

Università degli Studi di Napoli Federico II

Supermarket Simulation

Author: Vincenzo Riccio N86004441

Marzo 2025

Contents

Contents	1
1 Introduzione	2
2 Analisi dei requisiti	2
3 Il problema produttore consumatore	2
4 Architettura del sistema	2
4.1 Schema del server	2
4.2 Client	3
5 Implementazione	5
5.1 Thread "cassiere"	5
5.2 Thread "spesa"	6
5.3 Thread "client"	7
5.4 Thread "direttore"	8
5.5 Thread "connection"	9
6 Conclusioni	10

1 Introduzione

Il progetto consiste nella simulazione di un supermercato, modellando le interazioni tra quest'ultimo, i clienti e le casse in un contesto multi-threaded. Questo documento si divide in diverse sezioni che forniscono una panoramica dei requisiti, delle scelte architetturali e delle implementazioni adottate.

2 Analisi dei requisiti

Il presente progetto prevede la realizzazione di un' **architettura client-server** in grado di **simulare un supermercato** dotato di **K casse** e con un **limite massimo di C clienti** presenti simultaneamente all'interno della struttura.

All'inizio della **simulazione**, un totale di **C clienti** accede contemporaneamente al supermercato. Successivamente, ogni volta che **E clienti** terminano i propri acquisti ed escono, un numero equivalente di **E nuovi clienti** viene ammesso all'interno.

Ogni **cliente** trascorre un periodo di tempo variabile dedicato alla **spesa**. Una volta terminato il proprio **tempo di acquisto**, si dirige verso una **cassa**, mettendosi in **fila** e attendendo il proprio **turno di pagamento**. Completato il processo di **pagamento**, il cliente esce definitivamente dal supermercato.

La gestione delle **casse** è affidata a **cassieri**, ciascuno dei quali serve i clienti seguendo una **politica FIFO** (First In, First Out). Il **tempo di servizio** di ogni cassa è determinato da due componenti principali:

- **Una componente fissa**, specifica per ogni cassiere.
- **Una componente variabile**, che dipende **linearmente** dal numero di **prodotti acquistati** dal cliente.

Questa struttura permette di simulare un **flusso dinamico di clienti** all'interno del supermercato, regolando il numero di ingressi in base alle uscite e garantendo un servizio organizzato ed efficiente.

3 Il problema produttore consumatore

Il **server** è caratterizzato dal classico problema del **produttore-consumatore**. In questo contesto:

- I **clienti** agiscono da **produttori**, generando **richieste di servizio**.
- I **cassieri** operano come **consumatori**, elaborando le richieste secondo una politica **FIFO**.
- Il **direttore** regola il **flusso di ingresso** per prevenire il **sovraffollamento**.

Per gestire questa problematica, vengono utilizzate **primitive di sincronizzazione** come **mutex** e **variabili di condizione**, che garantiscono il corretto accesso alle **strutture dati condivise**, evitando **race condition** e **deadlock**.

