# Progetto di Laboratorio di Sistemi Operativi

a.a. 2020-21

Lorenzo Amorelli 601241

July 13, 2021

# Il codice e l’esecuzione

All’interno della cartella principale si trovano 3 cartelle utili. La cartella Server contiene tutto il codice necessario ad eseguire, appunto, il server. Lo stesso vale per la cartella Client. Entrambe contengono un Makefile utilizzato per una compilazione indipendente dei due sistemi. Nella cartella principale si trova un terzo Makefile, che può essere utilizzato per compilare sia Client che Server (*make*) e per avviare i test richiesti (*make test1*, *make test2_1*, *make test2_2*). E’ inoltre possibile ripulire le cartelle dopo un test/lavoro tramite *make cleanall* o *make clean*.
In caso di una esecuzione stand alone il server scriverà sullo standard output, alla fine dell’esecuzione numero di file massimo memorizzato nel server, la dimensione massima in Mbytes raggiunta dal file storage, il numero di volte in cui l’algoritmo di rimpiazzamento della cache è stato eseguito per selezionare uno o più file “vittima” e la lista dei file contenuti nello storage al momento della chiusura del server, come richiesto.
I client scriveranno invece, sempre sullo standard output, le seguenti informazioni: tipo di operazione, file di riferimento, esito e, dove è rilevante, i bytes letti o scritti.
Nel caso dei test il server verrà avviato con valgrind e scriverà l’output su un file di log che verrà creato nella cartella principale del progetto.

# Note di rilievo relative alla struttura del server

Il codice che implementa il server è strutturato nel seguente modo:
**main.c**:contiene il sorgente con la funzione main. Da qua, nell’ordine, viene parsato il file di configurazione (usando **server_config.c/h**), inizializzate le strutture dati per la gestione della coda di richieste in entrata e del filesystem, inizializzate le pipe per la comunicazione tra threads e creati/lanciati i thread master (per la gestione delle richieste in entrata, implementata tramite *select*), signal handler (per la gestione dei segnali di terminazione) e i vari workers (per la gestione delle richieste e l’esecuzione delle varie operazioni richieste dai client).

Si è scelta una soluzione Master-Workers con thread pool ed è a questo proposito che si è usata una pipe tra i thread workers e il thread master: per l’inserimento dei descrittori dei client, ancora attivi, all’interno dell’insieme di descrittori utile alla select.
Per la comunicazione inversa, tra thread master e thread workers è stato invece usato un buffer di richieste che viene riempito dal master e svuotato dai workers. Il buffer (buffer ad anello, **generic_shared_buffer.h**) è acceduto concorrentemente tramite un protocollo atto a far attendere la lettura se il buffer risulta vuoto e a far attendere la scrittura se il buffer risulta pieno.
Per la gestione della terminazione tramite segnali, un thread gestore è in ascolto (su *sigwait*) dei segnali di terminazione ed una volta ricevuto informa il master attraverso una pipe, svegliando così la *select*.
Per quanto riguarda la terminazione con segnale *SIGHUP* (terminazione soft), questa avviene una volta che il valore *currentClientConnections.value*, condiviso tra master e workers con un protocollo Lettori/Scrittori fair per entrambi (ottenuto tramite l’utilizzo di **shared_values_util.c/.h**), diventa 0.

**connection_util.c/h**:contiene il sorgente che si è ritenuto utile a ciò che riguarda la connessione, interna ed esterna. Contiene quindi le definizioni delle variabili relative alle pipe sopra descritte, alla coda di richieste, la definizione della struttura dati per l’astrazione delle richieste da parte di un client, delle funzioni utili alla comunicazione con l’esterno (*getClientRequest, parseMessage*). Inoltre realizza la comunicazione tra il front-end del server (implementato dal main) e il middleware (implementato dal *filesystem_util.c/h*).

**filesystem_util.c/h**: contiene il sorgente relativo al lato middleware e al back-end del server. Il back-end è l’astrazione di tutte le funzionalità utili alla inizializzazione / modifica / distruzione del filesystem. Il middleware usa le funzioni del back-end per eseguire le operazioni richieste dai client; questo lato del server risponde in modo biunivoco alle operazioni presenti all’interno dell’API implementata lato client.

