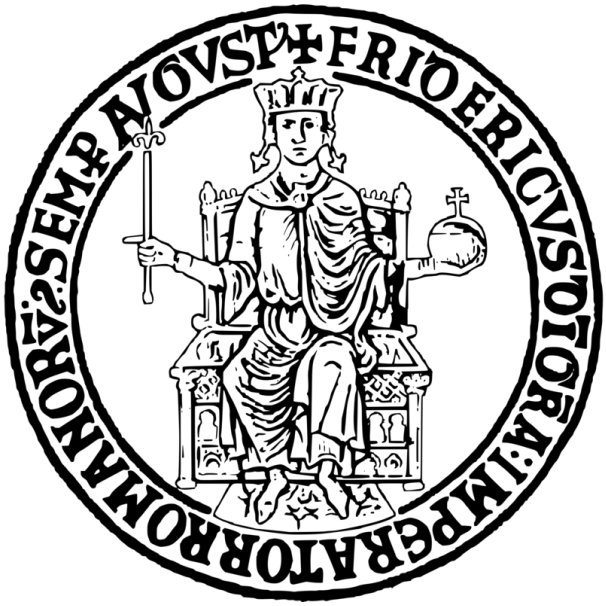


Università degli Studi di Napoli Federico II

Supermarket Simulation

Author: Vincenzo Riccio N86004441

Marzo 2025

Contents

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.

L'obiettivo principale della simulazione è analizzare e ottimizzare la gestione delle risorse all'interno di un supermercato, garantendo un flusso efficiente di client. Attraverso l'uso di una **architettura client-server**, la simulazione permette di gestire dinamicamente l'ingresso e l'uscita dei clienti, l'operatività delle casse e l'interazione tra i diversi attori del sistema.

Questo documento è suddiviso in più sezioni che forniscono una visione dettagliata dell'intero progetto:

- **Analisi dei Requisiti**: descrive le specifiche funzionali e non funzionali del sistema, nonché le regole di base che governano la simulazione.
- **Problema Produttore-Consumatore**: introduce il modello teorico che rappresenta il comportamento del sistema, illustrando come clienti e cassieri interagiscono.
- **Architettura del Sistema**: approfondisce le scelte di progettazione e le tecnologie utilizzate per implementare la simulazione.
- **Implementazione**: descrive nel dettaglio il codice e le tecniche usate per sviluppare il sistema multi-threaded.
- **Risultati e Conclusioni**: analizza i risultati ottenuti attraverso la simulazione.

Attraverso questo studio, sarà possibile comprendere le difficoltà legate alla gestione di ambienti condivisi in contesti altamente dinamici, come quello di un supermercato. Inoltre, l'approccio multi-threaded utilizzato permette di esplorare concetti avanzati di programmazione concorrente e sincronizzazione dei processi.

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**[1]. 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**.

References

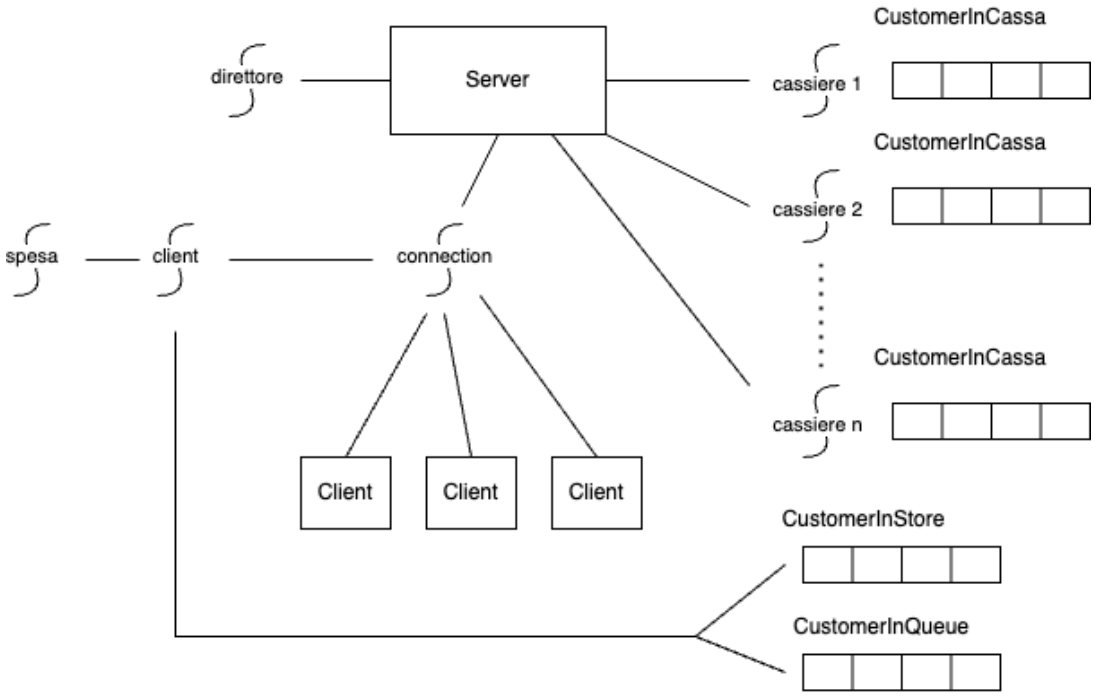
[1] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 10th. John Wiley & Sons, 2021. ISBN: 978-1119800361.

