RELAZIONE PROGETTO- Versione Semplificata

Francesco Grillea
582653- Corso B

Nell'implementazione prodotta, il supermercato viene visto come un processo multithreaded i cui thread sono i clienti, le casse e il direttore.

Il supermercato è rappresentato dalle seguenti strutture dati:

la struttura ***cassiere*** che rappresenta una singola cassa: oltre i parametri per la gestione delle statistiche, troviamo una variabile di mutua esclusione (*mtx*) necessaria per accedere alle informazioni in sicurezza, ad esempio per garantire che alla stessa cassa si accodi un cliente alla volta; due variabili di condizione(*empty, open*) che fanno si che la cassa attenda rispettivamente nel caso in cui non ci siano clienti in coda, e quando questa viene chiusa; infine una coda di clienti che attendono il proprio turno per pagare;

questi ultimi vengono rappresentati dalla struttura dati ***cliente*** che, anch'essi, oltre ai parametri per la gestione delle statistiche comprendono una variabile di condizione (*turno*) per far si che un cliente in fila ad una cassa aspetti il proprio turno. Notare che un cliente non necessita di una variabile di mutua esclusione poiché l'accesso ai campi di un singolo cliente dev'essere necessariamente sequenziale;

una lista (*pending_client*) di elementi di tipo *cliente* che rappresenta i clienti che non hanno fatto acquisti e attendono il permesso del direttore per poter uscire dal supermercato.

Il programma inizia leggendo i parametri iniziali dal file di configurazione, inizializzando *n_clienti* e *counter* rispettivamente a *C* e *C-1*: questo perché il primo identifica il numero di clienti attualmente all'interno del supermercato (dato che i primi C clienti entrano contemporaneamente) e il secondo indicizza i clienti addizionali tenendo il conto del numero di clienti totali entrati nel supermercato. Successivamente vengono dichiarate due sigaction (*s_quit, s_hup*) per gestire rispettivamente i segnali SIGQUIT e SIGHUP con un proprio handler che setta rispettivamente i flag *quitval* e *hupval*. Quindi quando il supermercato troverà uno di questi due valori a 1 grazie alle funzioni *getQuit()* e *getHup()* in cui viene letto il rispettivo flag in modo sicuro grazie alla corrispondente variabile di mutua esclusione (*quitmtx, hupmtx*), si potra' procedere con la chiusura del supermercato.

A questo punto vengono allocate e inizializzate le *K* casse tramite la funzione *crea_casse* (all'interno di *datastruct.c*) assicurandomi di aprirne un numero randomico ma maggiore di zero: per fare questo la prima cassa inizialmente sarà aperta; vengono creati i vari thread a cui vengono associate le seguenti funzioni: *fun_direttore* per il thread direttore, *fun_cassiere* per i thread cassieri e *fun_cliente* per i thread clienti. Notare che l'ordine in cui vengono creati non è casuale poiché, in un supermercato ideale, il direttore deve essere "creato" prima dei cassieri e questi prima dei clienti. Stessa cosa per quanto riguarda la terminazione: in un supermercato ideale, quando questo dovrà chiudere, si aspetterà prima l'uscita di tutti i clienti presenti, successivamente le casse potranno chiudere ed infine il direttore potrà terminare e con lui il processo stesso.

Nella funzione del **thread cliente**, la prima operazione da fare è quella di creare e inizializzare un cliente, successivamente viene chiamata la funzione *nanosleep* per simulare il tempo impiegato dal cliente per acquistare i prodotti. Se il cliente ha comprato zero prodotti, deve attendere l'autorizzazione del direttore per poter uscire: per fare questo viene inserito (in maniera sicura grazie alla variabile di mutua esclusione *pending_access*) in una coda gestita proprio dal direttore, segnalandogli che in quella coda vi è sicuramente un cliente. Se il cliente ha comprato almeno un prodotto, questo si dirige verso una cassa aperta scelta in modo randomico tramite la funzione *randomizza_cassa()*. Siccome per accodare il cliente alla cassa facciamo accesso ad una struttura condivisa (quale la cassa scelta), per modificarla, è necessario accedervi in mutua esclusione tramite l'apposita variabile relativa alla cassa scelta: *casse[r].mtx*). Conseguentemente si incrementa il numero dei clienti in coda a quella cassa e viene segnalato che quella cassa ha sicuramente un cliente in attesa di essere servito. Quando un cliente si mette in una qualsiasi coda d'attesa, deve attendere il suo turno per poter essere servito: per far questo si mette in attesa sulla sua variabile di condizione *turno* che verrà poi svegliata da un cassiere o dal direttore il quale setterà in oltre il flag *servito* per poter negare la condizione di attesa (la guardia del while) e consentire al thread di proseguire nella sua esecuzione.

