Università di Pisa -- Dipartimento di Informatica
Corso di Laurea in Informatica

Progetto di Laboratorio di Sistemi Operativi

a.a. 2020-21
per Corso A e Corso B
Data di pubblicazione: 4 Maggio 2021

Luca Agnello - 585063 - Corso B

1. Introduzione

Il progetto è suddiviso in sette moduli, ognuno dei quali contiene gli "include", una cartella "src" e gli unit test:

- **client**: contiene il codice sorgente di una possibile implementazione del client
- **data-structures**: contiene le strutture dati utilizzate nel progetto (*linked list, min heap, queue, blocking bounded queue, non-blocking bounded queue, ecc*)
- **file-system**: contiene il file system del progetto, le sue strutture dati (*inode, file table, file descriptor table*) e alcune sue utility
- **helpers**: contiene le librerie di supporto per operazioni generiche, e quindi utilizzabili in ogni contesto (logging, caricamento di un file nell'environment, lettura di un file, scrittura di un file, ecc)
- **message**: questo modulo permette di creare facilmente dei messaggi da inviare attraverso un socket o una pipe
- **server**: è il cuore del progetto, contiene il server e la libreria di api da utilizzare nei client
- **socket**: questo modulo permette di creare delle *request* e di ricevere delle *response* attraverso un socket

È possibile inviare **qualsiasi** tipo di file al server. Il progetto è quasi interamente coperto da *unit test* e, in piccola parte, da *feature test*. La documentazione delle librerie non è sempre esaustiva, ad esempio la documentazione sui possibili *errno* impostati in caso di fallimento è mancante. Il progetto utilizza solamente le interfacce e le librerie conformi allo standard POSIX. Il progetto non ha la pretesa di essere privo di errori, bug o bad practice. Qualsiasi feedback è ben accetto.

2. Il server

Il **formato** del file "config.txt" è ***chiave=valore***, una riga che inizia con con il carattere # è considerata un commento e le righe vuote sono ignorate. Al momento dell'avvio del server, il suddetto file è caricato nell'*environment*, in modo da disaccoppiare la configurazione dal server.

All'avvio, il server installerà immediatamente un **signal handler** per intercettare i segnali **SIGINT**, **SIGQUIT** e **SIGHUP**, utilizzando la flag **SA_RESTART** per riavviare automaticamente le chiamate di sistema ***read*** e ***write*** in caso si verifichi una *interrupt*. Così facendo non sarà necessario "catturare" l'errore EINTR in caso di fallimento delle suddette chiamate. Nel caso il server riceva uno dei due segnali SIGINT o SIGQUIT, attiverà una variabile dichiarata ***volatile*** e di tipo ***sig_atomic_t***, tale variabile è testata immediatamente al fallimento della *select* con errore EINTR (la chiamata di sistema *select* non viene mai riavviata a prescindere dall'impostazione della flag SA_RESTART) e, se attivata, causa l'uscita (con successo) immediata dal server. Analogamente, nel caso il server riceva il segnale SIGHUP, attiverà una variabile diversa dalla prima, tale variabile è anch'essa testata al fallimento della *select* e, se attivata, termina immediatamente l'esecuzione del server nel caso non ci siano connessioni attive, altrimenti riavvierà la *select*. In questo caso la variabile è ulteriormente testata non appena si accetta una nuova connessione e, se attivata, la nuova connessione verrà immediatamente chiusa. Allo stesso modo, non appena un client chiude la connessione con il server, un ulteriore test sulla variabile verifica la presenza di altre connessioni attive e, in caso negativo, termina immediatamente l'esecuzione del server.

La gestione dello **spazio** di memorizzazione del server è dinamica: all'avvio lo spazio allocato in memoria è nullo, e può crescere fino alla capacità indicata nel file di configurazione del server. Al raggiungimento della capacità massima uno o più file saranno **espulsi** per guadagnare spazio secondo la

policy indicata nel file di configurazione. Le policy implementate sono FIFO, LIFO, LRU, MFU e LFU, la loro implementazione è spiegata nel capitolo 7.

