UNIVERSITÀ DI PISA

Corso di Laurea in Informatica
Anno Accademico 2021/2022

Relazione del progetto per il corso di

Sistemi Operativi e Laboratorio

Gianluca Luparini
MAT. 558990
Corso A

1. Introduzione

Il progetto è stato sviluppato in ambiente Fedora Linux 36 (64 bit) e testato su quest'ultimo, in aggiunta all'ambiente Linux Xubuntu, fornito durante il corso, virtualizzato con Gnome Boxes (configurato con 2 Cores e 4GB di RAM).

Il codice del progetto è disponibile anche in un repository pubblico presente su GitHub (https://github.com/DraconYale/ProgettoSOL-20-21).

Sono state implementate tutte le funzioni opzionali dichiarate nella consegna (generazione del file di log da parte del Server, script *statistiche.sh*, target *test3* del Makefile, operazioni di *lockFile* e *unlockFile*, opzione *-D* del Client). In aggiunta, è stata implementata la politica di rimpiazzamento LRU, oltre alla politica FIFO. Non è stata implementata la compressione dei file.

L'implementazione della tabella hash (src/icl_hash.c e headers/icl_hash.h) utilizzata dallo storage è codice di terze parti, proprietà di *Jakub Kurzark*.

2. Server

2.1. File di configurazione

Per essere avviato, il server richiede, come argomento, uno dei file di configurazione forniti insieme al codice nella cartella *config/* e nelle sue sottocartelle. Questi file sono in formato *.txt* e devono essere formattati come segue:

Worker Number =	Numero di thread worker da avviare
Storage Size =	Capacità massima del server (in Bytes)
Storage File Number =	Numero massimo di file memorizzabili dal server
Path To Socket =	Path del socket utilizzato dal server
Replacement policy =	Politica di rimpiazzamento (0 = FIFO, 1 = LRU)
Path to log file =	Path del file di log generato dal server

Nel caso il valore inserito in "Replacement policy" sia diverso da 0 (FIFO) o 1 (LRU), il server sarà avviato con la politica di rimpiazzamento FIFO.

2.2. Segnali

Prima ancora di leggere il file di configurazione, il server maschera i segnali SIGHUP, SIGINT e SIGQUIT e avvia un thread dedicato alla gestione di tali segnali. Il Signal Handler così creato manipola due variabili globali *volatile sig_atomic_t*: ***term*** e ***blockNewClients***, entrambe inizialmente settate a 0. Una volta ricevuti i segnali SIGINT o SIGQUIT, *term* viene settato a 1 e il server termina il prima possibile (eseguendo comunque il cleanup della memoria). Alla ricezione del segnale SIGHUP, il server non accetta più connessioni e finisce di elaborare le richieste dei Client ancora connessi.

La funzione di cleanup viene registrata all'avvio del server tramite la chiamata della funzione ***atexit(cleanup)***.

2.3. Coda richieste e concorrenza

La coda delle richieste è implementata come una coda circolare e l'accesso alla stessa avviene in mutua esclusione da parte dei Workers e del Manager. Il Manager aggiunge alla coda i file descriptors dei Client che vogliono inviare una richiesta e i Workers che tentano di leggere da essa vengono risvegliati con una *broadcast* non appena la coda non è vuota. Una volta scaricato dalla coda il file descriptor, i Workers attendono un messaggio da parte del Client corrispondente e, ricevuta la stringa contenente la richiesta, essi utilizzano un'interfaccia per comunicare con lo Storage del Server e soddisfare tale richiesta.

L'accesso allo Storage e ai suoi file avviene sempre in mutua esclusione grazie ad una lock lettori-scrittori (*src/locker.c* e *headers/locker.h*). I file sono salvati in una tabella hash e vengono identificati dal loro path assoluto.

Gli errori che vengono generati da una richiesta nello Storage vengono inviati prima al Worker che gestisce tale richiesta (scriverà sul file di log l'esito dell'operazione) e successivamente al Client (che stamperà in output le informazioni riguardanti tale errore, nel caso sia stato avviato con l'opzione *-p*). In caso di codice di errore "-2" (errore fatale), il Server termina.

2.4. File di Log

Al suo avvio, il Server crea un file di log con il path specificato nel file di configurazione (il formato del log scelto in fase di sviluppo è stato il *.txt*) . Qui vengono registrati da parte dei Thread gli esiti di ogni richiesta dei Client, i Bytes letti/scritti, il momento in cui un client si connette/disconnette e parte del sunto delle informazioni dello Storage al momento della chiusura del Server. L'accesso a tale file avviene in mutua esclusione.

