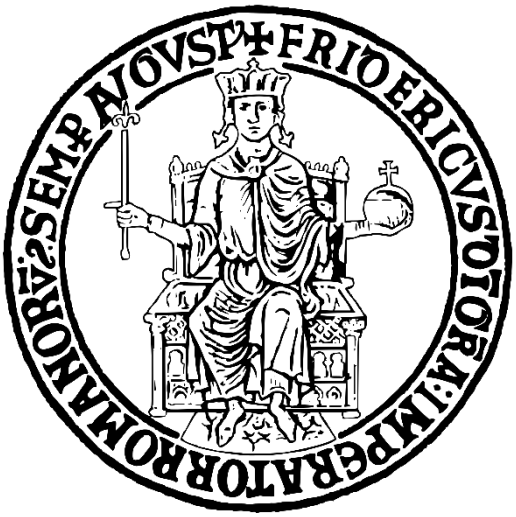
Università degli Studi di Napoli Federico II
Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione



Corso di Laurea in Informatica - Insegnamento di Laboratorio di Sistemi Operativi

Anno Accademico 2023/2024

Progettazione di "Miao Market", una Piattaforma per la Simulazione di un Supermercato

---

Vincenzo Marotta - N86004151

Antonio Abbatiello - N86003037

Leonardo Colamarino – N86003586

24/7/2024

Indice

Capitolo 1 – Descrizione del progetto

Introduzione al progetto

Il progetto “Miao Market” ha l’obbiettivo di realizzare una simulazione di un sistema che modella il funzionamento di un supermercato con casse e clienti. Il sistema prevede un afflusso di deflusso regolato di clienti all’interno del supermercato con l’intento di gestire efficientemente il flusso di persone e il processo di pagamento alle casse.

La piattaforma è composta da un server che gestisce le richieste e la logica di funzionamento e un client tramite cui è possibile interagire col server sia in modalità automatica che interattiva.

1.1 Server

Il server consiste nel supermercato vero e proprio, esso gestisce la logica di funzionamento e permette di far connettere i client tramite socket all’indirizzo definito nell’header file **parameters/parameters.h** dalla costante SERVER_ADDRESS (di default sarà localhost) e alla porta definita dalla costante PORT (di default sarà 9090).

Inoltre è possibile specificare dei parametri aggiuntivi nel file **parameters/parameters.h**:

- **MAX_CASHIERS** che definisce il numero massimo di cassieri nel supermercato,
- **ACTIVE_CASHIERS** che definisce il numero di cassieri attivi nel supermercato,
- **MAX_CLIENTS** che definisce il numero massimo di clienti che possono essere presenti all’interno del supermercato contemporaneamente,
- **CLIENTS_BATCH_SIZE** che definisce il numero di clienti che possono entrare per volta quando il numero massimo di clienti scende sotto la soglia **MAX_CLIENT** – **CLIENTS_BATCH_SIZE**.

Il server dispone di una struttura dati coda di tipo FIFO con la peculiarità di essere thread-safe, in particolare:

- **waiting_to_enter** che rappresenta la coda dei clienti in attesa di entrare,
- **clients** che rappresenta la coda di clienti attualmente impegnati negli acquisti nel supermercato,
- **waiting_to_exit** che rappresenta la coda dei clienti che non hanno fatto acquisti in attesa del permesso del supervisore per uscire.

Una volta avviato vengono inizializzati i cassieri e il supervisore con un tempo di servizio casuale e le loro rispettive code. Successivamente verranno creati i thread corrispondenti al numero di cassieri attivi ed uno per il supervisore che si occuperanno di gestire le loro logiche di funzionamento successivamente approfondite, infine verrà effettuato il bind della socket. Una volta fatto ciò il server eseguirà il loop principale che consiste nel mettersi in ascolto di connessioni da parte dei client tramite l'apposita funzione **listen**.

Quando un client si connette il server creerà un nuovo thread che gestirà il client tramite l'apposita funzione **void* handleClient(void* arg)**.

Il thread che gestisce il client controlla che il numero dei clienti attualmente connessi non superi il massimo numero consentito, se si è superato il massimo il thread accoderà il client nella coda **waiting_to_enter** e si metterà in attesa di essere risvegliato (in caso contrario verrà fatto entrare direttamente nel supermercato). Una volta risvegliato (da un cassiere o il supervisore) il thread estrae il client dalla coda **waiting_to_enter** e lo accoderà a **clients** mandando al client tramite **send** la lista dei prodotti disponibili e aspetterà di ricevere il carrello tramite **recv** dal client. Una volta ricevuto il carrello se il client ha preso almeno un prodotto lo accoderà al cassiere con la coda più breve scelto tramite la funzione **get_shortest_queue**, altrimenti se non ha preso nulla lo accoderà alla coda del supervisore **waiting_to_exit**.

Il thread dei cassieri che esegue la funzione **cashier_logic** consiste in un loop che controlla quando viene accodato un client alla sua coda (come un cliente che si mette in fila alla cassa), quando ciò accade il thread leggerà il numero di prodotti acquistati e calcolerà il tempo necessario a servire il client e simulerà il tempo di servizio tramite la funzione **sleep**, una volta finito di servire il client lo rimuoverà dalla sua coda e comunicherà al client il completamento e chiuderà la socket di comunicazione.

