Laboratorio di Programmazione di Sistema

a.a. 2015/2016

MEMBOX++

Daniele Trezza

489554

daniele.trezza@live.it

Danilo Spanò

465347

spano25@tiscali.it

Indice

1. Introduzione

Il progetto membox++ è stato realizzato dagli studenti Trezza Daniele (489554) e Spanò Danilo (465347) su proposta dei docenti Pelagatti e Torquati per la verifica delle competenze acquisite nell'ambito del modulo di Programmazione di Sistema del corso di Sistemi Operativi e Laboratorio (277AA).

Il progetto è stato realizzato, come da specifiche, interamente in linguaggio C in ambiente LINUX (distribuzione: Xubuntu 15.04) a 64 bit.

2. Descrizione

Il progetto prevede la realizzazione di un server concorrente per la gestione di uno spazio di memorizzazione (*repository*) in grado di collegarsi sulla stessa macchina a più client contemporaneamente per: ricevere oggetti da memorizzare, inviare gli oggetti richiesti, rimuoverli, aggiornarli, riservare un oggetto o l'intero repository ad un client, salvare su file tutti gli oggetti memorizzati o le proprie statistiche interne.

3. Le principali scelte di progetto: strutture dati

Il repository è stato realizzato con una tabella hash, sfruttando i file suggeriti dai docenti. In questo modo, le funzioni di ricerca ed eliminazione hanno un costo computazionale vantaggioso rispetto ad altri tipi di strutture dati presi in considerazione quali liste o code.

Per tenere traccia dei client connessi e servirli con politica FIFO, è stata implementata una coda in grado di memorizzare i file descriptors dei client da servire. A questa coda sono state associate una variabile mutex e una variabile condition:

- La variabile lock_coda di tipo pthread_mutex_t è necessaria per garantire la mutua esclusione in fase di lettura o modifica della coda.
- La variabile cond_coda di tipo pthread_cond_t è necessaria per segnalare al thread worker in attesa quando una nuova connessione da gestire è stata inserita in coda.

Per implementare la lock su oggetti, si è scelto di usare una lista i cui nodi contenessero in un campo il file descriptor del possessore della lock e in un altro campo la chiave dell'oggetto riservato a quel client.

La funzione di inserimento inserisce un nuovo elemento in testa, per l'operazione di ricerca sono state implementate 2 funzioni: la prima (*cerca (lista_t l, unsigned long n)*) cerca ricorsivamente la chiave passata come argomento e restituisce l'fd del possessore (se esiste) altrimenti 0, al fine di capire se l'oggetto preso in esame è in stato di lock.

La seconda funzione di ricerca (*cerca_client (lista_t l, int fd, lista_t * out)*) si occupa di cercare tutti i nodi che hanno come campo fd_possessore il file descriptor passato come argomento e inserirli in una lista ausiliaria al fine di cancellare tutte le lock di un client nel caso in cui quest'ultimo non abbia richiesto la UNLOCK_OBJ_OP prima di chiudere la connessione.
A questa lista è associata una variabile condition *condLista*. In questo modo, quando un client richiede la LOCK_OBJ_OP su un oggetto già in stato di lock, il thread ad esso associato si mette in attesa che l'oggetto venga liberato. Il segnale per risvegliare il primo thread in attesa viene inviato quando la UNLOCK_OBJ_OP va a buon fine, oppure quando vengono cancellati degli elementi dalla lista al momento della chiusura di una connessione. Grazie all'uso della variabile condition, la politica di riattivazione dei thread in coda è FIFO.

4. Le principali scelte di progetto: Algoritmi

A partire dalla funzione main contenuta nel file membox.c, in questa sezione verranno commentati tutte le funzioni e gli algoritmi utilizzati, soffermandosi sui più significativi.

Dopo aver controllato il numero di parametri passati come argomenti alla funzione *main*, vengono mascherati tutti i segnali e installati i gestori solo per i seguenti segnali:

- SIGINT, SIGQUIT, SIGTERM: hanno comportamento identico, viene stampato un messaggio di successo, cancellate tutte le strutture dati in memoria principale e il server termina immediatamente.
- SIGUSR1: vengono stampate le statistiche interne sul file specificato nel file di configurazione del server, nel formato richiesto dalla specifica.
- SIGUSR2: Il server smette di accettare nuove connessioni settando la variabile `closing` a 1, in questo modo si conclude il ciclo infinito del server senza eliminare i thread worker che non hanno ancora finito, viene poi inserita in coda una connessione fittizia con fd = -1 per ogni thread del pool, in modo che questi ultimi terminino quando estraggono dalla coda tale connessione. Infine, il server esegue la join per attendere la conclusione di tutti i worker, ripulisce la memoria e conclude.
- SIGPIPE: viene ignorato, per evitare terminazioni indesiderate dovute alla scrittura su un socket chiuso.