In particolare, il formato della stringa che contiene le informazioni sull'esito delle operazioni è il seguente:

[<*tempo*>] Thread <*TID*>: <*nome_operazione*> <*parametri*> exited with code: <*codice_return*>. 'Read size: <*numero_bytes_letti*>/Write size: <*numero_bytes_scritti*>'.

Con <*tempo*> si intende il tempo trascorso dall'avvio del server. "*Read size:*" e "*Write size:*" sono presenti solo con le operazioni di Read/Write.

I vari file di log possono essere analizzati tramite lo script *statistiche.sh* (fornito in *script/statistiche.sh*).

3. Client

I Client inviano richieste al Server tramite un'interfaccia (*src/interface.c* e *headers/interface.h*) e devono essere avviati con l'opzione *-f <path_socket>* per poter comunicare con il Server tramite il socket *path_socket*.

Nel caso siano presenti operazioni di lettura e il Client sia stato avviato con l'opzione *-d* <*dirname*>, i file letti vengono salvati su disco nella cartella *dirname.* Invece, se sono presenti operazioni di scrittura e il Client è stato avviato con l'opzione *-D* <*dirname*> i file espulsi dallo Storage del Server sono salvati su disco nella cartella *dirname*. In entrambi i casi, per poter salvare i file su disco, viene ricreato il path per ognuno di essi (funzione presente in *functions.h*).

Come da specifica, se il Client non riesce ad ottenere una lock tramite la chiamata di lockFile, esso attende fino a quando non viene resettata dal detentore della lock. Ciò è stato implementato in modo che il Client la richieda ciclicamente: se l'errore ricevuto è EPERM ("EPERM 1: operazione non permessa"), esso la richiederà ripetutamente finché non otterrà la lock o l'errore inviato dal Server sarà diverso da EPERM (in questo caso, fallirà).

In caso di errore fatale all'interno dello Storage, il Client termina.

4. Possibili codici di errore

I codici di errore che possono essere restituiti dal Server o dal Client sono stati scelti consultando il comando "*errno -l*", disponibile nel pacchetto *moreutils* (*sudo dnf install moreutils* in ambiente Fedora Linux). Di seguito, una lista che li enumera:

EPERM 1	Operazione non permessa
ENOENT 2	File o directory non esistente
EAGAIN 11	Risorsa temporaneamente non disponibile
ENOMEM 12	Impossibile allocare memoria
EACCES 13	Permesso negato
EEXIST 17	File già esistente

EINVAL 22	Argomento non valido
ENAMETOOLONG 36	Nome del file troppo lungo
EISCONN 106	Il socket di destinazione è già connesso
ENOTCONN 107	Il socket di destinazione non è connesso

5. Makefile

Nel Makefile, sono presenti tutti i target necessari per generare gli eseguibili e condurre i test richiesti dalla consegna. Il target *all* è quello di default (quindi è possibile compilare il codice semplicemente utilizzando il comando *make* all'interno della cartella del progetto). I target *test1*, *test2* e *test3* compilano e lanciano i test corrispondenti (per ognuno di essi, vengono generati dei file randomici). I target *clean* e *cleanall* ripuliscono le cartelle *bin/* e *obj/* ed eliminano eventuali file generati dopo i test ed, eventualmente, anche i file di log.

Test 1: vengono testate le operazioni dell'interfaccia. Come risultato, non sono attese vittime da parte dell'algoritmo di rimpiazzamento, ma saranno presenti dei file letti salvati su disco.

Test 2: i Client inviano solo richieste di Write in modo da costringere lo Storage a espellere vittime. Dunque, il risultato atteso consiste in file vittima salvati su disco.

Test 3: come nel test 2, i Client inviano solo richieste di Write (i file vittima non sono salvati su disco). Lo script *test3.sh* utilizza lo script *test3Clients* per mantenere almeno 10 connessioni. Le cartelle contenenti i file da inviare vengono scelte randomicamente con il comando bash "*shuf*". I risultati attesi consistono nella generazione di molti errori registrati nel file di log (writeFile fallite a causa di permessi di accesso) ed un alto numero di vittime da parte dell'algoritmo di rimpiazzamento.