Progetto di Laboratorio di Sistemi Operativi

a.a. 2020/2021

Navari Rebecca 553433

Il progetto consiste nella realizzazione un file Storage Server.

Il progetto che ho creato consta dei file

- Server.c
- Client.c
- Makefile
- config1.txt e config2.txt
- script1.sh e script2.sh

e delle librerie statiche e le relative implementazioni

- myAPI.h
- parser.h
- mylib.h
- icl_hash.h

IL SERVER

Il Server è strutturato secondo il paradigma Master-Worker, in cui il main prende la parte del master ed alcuni dei thread creati fungono da worker.

Il master è colui che, in un ciclo while e dopo aver creato i thread e provveduto alla socket e alla gestione delle interruzioni, accetta le connessioni dai diversi clients che si connettono e delega tutte le altre richieste da parte dei clients agli workers.

Il ciclo continua fino a che il Master non preleva dalla pipe (adibita alla comunicazione tra Master e workers) il valore -1 oppure -2. A questo punto fa una push di tale valore nella coda condivisa con i workers, attende la terminazione dei threads e provvede alla deallocazione della memoria e chiusura della pipe.

I worker threads, come detto sopra, soddisfano le richieste dei clients chiamando le funzioni contenute nella libreria ‘mylib’, e utilizzano per la gestione dei file una tabella hash, i cui dettagli si trovano invece nella libreria ‘icl_hash’.

Essi prelevano dalla coda i file descriptors corrispondenti ai client che hanno instaurato una comunicazione, file desciptors che vengono inseriti nella coda dal Master ogni qualvolta un client esegue una connessione oppure un fd viene restituito da un worker dopo aver eseguito un’operazione richiesta dal client (a meno che tale operazione non sia proprio la chiusura di una connessione: in questo caso il fd non viene restituito al Master e non verrà più inserito).

I worker threads terminano la propria esecuzione nel momento in cui estraggono dalla coda un valore uguale a -1 o a -2. Questo valore viene soltanto "visto" dai workers, che non eseguiranno l’operazione di pop su di esso. Questo elemento della coda verrà successivamente eliminato dal Master.

Un altro thread creato dal Master si sospende con una sigwait in attesa di un segnale SIGHUP oppure SIGINT/SIGQUIT. All'arrivo di uno di questi segnali, tale thread invia al Master un valore negativo (-1 per SIGHUP, -2 per SIGQUIT/SIGINT) e termina. Dopo questa comunicazione il Master a sua volta inserisce il valore nella coda e inizia la terminazione dell'intero Server.
Da segnalare che in caso di segnale SIGHUP, il valore -1 viene inserito in coda, in modo tale che i worker possano soddisfare le richieste provenienti dagli altri file descriptors prima di visualizzare il valore di arresto e terminare; in caso di SIGQUIT/SIGINT, che prevedono una chiusura più repentina, il valore è inserito in testa: i workers terminano subito e sarà il Master ad avvertire i client ancora connessi dell'imminente disconnessione inviando a tutti la stringa "NO" (i client attendono di ricevere "OK" per continuare), chiudendo le connessioni e svuotando la coda di tutti i valori rimanenti.

IL CLIENT

Il file Client.c è costituito prevalentemente da uno switch che effettua, grazie a "getopt", il parsing dei comandi ricevuti come argomento (tutti i comandi, compresi di argomenti che richiedono e caratteristiche, sono visualizzabili lanciando il client con il comando -h, come nello script1.sh), ed in seguito chiama la funzione runClient().
Il client include la libreria parser.h che a sua volta include myAPI.h.

Nella prima libreria ci si occupa di inserire ogni comando individuato dal parsing in una coda con priorità: in testa alla coda vengono inserite le operazione di apertura del file (-o), poi quelle di scrittura (-w, -W), quelle di lettura (-r, -R) ed infine quelle di append (-a).
Questa priorità è stata data in modo da riprodurre quello che dovrebbe essere l'ordine con cui ha senso eseguire questo tipo di operazioni.
Una volta inserite tutte le operazioni previste nella coda, la funzione runClient() scorrerà elemento per elemento e chiamerà la funzione corrispondente. Al termine di ogni operazione, su questa viene chiamata la funzione pop e si provvede alla deallocazione della memoria utilizzata per quell'elemento.

La seconda libreria, myAPI, è quella che implementa tutte le funzioni previste dall'interfaccia API ed alcune altre ausiliarie.
Gli specifici dettagli delle librerie verranno discussi in seguito.