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:

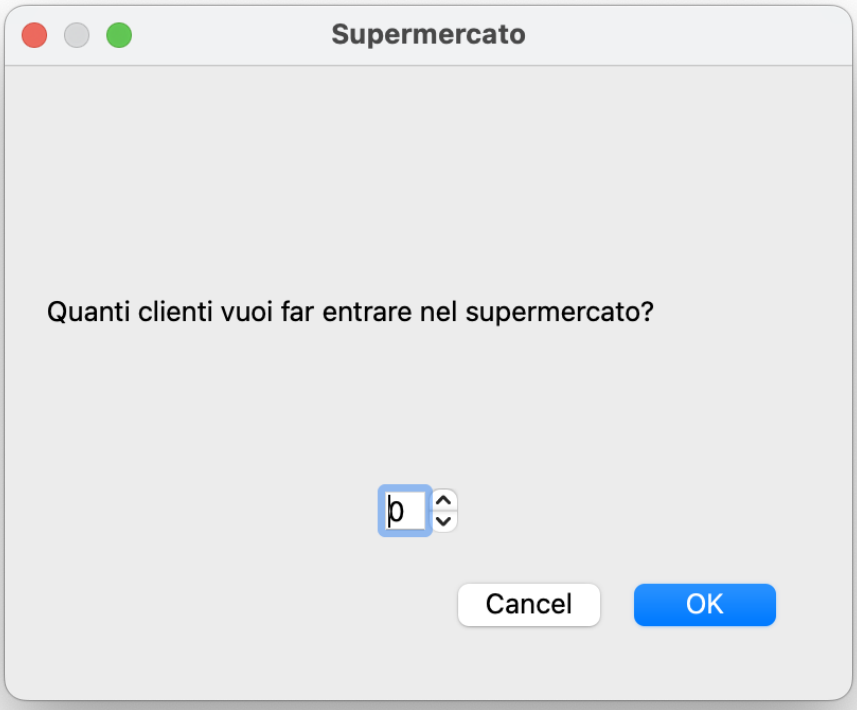
- **Thread "cassiere"**: Gestisce una singola cassa, servendo i clienti in ordine di arrivo. Il tempo di servizio dipende dal numero di prodotti acquistati.
- **Thread "spesa"**: Simula l'attività di un cliente che effettua acquisti per un tempo T prima di mettersi in fila alla cassa.
- **Thread "client"**: 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

