Report

The goal of the ClubSimulation project was to imitate clubgoer behavior in a virtual club setting. A multi-threaded technique was used to enforce simulation rules, with each clubgoer represented as a different thread. Using classes like Clubgoer, ClubGrid, and PeopleCounter to encapsulate functionality, the design adhered to the principals of object-oriented programming.

Clubgoers' movement, conversations at the bar, dancing, and an orderly arrival and leave were the key rules that were enforced. Each clubgoer's position was tracked within the grid as part of the design strategy, which comprised modelling the club as a grid of blocks. Threads were utilized to replicate the simultaneous movement and behaviour of many clubgoers.

Challenges Faced

Thread Synchronization: Preventing race situations and ensuring thread safety presented a substantial problem. Classes like ClubGrid, Clubgoer, and PeopleCounter have synchronization features added to them to manage access to common resources and avoid inconsistent data.

Preventing deadlocks while retaining liveness posed additional difficulties. It was essential to carefully design synchronization blocks and make sure they were adequate in order for threads to move forward and avoid becoming stuck permanently.

Synchronization Mechanisms and Their Appropriateness

ClubGrid: To ensure that patrons may enter, move through, and exit the club securely, synchronization mechanisms were introduced to methods like enterClub, move, and leaveClub. Blocks were properly locked and unlocked to prevent many clubgoers from occupying the same block at once.

Clubgoer: To avoid disputes among clubgoers and make sure they interacted with their surroundings appropriately, methods like enterClub, move, depart, and actions like going to the bar and dancing were synced.

PeopleCounter methods for updating counters were synchronized to prevent concurrent modifications, ensuring accurate tracking of the number of people inside, waiting, and leaving the club.

Ensuring liveness and preventing deadlock.

Liveness: By releasing locks after operations and employing the proper synchronization granularity, liveness was guaranteed. This made it possible for threads to proceed even while there were others waiting.

Deadlock Avoidance: Locks were released in the reverse order of acquisition, nested locks were avoided, and deadlocks were avoided by maintaining a consistent ordering of locks. This stopped threads from waiting in circles and made sure they could get the locks they needed without constantly obstructing one another.

Lessons Learned

Careful Synchronization: In multi-threaded contexts, accurate synchronization is essential for preventing data inconsistencies and ensuring proper behaviour.

Granularity: It's crucial to pick the right level of granularity for synchronization. Too coarse-grained locking can cause contention and reduced parallelism, whereas too fine-grained locking can cause performance problems.

Testing: Thorough testing revealed probable synchronization and race situations. Validating the accuracy of the simulation required both unit testing and stress testing with a large number of threads.