Tutto questo ovviamente è possibile soltanto se non è stato ricevuto un segnale SIGQUIT che causa l'uscita forzata dei clienti, mentre anche ricevendo SIGHUP il cliente, una volta entrato (quindi inizializzato), dev'essere gestito dal direttore o dal cassiere prima di uscire. Una volta che il cliente è stato servito o gli è stato permesso uscire, scrive all'interno del file di log le sue statistiche, può finalmente liberare la propria memoria e infine segnalare al main che un cliente è uscito in modo tale da poter controllare, nel caso in cui abbia ricevuto un segnale di terminazione, che non ci siano più clienti all'interno e conseguentemente far terminare il processo

Nella funzione del **thread cassiere**, la prima cosa è creare un sotto-thread (*th_comunicazione*) che implementa la funzione *timer* che permette, contemporaneamente alla gestione dei clienti, di inviare informazioni al direttore ad intervalli regolari. Un cassiere per poter inviare informazioni al direttore ha bisogno di salvare queste informazioni in una struttura dati che sia accessibile anche dal direttore: viene salvato, il numero di clienti in coda all'interno di un array lungo *K* (*send*) dove ogni posizione dell'array rappresenta il numero di clienti in coda alla cassa dello stesso indice. Essendo questo array appunto un elemento condiviso è necessario accedervi in mutua esclusione; al contrario non è necessario accedere in mutua esclusione al flag *open* della cassa poiché, nonostante sia un elemento condiviso che necessita di particolare attenzione, è modificato (all'interno di *comunicazione_casse*: funzione di comunicazione tra cassiere e direttore) mutuamente all'operazione di lettura: quindi non può capitare che il valore venga letto mentre un altro thread lo sta scrivendo. A questo punto, considerando che le casse sono già state inizializzate nella funzione *crea_casse* all'interno di *datastruct.c*, può iniziare l'esecuzione del cassiere vero e proprio. La gestione dei clienti viene messa all'interno di un *while(ture)* opportunamente interrotto dal costrutto *break*: questo perché ogni cassiere dev'essere sempre pronto a servire nuovi clienti anche nel caso in cui la cassa sia momentaneamente chiusa; ciò significa che una cassa può chiudere definitivamente (terminare) solo quando il supermercato deve chiudere. Prima di gestire qualsiasi cliente, il cassiere deve acquisire la rispettiva lock poiché essendo una struttura condivisa, è necessario leggere e scrivere informazioni in modo sicuro. Una volta che il cassiere ha acquisito la lock può controllare lo stato della cassa: se il supermercato è aperto, la cassa chiusa e non ci sono clienti che devono essere serviti, posso chiudere momentaneamente la cassa mettendola in wait sulla variabile di condizione *open*, altrimenti se c'è un cliente in coda (può capitare perché se una cassa viene chiusa, viene servito solo il primo cliente in coda mentre tutti gli altri andranno a disporsi in altre casse), viene servito normalmente; se il supermercato è aperto e non ci sono clienti in coda alla cassa, questa si mette in wait sulla variabile di condizionamento *empty* finché non c'è un cliente da servire. Se invece c'è almeno un cliente da servire, può essere gestito in due modi a seconda del fatto se si ha ricevuto SIGQUIT o meno: in caso negativo il cliente viene servito tramite la funzione *pay_client* (la cui implementazione è contenuta in *datastruct.c*) che, oltre ad aggiornare le statistiche del cliente e del cassiere, esegue una *nanosleep* che simula il tempo impiegato dal cassiere per servire il cliente (dato da tempo fisso + tempo elemento * # prodotti), successivamente viene settato il flag *servito* e viene segnalato al cliente che è stato servito in modo da poter uscire dal supermercato (far terminare il thread cliente); se si ha ricevuto SIGQUIT, viene eseguita la funzione *eject_client* (la cui implementazione è contenuta anch'essa in *datastruct.c*) in cui viene semplicemente settato il flag *servito* e segnalato al cliente che può uscire senza dover eseguire *nanosleep* poiché non viene servito ma "buttato fuori".
Nel caso in cui il supermercato debba chiudere (per SIGHUP) e ci sono clienti in coda alla cassa, questi vanno tutti serviti normalmente quindi tramite la funzione *pay_client* all'interno di un ciclo. Una cassa termina definitivamente quando non ci sono clienti in coda e nel supermercato in generale e il supermercato deve chiudere oppure se ho ricevuto SIGQUIT e non ci sono clienti in coda: nel primo caso il numero dei clienti attualmente all'interno del supermercato viene controllato attraverso la funzione *getClient_n()* che accede in mutua esclusione alla variabile *n_clienti* e ne ritorna il valore. Quindi vengono aggiornate le statistiche del cassiere e si esce dal ciclo. Una volta che si è usciti dal ciclo bisogna attendere la terminazione del sotto-thread precedentemente creato che può terminare soltanto quando viene ricevuto un segnale di uscita. A quel punto il thread cassiere è terminato definitivamente.

