Relazione Progetto di Laboratorio di Sistemi Operativi

SALVATORE CORRENTI

CORSO A, MATRICOLA 584136, 2° appello estivo A.A. 2020/21

1. Premessa

Parti opzionali svolte. Sono state svolte le seguenti parti opzionali: *test3.sh*, implementazione delle funzioni: *lockFile*, *unlockFile*, *removeFile*, *writeFile*; opzione *-D* del client.

Codice non-POSIX. La funzione *openConnection* dell'API utilizza le funzioni *Linux-specific* messe a disposizione in *sys/timerfd.h*, le quali consentono la creazione di un timer che notifica attraverso file descriptor anziché segnale SIGALRM, come le normali chiamate *timer_**. Si è scelto di utilizzare tali chiamate per ottenere una maggior precisione per il fallimento da timeout della *openConnection* rispetto a quanto si potrebbe avere semplicemente con una combinazione di *usleep/nanosleep* + controllo del tempo passato al risveglio, al prezzo di dover monitorare il timer con la *poll*.

Credits ad autori di terze parti Per la hashmap è stata utilizzata l'implementazione {icl_hash.h, icl_hash.c} di Jakub Kurzak fornitaci su Didawiki. Tutte le note relative al copyright sono nel file "icl_hash.h".

Repository Git. https://github.com/Servat0r/SOL21Project.

2. Il FileStorage / Filesystem

Struttura generale. Il "filesystem" del server, ovvero la struttura dati che permette la gestione dei files da parte del server, è implementato da un oggetto di tipo ***FileStorage_t***. Nel seguito si useranno i termini *filesystem* e "*file storage"* per riferirsi a oggetti del tipo *FileStorage_t*.

Ogni file storage contiene:

a) una hashmap che mappa ogni path di file presente nello storage al file corrispondente;
b) una coda FIFO concorrente utilizzata per il rimpiazzamento;
c) una mutex, 2 condition variables e altre variabili per gestire l'accesso al filesystem in modalità read/write;
d) dei campi di controllo dello spazio occupato e dei campi "statistici" che verranno stampati al termine dell'esecuzione.

All'interno del filesystem si assume che ogni client sia identificato da un id univoco, ma l'implementazione non pone restrizioni su come tale id debba essere ottenuto per ogni client: l'implementazione fornita identifica ogni client con il file descriptor della sua connessione, ma il filesystem permette anche altre implementazioni, con l'unico requisito che ogni identificatore sia ≥ 0.

Il singolo file è rappresentato da un oggetto di tipo ***FileData_t***, il quale contiene, oltre ai campi ovvii: un byte di flag "globali" sul file; un array dinamico di bytes tali che per ogni indice i, la posizione i dell'array si riferisce all'i-esimo client attuale e ospita dei flag "locali" del client; una coda concorrente che contiene gli id di tutti i client in attesa della lock; una pthread_rwlock_t utilizzata per le operazioni sui files.

Gestione della concorrenza L'accesso concorrente ai dati del filesystem è implementato attraverso una *doppia read-write lock*: la mutex, le condition variables e le variabili di stato del filesystem simulano una ***read-write lock di tipo writer-preferred*** con la possibilità per uno scrittore di passare atomicamente alla modalità lettore. In questo contesto, *lettore* e *scrittore* vanno intesi rispetto all'intero filesystem.

Il meccanismo si basa sull'osservazione che esistono 3 tipi di operazioni effettuabili sul filesystem:

a) creazione/rimozione di un file ed espulsione dei file(s) dallo storage: queste operazioni modificano sempre il filesystem, e vanno quindi eseguite in mutua esclusione con tutte le altre;

b) scrittura di un file: questa modifica certamente lo spazio occupato dall'insieme dei files, potendo causare espulsioni di file, dopodiché però l'insieme dei file non sarà più modificato: di conseguenza va eseguita in mutua esclusione fino all'eventuale rimpiazzamento di file, ma poi può essere eseguita insieme ad altre operazioni del tipo c).
c) altre operazioni: tutte le altre operazioni non modificano **mai** l'insieme dei file ospitati.

Inoltre, è necessario osservare che si ha la necessità di un ulteriore livello di *read-write locking* per impedire operazioni in conflitto sui dati condivisi in un singolo file.

