



UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea in Informatica

Prof.: Alessio Conte

Anno Accademico: 2020/2021

Studente: Marzeddu Simone

Matricola: 597134

Corso A

Note iniziali

Il progetto ha richiesto la realizzazione di uno **storage server** dove il server improntato sul multithreading è in grado di accogliere e rifiutare le richieste di connessione da parte di un numero non predefinito di client che sfruttano le funzioni implementate nel file **api.c** per interfacciarsi con esso.

Essendo stato consegnato in un appello successivo a quello di luglio 2021, questo progetto ha richiesto l'implementazione di tutte le funzionalità che erano facoltative in precedenza.

Lo sviluppo del progetto si è appoggiato all'utilizzo di un **repository pubblico** il cui link è riportato di seguito: <https://github.com/SimoneMarzeddu/StorageServer>

Il Server - Configurazione

Come richiesto, il server esegue al suo avvio il parsing di un file di configurazione e sarà l'esito di questa operazione a stabilire alcuni parametri fondamentali per l'esecuzione delle attività sul file storage.

I parametri che vengono inizializzati attraverso il suddetto parsing sono: il numero di thread workers all'interno del server, il nome del socket file, il massimo numero di files che possono essere contemporaneamente registrati sullo storage e la massima dimensione di questo indicata in Bytes.

La formattazione del file di configurazione deve essere strettamente simile alla seguente, se non per l'ordine delle righe ed ovviamente per i valori dei campi:

```
max_no=numero_massimo_di_file_nel_server
max_size=size_massima_del_server_in_bytes
thread_no=numero_di_thread
socket_name=./nome_del_socket.sk
replace_politic=POLITICA_SELEZIONATA
```

È possibile fornire come input al lancio del processo server l'opzione **"-cnfg"** seguita dal nome (separato da uno spazio) del file **".txt"** utilizzato per la configurazione.

Il Server – Thread Manager

Come richiesto il processo server è costituito da un numero variabile di thread, per scelta implementativa, un **thread manager** si occuperà della gestione dei segnali, del parsing sopra esposto, della gestione di strutture dati fondamentali per il server (che saranno riportate di seguito) e dell'amministrazione del socket dal lato server.

Il thread principale comunicherà con quelli restanti attraverso due diverse modalità:

- 1) Una **pipe** senza nome dove i vari worker restituiscono i descrittori di file relativi alle comunicazioni terminate,
- 2) Una **coda** in cui il thread manager inserirà i descrittori di file associati ai vari clients che hanno stabilito una connessione al server con successo e che sono pronti all'invio di richieste verso quest'ultimo.

Il Server – Thread Worker

Il fulcro delle attività svolte da un **thread worker** è la funzione **w_routine** dove, all'avvio o in seguito alla disconnessione di un client, viene prelevato dalla coda di comunicazione con il thread manager un file descriptor relativo al canale di comunicazione con il prossimo client che sarà servito dal worker.

Gli scambi di messaggi (input e output relativi) tra il server ed un client avvengono attraverso le funzioni **readn** e **writen** che si assicurano la corretta ripresa di una lettura o scrittura in seguito ad una possibile interruzione da parte di un segnale.

All'interno di un ciclo infinito si ha quindi il continuo scambio di richieste e risposte con il client, richieste che, all'interno della funzione **do_a_job**, vengono poi interpretate, se corrette, come veri e propri comandi riconoscibili ed eseguibili sullo storage.

Ogni operazione eseguita da un thread worker culmina con l'aggiornamento del **file di log** (descritto di seguito) attraverso una stringa opportunamente formattata.

Il Server – File di Log

Nel file “**log.txt**”, presente nella cartella principale del progetto, è tenuta traccia di ogni operazione terminata con successo; ogni stringa che qui viene stampata (prima della terminazione del server che stampa invece alcune statistiche) è riportata da un worker che vi scrive anche il proprio codice identificativo.

La struttura tipica di una scrittura sul file di log è la seguente:

```
/* FILE DI LOG:
 *
 * STRUTTURA MACRO : START...opx \n opy \n opz \n ...END;stats;s1;s2;...;sk;
 * STRUTTURA GENERICA: op/Thrd_id/Op_num/Succ/File_Path... \n
 *
 * open_Connection : op/Thrd_id/1/(0|1)/Sock_Name
 * close_Connection : op/Thrd_id/2/(0|1)/Sock_Name
 * open_File : op/Thrd_id/3/(0|1)/File_Path/flags
 * read_File : op/Thrd_id/4/(0|1)/File_Path/size
 * read_NFiles : op/Thrd_id/5/(0|1)/Read_no/size
 * write_File : op/Thrd_id/6/(0|1)/File_Path/size/Rplc(0|1)/Rplc_no/Rplc_size
 * append_to_File : op/Thrd_id/7/(0|1)/File_Path/size/Rplc(0|1)/Rplc_no/Rplc_size
 * lock_File : op/Thrd_id/8/(0|1)/File_Path
 * unlock_File : op/Thrd_id/9/(0|1)/File_Path
 * close_File : op/Thrd_id/10/(0|1)/File_Path
 * remove_File : op/Thrd_id/11/(0|1)/File_Path
 */
```

Il Server – Struttura dello storage

Per quanto riguarda invece lo **storage** vero e proprio, su cui l'intero server è edificato, la principale scelta implementativa è ricaduta sull'utilizzo di una **tabella hash**.

