**Melissa Githinji Due 13/08/23**

**CSC2002S Assignment 2**

*Multithreaded Concurrent Club Simulation*

**How The Rules Were Enforced**

As the common object, the rules of the simulation were enforced using the club grid. ClubGrid.class required the most synchronisation since it is what is accessed by all clubgoer, counter, and display threads. Synchronising on the main thread, ClubSimulation, causes liveness issues, does not OOP programming and provided issues with makefile circular dependencies.

1. The start button initiated the simulation by implementing a latch belonging to the ClubGrid. It acts as a gate for all club goer threads until the button is pushed and all can pass. The latch works here because once the simulation is started, the start button is evidently, not needed again, thus the ‘once only’ nature of the latch fits the problem.
2. The pause button is synchronised once again on the shared object club grid by using an atomic boolean called pushPause to act as the flag which clubgoers are synchronised on and to notify the clubgoer threads to unpause when the button is pressed again. I made use of a notifyAll since there are more than one threads. The main thread in ClubSimulation instantiates pushPause to false, which then changes value accordingly within the action performed.

In order to ensure that the waiting count still increases when the grid is paused, I created a method to access the overcapacity() method in PeopleCounter. If the club is over capacity, then the waiting threads will still continue to arrive at the club, allowing the counter to go up.

1. To control max capacity and the entrance door I added a synchronised block within the enterClub() method on the entrance GridBlock object. Updating the count of patrons waiting lies outside of the block to ensure that number still goes up. Within the block the current thread waits until the number of people in the club is less than the maximum. The people counter class already implements atomic integers so this comparison does not result in a race condition since they are protected.
2. To ensure that the exit door are accessed by one patron at a time, I synchronised the leaveClub() method in the ClubGrid class to ensure only one clubgoer thread uses the method at a time. This couldn’t be done to the enterClub() method because as in 3), the waiting count needs to be updated in this method by all threads. The lock on the entrance object locks the critical section of code which ensures that only one patron is admitted into the club at a time.
3. While at any time only one patron ‘owns’ a grid block, the need for synchronisation comes in the case that 2 threads want to move to the same grid block since they will both access the isOccupied value, and both will try to change it and move there. This variable needs to be protected, therefore I made isOccupied atomic. The entrance is the only block which more than one patron can access in the event that threads linger there unpredictably.
4. My implementation continued to allow the synchronised movement from the original program.

Challenges faced

At first it was difficult figuring out what to synchronise and I encountered a lot of illegal monitor state exceptions because I was trying to make the clubgoers wait on the monitor of club simulation for the pause and start button. This threw the exception because the clubgoers do not own the simulation class. By truly understanding the reason for the exception, it was easy to pick out what I should spend my time synchronising because there are only so many classes linking club goes and only one common one that they own (the club grid). This was also another issue with synchronising on the main thread.

Synchronising on the ClubView for the pause button caused deadlock because the atomic boolean in that class would make the simulation pause, but would kill interaction with the JPanel (since repaint would stop), despited the clubgoer threads carrying on. I could not click the pause button or quit the application. Therefore the resulting missed signal caused deadlock.