Ogni operazione sul filesystem avviene dunque acquisendo la *"read-write lock simulata"* nella modalità necessaria, dopodiché (se necessario) acquisendo le *pthread_rwlock_t* dei file interessati e rilasciando le due risorse in ordine inverso di acquisizione. In particolare, le operazioni di scrittura file avvengono acquisendo la read-write lock simulata in modalità scrittura, e dopo aver eseguito eventualmente la funzione di rimpiazzamento, si esegue l'operazione ***fs_op_downgrade***, la quale garantisce che il thread diventerà lettore senza che altri thread nel frattempo possano accedere al filesystem, garantendo così una scrittura per volta.

Gestione del rimpiazzamento. Il rimpiazzamento è realizzato attraverso la funzione ***fs_replace***, la quale è invocabile in modalità *"creazione"* oppure *"scrittura"*: il primo caso corrisponde ad un overflow della capacità di numero di files, e in tal caso fs_replace espellerà il primo file in coda, mentre il secondo caso corrisponde ad un overflow della capacità di storage, e in tal caso fs_replace espellerà i file in ordine FIFO. La fs_replace inoltre accetta come parametri due puntatori a funzioni, ***waitHandler*** e ***sendBackHandler***, che si occupano rispettivamente di notificare i client in attesa della lock che il file non esiste più e di inviare i file espulsi al chiamante, in quanto il filesystem non fa assunzioni su come vengano identificati i client.

Si osserva infine che l'invio dei file espulsi al client viene effettuato *solo* in caso di espulsione per storage-overflow, mentre per filecap-overflow il file viene semplicemente distrutto: questa decisione è stata presa in quanto la funzione openFile non dispone di un parametro che indichi una cartella dove salvare il file espulso, né da riga di comando esiste alcuna opzione che copra questo caso.

Semantica di open/close/write/lock/unlock. E' stata data un'interpretazione delle operazioni di apertura e locking di un file come "permessi" che un client può acquisire e rilasciare. Di conseguenza, le operazioni *openFile* e *closeFile* sono implementate in modo "idempotente", ovvero più chiamate della stessa funzione sullo stesso file terminano tutte con successo. Le operazioni *lockFile* e *unlockFile* sono state implementate in modo da essere "indipendenti" rispetto all'apertura / chiusura di un file, ovvero è possibile acquisire la lock su un file anche senza averlo aperto. L'unica eccezione è l'operazione *openFile(O_LOCK)*: in questo caso la richiesta è di acquisire *entrambi* i permessi di apertura e locking, e quindi se l'operazione fallisce non viene aggiunto nessuno dei due permessi al client chiamante.

Gestione delle operazioni di locking/unlocking e del cleanup. Il filesystem mette a disposizione un'operazione ***fs_clientCleanup*** per poter eliminare i dati di ogni client, la quale: azzera l'elemento corrispondente nell'array di bytes di ogni file; rilascia ogni lock posseduta da quel client e la riassegna al successivo in attesa (se presente); elimina il file da ogni lista di attesa di lock in cui si trova.

Le funzioni che possono causare il rilascio della lock su un file (*fs_unlock*, *fs_clientCleanup*) prendono come parametro un puntatore a una lista in cui scriveranno il prossimo/i prossimi detentori della lock: questa scelta è stata fatta per uniformare la notifica di acquisizione lock da parte dei thread worker, i quali forniranno sempre come parametro la lista necessaria (vedasi sez. ***Il server***).

3. Il client

Path assoluti e relativi. Le opzioni *'-W'* e *'-w'* accettano path relativi, ma i file corrispondenti vengono inviati al server con il loro path assoluto chiamando la funzione *'realpath'*; di contro, le altre opzioni accettano solo path assoluti: questa decisione è stata presa per evitare conflitti nel caso in cui due client inviino al server file che si trovano in path assoluti diversi, ma che coincidono nel path relativo rispetto a dove si trova l'eseguibile del client.

Flag -D e –d. In accordo con l'utilizzo del path assoluto per la memorizzazione dei file nel server, tutti i file vengono inviati al client con il path che avevano sul server, e il salvataggio avviene creando ricorsivamente nella cartella indicata *tutte* le directory presenti nel path.

