UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea in Informatica

Progetto di Laboratorio di Sistemi Operativi

Sistemi Operativi e Laboratorio

Prof. Alessio Conte

Giacomo Trapani
600124 - Corso A

Anno Accademico 2020/2021

1 Premessa.

Il progetto consiste nella progettazione e realizzazione di un **file storage** gestito da un server multithreaded in grado di comunicare con dei client attraverso le funzioni implementate all'interno della API.

Vengono implementate le seguenti parti opzionali: flag -D, funzioni adibite alle operazioni di lock/unlock su file, test3, file di log, script statistiche.sh, politiche di rimpiazzamento LRU, LFU. Il codice viene sviluppato all'interno di un repository pubblico.

2 Server.

Il server dipende dal file di configurazione il cui nome gli viene passato come unico argomento da linea di comando; il file passato dovrà rispettare la seguente sintassi, altrimenti il server non si avvierà:

```
NUMBER OF THREAD WORKERS = <num_workers>
MAXIMUM NUMBER OF STORABLE FILES = <max_files>
MAXIMUM STORAGE SIZE = <max_size>
SOCKET FILE PATH = <path/to/socket>
LOG FILE PATH = <path/to/log>
REPLACEMENT POLICY = <{0, 1, 2}> # 0 FIFO, 1 LRU, 2 LFU
```

Non vengono permesse righe vuote, un numero di spazi non standard o argomenti non validi (i.e. una stringa dove ci si aspetterebbe un valore numerico).

Struttura interna. Al momento dell'avvio del programma, il server maschera i segnali SIGHUP, SIGINT, SIGQUIT e ne affida la gestione a un thread dedicato: questo si occupa di segnare il flag "*no_more_clients*" (quando settato, il server smette di accettare nuove connessioni, finisce però di occuparsi delle richieste dei client connessi prima di terminare) nel caso in cui riceva il primo, "*terminate*" negli altri casi (il server termina lasciando il sistema in modo consistente non appena possibile). A questo punto vengono inizializzate le strutture dati necessarie (si sceglie di terminare se anche solo una delle operazioni necessarie non ha successo).

Il server funziona seguendo il modello manager-worker; il manager (i.e. il "main") si mette in ascolto chiamando la *select* e - basandosi sul descrittore pronto - distingue i seguenti casi:

- si sta leggendo dalla pipe di comunicazione tra thread workers e thread manager (si legge un valore diverso da zero se e solo se il descrittore del client deve essere riaggiunto all'insieme di quelli da cui si sta leggendo);
- si sta leggendo dal descrittore del socket (in questo caso si gestisce l'arrivo di un nuovo client);
- si sta leggendo dal descrittore di un client (in questo caso, il descrittore viene inviato alla coda concorrente in modo che poi un worker possa occuparsene e si elimina dall'insieme di quelli da cui si sta leggendo).

Thread workers. I thread workers si occupano di gestire interamente una richiesta ricevuta da un client traducendo il messaggio inviato dal client attraverso le funzioni implementate nella API client-server, le letture e le scritture vengono gestite chiamando i metodi *readn*, *writen* forniti a lezione. I workers - all'interno di un loop infinito - rimuovono dalla coda concorrente il descrittore messo in coda dal manager, leggono la richiesta inviata facendone il parsing e la traducono in una funzione sul filesystem implementato; a questo punto, seguendo un modello richiesta-risposta, invia al client il risultato e potenzialmente (sulla base del tipo della richiesta) si occupa di inviare anche i file letti o le vittime dell'algoritmo di rimpiazzamento; nel caso in cui il descrittore letto sia "0", il thread termina (è questo il messaggio di terminazione inviato dal manager).

Gestione degli errori. Gli errori vengono gestiti quando possibile, ma nel caso di errori ritenuti fatali (i.e. tutti gli errori che lasciano lo storage in uno stato inconsistente o che non permettano a un worker di occuparsi di una richiesta) si sceglie di far terminare il programma.