4 Architettura del sistema

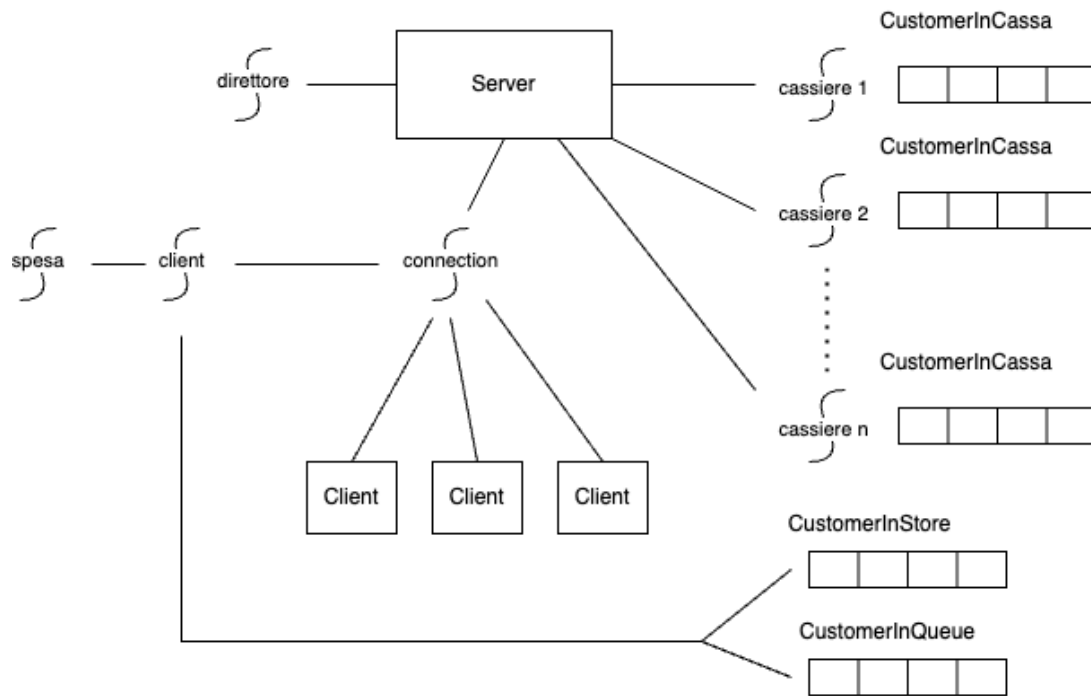
4.1 Schema del server

L'architettura del sistema si basa sul modello client-server.

Il server è stato sviluppato in C. La gestione del supermercato è stata implementata con l'utilizzo di thread multipli. Di seguito le scelte implementative dei thread:

- **Thread "cassiere"**: Gestisce una singola cassa, servendo i clienti in ordine di arrivo. Il tempo di servizio dipende dal numero di prodotti acquistati.

- **Thread "spesa"**: Simula l'attività di un cliente che effettua acquisti per un tempo T prima di mettersi in fila alla cassa.
- **Thread "client"**: Gestisce la ricezione delle connessioni dei clienti e li inserisce nella coda d'ingresso del supermercato.
- **Thread "direttore"**: Controlla il numero di clienti presenti e ne fa entrare di nuovi quando lo spazio disponibile lo consente.
- **Thread "connection"**: Gestisce le connessioni in arrivo creando un nuovo thread client per ogni richiesta



4.2 Client

Il client è stato implementato in **Python**. L'interfaccia è stata creata con **Qt Designer** e poi tradotta in codice Python con **pyuic**. L'utente tramite l'interfaccia può decidere quanti clienti far entrare nel supermercato e cosa acquistare. Di seguito gli screenshot dell'applicazione:









Supermercato

Quanti clienti vuoi far entrare nel supermercato?

10

Cancel OK

Carrello

Latte		10
Mela		0
Carne		0
Pasta		0
Cereali		0
Uova		0
Formaggio		0
Pane		0

Tempo trascorso: 3 secondi

Cancel OK

5 Implementazione

In questa sezione verrà illustrata l'implementazione del server secondo l'analisi fatta in precedenza.

5.1 Thread "cassiere"

```
1 void * cassiere(void * arg) {
2     const int numCassa = *(int *) arg;
3     // Ciclo infinito per servire i clienti
4     printf("Cassa%d aperta.\n", numCassa);
5     while (1) {
6         // Attende che ci siano clienti in coda
7         pthread_mutex_lock(&cassa[numCassa].mutexCassa);
8         while (isEmpty(&cassa[numCassa].customerInCassa)) {
9             pthread_cond_wait(&cassa[numCassa].condCodaVuota, &cassa[numCassa].
10                mutexCassa);
11        }
12        Customer *client = dequeue(&cassa[numCassa].customerInCassa);
13        pthread_mutex_unlock(&cassa[numCassa].mutexCassa);
14        // Simula il pagamento
15        if (client) {
16            printf(COLOR_MAGENTA "Il cliente%d'è in fase di pagamento alla cassa%d\n"
17                COLOR_RESET, client->id, numCassa);
18            sleep(cassa[numCassa].fixedTime + client->nProducts);
19            printf(COLOR_GREEN "Il cliente%d ha terminato il pagamento ed esce dal
20                supermercato.\n" COLOR_RESET, client->id);
21            // Libera la memoria e aggiorna il numero di clienti presenti nel
22                supermercato
23            free(client->products);
24            free(client);
25            // Aggiorna il numero di clienti presenti nel supermercato
26            pthread_mutex_lock(&mutexStore);
27            currentCustomers--;
```