MAKEFILE
Il makefile compila i file Server.c e Client.c e tutte le librerie annesse, e crea i target test1, test2, clean1, clean2.
Il target test1 esegue l'eseguibile di Server.c con valgrind ed il comando richiesto, e fa partire lo script1.sh. Al termine dello script invia al PID del server (memorizzato appositamente nel file t1.PID) il segnale SIGHUP.
Il target test2 esegue server.o, fa partire script2.sh, al cui termine avviene nuovamente l'invio del segnale.

I target clean1 e clean2 sono utili per eliminare tutti gli eseguibili, le socket, le librerie e i file e le cartelle creati dagli script1.sh e scrip2.sh rispettivamente.

GLI SCRIPT
In script1.sh vengono create cartelle e file utili alla simulazione del progetto e poi viene lanciato prima un processo client con il comando -h, in modo da poter visualizzare i comandi utilizzabili e le loro specifiche, poi un altro processo client con tutti gli altri comandi (questo poichè l'utilizzo del comando -h comporta l'arresto del processo client dopo la stampa su stdout).

In scrip2.sh avviene nuovamente la creazione di cartelle e file appositi e vengono lanciati 3 processi client, sufficienti a provocare l'utilizzo dell'algoritmo di rimpiazzo.

CONFIG.TXT
I file di configurazione sono di formato .txt e sono costituiti da 5 righe strutturate come di seguito:
riga 1: NFILE=#file contenibili nello storage contemporaneamente
riga 2: DIM=dimensione massima in un dato momento
riga 3: NT=#worker threads
riga 4: nome file di log
riga 5: nome socket

L'ALGORITMO DI RIMPIAZZO
Tale algoritmo implementa una politica FIFO: nel momento dell'inserimento di un file nella tabella hash, ad esso viene assegnato un numero che rappresenta l'ordine di entrata nella tabella(1 al primo, 2 al secondo e così via).
La tabella memorizza il numero maggiore ed il numero minore presenti in quel momento e i riferimenti degli elementi che sono rappresentati da questi numeri.
Ogni volta che si rende necessario l'algoritmo di rimpiazzo, esso semplicemente chiama la funzione delete sulla chiave key_mean (che è quella corrispondente all'elemento con numero minore) e poi seleziona il nuovo elemento più vecchio e memorizza i suoi dati.
Infatti, anche se non è previsto che un elemento venga eliminato se non perchè vittima dell'algoritmo, ogni volta che la funzione delete viene invocata, essa controlla se l'elemento appena eliminato fosse il maggiore oppure il minore o entrambi, e in ognuno di questi casi provvede ad aggiornare i riferimenti al minimo e/o al massimo.

LE LIBRERIE

PARSER
La libreria parser si occupa di inserire nella coda dei comandi ogni operazione di lettura, scrittura, apertura o append secondo un ordine già spiegato e tiene traccia, se specificati, di quali siano le cartelle in cui leggere o scrivere i file, del tempo di attesa tra un'operazione e la successiva, della socket sulla quale connettersi e del flag_p, che abilita o meno le stampe su terminale. Se questi valori non sono specificati, le cartelle sono

uguali a NULL, il tempo a 0 e il flag_p a 0, la socket invece deve sempre essere specificata.
Rispetto ai comandi richiesti, sono stati aggiunti i comandi -o e -a:
il primo è utile per richiedere la semplice apertura di un file, senza la quale un file non può essere scritto con l'opzione -W (al contrario di -w, in cui la openFile è automatizzata); il secondo è stato inserito per rendere utilizzabile la funione appendToFile e richiede come argomenti un file (che deve esser stato precedentemente aperto) e una stringa da appendere al file.

Una funzione di rilievo in questa libreria è findFile(dirname, n, t), utilizzata per il comando -w, che visita ricorsivamente la cartella specificata fino a che t (numero di file su cui sono state chiamate prima openFile e poi writeFile) non è uguale a n (se n>0) o fino a che non sono stati scritti tutti i file presente in dirname o nelle sue sottocartelle (se n<=0). Per semplicità, la openFile chiamata in questa funzione è invocata sempre con il flag O_CREATE.

