Short Report: Enforcing Simulation Rules and Synchronization Mechanisms

Introduction:

The goal of this assignment was to develop a multi-threaded simulation of a club environment where patrons interact with various constraints, including entrance limits, waiting queues, and door access. This report outlines how we enforced simulation rules, the challenges faced, synchronization mechanisms implemented, and the lessons learned throughout the development process.

Enforced Simulation Rules:

1. **Maximum Patron Limit:** I ensured strict adherence to the predetermined patron limit within the club by verifying both the presence of an existing patron within the club and evaluating whether the established limit had been attained. In cases where both criteria were confirmed, permission for the patron's entry was granted; however, in instances where either condition was not satisfied, the patron's entry was temporarily deferred while simultaneously assessing the availability of vacant space within the club.

2. **Waiting and Leaving:** Patrons were required to wait if the club limit was reached or if the entrance was occupied. Additionally, when patrons decided to leave the club, they were appropriately marked as wanting to leave. This was achieved through careful synchronization of shared resources and appropriate notifications.

Synchronization Mechanism

1. **Locks and Notifications:** I used locks and notifications to handle waiting patrons. When the club was at capacity or the entrance was occupied, patrons waiting to enter were appropriately synchronized and notified when they could proceed.

2. **AtomicBoolean for Pause:** To implement the pause functionality, I utilized an `AtomicBoolean` flag shared between the main thread and patron threads. Patrons paused when the flag was set to true, and resumed when it was set to false, ensuring smooth synchronization.

Challenges Faced:

1. **Concurrency Control:** Coordinating multiple threads accessing shared resources required meticulous synchronization. Ensuring correct order of execution and preventing race conditions was challenging.

2. **Deadlock Prevention:** Preventing deadlock was a concern due to the complexity of interactions. Careful design of synchronization points and proper resource management was crucial to avoiding deadlocks.

Lessons Learned:

1. **Thread Safety:** Developing thread-safe applications demands a deep understanding of synchronization mechanisms. Careful planning and implementation are necessary to maintain data integrity and prevent conflicts.

2. **Liveness:** By using appropriate synchronization mechanisms and avoiding long-held locks, I ensured liveness – the property that threads continue to execute in a reasonable time frame.

3. **Debugging Multi-threaded Code:** Debugging was more intricate due to the non-deterministic nature of thread execution. Printing logs, monitoring thread interactions, and step-by-step debugging proved vital.

Conclusion:

In conclusion, this assignment provided valuable insights into developing multi-threaded simulations and managing synchronization challenges. I successfully enforced simulation rules, utilized appropriate synchronization mechanisms, and prioritized liveness and deadlock prevention. This experience underscores the importance of careful design and synchronization in multi-threaded applications, contributing to my understanding of concurrent programming paradigms.