UNIX FILE STORAGE SERVER

CORSO SISTEMI OPERATIVI 2020/2021

Michele Montebovi – Mat. 599272

GitHub Repository: https://github.com/Montebovi/Progetto_SOL.git

COME ESEGUIRE IL PROGRAMMA:

```
1) make clean
2) make all
3) make test1
4) make test2
5) make test3
```

INTRODUZIONE

Il progetto consiste nel realizzare un file storage server, il quale mantiene file di qualsiasi formato all'interno della memoria principale. Il protocollo attraverso cui vengono inseriti, rimossi e letti i file dal server è il protocollo di tipo AF_UNIX. Il client comunica ed invia le richieste al server attraverso delle API. Il server esegue le richieste che provengono dai client connessi attraverso un numero di workers configurabile.

CLIENT

Il client viene eseguito con una lista di comandi, che specificano il nome del socket a cui connettersi e le directory utilizzate per la gestione dei capacity misses e per la lettura dei files; la lista dei comandi, inoltre, comprende tutte le operazioni che dovrà svolgere il server.
All'avvio del client vengono caricate all'interno di una struttura le varie opzioni passate come argomento ovvero:

- Abilitazione/Disabilitazione del comando di Help;
- Il nome del Socket a cui connettersi;
- La directory utilizzata durante la scrittura del file;
- La directory utilizzata nel caso di capacity misses del file storage;
- Tempo in millisecondi che intercorre tra due richieste successive;
- Abilitazione/Disabilitazione delle stampe sullo standard output per ogni operazione.

Non appena caricate le opzioni il client stabilisce la connessione di tipo AF_UNIX con il server. Nel caso in cui il server non sia ancora attivo per stabilire la connessione, il client tenta di effettuare la connessione ogni secondo per un totale di due minuti.
I comandi che il client supporta sono:

- **o:** lista di nomi di file da aprire nel server separati da ',’.
- **b:** lista di nomi di file da aprire in creazione e lock nel server separati da ',’.
- **l:** lista di nomi di file su cui acquisire la mutua esclusione separati da ',’ (lock).
- **u:** lista di nomi di file su cui rilasciare la mutua esclusione separati da ',’ (unlock).
- **c:** lista di nomi di file da rimuovere dal file storage separati da ',’.
- **e:** lista di nomi di file da chiudere nel server separati da ',’.
- **r:** lista di nomi di file da leggere dal server separati da ',’.
- **R:** legge n files qualsiasi contenuti nel server.

- **w:** invia al server n file da scrivere prendendoli dalla directory 'dirname'.
- **W:** lista di nomi di file da scrivere nel server separati da ','.

Il cliente effettua analizza i parametri di configurazione identificando quelli di applicazione trasversale come -d -D -, ecc. I parametri relativi ai comandi da inoltrare al server vengono inviati sequenzialmente attraverso una struttura che rappresenta la request.
In particolare ogni comando letto dal client viene passato ad una funzione chiamata *execute_x*, (x rappresenta il comando).
La funzione `execute` ha il compito di tokenizzare l'argomento del comando, ovvero ottenere tutti i parametri del comando, come ad esempio la lista dei nomi dei file per poi passarle all'API.
Le API di ogni singola operazione carica in una struttura chiamata `commandToSvr_t`, che contiene i parametri del comando che poi verrà servito al client.
La struttura `commandToSvr_t`, al suo interno tiene traccia di:

- Il tipo di comando
- I flags richiesti nelle open (lock / create)
- La `sizeData`, utilizzata ad esempio nelle `append`, `write`…
- Il `numFiles` ovvero il numero totale di file di cui eseguire un certo comando
- Il puntatore al pathname del file
- Il puntatore alla directory
- Il puntatore al buffer contenente i dati (es. dati del file da scrivere)

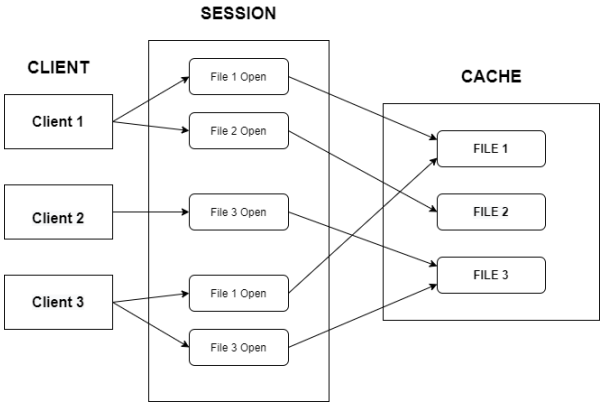
L'API dopo aver caricato i vari comandi all'interno della struttura `commandToSvr_t`, li invia i attraverso la `sendCommand` la quale impacchetta in una stringa ogni comando con i parametri separati dal carattere pipe '|' per poi inviarli al server ed attende il risultato da parte del server.

