CSC2002S 2023 ASSIGNMENT REPORT

MULTITHREADED CONCURRENT CLUB SIMULATION

BHMQAI001

Enforcing Simulation Rules

ClubSimulation Class

The start button initiates the simulation as required, and it is set to false once pressed so that it cannot be pressed again once the program has started running. In this same piece of code, I moved the code that starts the threads and the thread counter from the main method of the same class. This was done so that the start button initialises the threads.

For pausing the simulation, I initialised an AtomicBoolean variable pause and set it to false. In the actionPerformed(ActionEvent e), the code allows for toggling between states of pausing and running the simulation. If the value returned by pause.get() is currently true, the code sets the state to false, indicating that the simulation should resume running. If the value returned is false, the state is set to true and the simulation pauses.

Quit was already coded to terminate the simulation.

Clubgoer Class

In this class, the method startSim() is left empty as the start button initiates the simulation. Even though it is called in the run method, it is not necessary as the code needed to start the threads and the simulation is already present.

There is an AtomicBoolean pause declared in this class. The method checkPause controls the behaviour of threads when the pause condition returned true. The while loop will keep running as long as the pause state remains true (simulation paused). When the pause state becomes false (simulation resumed) , the pause condition will be lifted, and the threads will continue running.

ClubGrid Class

Inside the method Gridblock enterClub, the Gridblock move and the GridBlock leaveClub, the object entrance is synchronized so that only one thread can access it at a time.

In the method Gridblock enterClub, there is a while loop that checks whether the club is at capacity. If it is, the threads wanting to come in must wait(). They will only proceed to the entrance when the condition doesn't hold (when there is space inside). In the GridBlock leaveClub method, threads are notified when a patron leaves a club and space becomes available through notifyAll().

The Gridblock move code is similar, but it ensures that all blocks can only be accessed by one patron at a time. The synchronized block manages the process of a patron leaving a block and notifies waiting threads about available space if a patron is leaving the entrance block. It ensures coordinated and clear communication about the shared objects between the threads. Therefore, only one patron can access the entrance and exit doors at a time.

PeopleLocation Class

This class employs the Java Monitor Pattern, where all the methods are synchronized and only one thread can access certain methods at a time. This ensures that the patrons remain the given distance away from each other (one per grid block).

Challenges Faced and Lessons Learnt

I faced great challenge with trying to ensure that patrons inside should not exceed the limit. I attempted synchronizing methods, but eventually decided to synchronize blocks in order to ensure simultaneous change between patrons leaving and entering the club. Once there is space inside, immediately a thread that has been waiting outside enters.

Synchronization Mechanisms and their Appropriateness

I employed the Java Monitor Pattern in the PeopleLocation class, so that all the methods are synchronized and only one thread can access certain methods at a time. This prevents race conditions and data corruption when multiple threads are accessing shared resources.

I also utilised synchronized blocks in the ClubGrid class, as the compiler gives instructions to get the lock on the specified object before executing the code and releases it afterwards. In this way, the thread 'owns' the lock while it is using it, and thus the object protects itself.

Methods to Ensure Liveness and Prevent Deadlock

There is no use of any thread priorities to avoid liveness problems. An AtomicBoolean is used instead of Volatile, for easier implementation and to make the class thread safe. The atomic variable is a singular variable that defines the class state, and there are no compound actions accessing the state.

There is implemented of fine-grained lock granularity (locked blocks instead of whole methods) to avoid liveness problems. This synchronized access to shared resources prevents threads from getting potentially blocked due to other threads wanting to access the same resource.

The use of wait() and notifyAll() prevents threads from being stuck indefinitely.

I avoided any nested locking, which commonly causes deadlocks. I also utilised bounded loops with a condition to wait for a certain condition to be met before proceeding. This ensures that threads don't wait indefinitely if the condition isn't met.