Sistemi Operativi e Laboratorio B

Progetto Membox

Relazione sul Secondo Frammento del Progetto

membox.c

main

Il nostro main si apre con una serie di veloci dichiarazioni di variabile. Un array di dimensioni fisse ‘config’ salverà i dati letti dalla funzione readConfig per poi essere immagazinati in opportune variabile condivise. Passando sopra le variabili dal significato ovvio uso un pthread_t* ‘thrds’ che poi alloco a dimensioni uguali al pool di thread, per salvare gli id unici ritornati dalla pthread_create.

Con le struct sigaction s ed r inizializzo due diverse funzioni da passare ai signal handler che ora andrò a bloccare. r.sa_handler punterà alla funzione printLog per curarsi delle richieste dell’interruzione SIGUSR1, mentre ad s.sa_handler connetterò shutDown, che si occuperà di chiudere “immediatamente” il server.

Da qui in poi procedo ad aprire il file di configurazione passatomi via argomento, ed a controllare che ciò che vi leggo sia legale ai fini del server. Mi prendo la libertà di eguagliare il numero massimo di connessioni possibili al numero di thread dichiarati nel pool, in caso quest’ultimi siano superiori al primo.

Alloco finalmente thrds ed inizializzo una single linked list che userò per salvare le connessioni in attesa, usando come puntatore alla testa l’elemento condiviso ‘head’.

Alloco una struttura dati d’hash condivisa usando la funzione icl_hash_create dall’incluso hash.c e le funzioni di hashing e comparazione trovate in hash_test.c. La scelta di 1087 come numero di bucket deriva dal fatto che esso è un numero primo, vicino a 1024, ed i numeri primi portano ad un minor numero di connessioni.

Ora se un file di log mi è stato indicato nel file di configurazione lo apro, e mi preoccupo di pulire eventuali socket rimasti in socketpath prima di chiamare la mia funzione startConnection ed assegnarne il valore di ritorno alla variabile condivisa highSocID, che ospiterà l'id del socket in socketpath.

Ora posso allocare i thread, partendo dal dispatcher che si occuperà di accettare le connessioni su highSocID e salvarle nella lista condivisa, per arrivare ai vari worker, allocati all'interno di un ciclo for.

A questo punto il main si mette in attesa del join sul thread dispatcher, e poi in un ciclo for attenderà il join sui thread del pool.

Poi chiamerà la funzione di log, la stessa usata da SIGUSR1, e si occuperà di chiudere i file descriptor ancora aperti e deallocare le varie strutture dati usate dai worker.

Quest'intero processo, dal join dei thread al return 0, avverrà ogni volta che uno dei segnali connessi a s.sa_handler verrà chiamato.

dealmaker: il worker

Inizia con una chiamata ad **initActivity**, che a seconda del flag (1, -1) passatogli incrementa o decrementa il numero di thread effettivamente attivi. Questa è una funzione non richiesta dalla consegna che è stata implementata principalmente a scopo di debugging. Essa infatti avrà come return value un int unico al thread chiamante, che esso potrà usare per identificare se stesso.

Entro in un ciclo controllato dalla variabile condivisa 'overlord'. Questa variabile sarà modificata solamente dalla funzione shutDown ed è usata per segnalare ai thread di concludere il lavoro.

Per prima cosa un'instanza di dealmaker cercherà di acquisire mutua esclusione sulla variabile 'coQU', abbreviativo per connection queue. coQU è usata ogni volta che un thread deve manipolare la lista delle connessioni. Se una volta ottenuta la mutua esclusione il thread troverà la lista vuota, e se overlord è ancora valido, esso si metterà in attesa sulla variabile di condizione 'coQUwait'. Altrimenti, in caso overlord sia stato

azzerato, sbloccarà il lock sulla mutua esclusione e dopo aver diminuito initActivity lancerà la pthread_exit, per essere poi raccolto dal main nella join.

In caso vi siano invece elementi in coda il thread eseguirà il minimo numero di operazioni necessarie a rimuovere l'elemento di testa e salvarlo per se, per poi rilasciare la mutua esclusione.

Da qui entra nel secondo blocco del suo ciclo principale.

Brevemente, acquisisce la mutex sulla variabile stCO che d'ora innanzi sarà usata per l'accesso alla struttura dati mboxStats ed incrementa il numero di connessioni attive.

Entra poi in un secondo ciclo while, dal quale può uscire solo se fallirà a leggere dal socket che ha preso dalla lista delle connessioni.

