Dylan Friedman
FRDDYL002
27 August 2023
CSC2002S

MultiThreaded Concurrent Club Simulation

Enforcing Simulation Rules:

The simulation follows a basic premise: Clubgoers (goers) wait to enter the club, once the door opens for the club (on start) the goers are allowed to enter and then are allowed to leave. There are 3 rules in regards to entering the club.
1. The goers must enter one at a time and exit one at a time.
2. There must not be an overcapacity of the club.
3. There cannot be more than 1 goer in a grid block at a time.

These rules are ensured using wait() and notify() to lock the grid block so that other threads cant access ones with locks as well as having the leaveClub gridblock notify entrance so that a new club goer can
The synchronisation and deadlock prevention is based on these 3 simple rules…

Challenges Faced & Lessons learned:

The main challenges faced was the implementation of the CountDownLatch as well as the locking on specific blocks. The notes provided in the lectures gave us exactly what we need to get the start and pause buttons to work. I learned a lot about Locking and how an object must be locked when it has a [1]shared mutable state. I learned that when you have another class doing a more complex action on an instance of a shared class then you must lock that as well. Whenever using a thread safe class with a compound action then you lock the compound action on the object you are modifying. I.e. Need to protect each grid block from concurrent access - anytime, anywhere you access a grid block = have to lock on that grid block, not on some other class.

[1]Shared Mutable State: Object is accessed by multiple threads and the value can change.

The Synchronisation Mechanisms:

Note: Whenever synchronising on a complex action that is being performed on a class - That class has to be a [2]thread safe class
The basic synchronisation patterns implemented are: 1) Mutual exclusion; 2) Latches; 3) Signalling

1) The grid blocks in the simulation are mutually exclusive i.e. 2 threads cannot access the same grid block at the same time.
2) The start button made use of a latch to wait for the start button to be pressed, then the threads can enter the club
3) The pause button made use of signalling by using an AtomicBoolean, when the pause button is clicked then the AtomicBoolean is set to true and all threads begin to wait until the button is clicked again which sets the AtomicBoolean to false.

[2]Thread safe class is one which the Java Monitor Pattern has been performed on

Ensured Liveness:

To ensure that every thread created must eventually get the lock they want as well as unlock the lock, the conditions regarding the locks were very specifically chosen. In the ClubGrid class, the Clubgoer threads want to enter the club - the conditions specifically chosen was that once the thread is alive it waits for the entrance block to be available as well as it waits on the club to not be at overcapacity. The substate of the thread waiting is implemented in the “enterClub” method (the thread is **holding the lock for the GridBlock object**) then the runnable substate is implemented in the “move” and “leaveClub" methods that notify the thread that it can enter the club (wakes up all the threads blocked under the specifically chosen conditions)

Prevented Deadlocks:

Deadlocks occur when 2 or more processors never stop waiting i.e. 2 or more threads reaching for 2 or more locks. This was avoided by only locking on once in a method and not synchronising two different methods calling on one another in the sense that there is no [3].deadly embrace. There was a possible deadlock: Clubbers trying to get to each other's locks (the lock is on grid block) but all the grid blocks around that thread are taken was avoided: If one thread cant get to a block then it must try somewhere else to solve this problem.

[3]. Deadly embrace: 2 or more threads not wanting to give up their locks.