Relazione del Progetto Sistemi Operativi e Laboratorio

Gabriele Pappalardo

a.a. 2019/2020

1 Introduzione

La seguente relazione tratta del progetto che ho sviluppato durante il corso di *Sistemi Operativi e Laboratorio* dell'a.a. 2019/2020 nella sua **versione semplificata**.
Il progetto consiste nella realizzazione di un programma che simuli l'ambiente di un supermercato dove i *clienti* effettuano gli acquisti mettendosi in coda dai *cassieri* per pagare per poi uscire.
All'interno della simulazione troviamo anche un *direttore* che svolge il ruolo di coordinatore delle casse e che provvede anche a far uscire i clienti che non effettuano acquisti dando loro un'autorizzazione.

2 Implementazione

Come da consegna nella specifica del progetto semplificato, all'interno del programma `supermarket` ho:

- K threads che simulano i cassieri. Ad ogni thread, all'atto della creazione, è stato associato una struttura `cashier_t` che rappresenta un cassierie;
- C threads che simulano i clienti i quali effettuano acquisti. Come i thread cassieri, ad ogni thread cliente è associata una struttura dati `customer_t`;
- uno ed un solo thread direttore che ordina l'apertura e la chiusura dei cassieri e che gestisce i clienti in attesa di uscire che non hanno acquistato prodotti. Il thread direttore come per i cassieri e i clienti ha una struttura dati associata `director_t`.
- una struttura dati per raccogliere tutte le variabili globali e di gestione del supermercato chiamata `supermarket_t`;

All'interno del supermercato esiste un altro thread chiamato *Listener*, il cui compito è quello di restare in attesa di un segnale, nello specifico del segnale `SIGQUIT` o `SIGHUP`. Quando uno dei due segnali viene catturato allora il thread listener segnala, attraverso un flag booleano, la chiusura al main thread supermercato.

2.1 Il file di Configurazione

Prima di avviare il programma è possibile specificare attraverso l'opzione `-c` un file di configurazione che verrà parsato all'interno di una struttura definita dal tipo `config_t`.
Il **config parser** controlla attraverso una bit field se tutti i parametri necessari sono stati inseriti all'interno del file di configurazione. Se uno dei parametri dovesse essere assente allora il programma terminerà con un messaggio di errore inerente.
Il config parser ha anche il compito di controllare se i parametri inseriti rispettano la specifica del progetto (ad esempio, il numero massimo di prodotti acquistabili $P > 0$, o che il tempo di acquisto di un cliente deve essere $T > 10$).

2.2 Il main thread

Il main thread si occupa di creare un'instanza dell'oggetto supermercato e ne avvia l'apertura. All'apertura del supermercato vengono creati i threads nel seguente ordine:

1. il thread *listener* (in modalità detached);
2. il thread *direttore*;
3. e infine K threads *cassieri*.

Durante l'apertura del supermercato viene impostata una maschera dei segnali per i thread creati successivamente in modo da bloccare `SIGHUP` e `SIGQUIT` (e anche `SIGINT` in fase di deployment).
Una volta aperto il supermercato il main thread richiama il `supermarket_loop()` che instanzia i

threads clienti (in modalità detached) e li fa entrare all'interno del supermercato in modo tale che possano effettuare i propri acquisti.
Ogni qualvolta un thread cliente termina, il main thread colleziona le statistiche ottenute dai clienti e le scrive direttamente sul file di log dei clienti. Se il numero dei clienti scende sotto una certa soglia allora il supermercato crea nuovi threads con dei nuovi clienti.
Ricevuto un segnale di chiusura il main loop del supermarket viene interrotto (a meno che non si abbia ricevuto un `SIGHUP` che fa terminare gli acquisti dei clienti) e comincia l'esecuzione della routine di chiusura.
La routine di chiusura è la funzione `supermarket_close()`. La close aspetta che il thread direttore e i threads cassieri terminino (attraverso una join), successivamente dealloca tutte le risorse utilizzate e termina l'esecuzione del programma stampando un messaggio di addio.