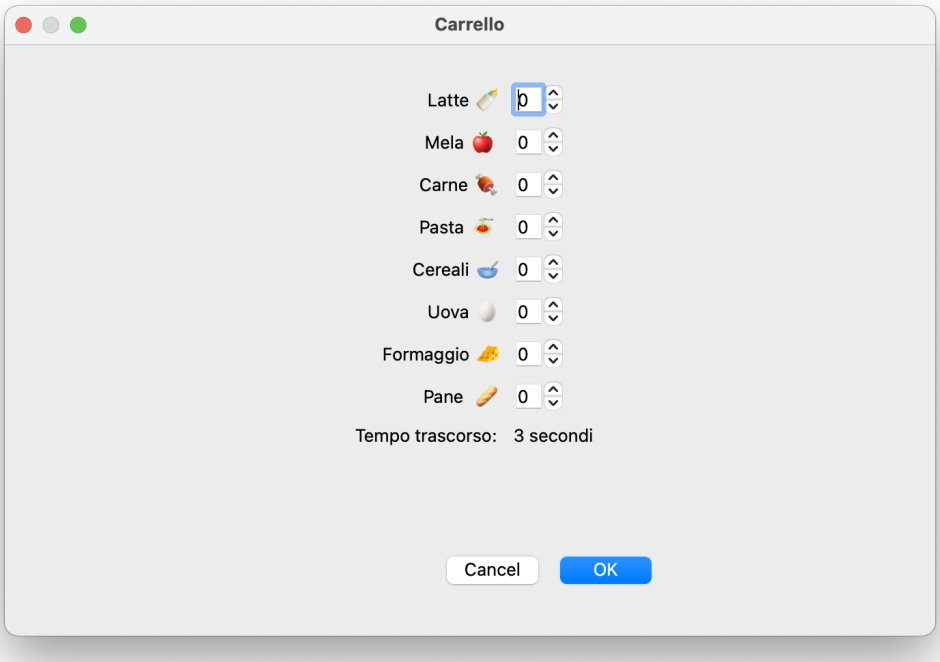


4.2 Client

Il client è stato implementato in **Python** e l'interfaccia è stata creata con **Qt Designer**. L'utente tramite l'interfaccia può decidere quanti clienti far entrare nel supermercato e cosa acquistare.

I clienti vengono generati in base al valore selezionato dalla spinbox; dopodiché, ogni cliente creerà la propria lista della spesa prima di entrare nel supermercato. Tutti i clienti entreranno contemporaneamente nel supermercato, rispettando il vincolo della traccia. Di seguito gli screenshot dell'applicazione:





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].
                   mutexCassa);
10         }
11         Customer *client = dequeue(&cassa[numCassa].customerInCassa);
12         pthread_mutex_unlock(&cassa[numCassa].mutexCassa);
13         // Simula il pagamento
14         if (client) {
15             printf(COLOR_MAGENTA "Il cliente %d e' in fase di pagamento alla cassa %d\n"
                    COLOR_RESET, client->id, numCassa);
16             sleep(cassa[numCassa].fixedTime + client->nProducts);
17             printf(COLOR_GREEN "Il cliente %d ha terminato il pagamento ed esce dal
                   supermercato.\n" COLOR_RESET, client->id);
18             // Libera la memoria e aggiorna il numero di clienti presenti nel
                   supermercato
19             free(client->products);
20             free(client);
21             // Aggiorna il numero di clienti presenti nel supermercato
22             pthread_mutex_lock(&mutexStore);
23             currentCustomers--;
24             pthread_mutex_unlock(&mutexStore);
25         }
26     }
27 }
```

La funzione `cassiere` rappresenta il comportamento di una cassa all'interno del supermercato, operando in un **ciclo infinito** per servire i clienti in base all'ordine di arrivo, secondo la politica FIFO (First In, First Out). Quando la funzione viene avviata, riceve come parametro un identificativo della cassa e stampa un messaggio per segnalare che è operativa. A questo punto, la cassa entra in una fase di attesa, durante la quale acquisisce il `mutexCassa` per proteggere l'accesso alla propria coda. Se la coda è vuota, il thread entra in attesa tramite la funzione `pthread_cond_wait`, bloccandosi fino all'arrivo di un cliente. Quando un cliente raggiunge la cassa, viene estratto dalla coda attraverso la funzione `dequeue` e il `mutexCassa` viene rilasciato, consentendo ad altri thread di accedere alle risorse condivise. Il pagamento viene quindi simulato con una pausa (`sleep()`) la cui durata dipende da due fattori: un tempo fisso specifico della cassa e un valore variabile legato al numero di prodotti acquistati dal cliente. Una volta completato il pagamento, viene stampato un messaggio di conferma e la memoria allocata per il cliente e i suoi prodotti viene liberata. Infine, il numero di clienti presenti nel supermercato viene aggiornato in modo sicuro utilizzando un `mutex` globale, garantendo la corretta gestione del flusso dei clienti. Questo approccio consente a ciascuna cassa di operare in maniera **parallela e indipendente**, assicurando un servizio efficiente e ordinato. Inoltre, grazie all'uso di **primitive di sincronizzazione** come `mutex` e `condition variable`, si evitano situazioni critiche come **race condition** e **deadlock**, garantendo il corretto funzionamento del sistema multi-threaded.

5.2 Thread "spesa"

