University of Cape Town
Department of Computer Science
CSC2002S-Report

Kananelo Chabeli

August 2023

Introduction

In recent years, concurrent programming has gained popularity in software development because of invention of multi-core processors.Different from parallel programming which is about running multiple tasks at the same time, concurrent programming describes principles of **effectively** managing and controlling access to shared resources. Typical application of concurrent programming is design and development of operating systems. An operating systems controls all programs or processes running on a given computer and effectively allocates resources to different processes.

A common challenge in concurrent programming is dealing with race conditions. There are two forms of race conditions namely; data race and interleaving. **Data race** is a race condition that occurs when different threads access same memory location, at least one of the threads writes to the memory with no synchronization that forces any particular order. It is important for software developers to have deep understanding of multi-threading and measures to prevent race conditions.

More often than not, prevention of race conditions is made by forcing ordering in the occurrence of events. This is called mutual exclusion, and process of enforcing this ordering is called synchronization. In Java, there a number of ways to achieve synchronization.The main purpose of this report is to document approach and results of using synchronization mechanisms in Java.

Objectives

The objective of this assignment is to utilize synchronization mechanisms in Java to ensure that the simulation of patrons in a club adhere to synchronization constraints and maintain thread safety and liveliness.

Problem Statement

This assignment entails a Java simulation of a club, ensuring that is adheres to defined software specifications. In this simulation, a Club is represented by a grid of specified size containing entrance door(dark gray in fig. 1), exit door (pink in fig.1), dance floor (yellow region in fig 1) and a bar(light gray). Patrons enter the club through the entrance doors, and exit through the exit door, one at the time. Within the Club, patrons cam dance, got to bar (to get drinks) or to wander about. Counters should keep track of the number of patrons inside the club, those waiting, maximum number of patrons as well as number of patrons that have left the club.

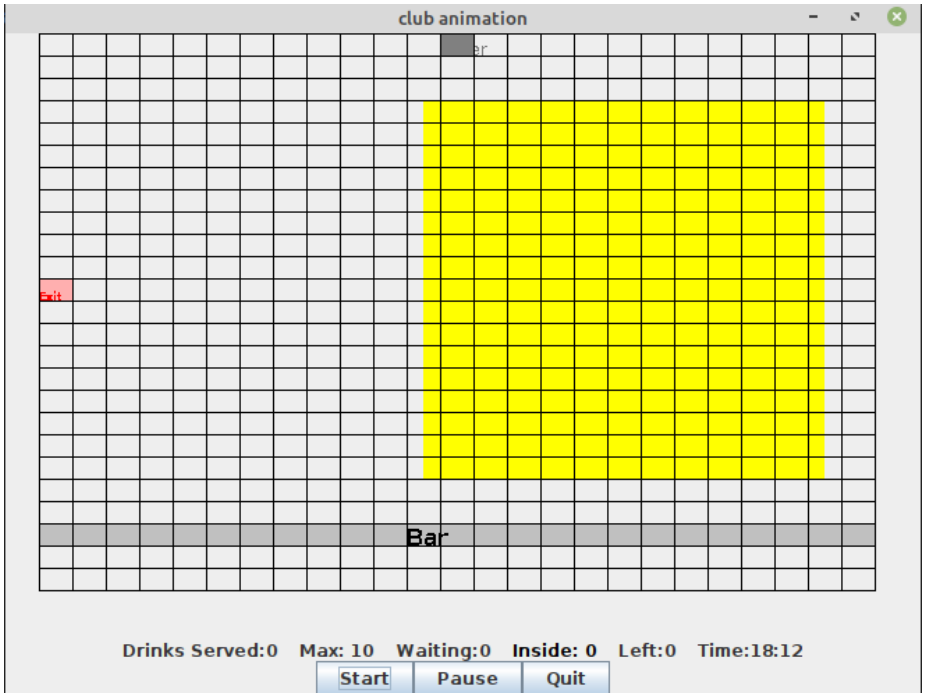

Figure 1: Screen shot illustration the simulated club

Problem Requirements

The following are the requirements of the club simulation:

- The Start button initiates the simulation.
- The Pause button pauses/resumes the simulation.

- The Quit button terminates the simulation.
- 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.
- The maximum number of patrons inside the club must not exceed a specified limit.
- Patrons must wait if the club limit is reached or the entrance door is occupied.
- Inside the club, patrons maintain a realistic distance from each other (one per grid block).
- Patrons move block by block and simultaneously to ensure liveliness.
- The simulation must be free from deadlocks.

1 Method

In this section, detailed description of how the problem was solved in presented. The approach deployed was to first identify all objects and variables which are shared by threads. There are many objects which are shared by different threads in this problem, few include, the grid blocks, the Club itself.