2.3 I threads dei clienti

I thread dei clienti vengono creati dal main thread. Appena creati i clienti cominciano la loro spesa, il tempo di attesa è pseudo-casuale e consiste in un intervallo tra [10, `MAX_BUYING_TIME`] dove la costante `MAX_BUYING_TIME` è definita all'interno del file di configurazione. Il thread cliente inoltre genera un altro numero pesudo-casuale per la quantità di prodotti acquistati. Appena trascorsi i millisecondi per fare shopping, se il cliente ha acquistato un numero di prodotti maggiore di zero allora quest'ultimo si mette in fila da un cassiere casuale aperto, altrimenti, se il numero di prodotti acquistati è uguale a zero allora il cliente dovrà attendere il permesso del direttore per uscire, mettendosi in coda da quest'ultimo.
Il thread cliente rimarrà in attesa finchè la sua variabile di condizione: `cond_waiting_queue` (se in coda dal cassiere) o `cond_waiting_director` (se in coda dal direttore) non vengano segnalate e i flag del cliente di conseguenza cambiati per uscire dal loop delle variabili di condizione. Si è deciso di usare due variabili di condizioni diverse soltanto per separare le logiche e garantire una migliore manutenibilità del codice.
Una volta che il cliente viene servito quest'ultimo scrive le statistiche collezionate nella coda condivisa con il main thread ed esce dal supermercato scrivendo un messaggio di saluto.

2.4 I threads dei cassieri

Quando si inizializza un thread cassiere, quest'ultimo genera un numero casuale compreso tra $[20, 80]$, che sarebbe l'$handlingCashierTime$. Un thread cassiere appena creato può entrare in uno dei due stati:

- *aperto*: il cassiere comincia a servire i clienti che sono in coda;
- *chiuso*: allora il cassiere si mette in attesa sulla variabile di condizione `cond_waiting_director` finchè la sua cassa non verrà aperta dal direttore.

Mentre la cassa è aperta il cassiere serve i clienti in coda nella propria fila. Durante il servizio dei clienti in fila il cassiere effettua una `nanosleep` per simulare il tempo di scansione dei prodotti. Possiamo calcolare questo tempo mediante la formula:

$$totalScanningTime = (productDelay * takenProductByCustomer) + handlingCashierTime \quad (1)$$

dove $takenProductByCustomer$ è il numero di prodotti presi dal cliente in coda e dove $productDelay$ è il tempo impiegato per scannerizzare un prodotto. Se il tempo calcolato è maggiore del tempo di intervallo per spedire le informazioni al direttore, allora per garantire la periodicità, il tempo totale viene diviso in parti dove allo scadere di ogni parte il cassiere invia le informazioni su quanti clienti si trovano in coda attualmente al direttore:

$$steps = totalScanningTime / sendInfoDelay \quad (2)$$

Altrimenti il cassiere serve direttamente il cliente.

Se non ci sono clienti disponibili allora il cassiere va in *waiting* sulla variabile di condizione `cond_can_work` e periodicamente invia informazioni al direttore sul numero di clienti in fila. Come detto precedentemente, quando un cassiere viene chiuso quest'ultimo va in uno stato di attesa finchè il direttore non lo risveglia, in modo tale da non sprecare cicli di clock inutilmente.
Quando il direttore segnala la chiusura del supermercato ai cassieri, questi ultimi rimangono attivi finchè tutti i clienti non escono. All'atto dell'uscita i cassieri scrivono le proprie statistiche all'interno della struttura `cashier_t` in modo tale che il main thread che effettua la join possa scriverle nel file di log dei cassieri.

2.5 Il thread del direttore

Il thread direttore controlla se ci sono clienti a cui dare il permesso e/o si occupa dell'apertura/chisura delle casse.
I clienti messi in coda dal direttore ricevono il permesso per potere uscire attraverso la funzione `customer_take_director_permission()` che si occupa di settare il flag booleano di attesa del cliente e la funzione effettua una signal per risvegliarlo.
Il direttore, inoltre, si occupa della gestione dei cassieri e decide se chiuderli e/o aprirli. Il direttore interviene solo quando i cassieri hanno inviato tutti i messaggi. Se il direttore non ha ricevuto tutti i messaggi allora quest'ultimo va in sleeping state.
Una volta ricevuti tutti i messaggi se vengono soddisfatti i criteri descritti dal progetto con i parametri di configurazione, allora il direttore decide di aprire o chiudere un cassiere. Quando un cassiere viene aperto viene estratto dalla cima della lista dei cassieri chiusi e inserito in fondo alla lista dei cassieri aperti. Quando, invece, un cassiere viene chiuso quest'ultimo viene estratto dalla lista dei cassieri aperti e inserito in fondo alla lista dei cassieri chiusi.
A causa del processo tramite cui il direttore riceve i messaggi, potrebbe accadere che il direttore sia risvegliato anche quando non necessario e quindi per evitare un problema di overhead di sveglia e attesa si è deciso di far dormire il thread direttore per 1 millisecondo, in modo da ridurre il carico della CPU, se quest'ultimo ha effettuato del lavoro (servito un cliente e/o aperto un cassiere).
All'atto della chiusura del supermercato, il direttore gestisce gli ultimi clienti in fila e organizza ancora le casse.
Appena tutti i clienti sono usciti il direttore viene notificato dal main thread della chiusura e avverte i cassieri, dopo di chè termina.

