Marcello Satta
580495 - CorsoB

Progetto di Laboratorio di Sistemi Operativi a.a. 2019-20
Relazione

Con la presente relazione si intendono delineare le scelte implementative che hanno portato alla realizzazione del progetto di Laboratorio di Sistemi Operativi da parte del candidato.

Introduzione

Il progetto vuole simulare il funzionamento di un supermercato con K clienti e C cassieri, rappresentati da thread. Esso è costituito da due processi principali: il processo direttore e il processo supermercato. In particolare, il primo, attraverso la chiamata di sistema fork() crea il secondo processo. Entrambi i processi rimangono in esecuzione per tutta la loro durata e si sincronizzano per la terminazione.
La loro comunicazione è garantita da un socket di tipo AF_UNIX e segnali POSIX.
I due processi posseggono dei compiti ben distinti: il processo supermercato acquista un peso sicuramente maggiore in termine di funzionalità offerte in quanto è modellato come un singolo processo multi-threaded, mentre il processo direttore si limita a gestire il numero delle casse aperte e a garantire l'uscita dei clienti che non acquistano prodotti.
Il progetto è composto dai seguenti file: director.c, supermarket.c, client.c, cashier.c, queue.c, cnfg.c e dai rispettivi header director.h, supermarket.h, client.h, cashier.h, queue.h e cnfg.h, dove sono presenti le intestazioni di funzioni e strutture dati utilizzate. Oltre ai file contenenti il codice, sono presenti tre script bash, uno di analisi (analysis.sh) e due di test (test1_script.sh e test2_script.sh), il Makefile, i due file .txt di configurazione (config1.txt e config2.txt) e il PDF contenente questa relazione.

Mi avvierò prima ad illustrare le strutture dati utilizzate per gestire l'intero progetto, per poi spostarmi sui meccanismi proposti e sulle funzionalità delle varie componenti del progetto.

Strutture dati

Vi sono tre componenti principali in gioco all'interno del progetto: i clienti, i cassieri e le code.

I primi sono implementati come un array di struct. All'interno della struct, che modella quindi il singolo cliente, sono presenti i seguenti campi: ID (tid del pthread creato), index (che indica la posizione sull'array), tre booleani (che indicano se il cliente ha pagato, è uscito, sta cambiando cassa o meno), la posizione in coda, l'indice del cassiere assegnatogli, il numero di code cambiate, il numero di prodotti acquistati e quattro struct timespec per salvare il tempo speso all'interno del supermercato, in cassa e in servizio. Ogni pthread cliente viene creato, come già sottolineato, all'interno del processo supermercato e le informazioni utili allo svolgimento dell'attività del cliente sono salvate all'interno della propria cella dell'array. Quando un nuovo cliente vuole entrare nel supermercato, occupa la posizione dell'array lasciata libera da un cliente già uscito.

I cassieri sono modellati, come i clienti, da un array di struct. L'entità struct cassiere presenta i seguenti campi: ID (tid del thread creato), index (che indica la posizione sull'array), un booleano (che indica se la cassa è aperta o meno), il numero totale di clienti serviti per cassa, l'intervallo di tempo totale in cui la cassa è rimasta aperta, il numero totale di chiusure, l'intervallo di tempo totale speso dal cassiere per servire i clienti e il numero totale di prodotti elaborati dal cassiere.

La terza entità facente parte l'ambiente supermercato sono le code (queues). Esse sono modellate come liste di elementi di tipo cliente. Ogni cassa ne possiede una e globalmente sono rappresentate da un array, i cui indici coda-cassiere corrispondono. La scelta di implementazione è ricaduta su liste concatenate. In particolare, se andiamo a scorgere per la prima volta la funzionalità di questo tipo di

struttura, ci rendiamo conto che, grazie alla possibilità di inserire elementi in testa e di rimuoverli in coda, esse simulano fedelmente i meccanismi di una coda di un supermercato reale.
Ricapitolando: un array di struct cliente; un array di struct cassiere; un array di liste concatenate che rappresentano le code (implementazione delle liste concatenate ripresa da quella proposta dai tutor del corso di Algoritmica A.A. 2018/19).
Un breve accenno alla struttura che modella il supermercato ma dal punto di vista del direttore: egli possiede una visione meno dettagliata del supermercato, infatti opera su una struct "sm" composta da un array di cassieri (composti a loro volta da un bool cassa aperta/chiusa e un int che rappresenta il numero di clienti presenti in coda) e un intero che indica il numero di casse aperte.

