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

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

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].

2 Direttore

Il processo Direttore, a regime, ha 2 thread:

- thread main
- thread signal handler (TSH)

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 *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 seguito all'imminente terminazione del processo. La comunicazione con il processo direttore avviene tramite l'utilizzo di un *Socket AF_ UNIX*.
Dato che il thread main deve anche attendere comunicazione dal TSH dalla pipe, utilizzo la *poll* per il multiplexing, ascoltando così sia dal fd del socket che da quello della pipe.
La comunicazione con il Supermercato avviene rispettando un protocollo di comunicazione, si inviano in ordine:

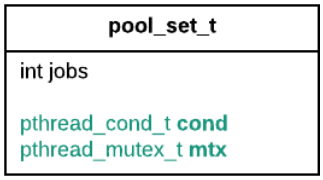
- tipo del messaggio da inviare (di tipo *sock_ msg_ code_ t*)
- 0 o più parametri interi a seconda del tipo di messaggio

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.

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*.

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



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 ($jobs \mathrel{+}= E$)
- **casse** quando riceve la comunicazione dal direttore di aprire una cassa, incrementa di 1 *jobs*

4 Manager

4.1 Comunicazione con i Clienti

Il manager gestisce le comunicazioni dai clienti. Essi 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.

4.1.1 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 anche il Thread Signal Handler

La comunicazione avviene seguendo un protocollo: viene inviato un messaggio di tipo *pipe_msg_code_t* (mostrato in figura 1)

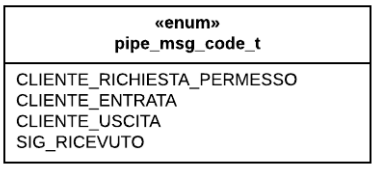


Figura 1: pipe_msg_code_t

5 Cliente

5.1 Cambio cassa

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*

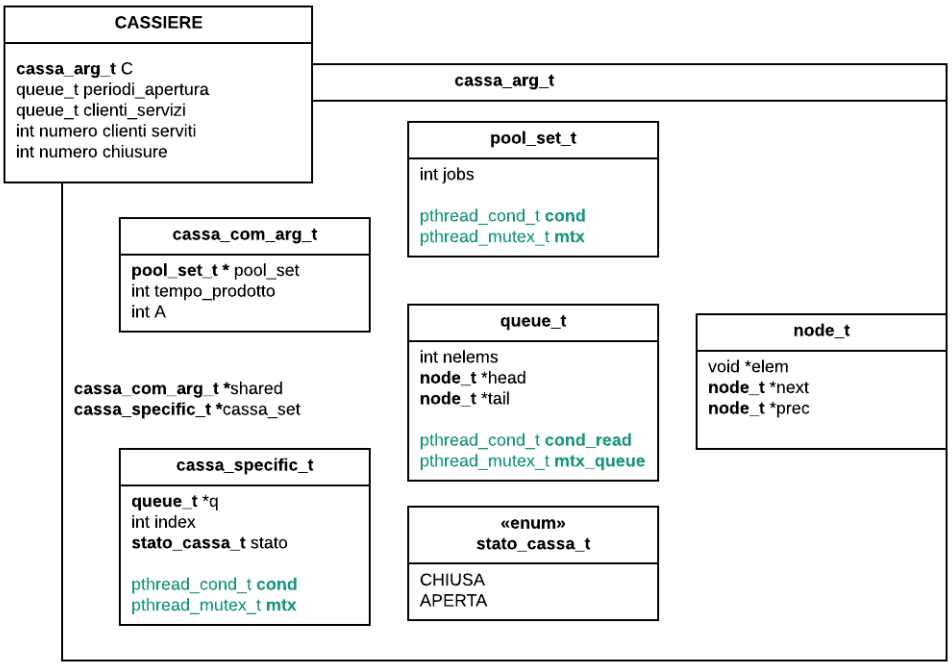
L'algoritmo non è ottimo, se trova una cassa posso solo dire che è migliore della mia. Non ho implementato una soluzione ottima dato che i vincoli principali da rispettare sono l'efficienza e il non

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*.

5.2 Concorrenza

- elem: **stato_attesa**
- **permesso_uscita** scritto solamente dal Manager e letto dal rispettivo cliente; reinizializzato dal cliente
  => va acquisita la risorsa attraverso *lock(cond_ cliente)*

6 Cassiere



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 incapsulamente e riservatezza dei dati delle che dovranno essere letti dai clienti

6.1 Concorrenza

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

- q
- stato

7 Utilizzo

Il makefile presente nel progetto contiene tutto 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 FLAGS = -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*.