Corso di Laurea in Informatica (Classe L31)

Laboratorio di Sistemi Operativi

RELAZIONE PROGETTO FILE STORAGE SERVER

Fabrizio Pini matricola 530755

Sommario

1 Strutturazione del codice

Di seguito la lista dei moduli che compongono i file di progetto.

api.c / .h Contiene prototipo e implementazione delle funzioni per inviare richieste per creare/rimuovere/aprire/scrivere/… file nel/dal file storage server

client_def.h Contiene la definizione del tipo di dato che identifica le operazioni (passate come opzioni al client(.c))

client.c Contiene il *main* del client

common_def.h Contiene la definizione di vari tipi di dato, tra cui il messaggio di richiesta (*msg_request_t*) e quello di risposta (*msg_response_t*)

common_funcs.c / .h Contiene prototipi e l'implementazione delle funzioni per leggere e scrivere da/su un descrittore; creare e scrivere un file in una directory

config.c / .h Contiene i prototipi e le implementazioni delle funzioni per la lettura dei parametri di configurazione, in particolare *read_config_file,* nonché la definizione del tipo di dato dove si memorizzano i campi letti dal file. Tutte le variabili di configurazione, una volta lette, vengono memorizzate nella struttura dati globale *config*.

intWithLock.c / .h Contiene i prototipi e l'implementazione di un intero con lock, con la definizione del tipo di dato *IntWithLock_t* e delle operazioni per lavorarci

Makefile Contiene le regole per automatizzare la compilazione. I target all per generare gli eseguibili del programma server e del programma client; clean/cleanall per ripulire la directory di lavoro dai file generati, moduli oggetto, socket file, logs, librerie, etc.; tre target di test: test1, test2 e test3.

opt_queue.c / .h Contiene i prototipi e l'implementazione della coda delle operazioni, con la definizione del tipo di dato *OptQueue_t* (ed *OptNode_t*) e delle operazioni per lavorarci.

queue.c / .h Contiene i prototipi e l'implementazione di una semplice coda di interi, con la definizione del tipo di dato *Queue_t* (ed *Node_t*) e delle operazioni per lavorarci.

server.c Contiene il *main* del server, la funzione eseguita dai thread worker (*Worker*)

common_funcs.c / .h Contiene il prototipo e l'implementazione della funzione che crea il thread che gestisce i segnali

storageQueue.c / .h Contiene i prototipi e l'implementazione della coda dei file dello storage, con la definizione del tipo di dato Storage*Queue_t* (ed Storage*Node_t*) e delle operazioni per lavorarci.

util.h Contiene alcune macro di utilità generale (lock e unlock dei mutex)

2 Lato Server

Il server consiste di un singolo processo multi-threaded, secondo lo schema "master-worker", in cui il numero di threads *Worker* è fissato all'avvio del server sulla base delle informazioni di configurazione, ed ogni *Worker* può gestire richieste di client diversi.

Il processo server accetta connessioni di tipo AF_UNIX da parte dei client.

Vi è anche un altro thread, di supporto, per la gestione dei segnali (*SignalHandler*).

2.1 Threads

2.1.1 Main

All'avvio del server, il thread **main**:

- legge il file di configurazione, passato come parametro e strutturato secondo il formato definito più avanti
- crea ed inizializza la coda *Queue_t*, utilizzata per comunicare i fd (pronti) ai threads *Worker*
- crea ed inizializza la coda *StorageQueue_t,* utilizzata per memorizzare i file e le informazioni su di essi necessarie
- crea una pipe per la comunicazione (dei *fd*) tra i threads *Worker* e il main
- crea una pipe per la comunicazione dei segnali tra il thread *SignalHandler* ed il main (nella pipe concretamente non scriviamo nulla, il *signalHandler* chiude il descrittore di scrittura per svegliare il main)
- inizializza lo stato del server (*server_status*) (RUNNING)
- crea e fa partire (passando opportunamente gli argomenti) il thread *SignalHandler* e i threads *Worker*
- crea il socket di connessione, e lo "registra" insieme ai descrittori in lettura delle due pipe nel set (prepara la maschera)
- si mette in un ciclo, in cui (tramite la SC *select()*) gestisce l'accettazione di nuove connessioni da parte dei client, le richieste funzionali da essi (utilizzo coda *Queue_t* e pipe tra *Worker* e main) e la ricezione dei segnali (il descrittore in lettura della pipe tra il main ed il thread *SignalHandler* diventa pronto quando il thread *SignalHandler* dopo aver cambiato lo stato del server (in seguito alla ricezione di un segnale a cui siamo interessati) chiude il descrittore in scrittura)
- esce dal ciclo in fase di chiusura del server
- termina i threads *Worker*, stampa il sunto delle operazioni effettuate, chiude e libera memoria opportunamente.

2.1.2 Worker (Gestisce le richieste dei client)

Dopo aver estratto i parametri passatigli dal main, si mette in un ciclo, in cui: aspetta sulla coda *Queue_t* un fd (identificativo di un client), da cui leggere la richiesta (*msg_request_t*), tramite la funzione *requestHandler* va concretamente a gestire la richiesta, interagendo con lo storage (coda *StorageQueue_t*) del server e comunicando con il client.

2.1.3 SignalHandler

La funzione createSignalHandlerThread (chiamata dal main) prima di far partire il thread SignalHandler maschera tutti i segnali, per evitare interferenze finchè la loro gestione non sarà completa.

Il thread SignalHandler maschera i segnali: SIGINT, SIGQUIT, SIGHUP che devono essere gestiti.

Alla ricezione di uno di essi, viene semplicemente cambiato lo stato del server in CLOSING (per SIGHUP), e CLOSED (per gli altri), e viene chiuso il descrittore di scrittura per svegliare il main.

2.2 Terminazione

La terminazione effettiva è gestita attraverso la variabile globale di stato, che assume i valori CLOSED per quella immediata, o CLOSING, per la chiusura "dolce" con il controllo dei client attivi.

Il thread main del server in caso di terminazione immediata (stato CLOSED impostato dal SignalHandler), semplicemente esce dal while di gestione delle attese. Mentre per la terminazione del server "dolce" (SIGHUP) chiude il socket di accettazione (di nuove connessioni) e continua a ciclare fino a quando vi sono dei client connessi. La variabile *iwl*, condivisa tra main e worker, contiene in numero di client attivi in ogni istante, quando diventa 0, allora il main esce dal while e procede con la chiusura.

2.3 Strutture dati principali

Il file storage server utilizza le seguenti variabili globali:

- *struct config_struct config* variabile che memorizza i dati della configurazione con cui è stato avviato il server
- *StorageQueue_t *storage_q* puntatore alla coda dei file dello storage
- *server_status status* variabile che rappresenta lo stato del server

Un'altra variabile importante (non globale, ma dichiarata nel *main* e passata ai threads *Worker*) è:

- *Queue_t *q* puntatore alla coda dei fd pronti

2.4 File di configurazione

Il file di configurazione ha un formato molto semplice.

Sono previste linee di commenti, che sono marcate con # e assegnamenti, che seguono la struttura

<chiave> = <valore>

Le chiavi sono associate ai parametri di configurazione del server e nello specifico:

- numero di workers: es. *Num_workers = 8*
- path utilizzato per la creazione del socket AF_UNIX: es. *Sockname = ./mysock*
- numero massimo di connessioni pendenti: es. *MaxConnections = 128*
- numero limite di files nello storage: es *LimitNumFiles = 100*
- dimensione dello storage in bytes: es. *StorageCapacity = 32000000*
- livello di verbosità del server: es. *PrintLevel = 2*

Il livello di verbosità è stato aggiunto per poter cambiare in modo semplice la quantità e tipologia di messaggi stampati durante l'esecuzione del programma server. Sono stati previsti 3 livelli:

- quieto, ovvero *PrintLevel = 1*: dove vengono stampati solo il sunto finale e gli eventuali errori
- messaggio, ovvero *PrintLevel = 2*: dove vengono stampati i contenuti dei messaggi scambiati con il client
- test, ovvero *PrintLevel = 3*: dove vengono riportati i cambiamenti delle variabili principali (es. struttura config), le chiamate e gli esiti delle principali funzioni (es.

  if (config.v > 2) printf(" SERVER: queue_s_push, fd: %d, pathname: %s\n", fd, pathname);

3 Principali scelte progettuali

3.1 Struttura dello storage

Non disponendo di indicazioni sulle modalità operative del file storage, ovvero numero di files, numero di client, condivisione dei file e aperture concorrenti, dimensioni dei file, ecc..

La struttura dati del server che realizza lo storage dei file è una coda (*StorageQueue_t*), tale scelta si basa sulla flessibilità, la sua intrinseca politica FIFO (adottata come politica di rimpiazzamento dei file), e sulla sua facilità di implementazione.

L'elemento della coda (*StorageNode_t*), contiene tra le variabili il nome del file, la lunghezza, ... ed il puntatore ad un'altra coda (*Queue_t*) contenente i client (identificati dal fd) che hanno aperto il file.

È stato fatto ciò, presupponendo che:

- il mio file storage server abbia come scopo servire pochi client che utilizzano molti file ciascuno (o comunque che condividono pochi file tra di loro)

- e che i client facciano varie operazioni, mentre l'uscita sia un evento raro

Con tali presupposti lo svantaggio, che ricorre all'uscita del client, di scorrere tutti i nodi della coda storage, e di ciascun nodo scorrere la coda dei client opener, per togliere il client (se c'è), è limitato, ed oscurato dai vantaggi.

3.2 Gestione dei segnali

Per la gestione dei segnali la scelta è stata di non usare gli handler, ma di avere un thread dedicato (*SignalHandler*) che si attiva quando è generato una richiesta di interruzione.

Tale soluzione permette di accedere senza problemi alle variabili globali e di invocare funzioni non signal-safe durante la loro gestione.

3.3 Politica di rimpiazzamento

L'algoritmo di rimpiazzamento dei file nella cache utilizza una politica FIFO che viene naturalmente gestita con la struttura a coda.

Utilizzando questa politica di selezione del file da espellere, è stato rilevato il problema che la rimozione di file poteva coinvolgere file sui quali fossero in corso operazioni da parte di altri client, che potevano (o non) avere applicato esplicitamente un lock. È stato quindi ipotizzata una attesa che le operazioni fossero terminate, ma dopo l'implementazione abbiamo rilevato varie situazioni di stallo, la cui gestione sarebbe stata abbastanza complessa, ed è stata quindi abbandonata.

Per risolvere il problema è stata valutata la revisione dell'algoritmo di selezione. Applicando un algoritmo di Seconda Chance, che per sua natura avrebbe selezionato i files più vecchi (FIFO), ma non usati recentemente (dall'ultimo rimpiazzamento e per un certo periodo). Il problema sopra indicato, della rimozione di file in uso, sarebbe stato risolto, ma probabilmente saremmo andati incontro al problema di non avere trovare un file da rimuovere.

L'alternativa un algoritmo LRU puro era scartata in fase di progettazione, e sarebbe stato complesso una revisione dell'implementazione fatta (stack al posto di coda).

Una variante della Seconda Chance, con i file marcati all'apertura e smarcati alla chiusura, è stata ipotizzata ma non implementata, sia per motivi di tempo che per gli stessi dubbi sul fatto saremmo andati incontro al caso di non avere trovare un file da rimuovere.

4 Lato Client

Il programma client si occupa di effettuare il parsing delle opzioni a riga di comando, e di inviare richieste al server sulla base di esse (come richiesto nella specifica), esclusivamente attraverso l'API.

