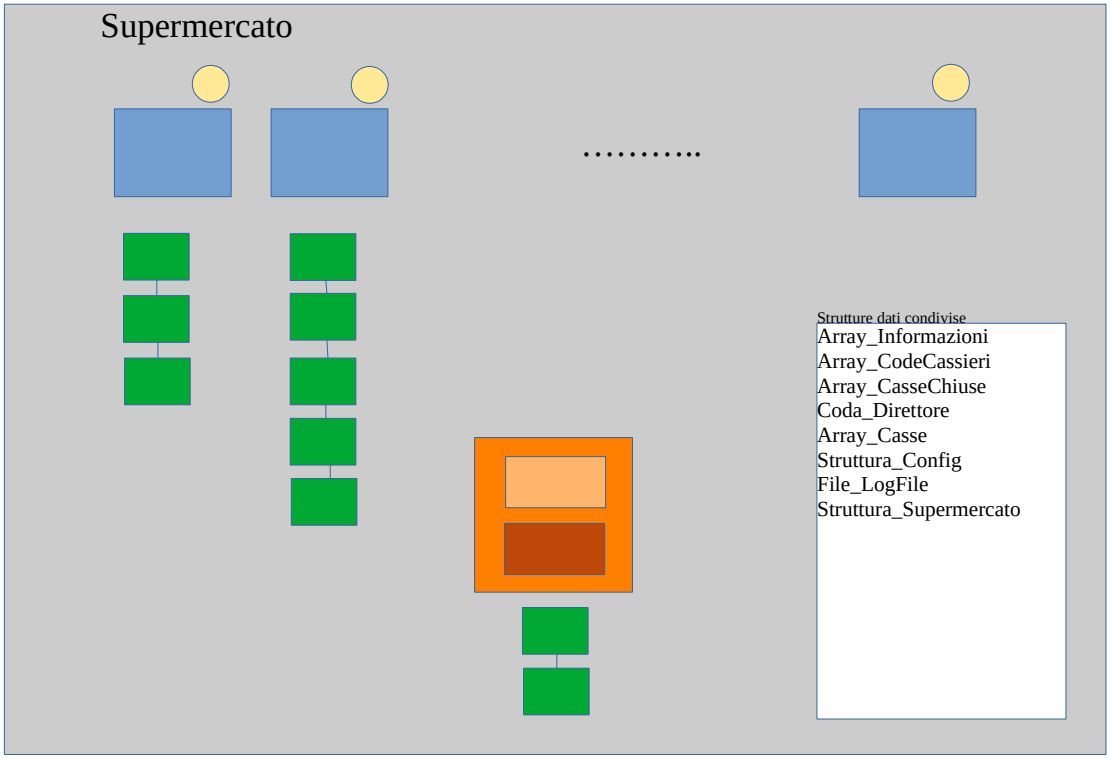
RELAZIONE PROGETTO DI LABORATORIO DI SISTEMI OPERATIVI

a.a. 2019/2020

Studente:
Luigi Gesuele
Matricola: 000000
Corso: A

Docenti:
G. Prencipe
M. A. Bonuccelli

Architettura



→ *Thread Timer*
→ *Thread Clienti*
→ *Thread Casse*
→ *Thread Direttore*
→ *Sotto-thread Direttore Casse*
→ *Sotto-thread Direttore Clienti*

Scelte progettuali

La versione semplificata del progetto richiedeva l'implementazione di un supermercato come unico processo multithread, per cui tutte le strutture dati utilizzate sono definite globalmente e quindi condivise tra tutti i thread, che vi accedono in mutua esclusione mediante mutex e condition variables. Questo non è stato fatto invece per le variabili relative ai due segnali da gestire, SIGHUP e SIGQUIT, poiché le due variabili globali associate a questi segnali sono state definite di tipo volatile sig_atomic_t, ed essendo gestite quindi con accessi atomici non erano necessarie le mutex o le condition variables.