Successivamente il thread controllerà se la condizione di ingresso di nuovi clienti è soddisfatta per notificare i thread che sono in attesa di entrare tramite l'apposita funzione **pthread_cond _broadcast**.

Il thread del supervisore esegue la funzione **supervisor_logic** che è speculare alla funzione **cashier_logic** con la sola differenza che il tempo di servizio è scelto casualmente tramite l'apposita funzione **rand**.

1.2 Client

Il client rappresenta un cliente che vuole entrare nel supermercato e fare eventuali acquisti.

Esso può essere avviato in due modalità:

- Automatica dove il comportamento del client sarà deciso in maniera automatica,
- Interattiva dove il comportamento del client sarà controllato da noi stessi.

Supponendo di avviarlo in modalità interattiva non appena il server darà il permesso di entrare ci sarà presentata un'interfaccia grafica dove è possibile scegliere i prodotti da acquistare e aggiungerli al carrello rappresentato da una coda non appena si finisce si verrà serviti (in caso non siano stati selezionati prodotti bisognerà aspettare che il supervisore ci dia il permesso di uscire) da un cassiere con un tempo pari al numero dei prodotti acquistati sommato al tempo di servizio del cassiere. Una volta terminato il client chiuderà la socket e terminerà il programma.

1.3 Dettagli implementativi

Per gestire i vari thread che girano nel server durante la sua esecuzione bisogna preoccuparsi di sincronizzare l'accesso alle risorse condivise e evitare le race condition, per fare ciò si è scelto di utilizzare strutture dati thread-safe e pertanto è stata implementata una coda che supporta l'accesso concorrente pertanto viene utilizzata la libreria **pthread** e in particolare le primitive di sincronizzazione basate sui mutex quali **pthread_mutex** e **pthread_cond**.

Inoltre, per assicurare che i client entrino *n* alla volta dove *n* corrisponde a **CLIENT_BATCH_SIZE** si è utilizzato **pthread_cond_wait** per mettere in attesa i thread che devono attendere che il numero dei clienti scenda e **pthread_cond_broadcast** per notificare i thread in attesa che possono proseguire il loro flusso di esecuzione

Capitolo 2 – Build ed Esecuzione del Software

1.1 Guida alla compilazione del progetto

Questa guida mostra come compilare il progetto usando lo strumento make. Di seguito i passi per effettuare la corretta compilazione del programma.

Prerequisiti:

Prima di iniziare assicurarsi di avere installati i seguenti strumenti:

- GCC (GNU Compiler Collection)
- Make
- Bash
- Docker
- Docker compose

Istruzioni di compilazione:

Per eseguire il programma sulla tua macchina

1. **Clona la Repository**

   Innanzitutto, clona la repository sulla tua macchina. Aprire il terminale ed eseguire:

   ```
   git clone https://github.com/leokm02/LSO-23-24.git
   cd LSO-23-24
   ```

   Se possiedi già una copia del progetto è possibile saltare questo passaggio.

2. **Compilazione programma**

Il progetto include un Makefile che gestisce l'intero processo di compilazione. Per compilare il programma eseguire il seguente comando:

```
make
```

Per eseguire il progetto in un container docker

1. **Clona la Repository**

   Innanzitutto, clona la repository sulla tua macchina. Aprire il terminale ed eseguire:

   ```
   git clone https://github.com/leokm02/LSO-23-24.git
   cd LSO-23-24
   ```

   Se possiedi già una copia del progetto è possibile saltare questo passaggio.

2. **Esecuzione programma**

   Il progetto include lo script 'launchApp.sh' che gestisce la fase di building ed esecuzione. Per lanciarlo esegui:

   ```
   ./launchApp.sh
   ```

3. **Interrompere il programma**

   Per interrompere l'esecuzione del server premere `Ctrl` + `C`.
   Questo arresterà il server e salverà un file log.txt contenente il log del server nella cartella del progetto.

Istruzioni per l'esecuzione

Per inizializzare il server eseguire i seguenti comandi dal terminale:

```
server.out
```

Se il server è già in esecuzione in un container docker è possibile saltare questo passaggio.
Per avviare un client bot in modalità automatica eseguire il seguente comando dal terminale:

```
client.out
```

Questo avvierà un client bot che sceglierà i prodotti in un tempo casuale e li comprerà.
Per l'esecuzione interattiva del client eseguire il seguente comando dal terminale:

```
client.out
```

In questo modo controlleremo il client noi stessi e decideremo cosa comprare.
Il progetto include lo script launchMultipleClient.sh che genera multipli clienti bot allo stesso tempo. Per utilizzarlo eseguire il seguente comando:

```
launchMultipleClient.sh CLIENTS_TO_LAUNCH
```

dove `CLIENTS_TO_LAUNCH` è un numero (es. 10).

Troubleshooting

• **Permission Denied**: Se ci si dovesse imbattere in un errore di permessi mentre si esegue launchApp.sh o launchMultipleClient.sh si potrebbe aver bisogno di rendere lo script eseguibile. Esegui:

```
chmod    launchApp.sh launchMultipleClient.sh
```

• **Missing Dependencies**: Assicurarsi che tutte le dipendenze siano installate. Se dovessero mancare delle dipendenze installarle usando il proprio package manager (e.g., apt, yum, brew).