TMTBIN004
TINASHE TIMBA

CSC2002 PCP2
MULTITHREADED CONCURRENT CLUB SIMULATION

1. INTRODUCTION

The goal of this assignment is to fix the code by identifying and correcting all the concurrency issues so that all rules are complied with all the time and the simulation suffers from no safety or liveness problems.

2. HOW RULES WERE ENFORCED

• ***The Start button initiates the simulation.***

An atomic Boolean variable named 'start' is used in the Clubgoer class. Threads are instructed to wait until 'start' is set to true by the user. The start button in the ClubSimulation class ensures that 'start' atomic Boolean is set to true once the button is pressed and then the start calls notifyAll() to allow the simulation to be started. Any further presses of the start button have no effect since it is only checked once. The 'start' AtomicBoolean is synchronized for the wait and notifyAll() blocks.

• ***The Pause button pauses/resumes the simulation.***

To achieve this an AtomicBoolean 'pause' and an object 'pauser' were created. Using the get and set methods of AtomicBoolean , 'pause' is toggled every time the 'pause' button is pressed (switching between "resume" when true and "pause" when false).The wait() is called when 'pause' is set to true in the checkPause() method of ClubGoer and notifyAll() method is called by the synchronized pauser in the ClubSimulation class.

• ***The Quit button terminates the simulation (it does).***
No changes were made to the Quit button as it already worked as specified.

• ***Patrons enter through the entrance door and exit through the exit doors.The entrance and exit doors are accessed by one patron at a time. Inside the club, patrons maintain a realistic distance from each other (one per grid block). Patrons move block by block and simultaneously to ensure liveness.***

The grid blocks are shared among the patrons and hence concurrency issues are possible. Since only one thread at a time should "own" a GridBlock, all getters and setters of this class were synchronized appropriately following the Montor Pattern. The ClubGrid class has no further synchronization as it utilizes the GridBlock class to make the grid. Synchronizing only the GridBlock class methods provides enough safety and ensures that there is liveness and no potential deadlocks.

• The maximum number of patrons inside the club must not exceed a specified Limit and Patrons must wait if the club limit is reached or the entrance door is occupied.

A wait() block synchronized on the entrance was used in a while loop with a check of whether that Club was at capacity or whether the entrance gridblock was occupied:

```
while (counter.overCapacity()==true||entrance.occupied()){…}
```

Patrons waiting outside the club are only notified when a patron moves off the entrance block(notifyAll() block in the move method), and when a patron leaves the club(notifyAll() block in the leaveClub() method).

The PeopleCounter class is another shared resource hence it was made to follow the Monitor Pattern to ensure atomic updates of the counters.

• The simulation must be free from deadlocks.

Double locks were avoided throughout the code as best as possible. GridBlock class methods were synchronized so that any other instance of GridBlock such as in ClubGrid would not need to be synchronized. The PeopleCounter class methods were synchronized as well. Only specific methods like enterClub() and leaveClub() had some synchronization for the wait and notify blocks.

All other classes and methods did not need synchronization because their methods are thread-specific.

CONCLUSION

The simulation adheres to the specified rules. Liveness was ensured and deadlock was prevented by synchronizing only the necessary methods and carefully placing wait() and notify() blocks in crucial places where patrons could respond without missing signals or entering endless cycles of waiting.