```
void * spesa(void * arg) {
    Customer *client = (Customer *) arg;
    printf(COLOR_MAGENTA "Cliente %d fa acquisti per %d secondi.\n" COLOR_RESET, client
        ->id, client->time);
    sleep(client->time);
    if (client->nProducts > 0) {
        const int chosenCassa = rand() % NUM_CASSE;
        pthread_mutex_lock(&cassa[chosenCassa].mutexCassa);
        enqueue(&cassa[chosenCassa].customerInCassa, client);
        pthread_cond_signal(&cassa[chosenCassa].condCodaVuota);;
        pthread_mutex_unlock(&cassa[chosenCassa].mutexCassa);
        printf(COLOR_CYAN "Cliente %d si mette in fila alla cassa %d.\n" COLOR_RESET,
            client->id, chosenCassa);
    } else {
        printf("Cliente %d non ha acquistato nulla ed esce dal supermercato.\n", client
            ->id);
        free(client->products);
        free(client);
        pthread_mutex_lock(&mutexStore);
        currentCustomers--;
        pthread_mutex_unlock(&mutexStore);
    }
    pthread_exit(0);
}
```

La funzione `spesa` modella il comportamento di un cliente all'interno del supermercato, simulando il processo di acquisto e il successivo pagamento. Ogni cliente esegue questa funzione come un thread separato, garantendo un'esecuzione parallela dell'intera simulazione.

Il cliente inizia la sua esperienza di acquisto ricevendo come argomento un puntatore a una struttura `Customer`, che contiene tutte le informazioni necessarie, come il numero identificativo e la lista della spesa. Una volta entrato nel supermercato, viene stampato un messaggio per segnalare l'inizio della spesa, e il cliente attende un tempo prestabilito, simulando il tempo impiegato per selezionare i prodotti dagli scaffali.

Dopo aver completato gli acquisti, il cliente deve scegliere una cassa per il pagamento. Se ha acquistato almeno un prodotto, seleziona casualmente una delle casse disponibili utilizzando una funzione di generazione casuale. A questo punto, per garantire un accesso sicuro alla coda della cassa, viene acquisito il `mutexCassa`. Il cliente viene quindi inserito nella coda della cassa attraverso la funzione `enqueue()` e, subito dopo, viene inviato un segnale tramite `pthread_cond_signal` per notificare alla cassa l'arrivo di un nuovo cliente. Una volta completata questa operazione, il `mutexCassa` viene rilasciato e viene stampato un messaggio per indicare che il cliente è in attesa di essere servito.

Nel caso in cui il cliente non abbia acquistato nulla, viene stampato un messaggio che ne segnala l'uscita senza acquisti. A questo punto, la memoria allocata per il cliente e i suoi prodotti viene liberata,

e il numero totale di clienti nel supermercato viene aggiornato in modo sicuro tramite un `mutex` globale, garantendo il corretto tracciamento degli utenti presenti nel sistema.

Infine, la funzione si conclude con `pthread_exit(0)`, terminando il thread in modo sicuro e liberando le risorse associate. Grazie a questo approccio, il ciclo di vita del cliente viene gestito in modo efficace, assicurando una corretta sincronizzazione con le casse e un aggiornamento coerente dello stato del supermercato.

5.3 Thread "client"