2.1 Storage.

L'elemento chiave dell'intero progetto è il filesystem implementato (si faccia riferimento a *src/storage.c* per i dettagli implementativi) dichiarato come una struct opaca (i.e. accessibile solo mediante le funzioni messe a disposizione dall' interfaccia); al suo interno i file vengono salvati come entries di una tabella hash le cui coppie chiave-valore sono formate dal nome del file (definito come il suo path assoluto) e da quello che concretamente è il file (definito come una struct di tipo "stored_file_t").

Accessi. Gli accessi allo storage vengono moderati da una read-write lock write-biased (si faccia riferimento a *src/data_structures/rwlock.c* per l'implementazione) in modo tale da permettere accessi in mutua esclusione o a più lettori alla volta o a un singolo scrittore evitando dunque potenziali colli di bottiglia: si accede in scrittura allo storage se e solo se l'operazione può modificare il numero di files (e.g. a seguito delle operazioni di "Storage_writeFile" e di "Storage_appendToFile" che possono far partire l'algoritmo di rimpiazzamento, della "Storage_openFile" se viene richiesta la creazione di un file o della "Storage_removeFile" che ne richiede la effettiva cancellazione).

Lo stesso tipo di lock viene utilizzato anche all'interno dei file salvati: si accede in scrittura se e solo se un parametro ne viene modificato, in lettura altrimenti.

Gestione degli errori. Si gestiscono gli errori facendoli galleggiare verso il chiamante; le funzionalità implementate restituiscono un valore definito come "OP_FAILURE" a seguito di errori non fatali (e.g. quando la semantica di una funzione non viene rispettata), "OP_FATAL" quando l'errore è di fatto fatale e lascia il sistema in uno stato non consistente (e.g. quando una operazione di lock fallisce).

Politiche di rimpiazzamento. Essendo lo storage gestito come una cache di file, esiste la possibilità di incorrere in *capacity miss*. È evidente che si possa avere capacity miss solo a fronte di 3 delle operazioni messe a disposizione: "Storage_openFile" (se viene richiesta la creazione di un nuovo file, il numero di files salvabili potrebbe superare il limite imposto), "Storage_writeFile" e "Storage_appendToFile" (possono scrivere all'interno del server una quantità di dati al momento non disponibile); si sceglie di non far partire l'algoritmo di rimpiazzamento nel caso della "Storage_openFile" bensì di restituire un errore.

Vengono implementate le politiche di rimpiazzamento FIFO (che trova una vittima in $O(1)$), LFU e LRU (che la trovano in $O(nlogn)$): la prima si occupa banalmente di eliminare il primo file salvato all'interno dello storage (in ordine cronologico), la seconda e la terza richiedono di ordinare sulla base o della frequenza o del tempo di utilizzo dei singoli file (viene implementato con una chiamata alla funzione di libreria *qsort*).

3 Client.

Il client è un programma che - a seguito di una analisi degli argomenti passati da linea di comando - manda al server le richieste opportune interagendo con la API richiesta dalla specifica a patto che questi siano validi (si faccia riferimento a *src/client.c:validate* per la semantica dei singoli flag); si richiede che gli argomenti passati a ogni comando siano validi, che ci sia uno spazio vuoto tra il comando e l'argomento (e.g. "-f sockname" e non "-fsockname"), che ogni comando sia valido e che - nel caso in cui il comando richieda esplicitamente il nome di un file - venga passato il suo path assoluto. Ogni comando viene eseguito nell'ordine con cui viene passato come argomento al programma.

Gestione degli errori. Si ricorda che l'interfaccia verso il server mette a disposizione un flag booleano "exit_on_fatal_errors" per decidere quale debba essere la risposta del client a fronte di una risposta segnalante il fatto che lo storage sia in una situazione di inconsistenza. Si utilizzano delle variabili globali per denotare tutte le risorse da rilasciare al momento della terminazione qualsiasi sia la sua causa.

Gestione dei file espulsi o letti. A fronte di una espulsione o di una lettura che implichi il salvataggio dei dati sul disco, viene ricreato ricorsivamente all' interno della directory scelta l'intero percorso verso ciascuno dei singoli file (il comportamento è analogo a quello del comando "mkdir -p").