# Filesystem

Astrazione implementata dalle funzioni e le strutture dati che si trovano in **filesystem_util.h**.
Permette l'esecuzione delle richieste provenienti dai client in modo concorrente. Un file all'interno dello storage è un'istanza di tipo *file_t* ed è un elemento della lista di file mantenuta dall'istanza *fs* di tipo *Filesystem*.
La struttura *file_t* contiene due mutex (*mux, order*) e una variabile di condizione (*Go*) utili all'implementazione del protocollo Lettori/Scrittori fair per entrambi i ruoli.
La struttura *Filesystem* contiene una mutex (*mux*) utilizzata per l'accesso in mutua esclusione al filesystem. Quando la lista di file deve essere acceduta per la ricerca di un file al suo interno, l'esecuzione dell'operazione inizia con una lock su *mux* per evitare che qualsiasi altro thread possa modificare il contenuto durante la ricerca. A meno dell'operazione di *openFile* una ricerca senza risultato porta all'immediato sblocco della *mux*. In caso di ricerca con risultato, il proseguo dell'esecuzione dipende dall'operazione che si sta eseguendo.
In linea generale, è possibile riassumere nel seguente modo:
- per operazioni che creano file all'interno dello storage è necessario agire nel seguente ordine <acquisire fs, cercare file/creare file, acquisire file, rilasciare fs, modificare file, rilasciare file>.
- per operazioni che distruggono file all'interno dello storage è necessario agire nel seguente ordine <acquisire fs, cercare file, acquisire file, distruggere file, rilasciare fs>.
- per operazioni che non modificano fs o file all'interno dello storage è necessario agire nel seguente ordine <acquisire fs, cercare file, acquisire file, rilasciare fs, leggere file, rilasciare file>.
- per operazioni che modificano file all'interno dello storage è necessario agire nel seguente ordine <acquisire fs, cercare file, acquisire file, scrivere file, rilasciare file, rilasciare fs>. Si noti che il fs viene rilasciato solo alla fine della scrittura. Questo per non avere conflitti con eventuali procedure di eviction concorrenti.

Per la ricerca di file all'interno dello storage si utilizza una libreria fornita dai docenti durante il corso (**icl_hash.c/h**). Questa implementa le funzionalità necessarie alla creazione/utilizzo/distruzione di una tabella hash. Il metodo di utilizzo che è stato scelto è basato sull'uso del path del file come chiave e il puntatore al file come valore.

Sono state implementate due politiche per la memorizzazione/eviction dei file:
- FIFO: i file nella lista non modificano la loro posizione. L'unica modifica che la posizione di un file può subire è in seguito alla rimozione di esso dalla lista.
-LRU: i file nella lista vengono posizionati in coda ogni volta che vengono utilizzati (questo non vale per l'operazione di readNFiles, vedi paragrafo successivo). In questo modo, quando avviene una eviction, è possibile partire dalla testa della coda per rimuovere tutti i file utilizzati meno recentemente.

# Il client

Il client è un processo single thread che utilizza una API per effettuare operazioni sul server. Le operazioni da effettuare vengono passate da linea di comando e parsate (**cmdLineParser.c/h**) in modo tale da ottenere una lista di operazioni da svolgere. L'interpretazione data all'acquisizione dell' opzione *-w* duplicata è la seguente: l'ultima opzione ottenuta e il relativo argomento è quella presa in considerazione. Portano invece ad errore le opzioni *-f , -h, -D, -d, -p* specificate più volte. Tutte le altre opzioni, se duplicate accumulano gli argomenti nelle varie liste utilizzate.
Le varie operazioni sono poi eseguite seguendo un ordine statico, chiamando le funzioni offerte da **client_API.c/h**.
La prima e l'ultima operazione sono la *openConnection* e la *closeConnection*. Alcune operazioni sono eseguite in sequenza, ad esempio la scrittura di un file su server si compone delle chiamate *openFile, writeFile, unlockFile, closeFile*. L'esecuzione di una operazione di append su un file avviene in seguito ad una chiamata *writeFile* che non è andata a buon fine.