```
void * client(void * arg) {
// Riceve il cliente e lo mette in coda nel supermercato
const int clientSocket = *(int *) arg;
Customer *customer = malloc(sizeof(Customer));
recv(clientSocket, customer, sizeof(Customer), 0);

pthread_mutex_lock(&mutexStore);
if (currentCustomers < MAX_CUSTOMERS) {
    enqueue(&customerInStore, customer);
    currentCustomers++;
    printf(COLOR_BLUE "Il cliente %d e' entrato nel supermercato.\n" COLOR_RESET,
        customer->id);
    pthread_t spesaThread;
    pthread_create(&spesaThread, NULL, spesa, (void *) customer);
    pthread_detach(spesaThread);
} else {
    enqueue(&customerInQueue, customer);
    printf(COLOR_RED"Il supermercato e' pieno, il cliente %d e' in attesa.\n"
        COLOR_RESET, customer->id);
}
pthread_mutex_unlock(&mutexStore);
pthread_exit(0);
}
```

La funzione `client` modella il comportamento di un cliente al momento del suo arrivo nel supermercato, determinando se può entrare immediatamente o se deve attendere in coda. Ogni cliente viene gestito come un thread separato, permettendo così un'elaborazione parallela delle richieste.

Il processo inizia con la ricezione del cliente, dove il thread accetta un socket come argomento, identificando la connessione con un cliente remoto. Per memorizzare le informazioni necessarie, viene allocata dinamicamente una struttura `Customer`. Successivamente, attraverso la funzione `recv()`, i dati del cliente vengono acquisiti dal socket e resi disponibili per la gestione.

Una volta ricevute le informazioni, il sistema verifica se il cliente può entrare immediatamente nel supermercato. Per fare ciò, si acquisisce il `mutexStore`, garantendo un accesso esclusivo alla variabile globale `currentCustomers`, che tiene traccia del numero di clienti attualmente presenti nel supermercato. Se il numero di clienti è inferiore alla soglia massima (`MAX_CUSTOMERS`), il cliente viene inserito nella coda dei clienti all'interno del supermercato tramite la funzione `enqueue()`, e il contatore dei clienti viene aggiornato di conseguenza. A questo punto, viene stampato un messaggio che conferma l'ingresso del cliente e viene avviato un nuovo thread, `spesaThread`, per simulare la fase di shopping (`pthread_create()`). Per ottimizzare la gestione delle risorse, il thread della spesa viene scollegato dal thread principale utilizzando `pthread_detach()`, permettendogli di concludersi autonomamente una volta terminato.

Se, invece, il supermercato ha già raggiunto la sua capacità massima, il cliente non può entrare immediatamente e viene inserito in una lista di attesa con la funzione `enqueue()`. Un messaggio informativo viene stampato per segnalare che il cliente è in attesa.

Infine, una volta completata la gestione del cliente, il `mutexStore` viene rilasciato per permettere ad altri thread di accedere alla risorsa in modo sicuro. La funzione termina con `pthread_exit(0)`, chiudendo il thread in modo sicuro ed evitando eventuali perdite di memoria.

Questo approccio garantisce un controllo efficiente e sincronizzato sull'ingresso dei clienti nel supermercato, evitando sovraffollamenti e gestendo correttamente la lista d'attesa quando il supermercato è pieno.

5.4 Thread "direttore"

