Concurrency-Based Online Market Simulation

1. Introduction

This project implements a concurrency-based simulation of an online shopping system using C and POSIX threads (pthreads). The objective is to simulate customer interactions such as adding products to a cart and completing purchases, with concurrency control for shared resources like stock and cashier counters.

The program is structured to align with the requirements defined in the project PDF, including:

- Parsing configuration and customer request data from an input file.
- Managing concurrent execution of customer threads within defined groups.
- Handling critical sections for stock and payment systems.
- Logging every action consistently both to the terminal and a log file.

2. Input Parsing and Group-Based Segmentation

The simulation begins by parsing the `input.txt` file, which includes:

- System configuration (e.g., number of customers/products, timeout settings, initial stock).
- Customer requests, grouped by blank lines.

The input parser extracts configuration values and splits the requests into logical groups. Each group corresponds to a batch of customer threads that should start concurrently. The following values are extracted:

- `num_customers`
- `num_products`
- `reservation_timeout_ms`
- `max_concurrent_payments`
- `initial_stock`

Customer requests are represented by `ProductRequest` structs, and each group is stored in a `RequestGroup`.

3. Thread Execution Strategy and Group Management

In the implemented system, customer requests are organized and executed based on a group-based threading model. Each group represents a distinct batch of customer product requests, as defined by the blank line separation format in the input file. This approach ensures compliance with the assignment requirement that threads within each group must begin execution simultaneously, while separate groups may start at different times.

Upon parsing the input, requests are segmented into logical groups. For each group, threads corresponding to individual product requests are created in rapid succession, ensuring they begin execution concurrently. This simulates simultaneous activity within each group, reflecting real-world scenarios where multiple users may interact with the system at the same moment.

To ensure separation between group executions without enforcing strict sequential dependencies, a short delay (`usleep(150000)`) is introduced after the processing of each group. This delay guarantees that the next group of threads starts with a temporal offset, preserving the simulation's intended staggered structure while allowing overlap due to timeout behavior or waiting for shared resources (e.g., cashier slots).

This execution strategy is fully aligned with the project specification:

- Threads within each group are launched to run concurrently.
- Groups begin at different times to reflect temporal separation in user interactions.
- Inter-group independence is preserved; a group does not need to wait for previous groups to complete.

4. Function Design and Responsibilities

a. `parse_input()`

This function reads `input.txt`, extracts configuration parameters and customer requests, and stores them into globally defined structures. It separates requests into groups whenever it encounters an empty line.

b. `get_current_time_ms()`

Utility function that returns the current time in milliseconds, used for timestamping log messages.

c. `reserve_thread()`

Each product request starts with a reservation attempt. This function checks if enough stock is available, reserves it if possible, and logs the outcome.

d. `log_purchase_attempt()`

After a successful reservation, this function logs the customer's intent to purchase the product. It prints the action with a timestamp.

e. `do_actual_purchase()`

This function simulates the final purchase process. It uses `rand() % 2` to randomly decide if the customer proceeds to purchase. If so, the thread tries to enter a critical section protected by a mutex to access a limited number of cashier slots. If it succeeds within the timeout window, the purchase is logged. Otherwise, the reserved stock is returned.

This function implements timeout logic and retry behavior as specified in the assignment.

5. Concurrency Mechanisms

To ensure thread-safe operations:

- `stock_mutex` is used to protect stock modifications.
- `cashier_mutex` and `cashier_cond` are used to coordinate concurrent access to payment slots.
- Condition variables allow threads to wait for available cashier slots and to wake up in FIFO order.

Each thread either completes its purchase or releases its reserved stock when the timeout expires.

6. Logging System

A unified `LOG(...)` macro logs messages to both `stdout` and `log.txt`. This ensures consistent and readable output for debugging and grading.

All critical actions (cart addition, purchase attempt, actual purchase, timeout expiry) are timestamped using millisecond-level precision.

7. Output and Observations

Each run produces a different log due to the random nature of purchase decisions (`rand() % 2`). The system respects concurrency constraints, such as limited cashier slots and timeouts, and prints retry behavior as needed.

The use of timed delays and synchronization primitives ensures that all concurrency behaviors are observable and verifiable against the expected output structure.

8. Alignment with Assignment Requirements

This implementation meets all the requirements stated in the assignment PDF:

- Input parsing and request grouping follow the specified format.
- Threaded execution respects group-based timing constraints.
- Proper concurrency control ensures safe access to shared resources.
- Timeout and retry mechanisms are implemented for realistic behavior.
- Logs provide clear, timestamped insights into the system’s runtime state.