Tutte le funzioni necessarie per l'utilizzo della sopra nominata struttura dati sono state implementate direttamente all'interno del file sorgente **server.c**, questo vale anche per i files (la cui astrazione nel server è definita dalla struct **file**), le liste di files (struct **f_list**), le liste di file descriptors relativi ai client e i rispettivi nodi (struct **c_node** e **c_list**) così come per la **coda di priorità** e i rispettivi nodi (struct **p_queue** e **p_queue_node**).

È importante spendere alcune parole sulla coda di priorità poiché è questa il nucleo delle politiche di rimpiazzamento adottabili dal server, in particolare questa si comporta proprio come una coda di tipo **FIFO**, ma subisce delle manipolazioni nel caso in cui la politica (espressamente richiesta nel file di configurazione) sia quella di tipo **LRU**, in questo ultimo caso, infatti, sarà presente una chiamata alla funzione **queue_lru**, per ogni operazione andata a buon fine su un file, che assicurerà l'ordinamento previsto.

È importante precisare che la manipolazione di tutti i dati implementati con l'adozione di queste funzioni può avvenire solo ed esclusivamente attraverso i vari thread del server e dunque da esterni solo sotto comandi opportunamente inviati al server e soggetti ad opportuni controlli.

Tutti gli errori non fatali per il lato server vengono fatti galleggiare verso il chiamante, il server continuerà dunque la sua esecuzione secondo il suo stato attuale.

Il Server – Mutua Esclusione

Il tema degli accessi in mutua esclusione è cruciale in un processo che, come in questo caso, è costituito da una potenzialmente vasta **thread pool** ed è questo il motivo per cui sono molteplici i files e le strutture dati per cui è stato necessario adoperare dei mutex.

La mutua esclusione è stata infatti implementata in accesso al **file di logging**, alle **variabili per le statistiche**, ma anche in accesso ai files dello storage.

Parlando di mutua esclusione sui files dello storage è doverosa una precisazione inerente ad una specifica scelta implementativa:

Le variabili di tipo **pthread_mutex_t** hanno un peso non trascurabile in termini di memorizzazione, questa è la ragione per cui è sembrata impensabile la possibilità di dedicarne una ad ogni singolo files ospitato dallo storage.

Un discorso parallelo al precedente vale invece per l'uso di un solo mutex a livello di tutto lo storage, soluzione che renderebbe decisamente poco proficuo il multithreading riducendone di gran lunga gli aspetti vantaggiosi in tal caso.

La scelta implementativa che è stata ritenuta più opportuna consiste nel garantire la mutua esclusione non tanto per accessi sull'intera `hashtable` o su ogni singolo `file`, quanto su diversi gruppi di files, motivo per cui nella struttura dati che astrae le liste di files (fondamenta della tabella hash) è contenuta anche una variabile di tipo `pthread_mutex_t`; due thread non possono dunque effettuare operazioni sulla stessa lista nello stesso istante.

Tra le varie operazioni che un client può richiedere sui dati presenti all'interno dello storage vi è anche quella di effettuare una lock, un'unlock od una open lock su un dato file; questo tipo di attività è ancora supportata dal server, anche se in maniera più astratta:

Ad ogni file presente sullo storage è di fatto associata una variabile, denominata `lock_owner`, il cui valore sarà equivalente a quello del file descriptor del canale di comunicazione del client che possiede la lock su quel file.

Alla disconnessione di un cliente dal server tutte le lock da lui effettuate vengono rilasciate, il codice relativo a questa attività è quello della funzione `hash_lo_reset`.

Il Server – Gestione dei rimpiazzamenti

Un lato decisamente cruciale dell'implementazione del server è quello relativo alla gestione della capacità ridotta di cui il server dispone.

Per scelta implementativa è stato considerato di trattare distintamente i due casi di `capacity miss` potenzialmente riscontrabili durante il periodo di attività del server:

- 1) Nel caso in cui il numero di files attualmente depositati nello storage sia uguale a quello massimo, previsto in seguito alla fase di configurazione, non sarà più possibile per i clients depositare ulteriori files all'interno del server, un errore sarà fatto galleggiare verso il chiamante,
- 2) Nel caso in cui la dimensione totale di quanto depositato sullo storage sia maggiore o uguale alla massima, prevista sempre in fase di configurazione, verrà avviato un algoritmo di rimpiazzamento con l'obiettivo di riportare i parametri ai valori accettabili e rendere comunque possibile l'ultima modifica, causa della situazione appena descritta.

La funzione che viene chiamata nella situazione citata, per la gestione dei rimpiazzamenti, prende il nome di `hash_replace`.