Start button: to meet the requirement that the start button should start the simulation, a CountDownLatch was used. This was found to be suitable synchronization tool because the was that latches work is that, they make threads wait until some event has happened. This is analogous to a situation where people must wait outside a lecture room until a lecturer is there to open the door. CountDownLatch was used, and all threads were forced to wait on this object. Clicking the button invokes countDown(), method of the count Down which releases the threads into the simulation. This is just the basic idea of of how start button was programmed.

There is an added feature is the solution of this assignment. The Barman, called Andre has been added to the simulation to provide drinks to patrons as they enjoy themselves. Because of this feature, pressing start button does not release all threads into the club, but the Barman is first let in and only after reaching the bar can threads be allowed in. This feature was also implemented using CountDownLatch synchronization mechanism.

Pause button-This behaviour of this button enforces spin-wait synchronization mechanism. Threads should repeatedly check if the button is paused in a while loop. This is actually a poor synchronization style because thread keep looping and wasting resources. A way that this was "prevented" was making thread check the while loop's condition after repeated interval.To achieve

this, threads are slept for some time between each successive check. when in sleep mode, they are kicked out the CPU,and allow other threads to execute, thus minimizing effect of resource wastage.

Quit button-meeting this requirement was fairly easy, because when pressed, the program executes System.exit(0), exiting the program.

For the rest o requirements, different synchronization mechanisms were used. For instance, for make sure that one thread access entrance or exit door at the time, the method that makes thread enter or leave the club were synchronized. To ensure that one thread occupies one grid block at the time, the class GridBlock was made to follow monitor pattern. This doesn't necessarily guarantees complete synchronization, but with additional synchronization of blocks of statements where objects of GridBlock are used, the requirement was fulfilled.

Moreover, to make threads wait when club limit has been reached, a different synchronization mechanism was used here. This is a conditional variable synchronization. Because every object in just is a conditional variable, all threads were forced to wait of Club object when the club is full, and a thread that leaves the club wakes then up, by invoking notify All().

Additional Features

As explained earlier, a thread labelled "BarMan", was added into the simulation and is responsible for serving other threads drinks. When a patron is thirty, it heads to the Bar counter (a grid block before the blocks colored light gray), the bar man then moves there and serves the drinks. No patron is allowed to be on the counter (otherwise they would still Bar man's money). A patron **waits** until served. This has potential of resulting in deadlock, where a huge number of thread will come and accumulate at the bar, seeking drinks. Because grid blocks are synchronized, a thread can not step on others to get the grid. Figure 2 below shows a scenario where deadlock could occur.

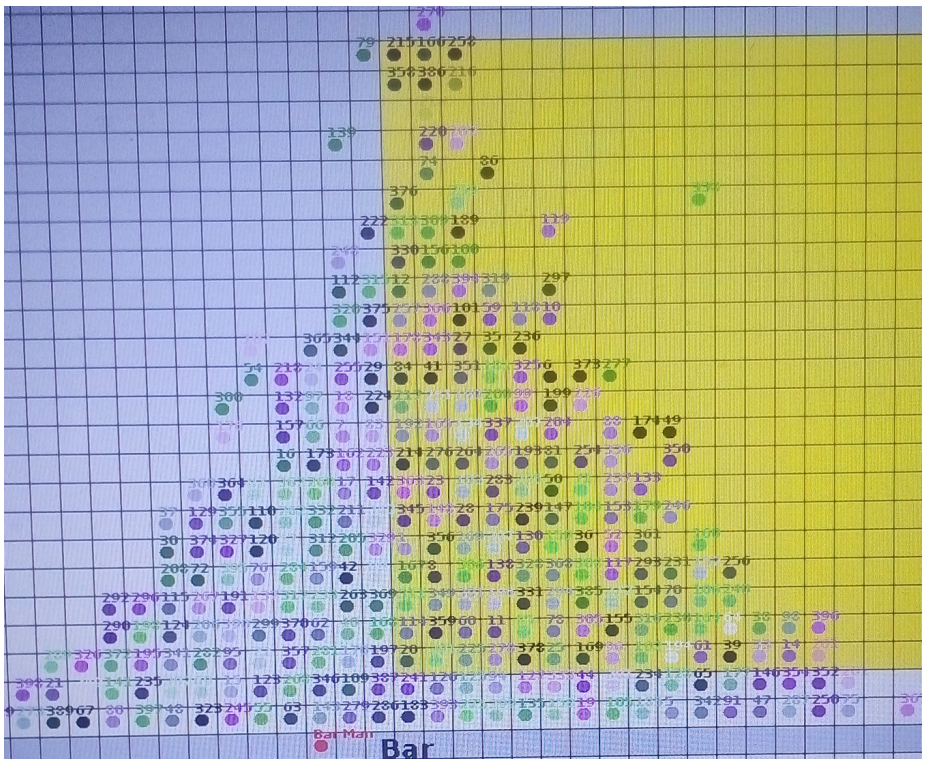

Figure 2: Screen sot showing possible deadlock that can result in the simulation

As shown in figure 2 above, threads have locked, and the simulation in said to be thread dead locked. After a thorough investigation on this problem, it was discovered that, the dead lock was influenced by how fast was the bar man serving drinks. If the bar man is slower than the number of drink requests, patron would cluster as shown blocking served patrons from moving and creating space for patrons not yet served.(A patron is only served when at the counter,

meaning when a grid closest to the bar grid blocks).