9. Conclusion and Reflections

This project reinforced the importance of thread synchronization and concurrency control in multi-threaded environments. Managing resource contention (stock, cashier slots) while maintaining real-time responsiveness was achieved using mutexes and condition variables.

The group-based execution model demonstrated a practical way to simulate wave-based concurrency in e-commerce systems. Future improvements could include:

- Better fairness in cashier allocation using semaphores or custom queues.
- More sophisticated logging (e.g., JSON or structured CSV).

Overall, the system is functionally complete, robust, and aligned with the academic specification.

9.Question Solving

1. Ensuring Maximum Two Concurrent Payments: Mutex and Condition Variable Strategy

In our implementation, we enforced the constraint that no more than two customers can make a payment at the same time using a combination of a mutex (`cashier_mutex`), a condition variable (`cashier_cond`), and a shared integer (`current_cashiers`). This was designed to simulate a maximum of two available cashier slots.

Whenever a customer thread wants to initiate a purchase, it first checks whether it can obtain one of the two available cashier slots. This check and acquisition are done within a `pthread_mutex_lock(&cashier_mutex)` block. If `current_cashiers` is less than `max_concurrent_payments` (which is set to 2 from the input file), the thread increments the counter, indicating that it has taken a cashier slot. If not, the thread waits on the condition variable using `pthread_cond_wait(&cashier_cond, &cashier_mutex)`, which releases the mutex and puts the thread to sleep until signaled.

After the purchase completes, the thread decrements the counter (`current_cashiers--`) and signals the condition variable (`pthread_cond_signal(&cashier_cond)`) to wake up another waiting thread.

This design ensures mutual exclusion while accessing `current_cashiers` and uses condition variables to manage access when the limit is reached.

Alternative Synchronization Mechanism

An alternative mechanism could have been to use a counting semaphore (e.g., POSIX `sem_t`) initialized with a count of 2. Each thread would call `sem_wait()` before proceeding to the payment and `sem_post()` after completing it. Semaphores offer a simpler interface for limiting concurrent access but lack the precise control and coordination offered by condition variables in more complex scenarios. In our case, condition variables provided finer-grained control and allowed us to log detailed information during each attempt and retry.

2. Reservation Timeout Strategy and Thread-Safe Stock Access

Reservation Timeout:
The timeout mechanism is implemented by recording the time a product is successfully reserved (`reservation_time`) and comparing it with the current time during the actual purchase attempt. If the elapsed time exceeds the `reservation_timeout_ms` threshold, the reservation is considered expired. This is simulated with a `usleep()` call for the full timeout duration in case the thread decides not to purchase.

Thread Safety:
To ensure thread safety during stock access (reservation and return), we wrapped every access or modification of the `stock[]` array within a `pthread_mutex_lock(&stock_mutex)` and `pthread_mutex_unlock(&stock_mutex)` block. This prevents race conditions where multiple threads might attempt to modify stock simultaneously, which would otherwise result in inconsistent states.

Conflict Handling:
When a thread attempts to reserve a product, it first checks whether the required quantity is available. If not, the reservation fails. If it succeeds but the thread does not complete the purchase in time, the stock is restored to its original state, again inside a mutex-protected block.

This logic ensures both integrity and consistency of shared data under concurrent operations.

3. Thread Termination Conditions and Resource Handling

Each product request is handled by a dedicated thread. These threads terminate under one of the following conditions:

- The reservation attempt fails due to insufficient stock.
- The purchase is not attempted due to a random `rand() % 2 == 0` outcome (simulating user behavior), and the timeout elapses.
- The purchase succeeds within the allowed timeout and cashier constraints.
- The purchase attempt fails to get a cashier slot within the timeout window.

Thread Exit Logic in Code:
Each thread has a clearly defined return point. After logging success or failure and making necessary changes to shared resources (like updating stock or decrementing `current_cashiers`), the thread returns `NULL` and terminates naturally.

We used the `pthread_join()` call on all threads in the main function to ensure that threads complete execution before moving on. This guarantees clean resource management and synchronization between stages (e.g., reservation, attempt, purchase).

Conclusion:
All synchronization mechanisms and logic are implemented in accordance with the assignment's requirements as detailed in the provided PDF:

- Concurrency is managed via per-request threads grouped logically.
- Timeouts are respected using timestamps and usleep.
- Mutexes and condition variables ensure safe, controlled concurrency.
- The simulation accurately reflects real-world constraints like limited cashier availability and user hesitation.

This implementation demonstrates practical concurrency control with POSIX threads and effective synchronization design.