La funzione `cassiere` rappresenta il comportamento di una cassa all'interno del supermercato. Ogni cassa lavora in un **ciclo infinito**, servendo i clienti in base all'ordine di arrivo (FIFO).

Funzionamento:

1. Inizializzazione:

- La funzione riceve un parametro `arg`, che rappresenta il numero identificativo della cassa.
- Viene stampato un messaggio per indicare che la cassa è operativa.

2. Attesa dei clienti:

- Si acquisisce il `mutexCassa` per proteggere l'accesso alla coda della cassa.
- Se la coda è vuota, il thread attende con `pthread_cond_wait` finché un cliente non arriva.

3. Gestione del cliente:

- Quando un cliente è disponibile, viene estratto dalla coda con `dequeue`.
- Il `mutexCassa` viene rilasciato per consentire l'accesso ad altri thread.
- Il pagamento viene simulato con `sleep()`, basato su un tempo fisso della cassa più un valore variabile dipendente dal numero di prodotti acquistati.

4. Uscita del cliente:

- Una volta completato il pagamento, viene stampato un messaggio.
- La memoria allocata per il cliente e i suoi prodotti viene liberata.

- Il numero di clienti nel supermercato viene aggiornato in modo sicuro utilizzando un **mutex** globale.

Questo approccio garantisce che ogni cassa operi in modo **parallelo e indipendente**, servendo i clienti in modo ordinato ed evitando race condition tramite **primitive di sincronizzazione** (mutex e condition variable).

5.2 Thread "spesa"

```

1 void * spesa(void * arg) {
2     Customer *client = (Customer *) arg;
3     printf(COLOR_MAGENTA "Cliente%d fa acquisti per %d secondi.\n" COLOR_RESET, client
4         ->id, client->time);
5     sleep(client->time);
6     if (client->nProducts > 0) {
7         const int chosenCassa = rand() % NUM_CASSE;
8         pthread_mutex_lock(&cassa[chosenCassa].mutexCassa);
9         enqueue(&cassa[chosenCassa].customerInCassa, client);
10        pthread_cond_signal(&cassa[chosenCassa].condCodaVuota);
11        pthread_mutex_unlock(&cassa[chosenCassa].mutexCassa);
12        printf(COLOR_CYAN "Cliente%d si mette in fila alla cassa %d.\n" COLOR_RESET,
13            client->id, chosenCassa);
14    } else {
15        printf("Cliente%d non ha acquistato nulla ed esce dal supermercato.\n", client
16            ->id);
17        free(client->products);
18        free(client);
19        pthread_mutex_lock(&mutexStore);
20        currentCustomers--;
21        pthread_mutex_unlock(&mutexStore);
22    }
23    pthread_exit(0);
24 }

```

La funzione `spesa` rappresenta il comportamento di un cliente all'interno del supermercato. Ogni cliente esegue questa funzione come un thread separato, simulando il processo di acquisto e successivo pagamento.

Funzionamento:

1. Inizio della spesa:

- La funzione riceve come argomento un puntatore a una struttura `Customer`, che contiene le informazioni del cliente.
- Viene stampato un messaggio per indicare che il cliente sta effettuando gli acquisti.
- Il cliente attende (`sleep()`) per un tempo pari alla durata della sua spesa.