Come richiesto dal testo, il supermercato deve essere aperto da un Direttore che farà entrare inizialmente tutti i clienti concorrentemente e aprirà un numero predefinito di casse minore o uguale al totale delle casse esistenti (K). Inoltre, il direttore deve poter controllare anche le entrate e uscite dei clienti stessi, e deve essere notificato costantemente sulla lunghezza delle code di ciascuna cassa per poter decidere quali aprire e chiudere. Sulla base di queste richieste, ho pensato di creare un thread **"Direttore"** che si occupasse di gestire sia le casse che i clienti mediante due sottothread.

Il sottothread **"Direttore Casse"** si occupa quindi di avviare i K thread Casse, dopodiché si mette in attesa di ricevere le informazioni sulle code di tutte le casse aperte da parte dei thread Timer. Quando il sottothread Direttore Casse viene svegliato, se non sono stati ricevuti segnali di SIGQUIT o SIGHUP, sulla base delle informazioni ricevute effettua i controlli delle soglie S1 e S2, fornite dal file di config, sceglie se aprire o chiudere una nuova cassa. Una volta terminati questi controlli, resetta a -1 i valori delle code di tutte le casse

aperte, per evitare che vengano riaperte/richiuse le casse da poco chiuse/aperte. Questi controlli vengono effettuati anche quando viene ricevuto il segnale SIGHUP, in quanto questo segnale sostanzialmente blocca solamente l'ingresso di nuovi clienti nel supermercato, ma quelli al suo interno vanno serviti. In caso di SIGQUIT invece, il sottothread attende che i thread Casse terminino per terminare a sua volta.

Il sottothread **"Direttore Clienti"** invece si occupa di avviare i C thread Clienti, dopodiché si mette in attesa di dover gestire i clienti uscenti dal supermercato, per poterne poi far rientrare E quando il numero di clienti all'interno del supermercato raggiungerà la soglia C-E. Per semplicità ho pensato di creare una coda gestita dal Direttore Clienti, che sarà riempita via via che un Cliente termina gli acquisti presso una qualunque Cassa oppure quando un Cliente decide di non acquistare nulla. Alla ricezione di un qualunque segnale il thread Direttore Clienti attende la terminazione di tutti i thread Clienti e termina a sua volta.

Il thread **"Cassa"** coincide invece con la figura del cassiere. Al suo avvio viene avviato anche il thread Timer corrispondente ad essa, dopodiché resta in attesa che venga aperta, e poi resta in attesa che la sua coda sia non vuota. Nel primo ciclo di attesa ho inoltre aggiunto un controllo sulla dimensione della coda, in quanto se quest'ultima risultasse piena vorrebbe dire che è stata appena chiusa e quindi i Clienti in coda devono cambiare cassa, e lo stato della cassa aggiornato viene scritto nel file di log. Vengono aggiornate anche le informazioni generali del supermercato. Se invece la coda di una cassa aperta è non vuota, il cassiere attende, prima di servire ogni cliente, un tempo fisso scelto casualmente con la **rand_r** tra **T_MIN_CASS** e **T_MAX_CASS.** Solo dopo che questo tempo è trascorso la Cassa inizia a servire il Cliente in testa alla coda relativa ad essa, così da permettere a eventuali clienti di accodarsi. Questo perché ho definito un array di K code globale, ciascuna relativa ad ogni cassa, per cui si ha bisogno di un accesso in mutua esclusione. Per ogni cliente in coda, il cassiere passa i prodotti di quest'ultimo attendendo il tempo fisso per passare un prodotto moltiplicato il numero di prodotti acquistati, e oltre ad aggiornare il tempo in coda del cliente che sta servendo, aggiorna anche i dati di clienti serviti e numero di prodotti elaborati. Il tempo in coda del cliente viene gestito tramite la funzione **clock_gettime(),** che salva nella struttura del Cliente il tempo di ingresso in coda e il tempo di uscita, in due strutture di tipo **timespec.** Alla ricezione del segnale SIGHUP, la Cassa controlla il suo stato. Se è chiusa, attende il termine del timer associato ad essa ed esce. Altrimenti, serve tutti i clienti nella sua coda con un ciclo, se ovviamente questa risulta non vuota. Se riceve un segnale di SIGQUIT invece, serve l'attuale cliente in testa alla coda e subito dopo che il timer ad essa associato termina, esce.