Ho deciso di dichiarare queste variabili globalmente; questa scelta mi ha permesso grande elasticità e praticità di utilizzo delle strutture, tutelate grazie al meccanismo della mutua esclusione.

Mi appresto ora a descrivere le funzionalità delle varie componenti del sistema.

Il processo direttore

Il processo direttore da avvio al programma leggendo dal file di configurazione i parametri di configurazione e installando gli handler per i segnali di terminazione (SIGHUP e SIGQUIT). In seguito apre le comunicazioni e si mette in ascolto del processo supermercato.
Esso è "forkato" immediatamente dopo aver creato il socket AF_UNIX e, grazie alla execl() da questo momento vive di vita propria. Ma torniamo al processo direttore: grazie alla select (implementazione ripresa e adattata dalla soluzione dell'esercizio sulla select della lezione 21 del corso tenuto quest'anno), egli riesce a gestire più comunicazioni con i diversi cassieri. In un determinato istante di tempo che definirò più dettagliatamente quando descriverò il cassiere, uno di questi invia al direttore il proprio indice e il numero di clienti presenti in coda in quel momento. Il direttore si impegna a decidere se tenere aperta la cassa, chiuderla, o aprire una nuova a seconda dei valori soglia S1 e S2 presenti nel file di configurazione. Inoltre, il direttore, se riceve uno 0, riconosce come mittente uno dei clienti che non ha fatto acquisti, e scrivendo 1 sul socket consente al cliente di terminare la sua esecuzione (astrattamente di poter uscire dal supermercato). Il processo direttore termina quando riceve dal processo supermercato la stringa "close", a cui risponde "ok". Egli termina le comunicazioni grazie alla unlink(), libera la memoria allocata, e termina.

Il processo supermercato

Il processo supermercato, come affermato in precedenza possiede al suo interno sia i thread cassieri, sia i thread clienti. La prima operazione svolta da questo processo è leggere dal file di configurazione allegato gli stessi parametri di configurazione del processo direttore. In seguito vengono aperti i file di log (ho preferito tenere due file separati, uno per i clienti e uno per i cassieri), installati gli handler per i segnali SIGHUP e SIGQUIT, e inizializzate le variabili di condizione e le variabili di mutua esclusione necessarie alla corretta sincronizzazione tra singolo cassiere e singolo cliente (il meccanismo che sta alla base dell'intero funzionamento del supermercato), tra cassieri e tra clienti.
Successivamente vengono prima creati i thread cassieri: faccio notare che essi vengono creati "detached" (come anche i thread clienti) grazie alla variabile attributo passata come parametro, di modo che alla loro terminazione liberino la memoria occupata senza la necessità che un altro thread ne esegui la "join". Il numero di cassieri aperti all'inizio del programma è specificato nel file di configurazione.
In seguito viene prima inizializzato l'array di clienti e poi creati i thread che ne rappresentano il comportamento.
Il processo poi entra in un ciclo (che termina solo se arriva uno dei due segnali per cui sono stati installati gli handler) in cui, se il numero di clienti diventa C-E, vengono creati e quindi fatti entrare nel supermercato altri E clienti. Il meccanismo è reso possibile grazie ad una variabile di condizione che avvisa il thread principale quando il numero di clienti usciti dal supermercato è uguale al

parametro E.
Una volta ricevuto uno dei segnali di terminazione, il main thread aspetta che tutte le casse chiudano e in seguito termina dopo aver distrutto le mutex e le variabili di condizione, e dopo essersi sincronizzato tramite “close” e “ok” sul socket, come già spiegato in precedenza.

Ma la peculiarità del progetto è rappresentata dal funzionamento dei thread cassiere e cliente e del loro delicato metodo di sincronizzazione.