# API

Una nota importante relativa alle API è la necessità di utilizzare un flag globale per la scrittura a video delle informazioni richieste dalla specifica in seguito ad alcune operazioni. E' dunque necessario, se si desidera specificare dal client la volontà di stampare o meno le informazioni a video, agire sulla variabile *stdout_print* (1 di default).
Ad ogni richiesta inviata corrisponde almeno una risposta del server. Il client che utilizza queste API attenderà la risposta da parte del server relativa al completamento dell'operazione richiesta. Esistono delle operazioni che possono aspettarsi anche risposte non definitive (volendo, non sbloccanti) per il client, come l'arrivo di informazioni sui file evictati in seguito ad operazioni di scrittura, informazioni sui file letti da una *readN*.
Si noti che una operazione di lock è bloccante per il client fino al momento in cui la lock richiesta non viene acquisita. In caso di chiusura senza unlock da parte del client è il server che provvede a rimuoverla.
Durante l'attesa di una lock è possibile che il file venga rimosso da terzi, in questo caso il server provvede ad avvertire chi era in attesa.
Per utilizzare le API, si noti la dipendenza di **client_API.c/h** da **general_utility.c/h** e **conn.c/h** !!

# Altre note interpretative

Per lo sviluppo del progetto sono state prese delle decisioni vista la libertà data dalla specifica. Queste sono riportate di seguito:
- Quando un client scatena una eviction cercando di scrivere il contenuto di un file avviene un "bytes size capacity miss". E' solo in questo caso che i file vengono spediti indietro al client.
Un "files number capacity miss" avviene infatti, eventualmente, solo in seguito ad una *openFile*, e dalla specifica non ci è possibile salvare il contenuto, lato client, dei file evictati a seguito di ciò.
-L'eviction avviene solo in base alla politica utilizzata (LRU o FIFO). Non siamo tenuti a evitare l'eviction di file lockati in quanto, al limite, non si avrebbe coerenza di esecuzione: la precedenza è, infatti, sempre data allo scrittore e se la differenza tra la grandezza del file da scrivere e la capacità (in bytes) del server fosse infinitesimale, allora sarebbe necessaria una eviction di tutti i file dello storage, anche quelli lockati.
- Allo scatenarsi di un errore fatale, il server prova a chiudersi rilasciando i lock che può rilasciare. Nessun messaggio viene inviato ai client, che potrebbero bloccarsi in attesa, in quanto l'invio comporta l'utilizzo (spostamento, allocazione) della memoria e, se questa fosse corrotta per qualche motivo, si rischierebbe di inviare messaggi senza significato o di non inviarne comunque.
- L'operazione di *readNFiles* , in caso di politica LRU, non muove i file in coda come le altre operazioni in quanto non essendo un'operazione mirata su un file non si considera come un vero e proprio utilizzo di esso. Inoltre la lettura dei file avviene a prescindere che questi siano lockati o meno da altri client.

# I test

All'inizio di ogni test viene creato il file di configurazione per il server con i valori riportati sulla specifica.
**make test1:** Avvia il server in background, con valgrind, ridirezionando l'output sul file *log.txt*. L'id del processo è salvato sul file server.PID in modo da poterlo killare con *kill -1* una volta terminato il test. Vengono quindi avviati 5 client che eseguono delle operazioni e scrivono le informazioni sullo standard output. I file all'interno di *test/testFolder/folder1* e le sue sottocartelle vengono scritti sul server e vengono poi letti e salvati in *test/testFolder/folder2*. Altre operazioni vengono effettuate, si rimanda alla lettura del file *test/test1.sh.*

Il secondo test è composto, in realtà, da due test. E' infatti richiesto di dimostrare la funzionalità dell'algoritmo di rimpiazzamento e lo si vuole testare sia con file capacity miss sia con byte capacity miss.
**make test2_1:** Avvia il server allo stesso modo del test1. I client runnati scatenano file capacity miss e nessun file viene rispedito ai client (vedi sezione precedente). Altre operazioni vengono effettuate, si rimanda alla lettura del file *test/test2_1.sh.*
**Make test2_2:** Avvia il server allo stesso modo del test1. I client runnati scatenano byte capacity miss e i file vengono rispediti ai client e salvati nella cartella *test/testFolder/evictedFiles*. Altre operazioni vengono effettuate, si rimanda alla lettura del file *test/test2_2.sh.*

Alla fine di ogni test è possibile leggere il file *log.txt* per ottenere informazioni sul server.