Relazione Progetto

Sistemi Operativi Laboratorio

Versione progetto: completa

Simulazione Multi-threaded, Multi-processo di un supermercato

Ludovico Venturi

Corso B
Matricola *578033*

Indice

Docente di riferimento: **Massimo Torquati**

Università di Pisa
Corso di Laurea in Informatica L-31

15 Luglio 2020

1 Introduzione

I dettagli implementativi delle scelte sono riportati sui commenti nel codice, in questa relazione ho cercato di spiegarle concettualmente (sono comunque presenti nel codice commenti riassuntivi che provano a spiegare la logica).
Per quanto mi fosse possibile ho cercato di incapsulare i dati al fine di garantire il privilegio minimo sulle variabili, nonostante ciò per utilizzare delle funzioni di cleanup (e per altri scopi) ho preferito lasciare alcune variabili globali.
Non sono state utilizzate librerie esterne, ho però inserito le funzioni *readn, writen* presenti sulla pagina del corso (la fonte è riportata nel codice) e la funzione *timeval_ subtract* [1].

Il processo principale del supermercato è il Direttore, sarà lui a far partire il processo supermercato; se si dovesse eseguire solamente il supermercato si otterrebbe una chiusura dovuta alla mancata connessione al Socket di comunicazione.

2 Direttore

Il processo Direttore, a regime, ha 2 thread:

- thread **main**
- thread **signal handler (TSH)**

Il direttore forka il processo Supermercato passandogli come parametro il file di configurazione.

2.1 Comunicazione con il Supermercato

La comunicazione con il processo direttore avviene tramite l'utilizzo di un *Socket AF_ UNIX*.
Dato che il thread main deve anche attendere eventuali comunicazioni del TSH dalla pipe, utilizzo la *poll* per il multiplexing, ascoltando così sia dal fd del socket che da quello della pipe (inizialmente attende anche su un altro fd: il socket di accettazione delle comunicazioni, che alla prima *accept* viene chiuso dato che ci troviamo in una situazione 1 client - 1 server).
La comunicazione con il Supermercato avviene rispettando un protocollo di comunicazione, si inviano in ordine:

- tipo del messaggio da inviare (di tipo *sock_ msg_ code_ t*)
- 0 o più parametri interi a seconda del tipo di messaggio

I vari tipi di comunicazioni sono spiegati da commenti nel codice. All'arrivo di un segnale SIGHUP rimane in attesa di altre comunicazioni dai cassieri/clienti: rimane possibile per il direttore decidere di aprire/chiudere casse amche nel momento che intercorre fra l'arrivo di SIGHUP e il servizio di tutti i clienti nel supermercato.
Assumo che il Direttore decida sempre di far uscire i clienti che richiedono il permesso di uscita.

Il Direttore non accede direttamente alle informazioni delle casse, su di loro conosce soltanto ciò che gli viene detto dai cassieri (in particolare dai *notificatori*).

Alla chiusura del supermercato il direttore attende la terminazione del processo Supermercato (suo figlio); sucessivamente vengono chiusi i vari *fd* usati (pipe e socket di ascolto/comunicazione) e viene cancellato il socket.

2.2 Segnali

Tutti i segnali vengono mascherati nel main così da lasciare la loro gestione al TSH, ad eccezione del segnale *SIGPIPE* che viene ignorato installando un gestore SIG_IGN.
Il TSH gestisce i segnali SIGQUIT e SIGHUP avvisando il thread main del loro arrivo attraverso una

[1] https://www.gnu.org/software/libc/manual/html_node/Calculating-Elapsed-Time.html

pipe unnamed adibita a tale scopo. Sarà quindi il main effettivamente a reindirizzare i segnali al processo supermercato. Il passaggio dal TSH al main potrebbe risultare superlfuo, ho preso questa strada poichè reputo che il TSH si debba occupare esclusivamente della gestione interna dei segnali.
Viene utilizzato sia nel Direttore che nel Supermercato il segnale *SIGUSR1* per far terminare il rispettivo TSH in previsione dell'imminente terminazione del processo.

