CSC2002S ASSIGNMENT PCP2 2023
PARALLEL PROGRAMMING WITH JAVA:
Multithreaded Concurrent Club Simulation

Murray Inglis

Student ID: INGMUR002

The University of Cape Town

August 2023

1 Introduction

This report covers the simulation of a club environment using methods of parallelization and concurrency.

2 Methods

2.1 Parallelization

The program creates threads for each individual customer, a thread for the barman, a thread for the counters and a thread for the interface. There are shared resources among these threads so care needs to be taken so that the program is safe but also doesn't suffer from liveness issues.

2.2 Concurrency

The *GridBlock* class is a shared resource between many of the classes. The *ClubGrid* class contains a 2D array of *GridBlock* objects. Instead of synchronizing the *ClubGrid* class, each *GridBlock* is synchronized to improve speed. It was therefore necessary to synchronize every method in the *GridBlock* class, especially the getters and setters. If this wasn't done, there would be issues of customers occupying the same block.
The *PeopleCounter* class is also a shared resource. The methods of this class need to be synchronized so that threads can't simultaneously update the counter or fetch the counter. This could cause issues like letting 2 people into the club at the same time.
The program does not encounter deadlocks as there is never a time when a thread tries to access a lock while it already has another lock.

2.3 Entrance and Exit

When the customer threads are in the club, they cannot occupy the same block when moving around. This ensures that only one customer can be on the exit block at a time and ensures that the customers exit the club one at a time.
The entrance mechanism is set up using the *wait/notify* methods synchronised on the entrance block. When a customer tries to enter the club, if the counter is full or the entrance block is occupied, *wait* is called on the entrance block. When a customer moves off the entrance block and there is space in the club, or if a customer leaves the club; *notifyAll* is called on the entrance block and a customer is allowed to enter. This ensures that customers enter one at a time, do not occupy the entrance block at the same time and also obey the

club capacity limit. The *waiting* counter will increment and decrement as customers wait and enter. These mechanisms work on the *enterClub*, *move* and *leaveClub* methods in the *ClubGrid* class.

2.4 Andre

Andre the barman is implemented in almost exactly the same way as a customer thread, except his movement is constrained to up and down the bar.
When a customer is thirsty and reaches the bar, synchronized on the *GridBlock* they are on, *wait* is called on that *GridBlock*. When Andre reaches that block, he calls *notifyAll* and his thread sleeps for a second to simulate serving the drink. The customer is now free to leave and Andre moves on to the next block.
An issue is when customers pile up and cannot move after being served a drink, since they cannot occupy the same block and essentially become trapped at the bar. However, solving this is beyond the scope of the assignment.

2.5 Buttons

The start button starts the program and after that does nothing when the user presses it. All the threads have a *AtomicBoolean* for a paused state that is set to true at the start of the program. Synchronized on this *AtomicBoolean*, *wait* is called to pause all the threads at the start of their *run* method. When the start button is pressed, the paused state is set to false and *notifyAll* is called, waking up all the threads. After this the start button is disabled.
The pause and unpause button work in the exact same way. They set the paused state to true or false and call *wait* or *notifyAll*.
The quit button terminates the simulation.