Lutho Mngqibisa PCP Assignment 2 CSC2002S

Introduction

The "Club Simulation" assignment is a java program that emulates nightclub dynamics according to established rules. The simulation represents the club as a grid with an entrance and an exit door, a dance area and a bar. The task involves resolving concurrency issues, ensuring that the program complies with rules, avoiding safety and liveness problems and optionally implementing a "barman". The rules encompass patrons entering through the entrance one at a time ,maintaining a limit of patrons inside the club at once, ensuring realistic spacing and simultaneous block movement as well as preventing deadlocks.

Enforcement of Simulation Rules

Start button

I used a **CountdownLatch** to enforce this simulation rule. The latch was initialised to 1 within the main method of ClubSimulation.java. As a result, the latch's CountDown() function was used within the ActionPerformed() method of the start button. To ensure that the patron threads followed this synchronisation mechanism, I used latch.await() within the Clubgoer class's StartSim() method. This method ensured that the simulation began only after the start button has been clicked.

Pause button

I implemented pause button functionality through the utilization of the **wait()**,**notifyAll()** and **AtomicBoolean**. I introduced a pause AtomicBoolean variable within the Clubgoer class, initializing it to false when the patron threads were started. In the actionPerformed() method of the pause button, I set the pause variable to true and invoked wait() on the clubgoer threads. When the pause button was clicked again to resume, I utilized notifyAll() to awaken the waiting threads, ensuring the smooth continuation of the simulation. This synchronized mechanism guaranteed the proper pausing and resuming of the simulation.

Door access

To make sure that only one thread could use the entrance and exit doors at a time, I used synchronized blocks on the entrance object, specifically focusing on the **enterClub()** and **leaveClub()** methods in the ClubGrid class. However, I observed that simply synchronizing the enterClub() method could cause some threads to appear crowded at the entrance even when it was locked due to threads sleeping after entering the door. To address this, I introduced a wait() mechanism when the entrance was occupied and a notifyAll() mechanism when it became available again. This helped in preventing threads from overlapping and ensured a more orderly entry process. This approach improved the synchronization and made sure that the entrance and exit were used by one thread at a time without any overlap.

Club limit

I implemented the club's capacity limit by utilizing a **while loop** within the **enterClub()** method. I integrated a condition that assesses whether the number of threads inside the club equals the maximum allowable limit. When this condition is met, **wait()** is called on the entrance object. This ensured that threads won't be allowed to enter when the club has already reached its maximum capacity. Subsequently, waiting threads gain entry when an existing thread within the club calls the **leaveClub()** method. This method triggers a **notifyAll()** action on the entrance object, thereby waking up and permitting the waiting threads to enter the club. This approach effectively controls the number of patrons inside the club and prevents overcrowding beyond the established limit.

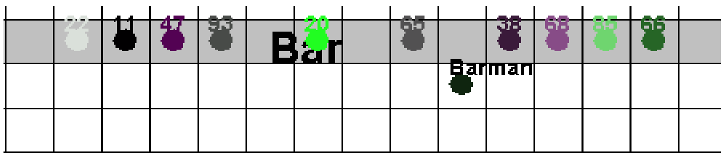
Realistic spacing

To maintain a realistic distance between patrons inside the club, I focused on synchronizing the movement and occupancy of grid blocks. I synchronized methods in the GridBlock class such as **get()** and **release()**. By synchronizing these methods, I ensured that only one patron could occupy a specific grid block at a time. This synchronization mechanism prevented multiple patrons from entering the same grid block simultaneously and maintained the one-patron-per-block spacing

Liveness

To ensure liveness, I was cautious of over synchronization and synchronized only critical sections in some of the methods to prevent nested locks and lock contention which would lead to non-simultaneous movement and potential deadlocks.

Andre the Barman



Design Approach

I wanted to create my barman thread such that it works with 3 mechanisms namely a First-In-First-Out (FIFO) queue, a wait() and notify() synchronization mechanism imitating signalling. The queue is a static variable in the barman class used such that patron objects when requesting a drink can add themselves to the queue and subsequently be put in waiting by calling a wait() on themselves. The barman can then continuously serve drinks to the patrons by constantly popping the head of the queue while it is not empty and accessing the patron's location in the bar area via their location.getX() method and then moving towards the patron by continuously changing the barman's x-value till it matches the patron's x-value. When the barman reaches the patron, it serves it and calls notifyAll() on the patron object to notify it that it has been served. The barman will keep serving patrons and clearing the queue until all the patrons have left the club.

Barman methods

- headTowardsPatron() : moves the barman towards a specified patron's location in the bar area using the move method in the Clubgrid class.
- headTowardsOriginalPos() : navigates the barman back to its original position at the centre of the bar area using the move method in the Clubgrid class.
- serveDrink (Clubgoer patron) : simulates the serving of a patron by calling the movement methods and using thread sleep mechanisms to simulate serving time. Once the barman reaches the patron's location , it calls notifyAll() to wake up the waiting patron as it has been served.
- requestDrink() : allows patrons to request a drink from the barman and wait for it to be served using synchronization mechanism wait(). (method in Clubgoer class)

Challenges encountered

- I encountered an illegalMonitorException when threads exited the club at first but I fixed that by making sure the notifyAll() in the leaveClub() method is inside a synchronized block.
- Despite synchronizing the entrance door access properly, the quick acquisition and release of the lock led to a visual impression of threads entering simultaneously.
- When I used nested synchronization in the enterClub() method , I encountered deadlocks when the club's capacity limit was reached , causing waiting threads to remain stuck despite available space.

Lessons Learned

- I learnt the importance of ensuring thread safety in a concurrent environment to prevent rare race-conditions and other synchronization issues.
- I learnt that the wait() and notify() methods work when used with shared or common objects to coordinate thread execution.
- I learnt that over-synchronizing could lead to cases where threads wait on other threads that are not releasing locks, ultimately causing deadlocks.