La gestione dei file è delegata ad un **filesystem** sviluppato allo scopo. Tale filesystem è implementato da tre strutture dati: la ***file table***, la ***file descriptor table*** e l'***inode***, per semplicità la capacità del server non tiene conto dello spazio allocato per la loro gestione. La *file table* contiene gli *inode* di tutti i file presenti nel filesystem e consiste in un ***ternary search tree*** che utilizza il nome del file come *key*, l'*inode* come *value* e in una ***linked list*** che contiene le *key* presenti nell'albero. La *file descriptor table* contiene una copia degli *inode* "aperti" ed è composta da un ***array*** di inode che cresce dinamicamente e da un ***min heap*** che ha una funzione simile ad una *free space map*. L'indice dell'*array* corrisponde al *file descriptor* da utilizzare per le operazioni sul corrispettivo file. Da notare che le copie degli *inode* contengono il puntatore al file originale. L'*inode* è una struct contenente i metadata del file ed un puntatore alla memoria dove esso risiede, la natura del filesystem richiede la presenza dell'attributo name nell'*inode* ma poiché gli *hard link* non sono supportati questo non causa nessun problema. La scrittura di un file avviene inizialmente in un *buffer* nell'*inode*, il filesystem proposto prevede il "flush" del *buffer* dell'*inode* al momento della scrittura del file, nulla vieta, con le dovute considerazioni, di posticipare il flush al momento della chiusura del *file descriptor*. Il filesystem opera sfruttando unicamente queste tre strutture dati e protegge il proprio stato da accessi concorrenti con una **mutex**. Ogni metodo dell'interfaccia è garantito essere atomico: non appena chiamato, prova ad acquisire immediatamente il lock, e lo rilascia subito prima di ritornare al chiamante secondo le best practices per la scrittura di codice concorrente[1]. Le tre strutture dati citate non necessitano di protezioni da accessi concorrenti perché operano ad un livello inferiore, e non possono quindi essere accedute se non attraverso l'interfaccia del filesystem.

Una delle funzionalità offerta dal filesystem è quella di acquisire un **lock** su uno o più file, la policy adottata è la seguente: per acquisire il lock su un file, il file non deve essere *locked* e non deve essere aperto da altri thread; se il file è *locked* allora l'operazione fallisce; se il file è aperto da un qualsiasi altro thread allora il thread attende passivamente la chiusura del file da parte di tutti i thread che lo hanno aperto prima di tentare nuovamente l'acquisizione del lock, in questo caso eventuali richieste successive di accesso al file falliranno con *errno* EBUSY, in questo modo chi richiede il lock ha garanzia che riuscirà ad ottenerlo. Degno di nota è il caso in cui due thread attendono la chiusura dello stesso file per acquisire il lock: poiché tutti e due i thread devono aver aperto il file per poter acquisire il lock, attenderanno all'infinito la chiusura del file da parte dell'altro thread! Questa situazione è stata gestita permettendo ad un solo thread alla volta di richiedere il lock, facendo fallire la seconda richiesta con *errno* EDEADLK ed evitando così che si verifichi un **deadlock**. Un altro caso interessante è il fallimento di una operazione (compresa la lock) perché il file è *locked*: in questo caso il server inserisce la richiesta del client in una lista di attesa condivisa con tutti i suoi worker, non risponde al client e si rimette in ascolto di altre richieste. La lista di attesa è implementata dalla struttura dati *waiting list*, composta da una lista di coppie <*target, queue*>, dove *target* è il nome del file e *queue* è la coda contenente le coppie <*client_id, request*>. Ad ogni unlock sul file, il worker che ha eseguito l'operazione si preoccupa di eseguire nuovamente le richieste nella *waiting list* (potrebbero quindi verificarsi degli "*spurious wakeup"*), questo meccanismo ha lo svantaggio di sbilanciare potenzialmente la distribuzione del carico su un singolo worker ma ha il vantaggio di delegare l'attesa al client e quindi di evitare che il server tenga inutilmente occupato uno dei suoi worker.

Il logging degli eventi avviene in modo più o meno approfondito in base al ***log level*** indicato nel file di configurazione. I livelli possibili sono (in ordine decrescente): ***error*** (errori di importanza considerevole che compromettono la normale esecuzione del programma, non è detto che determinano l'uscita dal programma), ***warn*** (situazioni pericolose che indicano potenziali problemi), ***info*** (informazioni sullo stato operativo del programma), ***debug*** (eventi interessanti per gli sviluppatori del programma), ***trace*** (dettagli degli eventi interessanti per gli sviluppatori del programma). Il *log level* più basso include i livelli più alti, quindi il livello *debug* contiene anche i livelli *error*, *warn* e *info*. Il progetto utilizza il *log level "debug"*. Gli eventi loggati possono essere divisi in ***channel*** per migliorare la leggibilità del log, nel progetto è stato creato un *channel* per ogni componente del server (thread pool, worker, main, filesystem, ecc). Un esempio di output finale di un evento sul log è "*2021-09-22 23:23:03 gnl_fss_worker_1.DEBUG new message received"* dove *gnl_fss_worker_1* è il *channel* e *DEBUG* è il *log level*. La libreria responsabile della gestione del logging è l'helper *gnl_logger*.