A questo punto, dopo aver inizializzato la coda dei lavori, la tabella hash e la struttura dove memorizzare la configurazione (dichiarate nell'ambiente globale), si effettua il parsing delle opzioni passate come argomento, che hanno la seguente sintassi e semantica:

- -f membox.conf : parametro obbligatorio, indica il file di configurazione dal quale ricavare le opzioni principali del server. Viene invocata la funzione `static int leggi_File_config(FILE* fd, struct config *fc_t)` che legge il file una riga per volta, ignorando le righe che iniziano con il carattere '#' e assegna i valori ai campi della struttura *config*. I valori numerici vengono opportunamente convertiti mediante la funzione di libreria `strtol(const char *str, char **endptr, int base)`.
- [-d dumpfile] : parametro opzionale, indica il file di dump dal quale caricare il contenuto del repository elaborato in una precedente esecuzione del programma. Seguendo la struttura del file, vengono prima letti i valori corrispondenti alla capienza necessaria allo storage da confrontare con i parametri *StorageSize* e *StorageByteSize* dell'esecuzione attuale, poi per ogni oggetto viene allocata la quantità di memoria richiesta nella riga corrispondente alla size e si inseriscono i dati nella tabella hash. Il file viene infine rinominato aggiungendo al suo nome il suffisso *".bak*".

In seguito, viene inizializzato il pool di thread: un array di thread di dimensione specificata dal file di configurazione al parametro ThreadsInPool con funzione iniziale `void *worker_thread()` (descritta più avanti).

Dopo aver creato il socket di tipo AF_UNIX mediante la funzione *openConnectionServer* da noi aggiunta nel file *connections.h*, il server entra in un ciclo infinito, dove:

- Verifica, tramite la variabile condivisa `closing`, può accettare nuove connessioni
- Accetta una nuova connessione salvando il file descriptor relativo al client nella variabile `connfd`

Qualora il numero attuale di connessioni sia maggiore del numero massimo di connessioni possibili (specificato col parametro `MaxConnections`), il server invia un messaggio di errore al client, altrimenti inserisce in coda il client (identificato tramite il suo fd).

4.1. Thread Worker

Ciascun thread worker contenuto nel pool di thread ha come funzione iniziale `void*worker_thread()`. Viene qui descritto il comportamento generale di un thread:

la funzione principale dei worker consiste di un ciclo infinito dove:

- Si estrae dalla coda l'identificatore del client (se la coda è vuota, la funzione `estrai` attende che venga chiamata la signal sulla variabile condition associata alla coda), qualora il valore estratto fosse pari a -1, il thread termina con successo perché è stato ricevuto dal server il segnale SIGUSR2.
- Viene letto il messaggio in arrivo dal client.
- Si controlla se il repository è in stato di lock tramite la variabile condivisa "islocked" (posta a 0 in caso di assenza di lock, e pari all'fd del client in possesso della lock se il repository è bloccato). Se il repository risulta bloccato e la lock non appartiene al cliente che questo thread sta servendo, il thread invia un messaggio di errore OP_LOCKED.
- Se l'operazione richiesta è di lock su un oggetto (LOCK_OBJ_OP) , ma l'oggetto è già stato bloccato da un altro client, il thread si mette in attesa sulla variabile condition relativa alla lista di lock su oggetti. In questo modo, sfruttando la coda associata alle variabili condition, i client che richiedono lock sullo stesso oggetto, saranno serviti con politica FIFO.

- Attraverso uno switch, viene compresa ed eseguita l'operazione richiesta (le operazioni sono descritte in seguito)
- Se al termine delle operazioni il repository risulta ancora in stato di lock dal client che si sta servendo, si rilascia la lock.
- Si rilasciano anche eventuali lock su oggetti residue eliminandole dalla lista (si chiama la signal quando un oggetto viene eliminato dalla lista).
- Si chiude la connessione.