Per la gestione delle opzioni si utilizza (*OptQueue_t*), nella quale il parsing appoggia le opzioni che poi vengono estratte. All'estrazione ogni opzione viene gestita con le opportune chiamate alle API.

Si è scelto che il programma client al primo fallimento di un'operazione, chiuda la connessione e termini.

4.1 Interfaccia per interagire con il server (API)

Il prototipo e l'implementazione delle funzioni per inviare richieste per creare/rimuovere/aprire/scrivere/… file nel/dal file storage server, seguono la segnatura del testo del progetto, ad eccezione del metodo *openFile*.

In esso è stato aggiunto il parametro *dirname*, indicante il nome della directory dove scrivere l'eventuale file espulso dal server.

Si è ritenuta opportuna tale scelta, considerando che un file, seppur vuoto, esista e debba essere conteggiato nel numero totale di file contenuti nel server, e così facendo la creazione di esso (tramite la *openFile* con flag O_CREATE), possa comportare l'eventuale espulsione di un file.

Si noti che tutti i nomi di file, se ricevuti come path relativi, vengono trasformati in path assoluti prima della chiamata al server, in quanto, come specificato nel testo del progetto, il server li memorizza usando il path assoluto.

4.2 Funzione di append

Come estensione è stata sviluppata la funzione di **append** che permette al client di appendere ad un file presente nello storage server il contenuto di un file presente sul file sistema locale del client.

Per l'invocazione sono state introdotti due nuovi argomenti che il client può utilizzare da linea di comando, nello specifico:

- *-a : file,localfile*: file del server a cui fare append e file locale del file system da cui leggere il contenuto da aggiungere
- *-A dirname* : che specifica la cartella del file system del client dove memorizzare i file che il server rimuove a causa dell'aumentata dimensione del file specificato con l'opzione *-a*

5 Test

Per il test l'opzione *valgrind* è stata utilizzata anche per il programma client. È stato quindi verificato che anche per il client il numero di *malloc* ed il numero di free coincidono e non ci sono errori.

Prima dell'esecuzione dei test finali sono stati condotti test specifici per tutte le funzioni attraverso un programma driver di prova. Per ogni funzione sono stati condotti sia per i casi di successo, con vari parametri che le varie casistiche degli errori.

Il Test1 verifica l'esecuzione di tutte le funzioni esercitabili dal cliente attraverso i parametri passati per argomento.

Il Test2 è stato scritto per attivare richieste da più client contemporaneamente ed innescare dell'algoritmo di rimozione. A seconda della tempistica di elaborazione delle richieste può quindi generare errori di *ENOENT No such file or directory*, causati dal fatto che l'algoritmo di rimozione può eliminare files sui quali sono in corso operazioni da parte di altri client. Tale esito è quello atteso e anche se nell'out sono presenti messaggi di errore.

6 Note

Nel progetto, oltre alla funzione append (si veda par. 4.2.), sono state sviluppate anche le seguenti parti, non richieste nella versione ridotta:

- le operazioni *lockFile* e *unlockFile*
- l'opzione del client *-D*

Il codice è stato sviluppato utilizzando un repository Git, che è accessibile al seguente link https://github.com/fabriziopini/unipi-sol-filestorage.

6.1 Opzioni di compilazione

Nella fase di test le opzioni del compilatore utilizzate sono state: *-Wall -pedantic*

Verificati che gli warning non fossero rilevanti, per una maggiore pulizia, nel make del pacchetto di consegna sono state rimosse.

6.2 GNU extension usate

Di seguito le GNU extension utilizzate:

- Subscript of a pointer to void
- Arithmetic on a pointer to void

6.3 Opzioni di consegna

Per la consegna è stato creato il target *consegna* nel makefile di progetto che crea il file *tar.gz*.

Il comando per l'estrazione è quindi: *tar –xvzf Fabrizio_Pini- CorsoA.tar.gz –C <dirname>*