2. Scelta della cassa:

- Se il cliente ha acquistato almeno un prodotto, sceglie casualmente una delle casse disponibili (`rand() % NUM_CASSE`).
- Si acquisisce il `mutexCassa` per garantire accesso esclusivo alla coda della cassa selezionata.
- Il cliente viene inserito nella coda della cassa con `enqueue()`.
- Viene inviato un segnale con `pthread_cond_signal` per notificare alla cassa che un nuovo cliente è arrivato.
- Il `mutexCassa` viene rilasciato.
- Viene stampato un messaggio per indicare che il cliente è in fila alla cassa.

3. Gestione del cliente senza acquisti:

- Se il cliente non ha acquistato nulla, viene stampato un messaggio che indica la sua uscita senza acquisti.
- La memoria allocata per il cliente e i suoi prodotti viene liberata.

- Il numero totale di clienti nel supermercato viene aggiornato utilizzando un `mutex` globale.

4. Terminazione del thread:

- La funzione termina con `pthread_exit(0)`, chiudendo il thread in modo sicuro.

Questa funzione gestisce in modo efficace il ciclo di vita di un cliente, garantendo una corretta sincronizzazione con le casse e aggiornando in modo sicuro il numero di clienti presenti nel supermercato.

5.3 Thread "client"

```

1 void * client(void * arg) {
2 // Riceve il cliente e lo mette in coda nel supermercato
3 const int clientSocket = *(int *) arg;
4 Customer *customer = malloc(sizeof(Customer));
5 recv(clientSocket, customer, sizeof(Customer), 0);
6
7 pthread_mutex_lock(&mutexStore);
8 if (currentCustomers < MAX_CUSTOMERS) {
9     enqueue(&customerInStore, customer);
10    currentCustomers++;
11    printf(COLOR_BLUE "Il cliente %d è entrato nel supermercato.\n" COLOR_RESET,
12           customer->id);
13    pthread_t spesaThread;
14    pthread_create(&spesaThread, NULL, spesa, (void *) customer);
15    pthread_detach(spesaThread);
16 } else {
17     enqueue(&customerInQueue, customer);
18     printf(COLOR_RED "Il supermercato è pieno, il cliente %d è in attesa.\n"
19            COLOR_RESET, customer->id);
20 }
21 pthread_mutex_unlock(&mutexStore);
22 pthread_exit(0);
23 }
```

La funzione `client` rappresenta il comportamento di un cliente all'arrivo nel supermercato. Ogni cliente viene gestito come un thread separato, che determina se può entrare immediatamente o deve attendere in coda.

Funzionamento:

1. Ricezione del cliente:

- Il thread riceve un socket come argomento, identificando la connessione con un cliente remoto.
- Viene allocata dinamicamente una struttura `Customer` per memorizzare le informazioni del cliente.
- Attraverso la funzione `recv()`, vengono ricevuti i dati del cliente dal socket.

2. Gestione dell'ingresso nel supermercato:

- Si acquisisce il `mutexStore` per garantire accesso esclusivo alla variabile globale `currentCustomers`.
- Se il numero di clienti nel supermercato è inferiore a `MAX_CUSTOMERS`:
 - Il cliente viene aggiunto alla coda dei clienti nel supermercato con `enqueue()`.
 - Il contatore dei clienti nel supermercato viene incrementato.
 - Viene stampato un messaggio per confermare l'ingresso del cliente.
 - Viene creato un nuovo thread `spesaThread` per simulare la fase di shopping del cliente (`pthread_create()`).
 - Il thread della spesa viene scollegato dal thread principale con `pthread_detach()`, permettendogli di terminare autonomamente.

3. Gestione dei clienti in attesa:

- Se il supermercato ha raggiunto la capacità massima, il cliente viene messo in coda nella lista d'attesa con `enqueue()`.

- Viene stampato un messaggio per indicare che il cliente sta aspettando di entrare.