```c
void *direttore(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutexStore);
        // Se il numero di clienti scende sotto la soglia, ne fa entrare altri E
        if (currentCustomers <= MAX_CUSTOMERS - E && !isEmpty(&customerInQueue)) {
            int clientiDaFarEntrare = E;
            while (clientiDaFarEntrare > 0 && !isEmpty(&customerInQueue)) {
                Customer *client = dequeue(&customerInQueue);
                if (client->nProducts == 0) {
                    printf("Il cliente %d non ha acquistato nulla ed esce immediatamente
                        .\n", client->id);
                    free(client);
                } else {
                    enqueue(&customerInStore, client);
                    currentCustomers++;
                    printf(COLOR_BLUE"Il cliente %d e' entrato nel supermercato.\n"
                        COLOR_RESET, client->id);
                    pthread_t clientThread;
                    pthread_create(&clientThread, NULL, spesa, client);
                    pthread_detach(clientThread);
                }
                clientiDaFarEntrare--;
            }
        }
        pthread_mutex_unlock(&mutexStore);
        sleep(1); // Controlla periodicamente
    }
}
```

La funzione `direttore` gestisce il controllo del numero di clienti all'interno del supermercato, regolando il loro ingresso per evitare sovraffollamenti. Il suo compito principale è monitorare costantemente la situazione e decidere quando consentire l'accesso ai clienti in attesa.

Il funzionamento della funzione è basato su un ciclo infinito, che esegue controlli periodici con una cadenza prestabilita (`sleep(1)`). In ogni iterazione, il direttore verifica la situazione attuale del supermercato per determinare se ci sia spazio sufficiente per far entrare nuovi clienti.

Per garantire un accesso sicuro alle strutture dati condivise, il direttore acquisisce il `mutexStore`. Se il numero di clienti presenti è inferiore a una soglia (`MAX_CUSTOMERS - E`) e vi sono clienti in attesa, il sistema determina quanti di essi possono entrare, gestendo gli ingressi a gruppi di `E` persone alla volta. A questo punto, il direttore esamina la lista di attesa e permette ai clienti di entrare uno alla volta, fino al raggiungimento della capacità disponibile.

Un'ulteriore gestione riguarda i clienti che, una volta entrati, non hanno effettuato alcun acquisto (`nProducts == 0`). In questo caso, viene stampato un messaggio che indica la loro uscita immediata e la memoria allocata per tali clienti viene liberata con `free()`.

Se invece il cliente ha effettivamente acquistato dei prodotti, viene inserito nella coda dei clienti presenti nel supermercato tramite `enqueue()`, e il contatore dei clienti viene aggiornato. Contestualmente, viene creato un thread separato per simulare la fase di shopping del cliente (`pthread_create()`) e, per una gestione più efficiente delle risorse, il thread viene scollegato dal principale attraverso `pthread_detach()`, permettendogli di terminare autonomamente.

Al termine del processo, il direttore rilascia il `mutexStore` per consentire ad altri thread di accedere alle risorse condivise. Infine, la funzione si mette in pausa per un secondo (`sleep(1)`) prima di ripetere il controllo, garantendo così un monitoraggio costante dell'afflusso di clienti.

Questo meccanismo consente di gestire in modo efficiente il flusso di persone all'interno del supermercato, evitando sovraffollamenti e assicurando una distribuzione equilibrata degli ingressi.

5.5 Thread "connection"