Oltre al logging degli eventi, il server è in grado di produrre alla sua terminazione un output con il sunto delle operazioni effettuate durante l'esecuzione, per farlo utilizza l'interfaccia del filesystem che a sua volta utilizza un componente denominato ***monitor*** per tenere traccia delle operazioni. Ad esempio, ad ogni

[1] T. Anderson, M. Dahlin, *Operating Systems: Principles & Practice*, second edition, Recursive Books, cap. 5.5.2

scrittura o rimozione di file il filesystem notifica al monitor i byte scritti o rimossi, il monitor somma o sottrae i byte notificati al totale dei byte notificati in precedenza. Anche il *monitor* non necessita di protezioni da accessi concorrenti perché opera ad un livello inferiore rispetto al filesystem, ed è protetto quindi dalla sua interfaccia.

3. Il client

Il parsing degli argomenti da linea di comando avviene sfruttando la libreria *getopt*. Il client costruisce prima una *queue* di comandi ed effettua un'analisi sintattica degli stessi, in un secondo momento elabora la *queue* eseguendo un comando alla volta. Questa soluzione disaccoppia la lettura dei comandi dalla loro gestione. In accordo con lo standard POSIX, la gestione della option ***-R [n=0]*** è particolare: il supporto degli argomenti facoltativi di *getopt* è infatti possibile solamente tramite l'estensione GNU ed è espressamente vietata da POSIX[2]. Si noti la presenza di un doppio ":" nella *optstring* ":hf:w:W:D:r:R::d:t:l:u:c:p" subito dopo "R": questo indica a *getopt* che la option *-R* ha un argomento facoltativo, il risultato su codice conforme POSIX è che *optarg* sarà sempre nullo, per cui si utilizza l'indice *optind* per analizzare manualmente l'argomento successivo: se questo non è nullo e non è una "option" allora è l'argomento di "R", in tutti gli altri casi l'argomento di "R" viene impostato di default al valore "0".

Il client permette il salvataggio dei file espulsi dal server, tuttavia si limita ad indicare alle API la cartella dove scrivere i file espulsi, il suo funzionamento è quindi spiegato più avanti alla fine del capitolo 4. Per attivare questa funzionalità l'utente deve specificare la cartella tramite la option ***-D***, che deve obbligatoriamente essere immediatamente preceduta da una delle due option ***-w*** o ***-W***.

La maggior parte dei comandi del client utilizza le API con il pattern "open-do-close", ovvero verrà sempre aperto un file prima di operare su di esso e al termine verrà immediatamente chiuso, quindi due operazioni sullo stesso file provocano due aperture e due chiusure dello stesso. Se l'utente ha specificato la option ***-t***, allora tra una chiamata alle API e la successiva si attenderà il tempo specificato. Ad ogni operazione il client produce un log con i dettagli e l'esito dell'operazione appena effettuata, se l'utente ha specificato la option ***-p*** allora il log viene scritto nello *stdout*, in caso contrario viene immediatamente eliminato.

4. Interfaccia per interagire con il file server (API)

Le comunicazioni con il server avvengono attraverso l'utilizzo dei moduli ***message*** e ***socket***. Il modulo *message* stabilisce un protocollo per generare un messaggio partendo da una *struct* e per generare una *struct* partendo da un messaggio. Il protocollo proposto è il seguente: una stringa viene codificata in "[lunghezza stringa][stringa]", un numero in "[numero]" con un pad di 10 caratteri, i bytes vengono accodati al messaggio con memcpy, e così via. Quindi ad esempio la *struct gnl_message_nnb* {4, 5, 'ABC'} (il suffisso *nnb* sta per *number*, *number*, *bytes*) sarà trasformata nella stringa *00000000040000000005* ed in coda si troveranno i bytes (nota: i bytes vengono accodati immediatamente dopo il carattere di terminazione stringa "\0"). Il modulo *socket* espone un ***service*** per comunicare con il server attraverso un socket, tale servizio è utilizzato sia dal server, sia dalle API. Si utilizzano due astrazioni: la ***request*** per inviare un messaggio al server e la ***response*** per ricevere la risposta dal server. Il tipo di *request* varia in base al comando che si vuole impartire al server e non è altro che una *facade*[3] del modulo *message* specifica per il progetto. Analogamente la *request* varia in base alle possibili risposte che il server può dare. Il *service* espone quindi un'interfaccia che permette di inviare *response* e ricevere *request*; l'invio di una *request* avviene con i seguenti passi: si verifica che la connessione al server è attiva; la *struct request* viene trasformata in stringa; la stringa viene scritta con il metodo *write* sul socket. La ricezione di una *response* avviene in modo analogo, con la differenza che si effettua una *read* invece di una *write* e si trasforma la stringa letta nella *struct response*. Si noti che anche qui è necessario definire un protocollo di comunicazione esattamente come fatto nel modulo *message*, in questo caso il protocollo è "[tipo di request/response][numero di byte del messaggio]" al quale sono accodati i byte del messaggio con *memcpy*. Un ulteriore fatto è che non possono essere usate direttamente le chiamate di sistema *write* e *read*, per le problematiche descritte durante il corso[4], si utilizza quindi l'implementazione tratta da "Advanced Programming In the UNIX Environment" by W. Richard