4. Terminazione del thread:

- Il mutexStore viene rilasciato.
- Il thread termina con `pthread_exit(0)`, chiudendosi in modo sicuro.

Questa funzione garantisce un controllo efficace sull'ingresso dei clienti nel supermercato, evitando sovraffollamenti e gestendo correttamente i clienti in attesa.

5.4 Thread "direttore"

```

1 void *direttore(void *arg) {
2     while (1) {
3         pthread_mutex_lock(&mutexStore);
4         // Se il numero di clienti scende sotto la soglia, ne fa entrare altri E
5         if (currentCustomers <= MAX_CUSTOMERS - E && !isEmpty(&customerInQueue)) {
6             int clientiDaFarEntrare = E;
7             while (clientiDaFarEntrare > 0 && !isEmpty(&customerInQueue)) {
8                 Customer *client = dequeue(&customerInQueue);
9                 if (client->nProducts == 0) {
10                     printf("Il cliente %d non ha acquistato nulla ed esce immediatamente\n", client->id);
11                     free(client);
12                 } else {
13                     enqueue(&customerInStore, client);
14                     currentCustomers++;
15                     printf(COLOR_BLUE "Il cliente %d e' entrato nel supermercato.\n"
16                           COLOR_RESET, client->id);
17                     pthread_t clientThread;
18                     pthread_create(&clientThread, NULL, spesa, client);
19                     pthread_detach(clientThread);
20                     clientiDaFarEntrare--;
21                 }
22             }
23             pthread_mutex_unlock(&mutexStore);
24             sleep(1); // Controlla periodicamente
25         }
26     }
}

```

La funzione `direttore` rappresenta il comportamento del direttore del supermercato, che controlla periodicamente il numero di clienti all'interno e decide se far entrare nuovi clienti in attesa.

Funzionamento:

1. Monitoraggio continuo:

- La funzione è un ciclo infinito che verifica costantemente la situazione del supermercato.
- Ogni iterazione esegue un controllo con cadenza regolare (`sleep(1)`).

2. Gestione dell'ingresso dei clienti:

- Si acquisisce il `mutexStore` per garantire un accesso esclusivo alle strutture dati condivise.
- Se il numero di clienti nel supermercato è inferiore a `MAX_CUSTOMERS - E` e ci sono clienti in attesa:
 - Viene determinato quanti clienti possono entrare (`E` alla volta).
 - Finché ci sono posti disponibili e clienti in coda, il direttore li fa entrare uno alla volta.

3. Gestione dei clienti che non acquistano nulla:

- Se un cliente prelevato dalla coda non ha prodotti da acquistare (`nProducts == 0`):
 - Viene stampato un messaggio che indica che esce immediatamente.
 - La memoria allocata per il cliente viene liberata con `free()`.

4. Avvio della fase di spesa per i clienti ammessi:

- Se un cliente ha prodotti da acquistare:
 - Viene aggiunto alla coda dei clienti nel supermercato con `enqueue()`.
 - Il contatore dei clienti nel supermercato viene incrementato.
 - Viene stampato un messaggio per confermare il suo ingresso.
 - Viene creato un thread separato per la fase di shopping del cliente (`pthread_create()`).
 - Il thread della spesa viene scollegato dal thread principale con `pthread_detach()`.

5. Rilascio del mutex e attesa del prossimo controllo:

- Dopo aver gestito gli ingressi, il `mutexStore` viene rilasciato.
- Il thread si mette in pausa per un secondo (`sleep(1)`) prima di ripetere il controllo.

Questa funzione garantisce un flusso controllato di clienti all'interno del supermercato, evitando sovraffollamenti e gestendo in modo efficiente la coda di attesa.