myAPI
myAPI implementa l'interfaccia API richiesta.
In questa libreria viene creata una coda composta da tutti gli elementi che sono stati aperti e non chiusi (i file vengono chiusi solo se vittime della politica di rimpiazzo: non implementando le operazioni di lockFile e unlockFile, e conseguentemente di removeFile e dei comandi -l, -u e -c, la scelta è stata di non poter richiedere di rimuovere un file lato Server, poichè non si ha l'esclusività su quel file).
In questa coda gli elementi sono contrassegnati da un flag op, utile per verificare se l'ultima operazione effettuata sia stata la open (op=1) o no (op=0). Tale flag viene impostato a 1 nel momento dell'inserimento nella coda, attuabile solo dalla openFile, e settato a 0 alla fine di ogni altra funzione chiamata su quel file che non sia stata chiusa in seguito ad errori, e che perciò abbia avuto successo.
Questo flag viene poi controllato da writeFile, che non può essere eseguita se op!=1.

All'inizio di ogni funzione, il client per prima cosa comunica al Server il tipo dell'operazione richiesta con una stringa di 7 caratteri (closeC, openFi, readFi, readNF, writeF, append) e attende che venga restituito "OK" dal Server.

Degna di nota, tra le funzioni ausiliarie, è saveFile, che viene chiamata, se dirname!=NULL, a seguito di un rimpiazzo, e fa in modo che nella cartella dirname venga aperto il file vittima e su di esso venga scritto il contenuto restituito.

ICL_HASH
icl_hash si occupa della gestione della tabella hash, riprende la libreria presentata a lezione ma viene modificata su alcuni aspetti per conformarsi meglio all'esecuzione del progetto.
Sono state alterate alcune funzioni e ne sono state aggiunte altre (come hash_ret_data(), che data una tabella ed una chiave restituisce il contenuto dell'elemento individuato dalla chiave).
La libreria è commentata.

MYLIB
Essa è la libreria utilizzata dai worker threads.
Un worker thread riceve, dal fd che ha prelevato, uno dei codici suddetti e chiama la funzione corrispondente. A questo punto avviene la comunicazione con il client, a seconda di ciò che deve accadere per quell'operazione.
In questa libreria si hanno anche quattro strutture diverse: coda, arg_s, arg_sig, conf.
Coda è la struttura grazie alla quale si implementa la coda dei file descriptors; arg_s è la struttura degli argomenti della funzione di inizio dei worker threads, che comprende l'indice del thread, il file descriptor di scrittura della pipe, utilizzato dai worker, e il file descriptor di lettura della pipe, utilizzata dal Master;
arg_sig viene utilizzata per passare gli argomenti della funzione di inizio del thread che attende i segnali, e comprende l'fd di scrittura della pipe e la mschera dei segnali. conf, infine, è utilizzata per memorizzare i dati prelevati dalle config1.txt e config2.txt e comprende i campi

- NFILE (numero di file contenibili nello storage contemporaneamente)
- DIM (dimensione massima in un dato momento)

- NT numero di worker threads
- fileLog nome del file di log sul quale scrivere alcuni dati durante l'esecuzione del porcesso server
- sock nome della socket su cui connettersi

Nel computo dei file contenuti in un dato momento sono esclusi i file solo aperti, cioè inseriti a seguito di openFile e mai scritti tramite writeFile o appendToFile.
La stessa cosa avviene per il conteggio della dimensione, poichè questi file sono tutti vuoti.
Si tiene traccia comunque del massimo numero di file inseriti e della dimensione raggiunta in totale, oltre che del numero di rimpiazzi. A seguito di ogni operazione viene scritto sul file di log un piccolo riassunto di quest'ultima, che comprende il nome dell'operazione, il file di riferimento, il numero di byte letti, se si è verificato un rimpiazzo e in tal caso il numero di byte restituiti al client e la vittima.

Come specificato per gli appelli estivi, alcune funzionalità opzionali non sono state sviluppate, ad esempio

- script statistiche.sh
- il target test3
- le operazioni lockFile, unlockFile e di conseguenza removeFile e i comandi -l, -u, -c

mentre altre sono state realizzate. Infatti sono presenti

- la scrittura sul file di log dei dati
- l'opzione -D, che è supportata
- la conseguente scrittura su disco dei file vittima di rimpiazzo

Per compilare ed eseguire il progetto è sufficiente

1. lanciare il comando make, che compila tutti i file necessari
2. lanciare il comando make test1 oppure il comando make test2, per verificare le simulazione del target test1 oppure del target test2

è inoltre possibile eliminare eseguibili, file di log eccetera, frutto di test1 e test2 quando non più necessari, con, rispettivamente, make clean1 / make clean2.

Link repository github https://github.com/hiimReb20/Sol21