RELAZIONE PROGETTO

LABORATORIO DI PROGRAMMAZIONE DI SISTEMA

Alessio Matricardi – Matricola 580159 – ultimo aggiornamento 31/05/2020

Breve introduzione

Il progetto in questione è costituito dalle cartelle src, include, lib, var, da 3 file di configurazione (input), un makefile per compilare ed eseguire il codice ed uno script bash per il parsing del file di log prodotto. La cartella *src* contiene tutti i file sorgente, la cartella *include* tutti i file header, la cartella *lib* contiene eventuali librerie prodotte. A run time viene prodotta la cartella *var,* contenente file utili alla fase di parsing del file di log prodotto al termine dell'esecuzione. Questi file contengono rispettivamente l'id del processo supermercato e il nome del file di log da parsare

Tutto il codice è conforme allo standard POSIX 2001, ma si è cercato di evitare l'utilizzo di versioni deprecate (anche recentemente) per permettere, volendo, un "salto" di versione.

Il progetto è stato realizzato con il sistema operativo Ubuntu 20.04 64 bit ed attraverso l'ausilio dei seguenti software: gcc 9.3.0, valgrind 3.15.0, gdb 9.1, make 4.2.1.

Parsing del file di configurazione

Il formato del file di configurazione è abbastanza semplice: ogni riga al suo interno può essere un commento (la riga deve iniziare per il carattere #) oppure un assegnamento. L'assegnamento è del tipo IDENTIFICATORE=VALORE. Sono ammessi spazi bianchi prima e dopo il simbolo =.

Il file viene letto e vengono salvate le associazioni identificatore-valore in una tabella hash. La tabella hash utilizzata proviene dalla soluzione proposta di un esercizio di laboratorio, tutti i diritti di copyright sono specificati nei file icl_hash.h e icl_hash.c.

Successivamente dalla tabella hash tutti dati nella tabella hash vengono opportunamente salvati in una struttura che li incapsula tutti. Il salvataggio su questa struttura viene effettuato con l'ausilio di macro che rappresentano ognuna un identificatore, è perciò opportuno non modificare in alcun modo la sintassi di un identificatore nel file di log, pena l'uccisione del processo per errato parsing. Sui valori viene effettuato un rigido controllo di qualità (devono essere rispettati vincoli quali dati non negativi, numero di casse inizialmente aperte minore del numero di casse totali, etc).

Il thread supermercato

Quando viene avviato il processo, il thread main rappresenta il thread supermercato. Inizialmente il thread alloca e inizializza tutte le strutture necessarie all'esecuzione. Dopo ciò, lancia in modalità detached il thread signal_handler (vedi poi) che si occuperà della gestione dei segnali. Successivamente, lancia il thread direttore e C thread clienti. Dopo aver fatto ciò, e fino a che non viene ricevuto un segnale SIGHUP o SIGQUIT, il suo compito si riduce ad aspettare che escano E clienti per inserire E nuovi thread all'interno del processo. Il supermercato è a conoscenza dei thread clienti terminati grazie ad un array condiviso con ognuno di loro, in modo da poter chiamare la pthread_join sui E thread clienti terminati ed immetterne di nuovi. Inoltre, il supermercato si occupa del salvataggio della struttura dedicata ad ogni cliente in una coda, in modo da poter stampare tutte le statistiche del caso nel file di log alla terminazione del processo. Quando il suo stato viene modificato dal signal handler (lo stato del supermercato varia a seconda del segnale ricevuto), attende la terminazione di tutti i thread clienti e successivamente avvisa il direttore, attraverso una variabile volatile (il cui accesso è atomico), in modo che anch'esso possa terminare. Dopo aver chiamato un'opportuna funzione che si occupa del salvataggio delle statistiche sul file di log, si occupa della deallocazione di tutte le strutture/mutex/variabili di condizione.

Il thread signal handler

Questo thread è forse il più "snello" di tutti, in termini di line of code. Dapprima vengono mascherati i segnali SIGHUP, SIGQUIT E SIGUSR1 (operazione effettuata da TUTTI i thread del processo).

Perché SIGUSR1? SIGUSR1 viene utilizzato da tutti i thread del processo per avvisare il signal handler che c'è stato un errore critico durante l'esecuzione (errori con variabili di mutua esclusione o di condizione, push/pop da coda, creazione/join di thread). Sarà il signal handler a doverlo gestire.

Fatto ciò, il signal handler si mette in attesa sincrona di un segnaled sulla sigwait. Ho deciso di adottare tale soluzione perché la preferisco di gran lunga all'installazione di un handler tramite sigaction: utilizzando un gestore asincrono sarei stato costretto ad utilizzare solo funzioni signal safe, mentre con la sigwait ho avuto la possibilità di utilizzare funzioni della libreria pthread senza alcun problema.