4. Il server

Gestione del dispatching richieste e del cleanup dei client. Il dispatching delle richieste avviene attraverso una coda FIFO concorrente in cui il manager inserisce i file descriptors pronti e da cui i workers continuamente estraggono i nuovi elementi inseriti. Il manager ascolta le nuove richieste utilizzando la chiamata *pselect*() e una volta ricevuta una richiesta da un client, lo toglie dal *readset* e lo accoda.

Dopo l'esecuzione della richiesta si hanno due possibili casi:

1. La richiesta prevedeva l'acquisizione di una lock detenuta da un altro client: in tal caso il worker passa all'iterazione successiva senza comunicare nulla al manager;
2. altrimenti, il worker invia il valore del file descriptor opportunamente trasformato (si veda la sezione successiva) per permettere al manager di riascoltarlo oppure di chiudere la connessione se il client ha a sua volta chiuso la connessione e in tal caso esegue il cleanup necessario.

Gestione riassegnamento lock e cleanup. Ogni worker all'avvio crea una linkedlist che conterrà tutti i file descriptor di client a cui va riassegnata la lock su un file dopo l'esecuzione di una *fs_unlock* o di una *fs_clientCleanup*, passando a queste ultime l'indirizzo della lista ad ogni invocazione. Al termine dell'esecuzione di ogni richiesta viene effettuata una scansione completa della lista per notificare ogni client che potrebbe essere in attesa sulla medesima. Se durante lo scambio di messaggi fra worker e client quest'ultimo chiude la connessione, viene chiamata la funzione *server_cleanup_handler* che si occupa di eseguire il cleanup dei dati del client con *fs_clientCleanup* e di notificare tutti i client restituiti da *fs_clientCleanup* che la lock che attendevano è stata sbloccata; inoltre, chiama ricorsivamente se stessa se durante la notifica a un client si accorge che quel client ha chiuso la connessione.

Gestione delle connessioni. È responsabilità del manager chiudere tutti i file aperti durante l'esecuzione del server: dopo ogni richiesta, i worker inviano al manager attraverso una pipe condivisa:

1. il valore del file descriptor se rappresenta una connessione potenzialmente ancora aperta, ed in tal caso il manager reinserisce il file descriptor nel listen-set per ascoltarlo di nuovo;
2. il valore del file descriptor a cui è stata applicata la funzione $f(x) \coloneqq -x - 1$ se rappresenta una connessione che è stata certamente chiusa: dato che il valore di un file descriptor x è sempre ≥ 0, si ha che $f(x) < 0$ e il manager può identificare univocamente che quel descrittore rappresenta una connessione ormai chiusa e ottenere il valore originario riapplicando f al dato ricevuto (in quanto $f(f(x)) = x$ per ogni x) e chiudere la connessione aggiornando il *clientset*.

Gestione della terminazione. Il thread manager utilizza la chiamata di sistema pselect per monitorare tutti i file descriptor attivi di interesse. Nel main, dopo l'installazione dei signal handler necessari, la signal mask originaria del processo viene ripristinata ma mascherando SIGINT, SIGQUIT e SIGHUP: la cattura di questi segnali avverrà soltanto durante la *pselect*, la quale prende come parametro la signal mask originaria. Questo meccanismo permette di fatto di gestire in modo "sincrono" SIGINT, SIGHUP e SIGQUIT, nel senso che se uno di essi viene inviato fra due pselect consecutive, verrà catturato dal thread manager solo alla prossima pselect: questo permette di evitare le race conditions menzionate nella pagina del manuale della *select* garantendo al tempo stesso che la semantica richiesta dei segnali viene rispettata, ovvero l'arrivo di un segnale in un qualunque momento ha effetto soltanto sulle richieste che arriveranno alla successiva pselect, e mai su quelle già in coda.

Al momento della terminazione, per notificare ai workers che non verranno più aggiunte nuovi elementi alla coda, quest'ultima consente di chiamare una funzione *tsqueue_close* che setta il valore di una variabile di stato per indicare che non verranno più inseriti altri elementi e risveglia tutti i workers in attesa sulla coda.

File di configurazione. Il file di configurazione ha il seguente formato: per ogni riga è ammessa al più una coppia *chiave = valore*, con almeno una coppia di spazi intorno all'uguale. Sono permesse righe vuote e commenti in linea singola che iniziano con #, e sono ammessi anche dopo una coppia chiave-valore.