4.2. Le operazioni fornite dal server

Il thread worker, tramite una operazione di switch, comprende quale operazione è stata richiesta e si comporta come descritto:

- PUT_OP : vengono letti i dati inviati, in caso di errore invia un messaggio con generico messaggio di fallimento OP_FAIL. Si verifica che la dimensione dell'oggetto non ecceda la massima dimensione possibile per lo storage e che sia possibile inserire un nuovo oggetto, qualora non fosse possibile si procede all'invio di un opportuno messaggio di errore, altrimenti si verifica che la chiave non sia già presente nel repository e si procede all'inserimento. Le variabili `ActualStorageSize` e `ActualStorageByteSize` vengono opportunamente incrementate.
- UPDATE_OP: si leggono i dati, si verifica che la chiave sia presente nel repository, si aggiorna l'oggetto dopo aver controllato che la lunghezza dei nuovi dati sia la stessa.
- GET_OP: se l'oggetto è presente, viene inviato al client, altrimenti si risponde con un messaggio di errore OP_GET_NONE

- REMOVE_OP: si verifica che la chiave sia presente e si procede alla rimozione dell'oggetto, decrementando opportunamente le variabili `ActualStorageSize` e `ActualStorageByteSyze`.
- LOCK_OP : il server concede la lock sull'intero repository settando la variabile `islocked` al valore del file descriptor del client.
- ULOCK_OP: se il server è in stato di lock, per i controlli effettuati in precedenza, la lock appartiene sicuramente al client che si sta servendo, quindi la variabile `islocked` può essere settata a 0 senza ulteriori controlli.
- LOCK_OBJ_OP : Si controlla che l'oggetto sul quale è stata richiesta la lock sia presente nel repository, si cerca nella lista degli oggetti bloccati se il cliente è già proprietario della lock sull'oggetto in questione, se tutti i controlli sono andati a buon fine, si aggiunge alla lista un nodo che rappresenta l'oggetto da bloccare, altrimenti viene inviato un opportuno messaggio di errore.
- UNLOCK_OBJ_OP : Il nodo relativo all'oggetto da sbloccare viene eliminato. Non è necessario controllare in questo punto che il client sia proprietario della lock perché tale controllo è già stato effettuato in precedenza.
- DUMP_OP : si controlla che il repository sia in stato di lock e che nel file di configurazione sia stato specificato il path per il dumpfile. Se nel percorso specificato è già presente un file di dump, quest'ultimo viene rinominato aggiungendo al suo nome il suffisso ".bak". A questo punto, può essere creato un nuovo file binario che conterrà tutti i dati dell'attuale elsecuzione. Si è ritenuto necessario modificare la funzione `icl_hash_dump` fornita dai docenti in modo che fosse in grado di stampare i dati contenuti nel repository su un file di tipo binario nel formato richiesto dalla specifica.

4.3 Statistiche interne

Le statistiche interne del server sono calcolate usando una struttura dichiarata nell'ambiente globale (insieme ad una variabile mutex ad essa legata). Sono state dichiarate alcune funzioni ausiliarie per l'aggiornamento dei valori. In particolare, la funzione `aggiorna_operazioni(int op, int esito)` prende in ingresso l'intero relativo all'operazione da aggiornare e, in base al valore del parametro esito, incrementa il numero di operazioni di quel tipo, oppure incrementa di uno il fallimento per l'operazione in esame. Dopo aver letto il messaggio proveniente dal client, il thread worker si occupa di incrementare immediatamente il contatore relativo all'operazione richiesta e, solo in caso di esito negativo dell'operazione, aggiorna il numero di fallimenti. Il numero di connessioni in coda viene calcolato all'interno del gestore di SIGUSR1 con la funzione `update_concurrent_connections(Coda_t *q)`.

La funzione `aggiorna_oggetti(int a)` incrementa il numero di oggetti (settando anche il massimo numero di oggetti raggiunto) o lo decrementa, in base al valore del parametro 'a' e viene chiamata al termine di una PUT_OP o REMOVE_OP insieme alle funzioni `void incrementa_current_size(int val)` e `void decrementa_current_size(int val)` che si occupano di conteggiare le dimensioni dello storage espresse in numero di byte aggiornando, eventualmente, anche la dimensione massima raggiunta.

Script

Al progetto, è stato associato uno script bash per rendere più semplice la visualizzazione delle statistiche interne del server. Lo script, infatti, dato in input un file di statistiche, e alcuni parametri, stampa a video tutti i valori dei parametri richiesti. In assenza di parametri, stampa tutte le statistiche.

Si esegue col comando `./mboxstat.sh filename[--help] [-p] [-u] [-g] [-r] [-c] [-s] [-o] [-m] [-d]`

5. Conclusioni

In questo file sono state riportate le principali scelte di progetto ritenute essenziali alla comprensione del lavoro svolto.

Per eventuali chiarimenti, si rimanda al codice corredato dagli opportuni commenti.