La politica per la selezione dei files che dovranno essere eliminati dallo storage per far posto a quelli nuovi è sì una politica di tipo FIFO (o LRU in base alla configurazione), ma in fase di implementazione è stato ritenuto più opportuno tenere in considerazione anche lo stato di lock o unlock dell'eventuale file bersagliato dall'algoritmo di rimpiazzamento; pertanto, la "rimpiazzabilità" di un file è vincolata dallo stato di lock o unlock di questo oltre ad essere determinata dalla politica selezionata.

Il Server – Altre scelte implementative

Per quanto il client non possa generare una chiamata della funzione `writeFile` offerta dalla api senza prima generare una chiamata alla funzione `openFile` con flag di apertura in lock o creazione in lock, la scelta implementativa è comunque stata quella di progettare un controllo sul fatto che il client chiamante del comando di scrittura abbia, prima di questa, richiesto e completato con successo un comando di apertura del file secondo le flags richieste dal progetto (OR bit a bit di `O_CREATE` ed `O_LOCK`).

La struttura dati che fornisce l'astrazione sui files contiene infatti tra i suoi campi una lista contenente tutti i file descriptors relativi ai client che hanno come ultima richiesta effettuata e completata con successo nel server quella di aprire il file con flags di creazione o lock, la semplice lettura del contenuto di questa lista determinerà la possibilità o meno di completare correttamente la scrittura del file.

Anche la gestione delle flags nella chiamata di `openFile` è stata soggetta ad una interpretazione magari peculiare rispetto a quella di altri progetti, per questo motivo sarà discussa di seguito:

Il parametro di tipo intero, rappresentante le **flags** e ricevuto in input dalla funzione, disporrà di un totale di quattro valori interpretabili dal server come corretti, la selezione di questi valori non è stata casuale. Considerando infatti la conversione in base due dei numeri: 0,1,2,3 si ottengono tutte le quattro possibili combinazioni di due bit, uno per flag in questo caso:

- a) 0: 00: O_CREATE = 0 && O_LOCK = 0
- b) 1: 01: O_CREATE = 0 && O_LOCK = 1
- c) 2: 10: O_CREATE = 1 && O_LOCK = 0
- d) 3: 11: O_CREATE = 1 && O_LOCK = 1

Il Client

Il client, il cui codice è reperibile nel file sorgente **client.c**, è un processo eseguibile con un'ampia gamma di opzioni opportunamente formattate, da questi comandi cedutigli all'avvio sarà determinato il suo comportamento con particolare riferimento alle sue interazioni con il server; non sono stati aggiunti comandi extra rispetto a quanto richiesto dal progetto; pertanto, non vi sarà una sezione dedicata all'interno della relazione.

È importante evidenziare il fatto che il processo client si interfacerà con il processo server esclusivamente tramite una **API**, il cui codice è reperibile nel file sorgente **api.c**, che genererà delle stringhe formattate secondo il formato richiesto dal server e dipendenti dalle opzioni con le quali il client è stato eseguito.

Tutti i riferimenti ai files presenti nei comandi di avvio del processo client devono essere espressi almeno come path relativo di questi rispetto alla cartella in cui è presente l'eseguibile client, sarà poi mansione del programma quella di recuperare il path assoluto del file che ricordo essere l'**identificativo univoco** di un file all'interno del server.

Esistono diverse richieste che un client può effettuare verso il server e che prevedono in output uno o più files, in questi casi, qualora la cartella di output fosse stata specificata secondo quanto richiesto dal progetto, il programma si occuperà autonomamente della creazione dei files nella cartella o della loro sovrascrittura, generando di fatto dei percorsi all'interno della suddetta.

Test 1

L'esecuzione del primo test **non** evidenzia memory leak e tutte le operazioni richieste dal client terminano con successo.

Lanciare da shell "**make test1**".

Test2

I valori di configurazione richiesti sono stati **modificati** per far sì che il test fosse utile al mostrare correttamente la gestione dei rimpiazzamenti: la dimensione massima di un file è per **scelta implementativa** uguale a 1024 Byte, la dimensione massima dello storage è stata dunque ridotta ad 1 Kbyte.

Il secondo test è stato implementato in due diverse varianti, entrambe con comportamento durante l'esecuzione equivalente a quello atteso; lanciare da shell "**make test2**" e "**make test2var**".

Test3

Il terzo test **non** evidenzia alcun tipo di errore dal lato server, dopo l'esecuzione del test è consigliato il lancio di "**make stats**" per maggiori dettagli sulle operazioni svolte dal server.

Lanciare da shell "**make test3**".

NOTA: tutti gli errori che presentano la stampa sullo standard error della stringa "**ERRORE**" sono generati lato client.

NOTA: oltre alle implementazioni obbligatorie sono state eseguite anche diverse attività extra come l'uso del **repository pubblico** e la possibilità di utilizzare la politica **LRU** per i rimpiazzamenti all'interno del server.