Un esempio di file di configurazione con spiegazione completa è *config.txt* nella root directory del progetto.

5. L'API e il protocollo di comunicazione

Struttura del protocollo di comunicazione. Il protocollo di comunicazione si basa su oggetti *message_t*, una struct che contiene: il codice che identifica il tipo di operazione richiesta; il numero di argomenti della richiesta; un array di coppie (dimensione, dati) per ogni argomento della richiesta.

Ogni codice ha un numero predefinito di argomenti e le funzioni che inviano/ricevono messaggi si aspettano di ricevere *esattamente* tale numero: ciò significa ad esempio che in caso di espulsione di più file, sarà inviato un messaggio per ogni file; di contro, questo rende lo scambio messaggi più sicuro.

Lo scambio di messaggi avviene attraverso le funzioni *msg_send* e *msg_recv*: la prima prende in input un *message_t** e invia nell'ordine il codice del messaggio, il numero di argomenti e per ogni argomento prima la dimensione e poi i dati, mentre la seconda prende in input un *message_t** e alloca dinamicamente l'array necessario per ospitare gli argomenti. Entrambe le funzioni falliscono con errore EBADMSG se non riescono a inviare/ricevere correttamente il messaggio.

Al di sopra queste funzioni sono definite altre due funzioni, *msend* e *mrecv*, che sono una sorta di "macro" che si occupano di gestire ricezione e invio in modo "transazionale", accettando un *message_t** **non** allocato e occupandosi dell'allocazione/liberazione in caso di fallimento di tutte le strutture necessarie.

Gestione dei path nell'API. Tutte le funzioni dell'API controllano che il path passato sia *assoluto*: malgrado i path siano forniti o vengano convertiti in assoluti già nel client, questo controllo è necessario perché in generale non è garantito che un client scritto male o malevolo invii solo path assoluti. Inoltre, dato che l'API deve permettere la creazione e la scrittura di contenuti in teoria anche di file *non* esistenti sulla macchina, il controllo che il path sia assoluto viene fatto semplicemente controllando che cominci con *'/'*, dato che si assume che ad esempio un path *"~/dir1"* si assume sia già stato tradotto dalla shell. L'unica eccezione è la funzione *writeFile*, la quale invece invoca la funzione *realpath*: questo perché non ha senso voler scrivere il contenuto di un file non esistente sulla macchina.

6. Strutture dati di supporto

Coda concorrente. I file *tsqueue.h* e *tsqueue*.c definiscono una coda FIFO concorrente che permette inoltre di iterare su se stessa in mutua esclusione rispetto alle operazioni di push e pop, supportando la rimozione interna alla coda durante l'iterazione.

Linkedlist. I file *linkedlist.h* e *linkedlist*.c definiscono una lista linkata doppia *non* concorrente con inserimento e rimozione in qualunque punto della lista e iterazione senza rimozione.

Parser riga di comando. I file *argparser.h* e *argparser.c* definiscono un parser di argomenti da riga di comando. Il parser *non* fa uso della funzione *getopt*, ma si basa invece su due struct: *optdef_t* che contiene la definizione di un'opzione e una stringa da stampare come messaggio di help/utilizzo attraverso la funzione *print_help*; *optval_t*, che rappresenta un'opzione effettivamente parsata con un puntatore alla sua definizione e una lista che contiene tutti gli argomenti forniti. La funzione di parsing principale *parseCmdLine* restituisce una lista di oggetti optval_t corrispondenti alle opzioni parsate, riconoscendo la virgola come separatore.

7. Makefile, librerie e test

Makefile. Il Makefile include tutti i target richiesti; in particolare il comando *make all* crea anche le directory che ospitano tutti i file binari, mentre *make clean(all)* elimina anche i file scritti durante l'esecuzione dei test bash.

Librerie. Si è deciso di utilizzare un'unica libreria condivisa, *libshared.so,* necessaria sia per il client sia per il server.

Test bash. Tutti i *test1/2/3.sh* sono invocabili con *make* utilizzando i comandi *make test1/2/3* rispettivamente. In aggiunta a ciò, è fornito un ulteriore ***test0.sh***, un semplice test aggiuntivo il cui scopo è mostrare la corretta terminazione del server a seguito di ricezione di un segnale anche in presenza di molti client ancora attivi. Accetta come argomento il codice di un segnale da inviare al server dopo 3 secondi, garantendo che in quel momento siano attivi 30 processi client.