Quando viene catturato il segnale SIGHUP, il signal handler modifica lo stato di supermercato e direttore a CHIUSURA. Se il segnale è SIGQUIT, il signal handler modifica lo stato di supermercato e direttore a CHIUSURA_IMMEDIATA e setta una speciale variabile volatile che serve ad avvisare i clienti che da quel momento in poi non dovranno più chiedere l'autorizzazione al direttore in caso di uscita senza acquisti (in dettaglio poi).

Inoltre, se il segnale catturato è SIGUSR1 (questo viene lanciato da un thread casuale in caso di errore critico), il signal handler UCCIDE l'intero processo lanciando una SIGKILL verso se stesso (id del processo). La chiamata SIGKILL, non gestibile a qualsiasi livello, terminerà l'esecuzione non preoccupandosi di liberare la memoria (non sarà più una nostra priorità).

Il thread cassiere

Il thread cassiere viene lanciato dal direttore all'apertura del supermercato (se è una delle casse iniziali) o quando si vengono a creare le condizioni (in dettaglio nella gestione casse del direttore).

Fino a che resta aperta (stato = APERTA) ogni cassa a guardare la propria coda.

Ogni coda di una cassa è una boundered queue ad accesso esclusivo (utilizzando mutex/variabili di condizione) di dimensione massima uguale a C (numero massimo di clienti presenti nel supermercato). È stata realizzata con un array di puntatori a void* in modo da potervi immettere qualsiasi tipo di variabile e/o struttura dati. La sua particolarità è che la pop, in caso di coda vuota, mette in attesa il thread invocante per un massimo di X millisecondi utilizzando la chiamata di libreria *pthread_cond_timedwait.*[1]

Se, scaduto il quanto di tempo, non è arrivato alcun cliente, allora il cassiere invia la notifica al direttore scrivendo su una variabile condivisa con lui la dimensione attuale della sua coda, altrimenti il cliente deve essere servito. Dopo aver calcolato il tempo di servizio necessario questo viene confrontato con l'intervallo di tempo che ogni cassiere deve rispettare per inviare una notifica al direttore: se il tempo di servizio è maggiore dell'intervallo di una notifica, il tempo di servizio viene "spezzettato" in modo da garantire sia che le notifiche vengano inviate in tempo, sia che il cliente venga servito senza alcun ritardo. Ad esempio, se t_servizio fosse 1300 e t_notifica fosse 300, allora il tutto viene gestito nel seguente modo:
attesa 300 ms – invio notifica – attesa 300 ms – invio notifica – attesa 300 ms – invio notifica – attesa 300 ms – invio notifica – attesa 100 ms. Questo comporta, ovviamente, che la notifica successiva dovrà essere inviata dopo 300-100=200 ms. L'attesa passiva viene realizzata attraverso la funzione *msleep*, una funzione che adatta la funzione nanosleep per un'attesa in millisecondi.Al termine del servizio, sveglia il cliente che si era immesso nella coda.

Questa soluzione è stata preferita alla creazione di un secondo thread POSIX, che avrebbe permesso una gestione più "semplificata" dei tempi, poiché avremmo appesantito ulteriormente il processo di K ulteriori thread cassieri e si sarebbe dovuta gestire la doppia creazione/terminazione di entità attive che concettualmente lavorano come fossero una sola.

Quando una cassa viene chiusa in modo decisionale dal direttore, questa la libera invitando i clienti a cambiare cassa e successivamente termina la sua esecuzione.

Quando viene chiusa dal direttore a causa di un segnale ricevuto, allora si occupa della gestione dei clienti servendoli (se stato = SERVI_E_TERMINA) o facendoli uscire immediatamente (stato = NON_SERVIRE_E_TERMINA). In entrambi i casi, la cassa non terminerà fino a che ci saranno ancora clienti all'interno del supermercato.

[1] https://linux.die.net/man/3/pthread_cond_timedwait

Il thread cliente

Il thread cliente, all'avvio, fa subito attesa passiva di 0 < t < T millisecondi in modo da simulare il tempo passato all'interno del supermercato a fare acquisti. Successivamente controlla quanti prodotti ha acquistato: se ha acquistato 0 < p < P prodotti, allora va a scegliere in modo totalmente casuale (il suo seed è uguale al suo idcliente * time) una delle casse attualmente aperte, si inserisce in coda e fa attesa passiva aspettando di essere risvegliato dalla cassa. Dopo essere stato risvegliato, se non deve cambiare cassa, termina avvisando il supermercato (attraverso variabili condivise tra loro), altrimenti torna a scegliere in modo casuale una delle casse aperte. Se ha acquistato p = 0 prodotti ed *il supermercato non ha ricevuto il segnale SIGQUIT (regolato attraverso la variabile volatile need_auth)* si mette in attesa dell'autorizzazione del direttore su una variabile di condizione.