Legge il messaggio salvato sul socket, salva l'operazione richiesta e passa l'intero messaggio insieme all'id del socket che lo ha inviato e la copia dell'operazione originale alla funzione selectorOP, che snocciolerà il messaggio, ne eseguirà l'operazione e restituirà una risposta.

Chiama sendReply su messg, che adesso è stato modificato nel messaggio di risposta atteso dal client durante il suo passaggio per selectorOP, dopodiché libera gli spazi allocati e ricomincia il ciclo, aspettandosi un nuovo comando sullo stesso socket. In caso di errore di lettura, uscirà dal ciclo while più interno.

Uscito controllerà la variabile globale replock che governa il lock sulla repository, per assicurarsi che il client non l'abbia lasciata settata in suo favore.

Chiude così il socket temporaneo e decrementa il numero di connessioni attive, loopando su overlord.

Se overlord è stato settato a 0 da un'interruzione è il momento di diminuire il conto dei thread attivi ed eseguire la pthread_exit.

dispatcher: Gestisce le connessioni

Anche questa funzione lavora in un ciclo dominato dalla variabile globale overlord. Al suo interno eseguirà la accept sulla variabile globale highSocID ed in caso di retval > 0 cercherà d'acquisire la mutex su coQU.

Acquisita, controllerà il numero totale di connessioni attive + in coda, ed in in caso esse siano < delle massime connessioni consentite aggiungerà un elemento alla coda ed eseguirà la signal su coQUwait. In caso siano > invierà un messaggio di OP_FAIL al client che avrà tentato la connessione, e ne chiuderà il socket.

Uscito dal ciclo while eseguirà una broadcast su coQUwait, per svegliare ogni possibile thread e spronarli a controllare overlord, e poi eseguirà la pthread_exit.

selectorOP: gestisce i messaggi letti

Ogni visita a selectorOP inizia acquisendo la mutua esclusione sulla struttura dati condivisa che implementa la mia tabella d'hash. Anche se non sto andando a guardare specificatamente dentro quella struttura, blocco altri dal modificare la variabile 'replock', che controllo e comparo con l'id del socket dal quale mi è arrivato il messaggio di cui mi sto curando.

Se replock è != da quest'id ed è != da -1 (valore di default scelto poiché impossibile per un socket) setto 'result' ad 1, stato di fallimento, e l'operazione nell'header del messaggio a OP_LOCKED.

Immediatamente sotto, ancora prima di entrare nello switch che gestisce le altre operazione, controllo che result non sia stato settato ad 1 dall'aver trovato la repository bloccata. Se ciò non si è verificato sono libero di continuare.

La PUT_OP è forse la più prolissa delle operazioni, dato che deve controllare se il messaggio che cerco di aggiungere supera i limiti impostati per la mia repository nel file di configurazione. Per far ciò si aiuta con mboxStats, acquisendo la mutex su stCO per poterla osservare indisturbato.

Se il dato da inserire passa questi test creo una copia delle parti del messaggio inviatomi che voglio salvare: key, len and buf. Passo alla funzione icl_hash_insert questi nuovi puntatori invece di quelli del messaggio, poiché li libererò uscito da selectorOP.

Se fallisco l'inserimento eseguo una find di conferma sulla chiave che volevo inserire, e poi procedo a settare l'op del messaggio originale come è più consono, ed a liberare i nuovi puntatori allocati per l'inserimento nella dataTable.

UPDATE_OP funziona in maniera molto simile, anche lui copia tutti i dati che voglio andare ad inserire in nuovi puntatori, ed esegue la mutex sulla struttura dati. Ma anziché disporre di una funzione propria, UPDATE_OP si affida ad una find seguita da una delete ed una insert per ottenere il suo scopo.

Spiegate queste le altre operazioni seguono la stessa forma.

All'uscita dallo switch acquisisco la mutex su stCO per aggiornare le statistiche di mboxStats tramite statOP.

shutDown: per chiudere in maniera pulita

Questa funzione si occupa di girare un singolo flag, la variabile statica overlord, da 1 a 0. Overlord è una variabile di tipo volatile sig_atomic_t, considerata sicura per azioni all'interno di un interruption handler.