Il thread cassiere

Il thread cassiere inizia la sua esecuzione connettendosi immediatamente al socket creato dal direttore che gli servirà più tardi per comunicare il numero di clienti presenti in coda.
Dopo aver modificato il suo stato in “aperto” grazie al booleano ed aver incrementato il numero totale di cassieri aperti entra nel ciclo while che gestisce ad ogni giro un’interazione col cliente.
Se la coda è ancora vuota, egli si mette in attesa di un eventuale cliente. Se un cliente risveglia il cassiere, egli calcola il tempo di servizio grazie ai dati del cliente e grazie ad una nanosleep aspetta suddetto tempo. Successivamente si mette nuovamente in attesa che il cliente paghi, di modo che possa terminare l’interazione.
Dopo aver servito il cliente, il cassiere controlla se il tempo in cui ha avuto l’interazione con il cliente è maggiore del tempo in cui esso deve aggiornare il direttore in merito al numero di clienti presenti in cassa. Se esso è maggiore o uguale, procede nel comunicarglielo grazie alla connessione sul socket, altrimenti procede al prossimo cliente. Questo meccanismo, seppur non precisissimo, mi è sembrato più efficiente che aspettare con delle nanosleep (anche eventualmente con un secondo processo timer).
I cassieri che devono chiudere escono dal ciclo e i clienti ormai già in coda vengono “accompagnati” verso la scelta di un’altra cassa facendoli uscire dal groviglio di variabili di condizione e mutex.
Stesso procedimento accade se viene chiamato il segnale SIGQUIT, ma i clienti in questo caso terminano.
Se una nuova cassa deve aprire viene creato un nuovo thread nel primo slot dell’array che descrive una cassa chiusa.
Se viene chiamato il segnale SIGHUP il cassiere termina di servire i propri clienti e poi termina scrivendo sul file di log.

Il thread cliente

Il thread cliente inizia la sua esecuzione facendo la lista della spesa e calcolando casualmente il numero di prodotti da acquistare. Se esso è 0, il cliente apre le comunicazioni con il direttore e chiede il permesso di poter uscire scrivendo 0 sul socket. Se la risposta è 1, scrive sul file di log ed esce.
Altrimenti, se il numero di prodotti è maggiore di 0, come nella stragrande maggioranza dei casi, sceglie, sempre random, una cassa aperta in cui mettersi in fila. Successivamente entra in un while che terminerà quando egli potrà uscire dal supermercato. Si entra poi in un secondo while (che termina se la posizione del cliente in coda è minore della soglia S2), in cui il cliente può decidere se andare in una cassa dalla coda più corta, se essa esiste, ogni S millisecondi e, in seguito, si mette in attesa di essere in prima posizione sulla coda. Ciò che condividono cassiere e cliente è proprio la prima posizione della coda, non l’intera coda. Egli verrà risvegliato dal cassiere che lo avviserà che è finalmente in prima posizione. Il cliente paga e si mette in attesa del segnale di poter uscire da parte del cassiere. Se la cassa sta chiudendo egli rientra nel while e sceglie un’altra cassa. Se viene chiamata la SIGQUIT egli termina ed esce dal supermercato senza pagare i prodotti acquistati.
Se la transazione ha invece successo, scrive sul file di log dei clienti e termina.
Se viene chiamata la SIGHUP ogni cliente termina normalmente.

Cnfg.c

Questo file (e il rispettivo header) contiene la funzione per effettuare il parsing del file di configurazione passato come parametro di esecuzione in entrambi i test.

Script di analisi

Lo script di analisi (./analysis.sh), che viene lanciato una volta che il processo direttore ha terminato la sua esecuzione, controlla se il file contenente le informazioni dei clienti (./statsClients.txt) è stato creato correttamente ed entra in un ciclo while in cui legge riga per riga dal file e sostituisce una formattazione più intuitiva alla verbosità delle stringhe stampate su file. In seguito, queste stringhe più compatte vengono stampate su terminale (echo). Ho aggiunto una sleep di 0.1 secondi all'interno del ciclo: in tal modo, specialmente durante l'esecuzione del test2 (SIGHUP), tutti i clienti hanno modo di stampare le proprie informazioni sul file prima che termini lo script. Se il file non viene aperto correttamente viene stampato un errore.
La stessa procedura viene ripetuta per il file ./statsCashiers.txt (che invece contiene le informazioni sui cassieri). Come già specificato, ho preferito mantenere i due file separati per una maggiore chiarezza. Un fac-simile delle stampe è il seguente:

- per i clienti:
  ID cliente: XXX | n. prodotti: XXX | tot. tempo: XXX s | tempo acquisti: XXX s | tempo in coda: XXX s | tempo di servizio: XXX s | n. cassa: XXX | n. code cambiate: XXX

- per i cassieri:
  ID cassiere: XXX | n. prodotti elaborati: XXX | n. clienti serviti: XXX | tot. tempo: XXX s | tempo di servizio medio: XXX s | n. chiusure cassa: XXX

N.B. - I clienti che non hanno acquistato alcun prodotto posseggono le sole informazioni "ID cliente", "n. prodotti" e "tot. tempo", le altre ovviamente non possono essere calcolate e sono sostituite con il carattere "-". Nei clienti con 0 prodotti acquistati il campo "tot. tempo" potrebbe essere "0.000 s": faccio notare che non si tratta di un errore, ma semplicemente questo tipo di cliente esce dal supermercato in modo molto veloce e la precisione a tre cifre decimali non rende giustizia al dato proposto.
I cassieri stampano i propri dati ad ogni chiusura, è quindi possibile trovare più stampe di uno stesso cassiere, che sono facilmente differenziabili le une dalle altre confrontando il numero di chiusure (sempre incrementato) e il tempo totale di esecuzione (sempre maggiore).