Il thread **"Cliente"** coincide con la figura del cliente. Una volta avviato, viene scelto casualmente sempre con la funzione **rand_r** sia un numero di prodotti da acquistare, sia un tempo che viene impiegato dal cliente stesso per effettuare gli acquisti. Se il numero di prodotti generato casualmente è 0, allora il cliente va direttamente nella coda del Direttore Clienti, altrimenti sceglie a caso una cassa aperta e ci si mette in coda, memorizzando nella struttura dati relativa al cliente il tempo di ingresso in coda. Infine il thread Cliente termina.

Il thread **"TimerCassa"** viene avviato insieme ad ogni thread Cassa, poiché per ogni cassa il direttore deve ricevere informazioni in maniera asincrona. Se la cassa è chiusa, il timer resta in attesa, altrimenti attende il tempo t_info del file di configurazione e poi scrive in un array definito globalmente le informazioni relative alla cassa associato ad esso, inviando un segnale al Direttore Casse in attesa su di esso. Questo viene fatto ciclicamente fino a che non viene ricevuto un segnale di SIGQUIT o SIGHUP, dopo il quale termina.

Il controllo da parte del thread Direttore Casse sullo stato delle code di ciascuna cassa, per semplicità lo ho scritto sotto forma di due funzioni a se stanti, chiamate S1() e S2(), per una migliore leggibilità del codice. Inoltre ho aggiunto nel controllo di S1 che vi siano almeno 2 casse aperte, altrimenti verrebbero chiuse tutte le casse, e per il controllo di S2 ho aggiunto che vi siano al più K-1 casse aperte. Il Direttore Casse resta quindi in attesa di ricevere tutte le informazioni solamente dalle casse aperte, per cui nelle posizioni relative alle casse chiuse ho inserito il valore -2, mentre nelle casse aperte da cui non sono ancora state ricevute informazioni ho inserito il valore -1. Ho inoltre ipotizzato che, alla ricezione del segnale SIGQUIT andavano finiti di servire solo i Clienti in testa e tutti gli altri andavano fatti uscire, per cui i controlli delle soglie erano superflui. Alla ricezione di un segnale, le casse vengono chiuse tutte, così si comporteranno in modo tale da svuotare tutte le code da tutti i clienti e poi terminano, stampando sul logfile.

Nell'handler dei segnali setto la variabile relativa ad esso a 1, ed eseguo la funzione wakeUp() che permette di svegliare eventuali thread "dormienti"(con signal e broadcast) per poter uscire dai rispettivi cicli e agire a seconda del segnale ricevuto.
Il file di log generato durante tutta l'esecuzione rispecchia la forma richiesta dalla specifica, e viene scritto in modalità "append". All'avvio del processo, se ancora esistente nella directory, esso viene cancellato per evitare dati corrotti.

Strutture dati

I dati relativi ai cassieri e ai clienti sono memorizzati in due **struct**, **client** e **cassiere.** Allo stesso modo, anche i dati per le statistiche generali del supermercato sono salvati nella **struct supermercato.** Infine, i dati del file di configurazione sono letti nel main e poi inseriti in una **struct config** globale chiamata cfg. Le varie code invece sono state implementate analogamente alle code viste nel corso di Algoritmica e Laboratorio, ovviamente riadattate a contenere elementi di tipo client. Ho inoltre creato due header file, **client.h** e **cassiere.h,** che contengono al loro interno le struct relative a cliente e cassiere e funzioni ausiliarie per la generazione casuale di prodotti, tempo per acquisto dei prodotti ecc…
Ho inoltre creato un ulteriore header file, **error.h,** contenente le macro con le definizioni di funzioni per la gestione degli errori, come visto a lezione.
Ho inoltre scritto due funzioni, nel file supermercato.c, chiamate **initAll()** e **cleanAll(),** che servono rispettivamente per inizializzare le varie strutture dati, mutex e variabili di condizione, e per liberare lo heap dalle allocazioni di memoria, per prevenire eventuali situazioni di memory leak latenti. Sono presenti nel codice anche due funzioni ausiliarie per contare il numero di casse aperte e restituire un array di casse aperte, poiché potessi poi riutilizzarle. Assieme alle funzioni ausiliarie è possibile, come scritto in precedenza, trovare le funzioni per controllare le soglie S1 e S2, la funzione per controllare lo stato dell'array contenente le informazioni di ogni cassa, una funzione per inizializzare le informazioni di questo array in caso di ricezione del segnale SIGHUP e una funzione per resettare a -1 i valori sempre del solito array. La funzione diffTime invece serve per poter calcolare la differenza tra il tempo di ingresso in coda e il tempo di uscita di un cliente, facendo le opportune conversioni per poter stampare infine un valore float. Sono inoltre presenti le funzioni **print()** e **printI(),** utili in caso il programma venga eseguito in modalità DEBUG per poter visualizzare tutti i passaggi intermedi durante l'esecuzione.