Inoltre si occupa di eseguire la shutdown sul socket condiviso highSocID, che allora interromperà la accept nel dispatcher ed assicurerà la chiusura dei thread.

cleanList, cleaninData, cleaninFun

Semplici funzioni che liberano spazio allocato per strutture dati temporanee. cleanList si occupa di ciò che è rimasto della lista delle connessioni, mentre le altre due sono usate come parametri per icl_hash_destroy e icl_hash_delete.

printLog: gestisce SIGUSR1

Acquisita la mutua esclusione sul file delle statistiche (variabile stCO) lancia la funzione printStats.

readConfig, readLine, readLocation

Diversi usi dello stesso concetto, implementato in maniera leggermente diversa ogni volta a seconda di cosa guardino. readLocation e readLine si specializzano nel leggere una stringa, mentre readConfig, usata per leggere e parsare i valori numerici del file di configurazione, si aspetta esattamente 5 valori in posizioni prestabilite. Scambiare la posizione dei valori nel file di configurazione non è supportato.

statOP, statConnections

Entrambi lavorano sulla struttura dati condivisa mboxStats e lasciano l'acquisizione della mutua esclusione su di essa ai loro chiamanti. Semplicemente aggiornano le statistiche interne alla struttura dati in base ai parametri passativi.

Connections.c

openConnection

Apre una connessione AF_UNIX dal client al server permettendo l'impostazione di un timeout ripetuto per un certo numero di tentativi, entrambi passati come argomenti, oltre i quali il client assumerà il server offline.

sendReply, readReply

sendReply si occupa di inviare una funzione di risposta dal server al client, scegliendo se mandare solo l'header del messaggio oppure sia l'header che i dati a seconda dell'operazione alla quale sto rispondendo.

readReply è la sua speculare, leggendo solo ciò che l'operazione richiede, per uso nel client.

readHeader, readData

Si comportano allo stesso modo, creando un buffer temporaneo storage nel quale salvare i dati letti dal socket prima di copiarli nel messaggio già allocato per la ricezione (nel server questa allocazione avviene all'interno di **dealmaker**).

In readData faccio particolare attenzione a non allocare un buffer in caso length sia zero, per evitare memory leak (anche un alloc di dimensione esattamente uguale a length crea un leak di 0 byte su tanti blocchi quanti alloc di size 0 sono stati chiamati).

sendRequest, sendHeader, sendData

sendHeader e sendData sono semplicemente la prima e seconda metà di sendRequest, separate per comodità di sendReply.

Come le funzioni di lettura da socket questa usa un buffer e la funzione memcpy per scrivere i dati della struttura messaggio passatale sul socket. Si occupa inoltre di liberare questo buffer (ma non la struttura dati originale).

membox.sh

Script diviso in due parti.

La prima esegue, attraverso il comando case sulla variabile key eguagliata agli argomenti (presi uno ad uno), il parsing dei dati.

L'ordine non è importante, e la path del file verrà assunta essere l'ultimo comando non riconosciuto. Uscito dallo switch, lo script controlla che il path passatogli abbia effetivamente significato, ed in caso d'errore termina con un messaggio appropriato.

Quando lo script identifica un comando esso setta alcune variabili che poi userà nella seconda parte. Innanzitutto setta il valore di una variabile chiamata come il comando al quale siamo interessati (in tutte maiuscole) a true. Poi cambia il valore delle variabile numerate, dal loro 1 di default al valore del loro nome. Questa è infatti la posizione del dato richiesto da quel parametro nella riga del file di log del server, contata dal comando 'cut' grazie al numero di spazi bianchi. Inoltre setta una variabile che per comodità ho chiamato uguale al nome del comando che il parametro vuole guardare (in sole minuscole) ad una riga di testo che sarà poi stampata per dare una formattazione all'output.

La seconda parte dello script controlla se una qualunque delle variabili è settata a true, ed in caso positivo sà che l'utente ha passato almeno un comando.

Grazie alla concatenazione di stringhe di testo vuote e non ed al fatto che all'uso del comando cut sulle variabile numeriche che non ripete il proprio output della riga per una posizione già stampata, posso creare un ciclo che stampa la combinazione di dati richiesta dall'utente senza dover testare ogni possibile combinazione.

Unica differenza è l'opzione -m, che richiede un conto per la stampa del valore più alto di connessioni contemporaneo mai raggiunto dal server. L'opzione -m inoltre fa uso di elementi stampati, nel file mbox_stats.txt, prima di elementi usati dall'opzione -o. Per una limitazione dell'utility cut non posso presentare questi dati in ordine diverso da quello col

quale appaiono sul file a meno di non chiamare cut una seconda volta, che è esattamente ciò che faccio dopo aver testato che l'opzione -m sia effettivamente richiesta. Uso una serie di tabulazioni, spazi vuoti e newlines per mantere consistente la formattazione.

Se nessuna delle variabili UPPERCASE è stata trovata uguale a true, lo script esegue tail dell'ultima linea del file di log, poiché l'utente non avrà passato alcun parametro.