Il **thread direttore** ha il compito di gestire più aspetti contemporaneamente: la comunicazione con i clienti che chiedono il permesso di uscire, la comunicazione con le casse per aprirne/chiuderne una, l'entrata di nuovi clienti e la gestione della terminazione di clienti e casse. La prima avviene tramite la funzione *comunicazione_cliente,* il thread *th_fromClient* controlla la presenza di clienti in coda dal direttore e, nel caso in cui ce ne fossero, ha il compito di dargli il permesso di uscire settando il flag *servito* e aggiornando le statistiche, altrimenti aspetta che ce ne siano; la comunicazione con i cassieri i

quali, periodicamente, inviano informazioni al direttore viene gestita tramite la funzione *comunicazione_casse* associata al thread th_fromCasse. Questa decide se aprire o chiudere una cassa in base al numero di clienti presenti in fila (indicati nell'array *send*), informazione contenuta in un elemento condiviso tra cassiere e direttore accessibile in modo sicuro tramite la variabile di mutua esclusione *sendmtx*. Quest'ultima viene acquisita due volte nello stesso thread *timer*: prima e dopo la nanosleep, questo approccio è stato pensato per aumentare la probabilità di accesso da parte degli altri thread al relativo array. Se in una cassa ci sono più di S2 elementi in coda, ne apro una casuale tra quelle chiuse (scelta tramite la funzione *randomizza_cassa_chiusa*), che, facendo accesso ancora una volta ad un elemento condiviso quale la cassa, bisogna accedervi in sicurezza per poter modificare il flag di apertura e aggiornare le relative statistiche. Se invece c'è al più un cliente in almeno S1 casse, posso chiuderne una in modo randomico spostando tutti i clienti (tranne il primo) in un'altra cassa tramite la funzione *move_client:* il primo verrà servito normalmente dal thread cassiere. Una volta letta e gestita l'informazione di una singola cassa si può resettare il valore del relativo indice nell'array *send* (a *-1*) in modo tale da indicare che il direttore sia consapevole di star leggendo un'informazione invalida poiché già gestita o non ancora inviata/ricevuta. Quanto detto fin ora ovviamente continuerà fin quando non si riceve un segnale di terminazione.
Il direttore nella sua funzione vera e propria dovrà gestire l'entrata di *E* clienti nel caso in cui il numero di clienti all'interno del supermercato scenda a *C-E*. Dev'essere possibile quindi, far entrare nuovi clienti ogni qual volta ci siano le condizioni necessarie per farlo: proprio per questo motivo, è necessario poter creare nuovi thread senza aspettare la terminazione dei thread creati precedentemente. Questa politica non è implementabile tramite una join (poiché bloccante), ma si utilizza l'approccio di marcare il relativo thread come detached, in modo tale che un thread cliente possa terminare/essere creato indipendentemente dagli altri. Un'altra operazione fondamentale del thread direttore è quella di risvegliare le casse e i clienti (in attesa dal direttore) che, alla chiusura del supermercato, sono in attesa di un evento. In questo modo una volta risvegliati possono terminare normalmente, altrimenti potremmo avere un'attesa infinita nel caso in cui siano fermi su una wait.

Infine il main attende che tutti i clienti terminino: i primi C clienti tramite una join e quelli addizionali tramite un'attesa sulla variabile di condizione *exitclient_wait* che viene "controllata" ogni qual volta un cliente addizionale termina. Successivamente, una volta che tutti i clienti sono fuori dal supermercato, si attende la terminazione di tutti i cassieri e del direttore tramite join. A questo punto si può liberare la memoria precedentemente allocata, chiudere i file aperti e far terminare l'esecuzione del programma.

Notare che tra l'invio di SIGHUP al processo e la terminazione dello stesso passa il tempo necessario per servire i clienti rimanenti all'interno del supermercato. Ciò vuol dire che tra l'invio del segnale e il sunto delle statistiche è necessario aspettare che il processo finisca per assicurarsi di non leggere informazioni parziali dal file di log.