3 Supermercato

Il processo supermercato, a regime, è strutturato a livello di thread in:

- **thread signal handler (TSH)**
- **manager**
- **C clienti**
- **K cassieri**, ognuno di loro genera un relativo *notificatore*
- **K notificatori**

La gestione dei segnali avviene analogamente a quella del direttore. La differenza sta nella gestione degli stessi: se riceve SIGHUP il supermercato va nello stato *CHIUSURA_SOFT*, in cui i notificatori continuano il proprio lavoro e quindi le casse potrebbero venire aperte e/o chiuse. Se viene ricevuto SIGQUIT il supermercato va nello stato *CHIUSURA_IMMEDIATA* e vengono gestiti solamente i clienti che i cassieri stanno già servendo.
All'uscita chiude il fd di comunicazione con il Direttore e i fd della pipe.
Il *manager* è colui che gestisce i vari thread creati, fa da intermediario fra i clienti ed il direttore, gestisce le notifiche di entrata/uscita dei clienti, le eventuali aperture/chiusure delle casse, consente l'accesso ad E clienti ogni qualvolta E clienti escono dal supermercato e si occupa delle strutture dei LOG e delle strutture passate ai thread (sarà lui a fare la principali free() del programma).

3.1 Comunicazione interna al Supermercato

I clienti comunicano la loro entrata, la loro uscita e l'eventuale richiesta del permesso di uscita nel caso in cui non abbiano effettuato acquisti.
Tale comunicazione è gestita con una *unnamed pipe* aperta da entrambe le estremità in ogni thread. Su tale pipe scrive anche il TSH per avvisare della ricezione di segnali. La comunicazione avviene nuovamente seguendo un protocollo di comunicazione, si inviano in ordine:

- tipo del messaggio da inviare (di tipo *pipe_msg_code_t*)
- 0 o più parametri interi a seconda del tipo di messaggio

Anche in questo caso i vari tipi di comunicazioni sono descritti da commenti nel codice. Ho aggiunto l'invio della notifica di entrata/uscita dei clienti come informazione aggiuntiva.

3.1.1 Concorrenza

- **pipe, fd[0]** l'unico lettore è il manager, non vi è concorrenza su tale fd
- **pipe, fd[1]** possono scrivere sulla pipe tutti i C clienti ed il Thread Signal Handler

Per ovviare a tale problema si può scrivere sulla pipe solamente acquisendo la relativa lock, ed è necessario che si invii il messaggio completo di tutti i parametri in una volta sola, altrimenti la comunicazione potrebbe non risultare consistente e perderebbe significato.
Lo stesso problema si ha sul *Socket* di comunicazione col Direttore:

- **socket direttore** scrittori: manager e K notificatori ; lettore: manager

Il manager lo usa per comunicare la richiesta del permesso di uscita dei clienti e per inviare il messaggio di imminente chiusura (utile soprattutto nel caso di SIGHUP), mentre i notificatori lo usano per inviare le informazioni sulle casse al direttore.
Anche per le scritture sul Socket ho adottato la medesima tecnica di mutua esclusione e invio completo del messaggio; per le letture non c'é bisogno dato che l'unico lettore è il manager.
Il manager attende tramite *poll* quindi sulla pipe e sul socket di comunicazione con il direttore.

3.2 Chiusura

La terminazione è un momento delicato della vita del processo Supermercato, l'ordine delle operazioni risulta fondamentale: sveglia gli eventuali thread cassieri e clienti in attesa di lavoro per farli terminare, successivamente aspetta la terminazione dei cassieri poi dei clienti. Scrive il LOG su file ed infine libera la memoria allocata dalla struttura della poll, dagli argomenti dei thread e dalle strutture del LOG.

3.3 Thread Pool

I clienti ed i cassieri vengono implementati come thread sempre 'vivi' all'interno del programma: vengono creati inizialmente K cassieri e C clienti e non ne verrano spawnati dinamicamente altri. Quando un cliente esce dal supermercato non viene terminato, esso va in attesa su una variabile di condizione. Per i cassieri il discorso è analogo, se la loro cassa è chiusa, attendono.
Per garantire il funzionamento di questo metodo ho strutturato i thread relativi ai clienti e ai cassieri utilizzando due *thread pool*.
Li gestisco attraverso una struttura dati da me definita di tipo *pool_set_t* che contiene oltre ad una lock ed una condition variable, il contatore di *jobs* disponibili.
Prima di eseguire il proprio lavoro un thread (cassiere o cliente) deve controllare che sia disponibile un lavoro: se *jobs > 0* allora il thread prende il lavoro e decrementa *jobs* di 1.
Se invece *jobs == 0* il thread si mette in attesa sulla condition variable del *pool_set_t*.

