker.
I thread si distinguono in tre tipi: master, worker e logger.
Tra **master** e **workers** la comunicazione avviene tramite una *coda* e una *pipe*.

Il **master** è il tread main del processo server. Tramite la SC `select` si mette in attesa che uno o più file descriptor specificati nel readset siano pronti.
Se il fd pronto è quello del socket allora accetterà una nuova connessione e aggiungerà al readset quello del client connesso;
se il fd pronto è uno dei client, lo inserirà alla coda in modo tale che la richiesta possa essere servita da uno dei workers.
Ovviamente l'accesso alla coda è garantito safe tramite la variabile di mutua esclusione **request_mux** e la variabile di condizione **cond** (sulla quale viene effettuata una **pthread_cond_signal** per risvegliare eventuali workers in attessa).
Il readset della select contiene anche l'endpoint di lettura della pipe in comune con i workers, dal quale può essere letto un fd > 0 sul quale ci si deve rimettere in ascolto (quindi da inserire al readset della select) oppure 0 e -1, necessari per una corretta terminazione del server a seguito della ricezione di un segnale (descritta in seguito).

I thread **worker** si occupano di leggere dalla coda i file descriptor dai quali è possibile leggere, senza bloccarsi, una nuova richiesta o una disconnessione da parte di un client. Una volta gestita una richiesta , il worker scriverà il file descriptor del client sulla pipe, in modo tale che il thread master possa inserirlo nuovamente sul set dei fd da ascoltare.
Anche da parte dei workers viene garantita la lettura in maniera safe dalla coda. Un worker si mette in attesa tramite **pthread_cond_wait** su **cond** quando la coda è vuota. Una volta letto un fd dalla coda, vengono fatte più read, al fine di poter leggere la richiesta inviata dal client. Dopo aver fatto la prima read per leggere l'operazione, seguendo lo schema descritto nella tabella 2, vengono letti i vari dati della richiesta. Dopo aver letto completamente la richiesta, ogni worker si serve delle funzioni implementate in **fs.c** per operare sul file storage.

Il thread **logger** ha il compito di scrivere il contenuto di un buffer (in memoria principale) sul disco. Tramite la funzione **logPrint**, i thread master e worker, hanno la possibilità di effettuare il logging delle operazioni e degli eventi sul buffer, senza effettuare operazioni I/O che potrebbero impiegare anche molto tempo. L'accesso al buffer è garantito in mutua esclusione tramite una mutex (lmux) e una variabile di condizione (lcond). Sul file di log vengono memorizzate: connessioni e disconnessioni dei client, tutte le operazioni effettuate sul file storage (con esito ed eventuali byte processati), file espulsi ed esecuzioni dell'algoritmo di rimpiazzamento.

3.3 Gestione Segnali

Quando il server viene avviato, per prima cosa, si procede all'istallazione dei signal handler per la gestione dei segnali. Vengono utilizzate 2 variabili volatile sig_atomic_t: *noMoreClient* e *noMoreRequest*, che vengono settate a 1 rispettivamente alla ricezione di **SIGHUP** e **SIGINT/SIGQUIT**. Viene ignorata la ricezione di SIGPIPE. Il thread master si assicura dell'arrivo di un segnale, controllando che siano settati i 2 flag.

- noMoreRequest è la guardia del ciclo while del thread master. Una volta arrivato SIGINT/SIGQUIT, il ciclo sarà interrotto solo alla fine dell'iterazione corrente, smettendo quindi di servire richieste,connessioni ecc...
- noMoreClient fa si che il server termini in maniera "tranquilla", permettendo ai client connessi di continuare a mandare richieste e disconnettersi, ma impedendo, tramite la chiusura del socket, la connessione a nuovi client. Il worker che servirà l'ultimo client si occuperà di scrivere sulla pipe il valore **0** che, una volta letto dal master, permetterà la terminazione.

Indipendentemente dal segnale ricevuto il server termina deallocando le risorse, joinando i thread (workers e logger) e stampando il sunto delle operazioni.