```c
void * connection(void * arg) {
    // Crea un socket , si mette in ascolto e crea un nuovo thread per ogni richiesta
    const char *ip = (char *) arg;
    //int clientSocket; // Descrittori dei socket
    struct sockaddr_in serverAddr, clientAddr; // Indirizzi del server e del client
    socklen_t addrSize = sizeof(struct sockaddr_in); // Dimensione dell'indirizzo
    const int serverSocket = socket(AF_INET, SOCK_STREAM, 0); // Creazione del socket
    if (serverSocket < 0) {
        perror("Errore nella creazione del socket");
        exit(1);
    }
    serverAddr.sin_family = AF_INET; // Dominio del socket
    serverAddr.sin_port = htons(PORT); // Porta del socket
    serverAddr.sin_addr.s_addr = inet_addr(ip); // Indirizzo del server
    if (bind(serverSocket, (struct sockaddr *) &serverAddr, sizeof(serverAddr)) < 0) {
        perror("Errore nel binding del socket");
        exit(1);
    }
    if (listen(serverSocket, MAX_CUSTOMERS) < 0) {
        perror("Errore nell'ascolto del socket");
        exit(1);
    }
    printf("Server in ascolto sulla porta %d...\n", PORT);
    while (1) {
        int clientSocket = accept(serverSocket, (struct sockaddr *) &clientAddr, &
            addrSize);
        printf("Nuova connessione accettata\n");
        if (clientSocket < 0) {
            perror("Errore nell'accettazione del client");
            exit(1);
        }
        pthread_t clientThread;
        pthread_create(&clientThread, NULL, client, (void *) &clientSocket);
        pthread_detach(clientThread);
        usleep(500);
    }
}
```

La funzione `connection` svolge un ruolo fondamentale nella gestione delle connessioni in ingresso al sistema del supermercato. Il suo compito principale è quello di avviare un server socket, metterlo in ascolto e accettare le richieste di connessione da parte dei clienti. Ogni volta che un nuovo cliente si connette, la funzione crea un thread dedicato per gestire la comunicazione con esso, consentendo un'interazione fluida ed efficiente senza bloccare il funzionamento generale del server.

Per comprendere meglio il funzionamento della funzione, possiamo suddividerlo in diverse fasi. Innanzitutto, il server crea un socket utilizzando la chiamata `socket(AF_INET, SOCK_STREAM, 0)`. Se questa operazione non va a buon fine, il programma interrompe l'esecuzione e restituisce un messaggio di errore.

Una volta creato il socket, viene configurato l'indirizzo del server. Questo avviene assegnando il protocollo IPv4 (`AF_INET`), impostando la porta con `htons(PORT)` e associando l'indirizzo IP corretto tramite `inet_addr(ip)`.

Successivamente, il socket viene legato all'indirizzo del server attraverso la funzione `bind()`. Anche in questo caso, se il processo fallisce, viene segnalato un errore e il programma viene terminato. Quando il binding ha successo, il server può finalmente mettersi in ascolto delle connessioni in entrata, utilizzando `listen()`, permettendo così a un numero massimo di `MAX_CUSTOMERS` client di connettersi simultaneamente. Se si verifica un errore in questa fase, il sistema stampa un messaggio di avviso e termina l'esecuzione.

A questo punto, il server entra in un ciclo infinito in cui attende e accetta le connessioni dei clienti tramite la funzione `accept()`. Ogni volta che un nuovo client si connette, viene generato un messaggio di conferma. Se, invece, l'accettazione della connessione fallisce, viene stampato un messaggio di errore e il server si arresta.

Per gestire le interazioni con i clienti in modo efficiente, il server crea un nuovo thread per ogni connessione accettata, sfruttando la funzione `pthread_create()`. Questo thread esegue la funzione `client()`, che si occupa della gestione specifica dell'utente. Inoltre, viene utilizzata la funzione `pthread_detach()` per scollegare il thread principale da quello del client, evitando così perdite di memoria. Infine, per

prevenire eventuali sovraccarichi del sistema, viene introdotta una breve pausa (`usleep(500)`) prima di accettare nuove connessioni.

Grazie a questa struttura, la funzione `connection` garantisce che il supermercato possa gestire in modo efficace più clienti contemporaneamente, evitando blocchi e rendendo l'esperienza di utilizzo del sistema fluida e reattiva.

6 Risultati e 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 che rispecchia situazioni reali di afflusso e gestione delle risorse.

Per valutare le prestazioni e l'affidabilità del sistema, sono stati effettuati diversi test mirati:

- **Test di carico:** Sono stati simulati scenari con un numero variabile di clienti, da un traffico ridotto fino a situazioni di sovraffollamento, per verificare la capacità del sistema di gestire molteplici connessioni simultanee.
- **Test di sincronizzazione:** È stata analizzata l'integrità dei dati condivisi tra i thread, verificando l'assenza di **race condition** e **deadlock**. L'uso di mutex e variabili di condizione ha garantito che i clienti venissero serviti correttamente senza conflitti di accesso alle risorse condivise.
- **Test di latenza:** È stato misurato il tempo medio di attesa dei clienti prima di essere serviti, in base al numero di casse disponibili e al carico di lavoro. I risultati mostrano che un numero adeguato di cassieri riduce significativamente i tempi di attesa, mentre in condizioni di sovraffollamento il sistema mantiene un comportamento prevedibile.

I risultati ottenuti confermano che il sistema è stabile ed efficiente nel gestire un supermercato simulato con più clienti e casse operanti in parallelo. L'uso del multi-threading ha permesso di ottenere un comportamento realistico e performante, dimostrando l'importanza delle tecniche di sincronizzazione per evitare problemi di concorrenza. Ulteriori miglioramenti potrebbero includere algoritmi avanzati per la gestione delle code, ottimizzazione della distribuzione del carico tra le casse e analisi predittiva dei flussi di clienti per un migliore bilanciamento del sistema.

References

[1] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 10th. John Wiley & Sons, 2021. ISBN: 978-1119800361.