Come usare il programma

Per utilizzare il programma è possibile crearne l'eseguibile eseguendo il comando "**make**" sul terminale. Dopodiché ci sono due possibili esecuzioni: "**make test**" e "**make clean**". Il primo serve per effettuare il test richiesto dalla specifica, che andrà quindi a prendere in input il file di configurazione "config.txt" che si trova nella cartella "config". Alla fine dell'esecuzione, è possibile eseguire "**make clean**" per pulire eventuali file superflui generati durante l'esecuzione del programma. Eseguendo make test, viene eseguito il programma principale con i parametri:
K=6 //numero casse totali
C=50 //numero clienti totali
E=3 //clienti che devono uscire per farne rientrare altri E
T=200 //tempo massimo che ciascun cliente può impiegare nell'acquisto
P=100 //numero massimo di prodotti acquistabile per ciascun utente
S=20 // tempo che un cassiere impiega per passare un prodotto
S1=2 //parametro da rispettare per la chiusura delle casse
S2=10 //parametro da rispettare per l'apertura delle casse
t_info=22 //intervallo di tempo per cui ogni cassa informa il direttore sullo stato della propria coda
c_iniz=1 //numero di casse aperte inizialmente
log=logfile.log //nome del file di log sul quale viene scritto il sunto della situazione
Al termine dei 25 secondi, viene mandato un segnale di SIGHUP al programma, quindi gli eventuali clienti all'interno del supermercato vengono processati ed infine esso termina. Al termine, il makefile esegue lo

script analisi.sh, dopo averne cambiato i permessi, che effettua il parsing del file di log sul terminale, sullo standard output. Sarà dunque possibile visualizzare i risultati nel formato richiesto dalla specifica.

Configurazione personalizzata

È inoltre possibile utilizzare un file di configurazione personalizzato, affinché questo sia di un formato consono. Per farlo, occorre scriverlo nello stesso format della struttura sopra descritta, ed infine eseguire il comando "**make**" per compilare il programma e generare l'eseguibile, ed infine eseguire da terminale:

make personal conf=pathToFile.format

dove pathToFile è il nome o il percorso del file di configurazione scelto e format è l'estensione di quest'ultimo.
Se si vuole passare un diverso tempo (in secondi) oltre che un file di configurazione diverso da quello di default, si può eseguire invece, dopo aver generato l'eseguibile con make:

make personalTime conf=pathToFile.format time=Xs

dove pathToFile è il nome o il percorso del file di configurazione scelto e format è l'estensione di quest'ultimo, mentre X è il numero di secondi di apertura del supermercato (la s va inclusa per indicare che l'unità di misura è il secondo). Nella cartella "config" è possibile trovare infatti due ulteriori configurazioni, se si vogliono provare, chiamate "config2.txt" e "config3.txt". Un esempio di esecuzione con il terzo file in 7 secondi:

make personalTime conf=config/config3.txt time=7s

Modalità DEBUG

Seguendo le stesse modalità precedentemente descritte, è possibile inoltre eseguire il programma in modalità DEBUG per poter visualizzare alcune stampe di procedimenti intermedi. Per farlo, occorre compilare con il comando:

make debug

Ed infine eseguendo come al solito:

make test