[2] POSIX.1-2008, section 12.2, "Utility Syntax Guidelines", Guideline 7: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html#tag_12_02

[3] Facade pattern: https://en.wikipedia.org/wiki/Facade_pattern

[4] Primitives Interrupted by Signals: https://www.gnu.org/software/libc/manual/html_node/Interrupted-Primitives.html

Stevens and Stephen A. Rago, 2013, 3rd Edition, Addison-Wesley. Tale implementazione è incapsulata in due metodi *write* e *read* pubblici dell'interfaccia del *service*, in modo da poter essere utilizzata anche nelle comunicazioni con *pipe* (è il caso della notifica da parte di uno *worker* al *master* nel pattern *Master-Worker*).

Come visto nel capitolo 2, il filesystem si aspetta di lavorare per lo più con dei *file descriptor*, si è deciso quindi di implementare le API come se utilizzassero direttamente il filesystem: la *request* di apertura di un file genererà una *response* contenente il *file descriptor* del file aperto, tale *file descriptor* andrà utilizzato per tutte le *request* future che coinvolgono il file. Ad esempio un'operazione di *read* di un file richiede una "traduzione" del parametro *pathname* nel relativo *file descriptor* ricevuto nell'operazione di *open* precedente; l'unica eccezione a questa regola, oltre al metodo *open*, è il metodo *remove*. Anche qui si è scelto di utilizzare un *ternary search tree* per gestire le associazioni tra file e *file descriptor*, questa soluzione ha il limite di poter gestire solamente un *file descriptor* per file. Nel caso l'operazione di *write* causi l'espulsione di uno o più file dal server, le API riceveranno una lista contenente i file espulsi, questa lista viene iterata ed ogni file e, se richiesto dall'utente, verrà salvato nella *dirname* passata come parametro.

5. Makefile

Ognuno dei moduli presenti nel progetto ha un proprio Makefile per produrre le librerie e gli eseguibili di sua competenza. Quando per la compilazione sono richieste librerie esterne al modulo, ogni Makefile legge la loro posizione sul disco dal file .env del progetto. Il Makefile principale, oltre ad utilizzare i Makefile dei moduli per la loro compilazione, definisce alcuni *target* per eseguire i test automatici, ad esempio il *target* "tests-valgrind-error" salva l'output di una esecuzione con *valgrind* in un file temporaneo e cerca in quel file errori o *memory leak*, in caso trovi almeno un errore allora la shell lanciata dal Makefile fallirà, in caso contrario terminerà normalmente. Questo meccanismo è molto utile ad esempio durante l'esecuzione dei test in una *pipeline* di *deploy*, in caso di errori il *job* fallisce fermando immediatamente il *deploy*.

Nel modulo *server* è presente uno *unit test* che testa la correttezza delle API. Poiché lo scopo del test è testare unicamente le API, piuttosto che utilizzare il server reale si è preferito creare un *mock* del modulo *socket*, in modo da simulare un reale server senza di fatto averne uno in esecuzione. Nel Makefile dei test del modulo *server* ("server/tests/Makefile") è quindi incluso l'header originale (riga 36) mentre l'implementazione è quella del *mock* (righe 30, 31). Di seguito sono indicati i dettagli dei *feature test test1*, *test2* e *test3*.

test1: avvia il server con socket file "/tmp/LSOfilestorage_feature_test.sk" e file di log "/tmp/gnl_fss_feature_test.log", questi file possono essere eliminati con il comando *make clean.*

test2: avvia il server con socket file "/tmp/LSOfilestorage_replacement_policy_test.sk" e file di log "/tmp/gnl_fss_replacement_policy_test.log", questi file possono essere eliminati con il comando *make clean.*

test3: avvia il server con socket file "/tmp/LSOfilestorage_stress_test.sk" e file di log "/tmp/gnl_fss_stress_test.log", questi file possono essere eliminati con il comando *make clean.*