3.3.1 Concorrenza

Chiaramente la variabile *jobs* contenuta in *pool_set_t* genera una race condition:

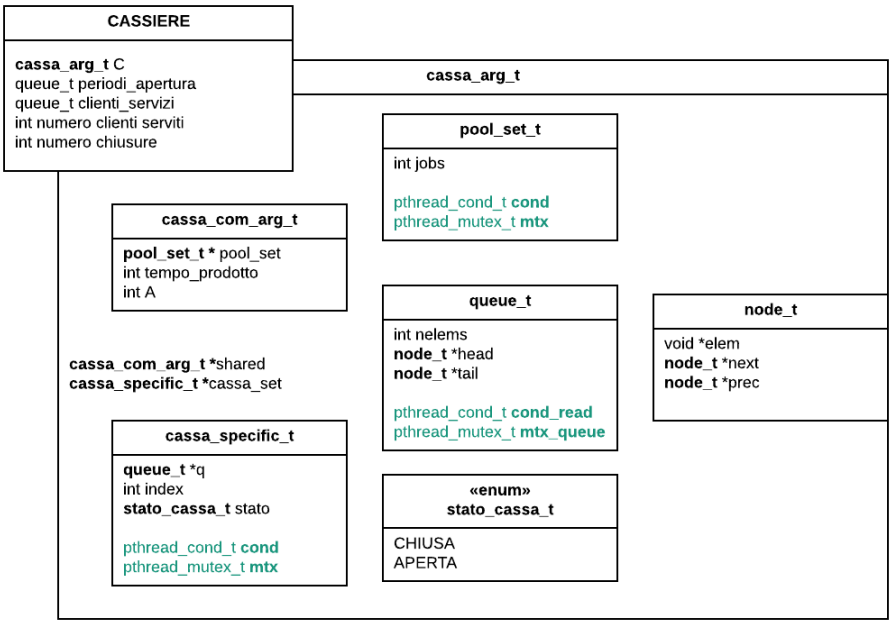
- **jobs** letta e scritta da tutti i thread del pool di appartenenza; inizializzata e scritta dal *manager*

Per accedervi è necessario acquisire la *lock* sulla mutex relativa al pool_set. Le variabili di pool nel programma sono 2: una per i cassieri ed una per i clienti. Esse sono scollegate fra loro infatti i cassieri controlleranno solo la loro variabile di pool e viceversa.
Il *manager* può modificare *jobs* dei 2 pool nel seguente modo:

- **clienti** ogni E clienti che escono dal supermercato vengono resi disponibili E lavori (*jobs += E*)
- **casse** quando riceve la comunicazione dal direttore di aprire una cassa, incrementa di 1 *jobs*

3.4 Cassiere

Inizialmente non tutte le casse sono aperte pertanto ogni cassa attende un lavoro. Appena aprono fanno partire un thread *notificatore* associato di supporto che svolgerà la comunicazione ad intervalli regolari col direttore.
Se la cassa viene chiusa dal direttore avvisa gli eventuali clienti in coda settando il loro stato di attesa in *CASSA_IN_CHIUSURA* e li sveglia. Se il supermercato sta terminando setta il loro stato in *SM_IN_CHIUSURA* svegliandoli. L'argomento passato ad ogni cassiere ha questa gerarchia (la figura ha attributi di una vecchia versione): *shared* è condiviso fra tutte le casse mentre *cassa_set* è specifico per ogni cassa la scelta di usare un'ulteriore struttura dati è per garantire maggiore incapsulamente e riservatezza, poichè i clienti dovranno accedere a delle informazioni sulle casse volevo che queste fossero il meno sensibili possibile.



