A black text on a white background

Description automatically generated

**CSC2044 CONCURRENT PROGRAMMING**

**Assignment 2 Report**

**22022123 THIEN TZE YEA**

1. **How did you block the other customers from selecting the same seat(s)?**

To ensure that no more than one customer can select the same seat(s) concurrently, the system employs thread synchronization mechanisms.

* **Synchronization with Locks**: The core mechanism used to prevent two customers from selecting same seat(s) is the ReentrantLock, which is part of the java.util.concurrent.locks package. In the Theatre class, this lock is applied to the methods that modify the seat availability array. When a customer attempts to reserve seats, the selectSeats method acquires the lock before performing any operations on the shared seat availability array.

A computer code on a black background

Description automatically generated

Figure 1: Synchronization with Locks

* **Seat Availability Check**: Inside the selectSeats method, once the lock is acquired, the system checks whether each seat in the seatsToReserve array is available. This is done by iterating over the array and verifying that the seat index is within bounds and that the seat is not already marked as reserved in the seatAvailability array.

A screen shot of a computer screen

Description automatically generated

Figure 2: Seat Availability Check

* **Seat Reservation**: If all seats are confirmed available, the system updates the seatAvailability array to mark these seats as reserved. This operation also occurs within the lock's protected region, ensuring atomicity and consistency.
* **Lock Release**: After updating the seat availability, the lock is released, making it available for other threads. This ensures that other customer threads can proceed with their seat reservation attempts without interference.

By synchronizing access to the seatAvailability array with the ReentrantLock, the system prevents race conditions where multiple threads could otherwise end up reserving the same seat(s).

1. **How to interpret the results / prove that the system has fulfilled the requirement?**

To verify that the system correctly prevents multiple customers from selecting the same seat(s) and fulfils the reservation requirement, the following steps and observations are critical:

* **Output Examination**: The program generates output logs for each reservation attempt, including details about customer IDs, requested seats, and reservation success or failure. By analysing this output, one can confirm that reservations are processed correctly and that no two customers are given the same seat(s).

A screen shot of a computer program

Description automatically generated

Figure 3: Output Examination

As shown in Figure 3, the seat 13 in Theatre 3 was selected and reserved successfully by Customer 12. Therefore, Customer 77 will not be able to reserve seat 13.

* **Final Seat Availability**: After all customer threads have completed, the final seat availability status is printed for each theatre. This shows the actual state of each seat (reserved or available) and confirms that the seat status reflects the reservations made during the simulation.



Figure 4: Seat Availability

This visual representation allows verification that each seat has been accurately marked as reserved or available according to the operations performed.

**3. Discuss whether careful design is required to avoid deadlock.**

Careful design is critical to avoid deadlock in concurrent programming. Deadlock is a situation where two or more threads are unable to proceed because each is waiting for the other to release a resource. In the context of the cinema seat reservation system, although the current design effectively avoids deadlock, understanding and applying principles of careful design are essential, particularly if the system were to become more complex. Here’s how careful design is employed to avoid deadlock in the system:

* **Single Resource Locking**: In the system, each thread acquires only one lock (the ReentrantLock for the specific theatre) at a time. This approach eliminates the possibility of circular wait—a common cause of deadlock where threads wait for each other to release multiple locks. By ensuring that only one lock is involved in each critical section, the system avoids creating a situation where threads could be waiting on each other in a cycle.
* **ReentrantLock**: The ReentrantLock used in the Theatre class provides additional safety by allowing a thread to re-acquire the same lock if needed without causing self-deadlock. This feature is beneficial in scenarios where the same thread might need to lock the same resource multiple times, ensuring that it does not accidentally cause deadlock by re-entering the lock.
* **Lock Management**: Locks are managed with a clear acquisition and release pattern using try-finally blocks. This design ensures that locks are always released, even if an exception occurs during the execution of the critical section. This approach guarantees that the lock is properly freed, preventing scenarios where a lock could be held indefinitely due to an unexpected error.
* **Simple Synchronization**: The synchronization logic is kept simple by limiting the scope of the locked code to only the necessary operations. This reduces the potential for contention and deadlock by ensuring that only the essential operations are protected by the lock. Simplified synchronization helps to minimize the time any lock is held, reducing the likelihood of contention among threads.