NOTE SU ALCUNE EXEC DI COMANDI:

- **execute_readN**: questa funzione legge N files qualsiasi dal server; se N è non specificato vengono letti tutti i dati contenuti all'interno del file storage del server. Essa è implementata attraverso la funzione **`readFileFromServer`** la quale, per ogni file, acquisisce dal server il pathname, i dati del file e la relativa size; i dati vengono poi salvati all'interno della directory di lettura (indicata dal comando -d) attraverso la funzione **saveFile**.
- **execute_w:** se viene passato 0 come parametro la `execute_w` ha il compito di prendere tutti i file presenti nella directory, navigando anche nelle sue subdirectory; essa utilizza la funzione `getNumOfFilesOfDirectoryRec`, la quale ricorsivamente conta tutti i file presenti nella directory, visita successivamente ricorsivamente tutte le sottodirectory, e ritorna come valore il numero di file trovati. Successivamente viene chiamata la funzione `getFilesOfDirectoryRec` che inserisce in un array i path di tutti i file presenti nella directory passata come parametro alla funzione. La funzione `execute_w` dopo aver calcolato il numero di files e dopo aver inserito in un array tutti i pathname, fa un ciclo for per invocare l'API per ogni file da scrivere; all'interno del ciclo vengono inviati comandi al server per aprire ogni file con flags **O_CREATE|O_LOCK**, inviare i dati e poi chiudere il file.
- **execute_W:** questa funzione dopo aver tokenizzato l'argomento e trasformato i path in assoluti, fa un ciclo for su tutti i file dei parametri ed utilizza la funzione ***writeFileCore*** passandogli il pathname assoluto del file e le varie opzioni.
- **writeFileCore**: questa funzione prende come argomento il pathname e le opzioni, chiama a sua volta la funzione ***loadFile*** la quale a lato client apre il file indicato dal pathname e carica nel buffer tutti i dati relativi al contenuto del file e la sua dimensione. Dopo aver caricato i dati del file la funzione ***writeFileCore*** chiama l'API relativa (***writeFile***) che avrà il compito di mandare la richiesta dell'operazione da svolgere al server (invia tutti i dati letti).

SERVER

Il server viene inizialmente configurato attraverso un file testuale, il quale indica il nome del socket a cui connettersi, il numero di thread worker del server, la capacità massima del file storage espressa in MBytes e il numero massimo di file che il file storage può contenere.
Il server crea inizialmente N thread worker, e successivamente entra in un ciclo *while(true)* in attesa di ricevere comandi. I comandi una volta ricevuti vengono inseriti all'interno di una coda per poi essere prelevati dagli worker.
I thread worker prelevano dalla coda i comandi inviati dai client e li eseguono.
Il server è strutturato in due strati: il primo strato ha la funzione di gestire e tracciare le sessioni di apertura e chiusura file dei client; il secondo strato gestisce la cache dei file fisicamente memorizzati in memoria.
In questo modo il server è in grado di gestire più file aperti da parte di più client garantendo la mutua esclusione in caso di lock.
Ogni worker dopo aver eseguito un comando invia un messaggio di ritorno per notificare al client il relativo risultato (0 operazione avvenuta con successo != 0 altrimenti).
Le sessioni vengono create ogni qual volta il client invoca la open di un file ed il file non è ancora mai stato aperto dal Client. Le sessioni vengono chiuse ogni qual volta un Client decide di chiudere il file in cui stava lavorando. Un client, pertanto, avrà tante sessioni quanti sono i file da lui aperti.
Quando arriva una richiesta per agire su un file (p.es. lettura o scrittura) il sistema individua la sessione del client per il file specificato ed effettua i relativi controlli.
L'apertura di una sessione avviene verificando se il file è già stato aperto dallo stesso client e se è in lock da un altro client.
Quando un client invia il comando di chiusura, la relativa sessione viene eliminata e il file viene sbloccato se era stato lock.
Lo strato delle sessioni effettua chiamate allo strato di cache per creare file, scrivere i dati o rimuovere file. Anche quest'ultimo strato mantiene traccia del lock e gestisce l'espulsione dei file quando si verifica il caso di capacity misses.
Tutti gli strati garantiscono il corretto funzionamento di azioni concorrenti di più workers con opportuni mutex (sulla coda, sulla struttura dati delle sessioni, sulla cache).



FILES_STORAGE

Il Files_Storage mantiene in memoria temporanea tutti i dati che i client richiedono.
Il Files_Storage è strutturato in questo modo:

- **FileCache**: il FileCache è dove effettivamente vengono memorizzati i Files. È composto da una struttura (`fileItem_t`***)*** che mantiene per ogni file: il pathname, la dimensione totale, il suo stato se è *locked* o *unlocked*, il contenuto del file ed infine data/ora dell'ultimo accesso.

Il FileCache si occupa di tutte le operazioni nei file ovvero: rimpiazzamento e salvataggio del file all'interno della directory in caso di capacity misses, lock e unlock di files, rimozione o inserimento di file all'interno dello storage.

- **FileSessions**: il FileSessions è una struttura che tiene traccia di tutte le sessioni di file aperti dai client collegati al server; ovvero per ogni client tiene traccia di tutti i suoi file aperti.

PARTI OPZIONALI SVOLTE

- Opzione -D svolta in modo da avere un riscontro visivo dei file espulsi dal file storage.
- Prodotto File di Log sia per il client che per il server.
- Realizzato le Lock e Unlock dei file.
- Realizzato il Test 3.
- Rimpiazzo dei file all'interno del File Storage utilizzando politica LRU.