Il thread direttore

Il thread direttore è l'entità attiva più interessante del processo. All'avvio ha il compito di creare i thread cassieri inizialmente aperti. Fino a che è aperto il supermercato (lo stato – o meglio dire la variabile di stato - di direttore e supermercato coincide) il direttore fa essenzialmente 2 cose: giudicare se chiudere o aprire una cassa e autorizzare i clienti che vogliono uscire senza acquisti. Per quanto riguarda la gestione delle casse: il direttore implementa un array di K posizioni di tipo struct timespec dove viene salvato, per ogni cassa, il momento ultimo in cui il suo stato è cambiato. Basandoci su questo dato, la politica è così definita:

Per aprire una cassa, se posso:

- apro la cassa chiusa da più tempo rispetto alle altre
- la apro se e solo se è chiusa da più di UPDATE_SECONDS secondi

Per chiudere una cassa, se posso:

- chiudo la cassa con meno clienti
- la chiudo se e solo se è aperta da più di UPDATE_SECONDS secondi

Dove UPDATE_SECONDS è una macro il cui valore è 4.
La precedenza viene data alla chiusura di una cassa. Questa politica implica che, nonostante ci siano casse aperte da chiudere e/o chiuse da aprire, è possibile che il direttore non intervenga in nessun modo per evitare la chiusura/apertura schizofrenica di una cassa.

Quando il processo riceve un segnale tra quelli mascherati, il suo stato viene modificato e, di conseguenza, si preoccupa di avvisare tutte le casse in quel momento aperte di procedere con il servire i clienti rimasti (SIGHUP) o farli uscire immediatamente (SIGQUIT). Dopodichè, fino a che sono presenti clienti nel supermercato, ogni 200 ms autorizza l'uscita di eventuali clienti che avevano richiesto l'autorizzazione per uscire senza acquisti. Termina dopo aver atteso la terminazione di tutte le casse ancora aperte.

Log di output

A cavallo tra la terminazione del direttore e la deallocazione, viene effettuato il salvataggio di tutte le statistiche sul file di log. Ad occuparsene è il modulo filestat. Il formato di salvataggio è quello CSV, in modo da poter facilmente tokenizzare nello script ogni riga letta. Al suo interno salviamo, in ordine:

- Le informazioni utili relative ad un cliente, per ogni cliente;
- Le informazioni utili relative ad una cassa, per ogni cassa;
- Numero di prodotti e di clienti processati dal sistema.

Per questioni legate alla semantica della pop da una coda, dopo aver stampato le informazioni relative ad un cliente, la sua struttura dati viene deallocata all'interno della funzione di stampa.

Lo script

Lo script analisi.sh aspetta che il processo termini interrogando con il comando *ps* ogni mezzo secondo. Quando è terminato, inizia la lettura del file di log prodotto (il cui nome è contenuto nel file *log_filename.txt* prodotto dal thread supermercato). Ogni riga viene analizzata con una pipe *grep | wc -l* che, tramite pattern matching, riesce a capire di che tipo di riga si tratta. Attraverso la calcolatrice *bc* vengono eseguiti tutti i

calcoli in virgola mobile e le approssimazioni a 3 cifre decimali. Per stampare in modo formattato utilizzo *printf*, omonimo della funzione di libreria C.

Istruzioni di compilazione

Per eseguire il test basta eseguire in sequenza i comandi *make* e *make test*. Al termine del processo verranno stampate sul terminale tutte le statistiche.

Alternativamente, è possibile eseguire il processo con

$ ./supermercato [-c config.txt] (config.txt è il file di configurazione di default)

$ ./supermercato -c <nome file di configurazione>

Note finali

1. Eseguendo il processo con valgrind si può notare che non ci sono memory leak, se non uno di 272 bytes apparentemente legato al thread signal handler (la cui unica particolarità rispetto agli altri thread è quella di essere detach). Cercando in rete ho potuto constatare che non si tratta di un vero e proprio memory leak, e che in ogni caso non è possibile eliminarlo.[2][3]
2. Per ottenere un esatto momento (utile per il calcolo dei tempi di chisura, valutare i ms trascorsi tra una notifica e l'altra, etc) ho utilizzato la funzione di libreria *clock_gettime* con clock_id CLOCK_MONOTONIC, preferito a CLOCK_REALTIME perché non soffre eventuali cambi di orario del sistema durante l'esecuzione.[4][5]

[2] https://sourceware.org/legacy-ml/glibc-bugs/2007-04/msg00036.html
[3] https://icevanila.com/question/memory-leaks-in-pthread-even-if-the-state-is-detached
[4] https://linux.die.net/man/3/clock_gettime
[5] https://stackoverflow.com/questions/3523442/difference-between-clock-realtime-and-clock-monotonic