Il ***test1.sh*** lancia in totale 5 processi client per testare tutte le funzionalità offerte, compresa una dimostrazione di un client che attende la lock su un file acquisita da un altro client.

Il ***test2.sh*** verifica il corretto rimpiazzamento dei file per entrambi i tipi di overflow nel seguente modo: prima invia 12 file molto piccoli al server per verificare l'espulsione per file-overflow; dopodiché invia altri 3 file la cui dimensione complessiva supera la capacità di storage del server per verificare l'espulsione per storage-overflow; infine invia un file di dimensione superiore alla capacità dello storage per verificare che il file viene creato ma la scrittura fallisce.

Il ***test3.sh*** è strutturato in questo modo: nella cartella test/test3files sono presenti 10 cartelle che contengono ognuna 10 files la cui dimensione complessiva supera la capacità di storage. *test3.sh* lancia 10 processi figli assegnando a ognuno una cartella diversa ed ogni processo figlio lancia lo script *client_factory.sh*, il quale lancia continuamente un processo client che prova a scrivere la cartella corrispondente e un processo client che prova a leggere tutti i file della cartella e salvarli. Dopo 30 secondi, viene inviato il segnale SIGINT al server e vengono uccise tutte le client factories, dopodiché vengono uccisi tutti i client rimasti orfani e il test termina. La correttezza dell'assenza di errori a runtime lato server è garantita dall'output finale del server.

Colorazione output su shell di test, client e server. Per facilitare la lettura dell'output dei test, si è deciso di colorare le parti più rilevanti dell'output prodotto da server, client e script bash.

8. Gestione degli errori

Generale. Si è optato in generale per la propagazione degli errori generati fino a *"top-level"*, ovvero i programmi server e client; per garantire una maggiore sicurezza, ove ciò sia possibile senza incorrere in race conditions/faults, all'inizio di ogni funzione vengono controllati i parametri di input. L'unico caso in cui si ha uscita istantanea è in caso di errori nell'utilizzo di *mutex/rwlocks/condition variables*, perché in tal caso non è più possibile garantire uno stato consistente.

Si è fatto uso estensivo delle macro *SYSCALL_** definite a lezione ed incluse nel file *defines.h*, anche per chiamate di funzione che *non* sono system call ma che ritornano -1 in caso di errore.

Per maggior sicurezza, nell'esecuzione di funzioni sulle strutture dati del filesystem che potrebbero causare errori non fatali ma non recuperabili (ad esempio non si riesce a deallocare una struttura sullo heap) si è usata una macro *SYSCALL_NOTREC* (e varianti per liberare delle lock) che ritorna immediatamente dalla funzione corrente settando errno a *ENOTRECOVERABLE*. Ogni volta che viene eseguita un'operazione sul filesystem lato server, la macro *CHECK_FATAL_EXIT* controlla se $errno \in \{ENOMEM, ENOTRECOVERABLE\}$ e in tal caso esce. Ogni uscita con lato server garantisce in ogni caso l'unlink del file creato con la funzione bind.

EBADE. In caso di errore nell'esecuzione di una richiesta, il server invia al client un messaggio con codice *M_ERR* e come unico argomento il valore di errno, il quale, se è stata fornita *–p*, vengono stampate dalle funzioni dell'API. Tuttavia, dato che un errore lato server può confondersi con un errore lato client (che invece va gestito nel client stesso), si è deciso di restituire -1 con errno settato a *EBADE* per tutte le funzioni dell'API che ricevono un errore lato server ma in cui non avvengono errori interni.

EBADMSG, EPIPE. Come già visto, le funzioni *msg_recv* e *msg_send* settano errno a *EBADMSG* nel caso in cui il messaggio è stato parzialmente inviato (*msg_send*), oppure non ricevuto completamente (*msg_recv*). Nel caso di una *msg_send*, l'impossibilità lato server di inviare un messaggio per chiusura connessione setta errno a *EPIPE*, pertanto per server e client l'insieme {*EBADMSG*, *EPIPE*} individua univocamente tutti i casi di connessione chiusa da parte di uno dei due processi oppure di impossibilità di scrivere ulteriormente, e si è scelto in quest'ultimo caso di agire come se la connessione sia stata chiusa.