6. Script Bash per le statistiche

Lo script per le statistiche fa largo uso del comando *grep* per ricavare le informazioni. Per calcolare il numero totale di operazioni di un determinato tipo, lo script esegue una *grep* con il comando *wc -l* in pipe. Per calcolare i dati aggregati di media e valore massimo, lo script utilizza due funzioni separate che a loro volta utilizzano il comando *grep* per estrarre il dato grezzo dal file di log, per poi lavorarlo con le usuali operazioni matematiche. Per estrarre le informazioni sui singoli *worker*, lo script fa uso del comando *awk*, che risulta molto più snello rispetto ad una funzione e/o all'utilizzo del comando *grep*. In questo caso è necessario eseguire alcune operazioni in pipe per formattare l'output, per esempio viene utilizzato il comando *sed* per eliminare o sostituire le stringhe superflue.

7. Sviluppo opzionale

Le politiche di rimpiazzamento della cache implementate sono: FIFO, LIFO, LRU, MFU, LFU. Si è scelto per semplicità di non sviluppare un modulo di *cache replacement* vero e proprio, l'implementazione consiste quindi in un metodo interno al *filesystem* che genera ogni volta le sue strutture dati e seleziona il file da espellere in *realtime*. L'algoritmo di rimpiazzamento viene attivato dal filesystem ogni volta che una

scrittura su file fallisce con *errno* EDQUOT, in questo caso viene chiamato il metodo predisposto che, in ordine, prende la lista dei file presenti nel server, si costruisce un *minheap* in base alla policy scelta dall'utente ed estrae il minimo. Ad esempio per la policy LFU la costruzione del *minheap* avviene utilizzando come *key* il *reference count* dell'*inode* del file, per la policy LIFO è invece utilizzato il *btime* dell'*inode* del file moltiplicato per *-1*, e così via. Il filesystem itera la chiamata all'algoritmo fino a che si libera spazio sufficiente per procedere con la scrittura del file.

La memorizzazione dei file all'interno del server avviene in formato compresso, tramite un'implementazione dell'algoritmo *Huffman Coding*. Questa implementazione utilizza delle macro di manipolazione di bit definite all'indirizzo http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html. I metodi principali dell'interfaccia sono due: *encode* e *decode*. Il metodo *encode* prende in input una serie di byte e produce un "artefatto", che dovrà essere passato al metodo *decode* per ottenere i bytes originali. L'artefatto contiene l'*huffman tree* e i byte compressi generati durante l'encoding. Da notare che durante l'encoding l'allocazione di memoria è limitata al minimo indispensabile grazie all'utilizzo della funzione *realloc*, che permette di espandere la locazione di memoria della sequenza compressa di un blocco di 32 bit (i.e. 4 byte) al bisogno. Tuttavia questa soluzione degrada le performance di valgrind, la cui analisi della memoria durante una qualsiasi esecuzione potrebbe impiegare diversi minuti. La funzionalità di compressione e decompressione dei file è racchiusa per semplicità nella struttura dati *inode* presentata nel capitolo 2 e non è esposta dall'interfaccia del filesystem, questa scelta causa una disottimizzazione delle performance, perché sarà necessario comprimere il file due volte: prima di effettuare i dovuti controlli di spazio (quindi al livello del filesystem) e al momento del salvataggio del file nella memoria (quindi al livello dell'*inode*).

8. Note finali

Il progetto è stato sviluppato in un repository github pubblico raggiungibile al seguente indirizzo: https://github.com/gnello/in-memory-file-storage-server. Sono presenti due branch: il branch *main* che contiene una versione pubblica del progetto e può essere ignorato (l'interfaccia delle API è in formato snake_case ed i suoi metodi sono denominati con il prefisso *gnl_fss_api_)*, e il branch *assignment_version* che contiene il progetto conforme alle specifiche dell'assingment. La correttezza del codice è stata costantemente monitorata, per quanto possibile, da un "github workflow" che si occupa di eseguire i test automatici ad ogni *push* sul repository. I test eseguiti comprendono un check con valgrind che si assicurano dell'assenza di errori e/o *memory leak*. La history delle esecuzoni degli workflow è visionabile al seguente indirizzo: https://github.com/gnello/in-memory-file-storage-server/actions. I test automatici coprono la maggior parte del codice e sono composti in larga parte da unit test e in piccola parte da feature test.