To solve this problem, the serving speed of Bar man was make to be proportional to the number of drink requests. This was implemented by making each thread notify the bar man when thirsty, and then depending on the number of requests made thus far, the bar man would adjust his speed accordingly. To symbolise "barman serving drinks", the barman is made to wait or sleep for few microseconds adjacent to the thread being served. This is shown is figure 3 below. Lastly, patrons are served drinks on the basis of first-come-first-serve.

It was thought that, for some reason, the Barman might wan to keep track of how many drink he has served so far, and decided if he can continue buying more stock, or what so ever. For this reason, another additional feature added to the simulation is the display of the number of drinks served.Lastly, the system time is again made to be displayed on the club, perhaps for patrons to see the time and decide if its late(behaviour isn't programmed though).

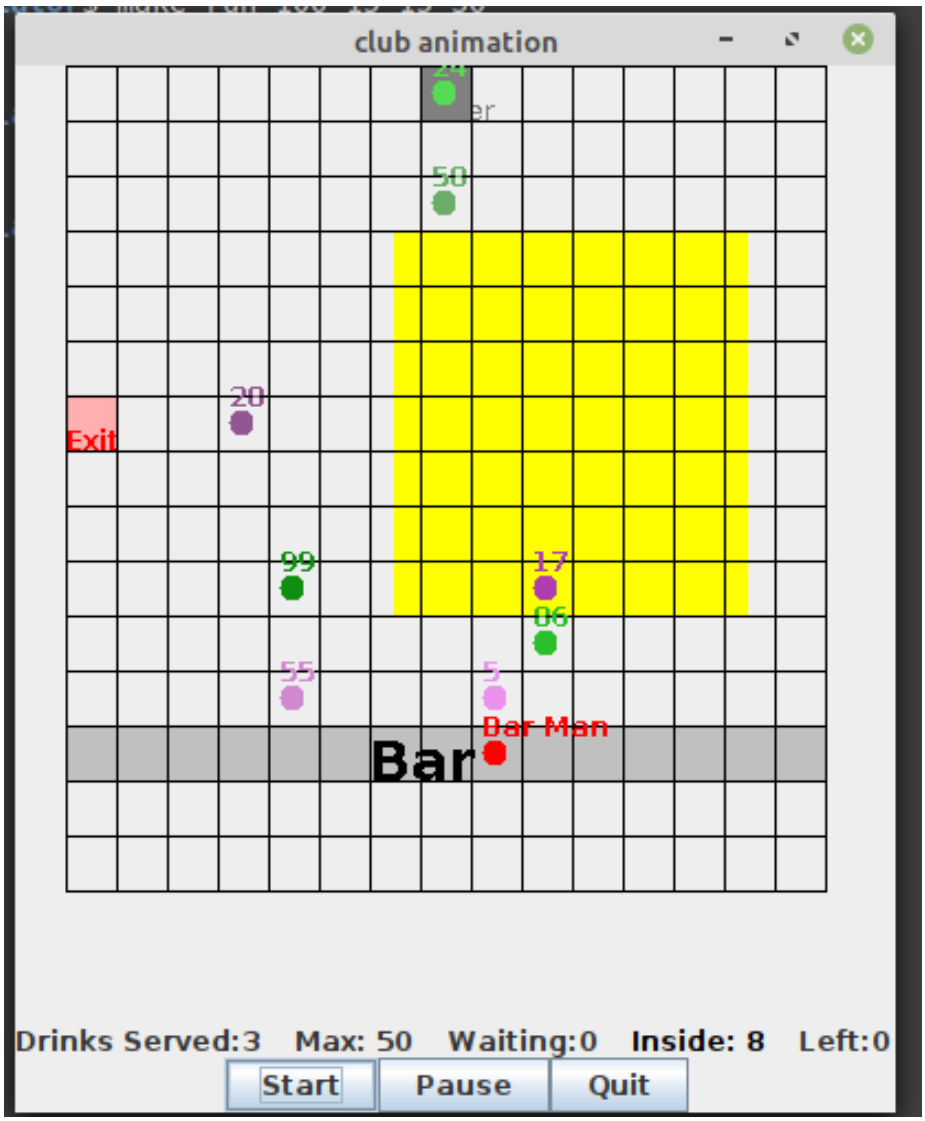

Figure 3: Andre serving drink