4 Interfaccia per interagire col server.

Come richiesto dalla specifica, viene messa a disposizione una interfaccia per interagire col server (si faccia riferimento a *src/server_interface.c* per i dettagli implementativi) che si occupa di convertire le richieste del client in richieste opportunamente leggibili dal server. Per ognuna delle operazioni implementate viene infatti definito un campo di una enum "opcodes_t" corrispondente e si invia al server un buffer contenente - in primis - questo valore (si faccia riferimento alla documentazione per la semantica dei singoli metodi).

openFile. Nonostante richiedere la creazione di un file possa causare capacity miss all'interno dello storage, il prototipo fornito per questa funzione non prevede la possibilità di salvare su disco all'interno di una qualche directory il file espulso: per questo motivo, si sceglie di non far partire l'algoritmo di rimpiazzamento a seguito di questo tipo di richiesta ma si preferisce restituire un messaggio di errore opportunamente definito.

lockFile. La specifica richiede che il client si blocchi su questa richiesta se il lock sul file specificato non può essere acquisito poiché posseduto da un altro client: si sceglie di implementare questa cosa rimettendo in coda la richiesta finché non ha successo o fallisce per un motivo diverso da quello appena citato.

Gestione degli errori. Si sceglie di far galleggiare gli errori verso il chiamante e si mette a disposizione un flag booleano "exit_on_fatal_errors" che, se settato, forza la terminazione a seguito di un errore fatale all'interno dello storage.

5 Makefile.

Viene messo a disposizione un Makefile che fornisce - tra gli altri - i target "all" (per creare tutti gli eseguibili), "clean" e "cleanall" (per pulire la directory), "test1" (che lancia il test1), "test2" (che lancia il test2) e "test3" (che lancia il test3).

test1. Il Makefile crea il file di configurazione necessario per avviare il server coi parametri richiesti dalla specifica, dopodiché esegue il file *script1.sh* che si occupa di avviare il test una volta per ogni politica di rimpiazzamento implementata e di eseguire almeno una volta ognuno dei comandi implementati. Durante l'esecuzione, lo script crea dei file da inviare al server, operazione che potrebbe occupare un po' di tempo a seconda della macchina su cui lo si sta eseguendo.

test2. Il Makefile crea il file di configurazione necessario per avviare il server coi parametri richiesti dalla specifica, dopodiché esegue il file *src/script2.sh* che si occupa di avviare il test una volta per ogni politica di rimpiazzamento implementata utilizzando dei file di input creati appositamente per far partire piú volte l'algoritmo di rimpiazzamento. Durante l'esecuzione, lo script crea alcune cartelle contenenti file da inviare al server.

test3. Il Makefile crea il file di configurazione necessario per avviare il server coi parametri richiesti dalla specifica, dopodiché esegue il file *src/script3.sh* che si occupa di avviare il test una volta per ogni politica di rimpiazzamento implementata assicurando - come da specifica - che ci siano sempre almeno 10 client connessi e che questi inviino (almeno) 5 richieste. Durante l'esecuzione, lo script crea alcune cartelle contenenti file da inviare al server.

6 Logging.

Durante l'esecuzione, il server scrive all'interno del file di logging specificato dal file di configurazione accedendovi in mutua esclusione. Precisamente, scrive l'identificativo del thread worker che ha gestito la richiesta, il tipo di richiesta, l'esito e - dove è rilevante - il numero di bytes letti, scritti, il numero di vittime in seguito a una delle operazioni che può far partire l'algoritmo di rimpiazzamento; salva inoltre i dati rilevanti per il server, come il descrittore del nuovo client connesso, il numero di client connessi al momento di una nuova connessione e - al momento della terminazione - il numero massimo (raggiunto) di file salvati, la massima dimensione (raggiunta) dello storage in MB.

statistiche.sh. Viene messo a disposizione lo script *src/statistiche.sh* per effettuare il parsing del file di log creato durante l'esecuzione che ne stamperà un sunto.