3.4.1 Notificatore

Avvisa il direttore ad intervalli regolari di ampiezza A riguardo il numero di clienti in coda alla relativa cassa. Invia un tipo messaggio *CASSIERE_NUM_CLIENTI* seguito dai parametri: indice della coda e numero clienti in coda.
Se la cassa viene chiusa si mette in attesa sulla var. di condizione della cassa.
Aspettano STABILIZATION_TIME (impostabile con una define, di default 300ms) prima di iniziare ad inviare notifiche al direttore ogni qualvolta la cassa si riapre (o viene aperta per la prima volta) per evitare che la cassa venga subito richiusa a causa dei suoi iniziali 0 clienti.

3.4.2 Concorrenza

Nel cassiere le variabili accedute in lettura e scrittura da più thread sono:

- **q (coda cassa)**
- **stato cassa**

Per accedere sia in lettura che in scrittura a tali dati è necessario acquisire la lock della cassa. Ho reputato migliore utilizzare una sola lock comune e non 2 differenti dato il tipo di operazioni che si dovranno effettuare: pressochè tutte hanno bisogno di accedere sia allo stato della cassa che alla coda e hanno bisogno di conoscere le 2 informazioni garantendo che nessuna delle 2 cambi.

3.5 Cliente

La vita del cliente è quella descritta nelle specifiche del progetto: per un maggior dettaglio si invita a consultare i commenti nel codice. Durante l'attesa del servizio vengnono acquisite 2 lock, quella della cassa e quella dell'elemento, perchè un cassiere potrebbe effettuare in quel momento una POP cambiando lo stato sa IN ATTESA a SERVIZIO IN CORSO (quindi prende la lock sulla coda) o potrebbe cambiare lo stato di attesa da SERVIZIO IN CORSO a SERVITO (prendendo quindi la lock sull'elemento). Per poter essere certo di controllare lo stato ho quindi bisogno di entrambe lo lock.

3.5.1 Cambio cassa

Si prega di controllare i commenti nel codice della funzione per il funzionamento della selta Ho reputato opportuno impostare un limite di 10 cambi cassa per cliente. Può accadere (raramente dalle mie

osservazioni) che un cliente decida di cambiare cassa ogni S ms, causando più overhead che guadagno. Se la cassa viene chiusa cambierà comunque cassa anche se lo ha già fatto 10 o più volte.
Ho utilizzato i *pthread_spin_lock* sperando di guadagnare qualcosa in termini di efficienza in quanto le sezioni critiche sono tutte molto brevi e molto accedute.
L'algoritmo non è ottimo, se trova una cassa posso solo dire che è migliore della mia. Non ho implementato una soluzione ottima dato che i vincoli principali da rispettare sono l'efficienza e il non causare deadlock:
se un cliente in coda ogni S millisecondi (con S=20/30) si sveglia e volesse controllare tutte le casse dovrebbe acquisire la lock di tutte le casse! Ma ogni client vorrebbe farlo, quindi si creerebbero deadlock.
Quindi un cliente si sveglia, acquisisce la lock sulla sua cassa, controlla la sua posizione e controlla quanto vale lo stato della cassa*min_queue*.

3.5.2 Concorrenza

- elem: **stato_attesa**
- **permesso_uscita** scritto solamente dal Manager e letto dal rispettivo cliente; reinizializzato dal cliente

Per potervi accedere è necessario acquisire le rispettive lock.

4 Utilizzo

Il makefile presente nel progetto contiene il necessario per compilare e testare il programma, usare:

- **make**
- make **clean** / **cleanall** per pulire dai file di lavoro usare clean, per ricompilare l'intero progetto usare cleanall (include l'effetto di clean)
- make **test1 test2**

La funzione di cambio cassa dei clienti in coda utilizza in genere molta CPU, è possibile disabilitarla settando DFLAGS = -DNO_CAMBIO_CASSA nel makefile.
Riguardo il parametro S ho reputato opportuno imporre che debba essere > 10 (millisecondi) per non causare troppo overhead.
Per una descrizione dei parametri aggiuntivi (A, J, ...) e del file di configurazione accettato si prega di eseguire *make help*.