5.5 Thread "connection"

```
1 void * connection(void * arg) {
2     // Crea un socket , si mette in ascolto e crea un nuovo thread per ogni richiesta
3     const char *ip = (char *) arg;
4     //int clientSocket; // Descrittori dei socket
5     struct sockaddr_in serverAddr, clientAddr; // Indirizzi del server e del client
6     socklen_t addrSize = sizeof(struct sockaddr_in); // Dimensione dell'indirizzo
7     const int serverSocket = socket(AF_INET, SOCK_STREAM, 0); // Creazione del socket
8     if (serverSocket < 0) {
9         perror("Errore nella creazione del socket");
10        exit(1);
11    }
12    serverAddr.sin_family = AF_INET; // Dominio del socket
13    serverAddr.sin_port = htons(PORT); // Porta del socket
14    serverAddr.sin_addr.s_addr = inet_addr(ip); // Indirizzo del server
15    if (bind(serverSocket, (struct sockaddr *) &serverAddr, sizeof(serverAddr)) < 0) {
16        perror("Errore nel binding del socket");
17        exit(1);
18    }
19    if (listen(serverSocket, MAX_CUSTOMERS) < 0) {
20        perror("Errore nell'ascolto del socket");
21        exit(1);
22    }
23    printf("Server in ascolto sulla porta %d...\n", PORT);
24    while (1) {
25        int clientSocket = accept(serverSocket, (struct sockaddr *) &clientAddr, &
26                                addrSize);
27        printf("Nuova connessione accettata\n");
28        if (clientSocket < 0) {
29            perror("Errore nell'accettazione del client");
30            exit(1);
31        }
32        pthread_t clientThread;
33        pthread_create(&clientThread, NULL, client, (void *) &clientSocket);
34        pthread_detach(clientThread);
35        usleep(500);
36    }
37 }
```

La funzione `connection` è responsabile della gestione delle connessioni in entrata al supermercato. Si occupa di creare un server socket, mettersi in ascolto e accettare connessioni dai clienti, creando un nuovo thread per ognuno di essi.

Funzionamento:

1. Creazione del socket:

- Il socket del server viene creato con la chiamata `socket(AF_INET, SOCK_STREAM, 0)`.

- Se la creazione fallisce, viene stampato un errore e il programma termina.
- 2. Configurazione dell'indirizzo del server:**
 - Il socket viene configurato per utilizzare il protocollo IPv4 (`AF_INET`).
 - La porta del server viene impostata con `htons(PORT)`.
 - L'indirizzo IP del server viene assegnato con `inet_addr(ip)`.
 - 3. Binding del socket al server:**
 - La funzione `bind()` associa il socket all'indirizzo del server.
 - Se il binding fallisce, viene stampato un errore e il programma termina.
 - 4. Messa in ascolto delle connessioni:**
 - La funzione `listen()` imposta il socket per accettare fino a `MAX_CUSTOMERS` connessioni simultanee.
 - Se l'operazione fallisce, viene stampato un messaggio di errore e il programma termina.
 - Un messaggio viene stampato a schermo per indicare che il server è in ascolto.
 - 5. Accettazione delle connessioni in un ciclo infinito:**
 - Viene chiamata la funzione `accept()` per accettare nuove connessioni dai client.
 - Se la connessione è accettata con successo, viene stampato un messaggio di conferma.
 - Se l'accettazione fallisce, viene stampato un errore e il programma termina.
 - 6. Gestione del cliente con un thread dedicato:**
 - Per ogni connessione accettata, viene creato un nuovo thread con `pthread_create()` per gestire il cliente.
 - Il thread esegue la funzione `client()` e viene scollegato con `pthread_detach()` per evitare memory leak.
 - Una breve pausa (`usleep(500)`) viene introdotta per evitare sovraccarichi.

Questa funzione consente al supermercato di gestire in modo efficiente le connessioni dei clienti, accettandole in modo continuo e creando un thread dedicato per ciascun cliente.

6 Conclusioni

Questa simulazione offre un modello realistico di gestione di un supermercato, sfruttando il multi-threading per rappresentare il comportamento parallelo dei clienti e delle casse. L'implementazione garantisce efficienza e sincronizzazione tra le diverse componenti del sistema, offrendo un ambiente simulato.