Un caso particolare è l'arrivo di un segnale tra il while e la select

```
1 while(!noMoreRequest){
2   ...
3   /*
4       SEGNALE!
5                 */
6   if(select(fd_num+1,&read_set,NULL,NULL,NULL) == -1){
7     ...
```

Vi è il rischio di rimanere in attesa sulla select per un tempo indefinito.
Per evitare ciò ho ritenuto opportuno, una volta arrivato un segnale, far scrivere dal signal handler il valore **-1** sulla pipe, che risveglierà la select in attesa.

3.4 File Storage

La struttura del file storage è implementata tramite una struct di tipo **fsT** che, oltre a contenere informazioni come capacità massima, capacità corrente, variabile di mutua esclusione e informazioni utili per le statistiche, memorizza una coda di file. Ogni file è una struct di tipo **fT** che contiene informazioni come pathname, size, contenuto e altre informazioni fondamentali per garantire la mutua esclusione e per il rimpiazzamento (di cui parlerò in seguito).

3.5 Strutture dati

Ho utilizzato due strutture dati:

- una *hash table* per la memorizzazione vera e propria dei file e le altre operazioni sul file system (rimozione, ricerca, lock ecc...);
- una *coda*, tenuta sincronizzata con la hash table, per mantenere un ordinamento sull'inserimento dei file per le politiche di rimpiazzamento.

3.6 Concorrenza

Il file storage possiede una variabile di mutua esclusione **smux**. Prima di eseguire un'opeazione viene fatta una **pthread_mutex_lock** su quest'ultima, viene cercato il file sulla hash table e, in base all'operazione, viene rilasciata o meno nell'immediato. Operazioni (di fs.c) come **close_file**, **read_file**, **lock_file** e **unlock_file** la rilasciano dopo la ricerca, mentre le restanti operazioni la tengono fino alla fine.
Per garantire la mutua esclusione in lettura/scrittura dei file ho seguito una delle soluzioni del paradigma *lettori-scrittori* fornite durante il corso di teoria, che garantisce un solo scrittore e più lettori allo stesso momento. Ogni file ha due variabili di mutua esclusione (**mux** e **ordering**) , una variabile di condizione (**go**) e due contatori (**activeReaders** e **activeWriters**). Il flag O_LOCK corrisponde alla variabile *fdlock*.
lock_file, se il file esiste, setta **fdlock** = **fdClient** che ha richiesto la lock. Se il file è già lockato da un altro client, fdClient viene messo in coda ai client che sono in attesa di acquisire la lock, per permettere al thread worker di continuare a gestire richieste.
unlock_file, se il file esiste ed è stato lockato dal client che ne ha fatto richiesta, resetta **fdlock** a uno dei client in attesa (se non esiste lo setta a 0).
Quando un client rilascia la lock su un file (o si disconnette), l'eventuale nuovo detentore della lock, che era in attesa della risposta, sarà avvisato con un *SUCCESS*. Se il file viene rimosso verrà avvisato con *FILE_NOT_EXISTS*

3.7 Scelte effettuate

- Un file *non* lockato può essere aperto (con qualsiasi flag) e chiuso;
- Un file *lockato* da un altro client non può essere aperto con il flag O_LOCK (risposta *LOCKED*) ma può essere in ogni caso chiuso;
- Un client deve prima lockare un file per poterlo rimuovere;
- Un client diverso dal detentore della lock che prova a fare una writeFile, appendFile, readFile riceve *LOCKED* come risposta;
- lockFile e unlockFile non necessitano che il file sia stato aperto dal client

3.8 Rimpiazzamento

Ho implementato tutte politiche di rimpiazzamento: **FIFO**,**LRU** e **LFU**. Quando si ha necessità di espellere un file, viene chiamata la funzione **eject_file** che si occuperà, scorrendo la coda dei file memorizzati, di scegliere secondo questi criteri il file da espellere:

- Non deve essere lockato da un client diverso di chi ha causato l'espulsione;
- Non deve essere lo stesso file di quello che si vuole scrivere/appendere;
- Non deve essere vuoto (tranne se l'espulsione è causata dalla openFile);

I file espulsi vengono mandati al client solo se il *dirty bit* (modified) del file è uguale a **1**. Se l'espulsione è causata da una openFile ho scelto di non inviare i file al client ma solo di espellerli.
La politica *FIFO* rimuove dalla testa della coda il primo file che rispetta i criteri sopra citati.
LRU ed LFU non tengono conto dell'ordinamento della coda ma delle variabili **accessTime** e **accessCount**, aggiornate ad ogni accesso al file.

4 Client

Per prima cosa viene effettuato il parsing da linea di comando, implementato tramite **getopt** in *clientParser.c*, che inserisce in una coda tutte le operazioni (e l'argomento) che comporteranno una richiesta al server (non vengono inserite -h,-p,-t,-f).
Esclusivamente i flag **-W** e **-w** accettano path relativi (convertiti in assoluti tramite *realpath* prima di inviarli al server). Tutte le altre operazioni necessitano del path assoluto con il quale il file è

memorizzato nel server.
I flag **-d** e **-D** devono essere utilizzati rispettivamente dopo -r/-R e dopo -w/-W (scrivono sul disco ricreando il percorso dei file a partire dalla directory passata come argomento).
Tutti i flags hanno associati una o più funzioni dell'API. In particolare le letture (-r) e le scritture (-w e -W) necessitano di fare una openFile prima di leggere/scrivere e una closeFile una volta concluso. I flags come -l,-u,-c non hanno bisogno di aprire/chiudere il file ma usano solo le corrispondenti funzioni lockFile, unlockFile e removeFile.
La openFile dell'opzione -W/-w viene effettuata di default con i flags *O_CREATE* e *O_LOCK* settati. Se il server dovesse rispondere *FILE_EXISTS* allora verrà fatta una openFile senza flag (seguita da una appendToFile).
La openFile di -r viene effettuata senza flags.
La funzione readNFiles invia solamente il numero di file che si aspetta di leggere.

4.1 API Client

Tutte le funzioni api della specifica sono state implementate in **api.c**. Ogni funzione invia una richiesta al server e attende una risposta attenendosi al protocollo di comunicazione descritto nella sezione 2.
In base alla risposta ricevuta dal server viene stampato un messaggio su stdout (se -p passato) e viene settato errno al codice di errore opportuno.
Sono presenti anche funzioni di accompagnamento (getFile, sendRequest, getResponseCode, ecc) al fine di rendere il tutto più modulare e leggibile.

5 Gestione Errori

Tutte le funzioni api chiamate dal Client ritornano -1 in caso di errore ed errno viene settato opportunamente. Le funzioni di **fs.c** ritornano la risposta da dare al client (inoltrata dal worker), in caso di errore ritornano *SERVER_ERROR* con il quale viene arrestato tutto il processo server.
In caso di fallimento di una operazione di read/write, il server continua indisturbato se errno è legato alla disconnessione di un client, invece il client termina.

6 Makefile e Test

Sono stati implementati i target test1, test2, test3 che eseguono i rispettivi script in **test/**.
La creazione degli eseguibili *server* e *client* è preceduta dalla creazione dei file oggetto per ogni file *.c* e da quattro librerie statiche (libclientApi.a, libfsApi.a, libdataStruct.a, libutils.a). Il target **clean** rimuove gli eseguibili client e server, **cleanall** rimuove i file oggetto, il file di log, il socket.sk (anche se rimosso comunque dal processo server), le librerie ed eventuali directory dei file letti/espulsi.
Tutti i test avviano il server passando il rispettivo file di configurazione (config1.txt, config2.txt, config3.txt). È necessario cambiare la politica di rimpiazzamento manualmente, modificando i file di configurazione. In particolare il **test2** evidenzia[1] maggiormente le differenze tra le varie politiche.

[1] il file di log mostra tutti i file espulsi, anche i non inviati