2.6 Gestione dei segnali

La gestione dei segnali `SIGHUP` e `SIGQUIT` è gestita da un thread listener creato all'apertura del supermercato.
Per motivi di debugging e di testing si è deciso di gestire il `SIGINT`, infatti, all'atto dell'esecuzione del programma viene installato un signal handler (signal safe) che stampa un messaggio di warning in modo tale da avere un feedback.
Il thread listener effettua in un ciclo while un'attesa con `sigwait`, aspettando un `SIGHUP` o un `SIGQUIT`. Quando un `SIGHUP` viene ricevuto, il Listener setta un flag booleano del supermercato per indicarne la chiusura e torna in attesa. Se venisse reinviato un altro segnale di `SIGHUP` consecutivo allora sul terminale verranno mostrati dei messaggi di feedback sulla chisura del supermercato, dove i clienti vengono invitati ad uscire al più presto. Se il listener riceve un `SIGHUP` allora, oltre a settare il flag booleano discusso prima, setta un ulteriore flag di uscita forzata che provoca la scrittura dei log dei cassieri sul file di log e ne termina l'esecuzione con un invocazione di `exit(SIGQUIT)`.

3 Eseguire il test

Per potere lanciare il test basta eseguire il comando: `sudo make test`.
Il comando di test genera automaticamente l'eseguibile del supermercato all'interno della cartella `build/`.
Il comando **sudo** è richiesto poichè all'avvio del programma quest'ultimo scrive il proprio `pid`

in un file temporaneo all'interno della cartella `/var/run/supermarket` (`test.pid`) che richiede i permessi di root.

4 Strumenti utilizzati

All'interno del progetto viene usata un'unica funzione che non è POSIX compliant, appartenente alla libreria pthread, ovvero la `pthread_setname_np` usata per scopi di debugging durante lo sviluppo.
Quando si riceve un segnale `SIGHUP` il processo dealloca tutta la memoria utilizzata durante il runtime, attraverso il programma `valgrind` è possibile verificare che all'atto dell'uscita il numero di blocchi allocati è uguale al numero di blocchi deallocati (ad eccezione di 272 bytes occupati dal thread Listener).

4.1 Debugging

La macchina che uso per lo sviluppo è un Macbook Pro con 16GB di RAM, una CPU Intel i7 a 6 core e *macOS Catalina* nella versione 10.15.5 , ma principalmente ho usato un VPS, tramite SSH e Visual Studio Code, per sviluppare su una macchina *Ubuntu* 18.04.4, con una CPU Intel Xeon E5-2630 a 4 core e 8GB di RAM, con il seguente kernel:

```
Linux 4.15.0-66-generic #75-Ubuntu x86_64 x86_64 x86_64 GNU/Linux
```

Il debugging dell'applicazione è avvenuto usando i seguenti strumenti:

- `gdb` per il debugging su Linux;
- `lldb` per il debugging su Linux / macOS;
- `valgrind` per monitorare l'uso della memoria (solo su Linux);
- `trace` per monitorare l'uso della memoria (solo su MacOS);
- `htop` per monitorare l'uso delle risorse all'interno della macchina.

In aggiunta a Valgrind, che ho notato che rallenta l'esecuzione dell'applicazione, ho usato i *sanitizers* forniti con `llvm` per avere un feedback migliore sul comportamento del processo.
Il programma è stato inoltre testato e compilato all'interno della macchina virtuale *Xubuntu* 14.04 fornitaci all'interno del DidaWiki del corso di SOL a.a. 2019/2020.
Nella pagina seguente è stata riportata un immagine dell'applicazione in fase di esecuzione monitorata da `htop`.