Test 1

Ho preferito scrivere degli script bash per i comandi di esecuzione dei test. Il primo (./test1_script.sh) lancia il programma ./director passandogli come parametro il file ./config1.txt. Il programma viene lanciato, come richiesto, da valgrind con l'opzione —leak-check=full. Contemporaneamente viene salvato il PID del processo direttore e viene lanciata la sleep di 15 secondi. Successivamente viene inviato il segnale di SIGQUIT (-3) tramite la kill. Un while attende la terminazione del processo direttore; poi viene lanciato lo script di analisi descritto in precedenza dopo averlo reso eseguibile tramite la "chmod -x". Lo script di test termina con la rimozione del file contenente il PID del direttore (./director.PID), del file di socket (./socketSM_D) e della cartella director.dSYM (simboli di debug).

Test 2

Il secondo script di test (./test2_script.sh) è sostanzialmente uguale al primo script di test con le seguenti uniche differenze:
- il programma non viene lanciato tramite valgrind;
- il file passato come parametro è ./config2.txt;
- la sleep non è più di 15 secondi ma di 25;
- il segnale lanciato dopo la sleep è il SIGHUP (-1).

Makefile

Il Makefile per la compilazione presenta, come da richiesta, un comando “all” (make) che compila i due file oggetto principali (./director e ./supermarket). A cascata, vengono compilati gli altri file di supporto (./client.o, ./cashier.o, ./queue.o, ./cnfg.o) che vengono linkati al file ./supermarket, mentre solo il file ./cnfg.o viene linkato al file ./director. Come già accennato in precedenza i comandi per i test (make test1 e make test2) lanciano i rispettivi script.
E’ presente il target .PHONY clean (make clean) che fa pulizia di tutti i file prodotti durante la compilazione e l’esecuzione.

File di configurazione

Primo file di configurazione (./config1.txt) riguardante il primo test:
- 2 K - numero casse;
- 20 C - numero clienti che entrano in un primo momento;
- 5 E - numero di clienti che entrano all'uscita di altrettanti;
- 500 T - tempo massimo speso per acquistare i prodotti da parte dei clienti;
- 10 L - tempo di servizio di un cassiere per singolo prodotto;
- 80 P - massimo numero di prodotti acquistabili;
- 30 S - intervallo di tempo in cui un cliente controlla la coda di una nuova cassa;
- 2 S1 - soglia di chiusura cassa da parte del direttore;
- 8 S2 - soglia di apertura nuova cassa;
- 1 openCD - n casse aperte inizialmente;
- 700 refreshD - intervallo di tempo in cui un cassiere aggiorna il direttore sul numero dei clienti in coda;
- statsClients.txt logname1 - nome file di log per i clienti;
- statsCashiers.txt logname2 - nome file di log per i cassieri.

Secondo file di configurazione (./config2.txt) riguardante il secondo test:
- 6 K - numero casse;
- 50 C - numero clienti che entrano in un primo momento;
- 3 E - numero di clienti che entrano all'uscita di altrettanti;
- 200 T - tempo massimo speso per acquistare i prodotti da parte dei clienti;
- 10 L - tempo di servizio di un cassiere per singolo prodotto;
- 100 P - massimo numero di prodotti acquistabili;
- 20 S - intervallo di tempo in cui un cliente controlla la coda di una nuova cassa;
- 3 S1 - soglia di chiusura cassa da parte del direttore;
- 12 S2 - soglia di apertura nuova cassa;
- 3 openCD - n casse aperte inizialmente;
- 300 refreshD - intervallo di tempo in cui un cassiere aggiorna il direttore sul numero dei clienti in coda;
- statsClients.txt logname1 - nome file di log per i clienti;
- statsCashiers.txt logname2 - nome file di log per i cassieri.

Istruzioni per la compilazione

Per compilare il progetto basta posizionarsi sulla cartella principale e digitare “make” da terminale. Per eseguire i test digitare “make test1” e “make test2”. Per la pulizia della cartella dai file prodotti digitare “make clean”. É possibile compilare solo il codice del direttore tramite “make director” o solo quello del supermarket tramite “make supermarket”.