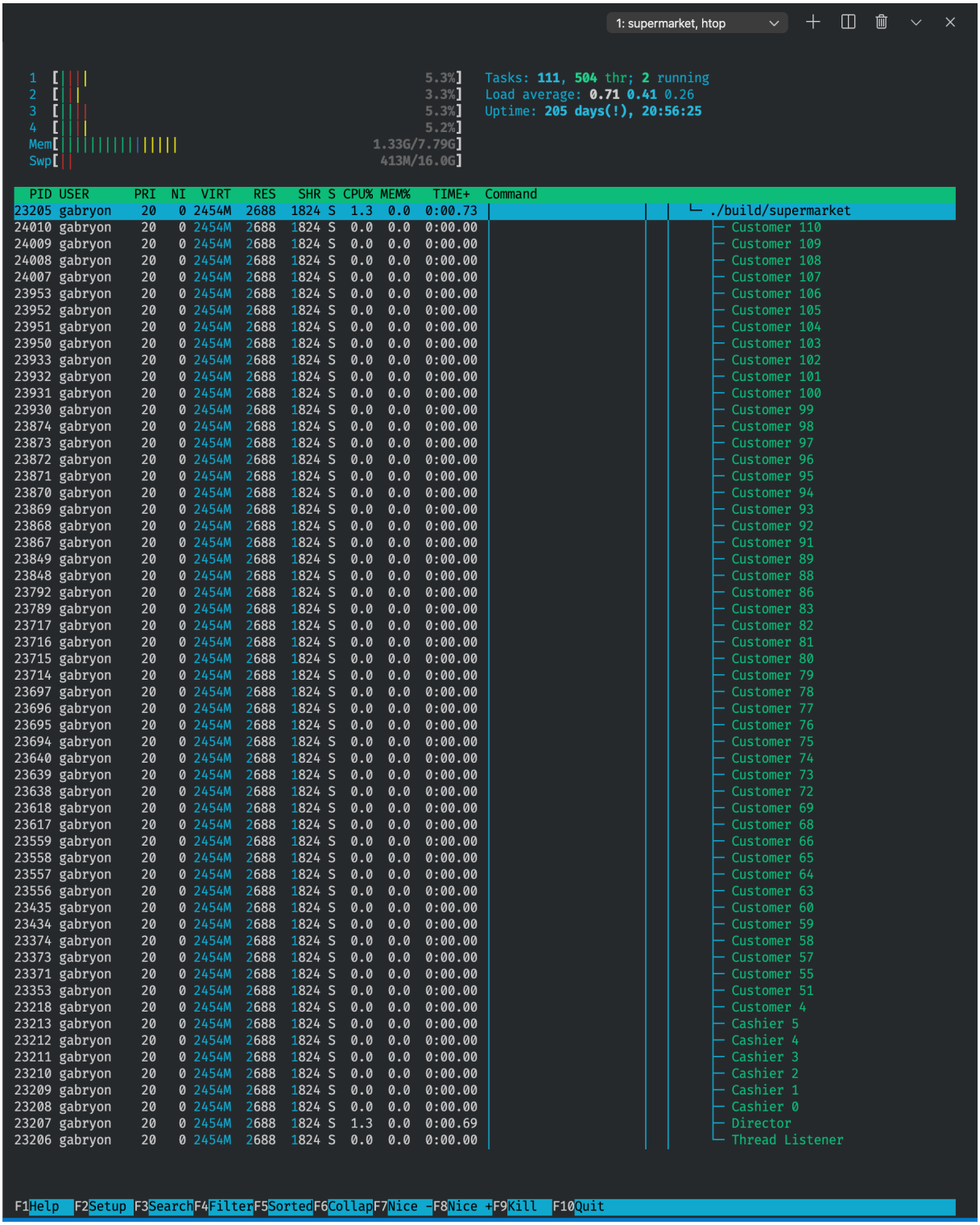

Figure 1: Come si può osservare all'interno dell'immagine, ogni thread ha un proprio nome e l'uso della CPU non è sovraccaricato.