CSC2002S 2023 ASSIGNMENT

VLNSHA004

MULTITHREADED CONCURRENT CLUB SIMULATION

CLUB RULES (VERSION 1)

Report

Enforcing simulation rules:

1) Start button:

   The start button in the program was an issue since the program was only meant to start once the start button is clicked. Threads are to wait at the beginning of the program until the start button is clicked, this was not the case as threads would start to enter the club as soon as the program started running.

   How this was modified – the program uses a countDownLatch with parameter (1) to ensure that threads are held back until the start button is clicked. On click of the start button the latch would be released using the latch.countDown() method called in the event of the start button being clicked. The threads would only then start once this event has occurred.

2) Pause button:
   The issue found with the pause button in the given program was that when the button was clicked, all clubgoers were meant to stop any further actions from being performed. The given program did not include a pause button that was functional in this way.

   To fix this problem, the program uses an Atomic Boolean flag (paused). The flag value is changed when the pause button is clicked, if the program is paused, i.e. the paused variable is set to true, all threads wait until the button is clicked again. Once the button is clicked another time, notifyAll() is called in the event for the button and the paused flag is set to false for threads to resume activities. The threads Andre and clubGoers have a method to check the flag before they take each step. Additionally, clubGoers waiting outside do not pause, only people in the club stop activities when the pause button is clicked. This was done by first checking if the thread is in the room before checking if the pause variable is set to true in the checkPause() method. This allowed the club to pause and resume when the pause button was clicked.

3) The grid blocks:
   Grid blocks are a shared resource between clubgoers, thus we identify a need for synchronization. To ensure that clubgoers never step in the same block, the move

method is synchronized and places a lock on the new GridBlock object. The lock only releases when the block that the entity is on is released. Thereafter the clubgoer can safely move to the new block as no other entity can take the lock on the new block while another entity is entering that block.

4) The entrance and exit doors:
The patrons at the club should only enter and exit the club one at a time, meaning, the block at the entrance and the block at the exit could only be accessed one at a time. Since gridblocks have been synchronized, the exit block cannot be accessed by multiple threads. This ensures only one thread can leave at a time. The entrance block is synchronized in the same way, additionally while an entity is on the entrance block, the threads waiting outside are set to wait until the block is empty. This ensures only one thread at a time can enter the club.

5) Ensuring threads wait at the door while club is at max capacity:

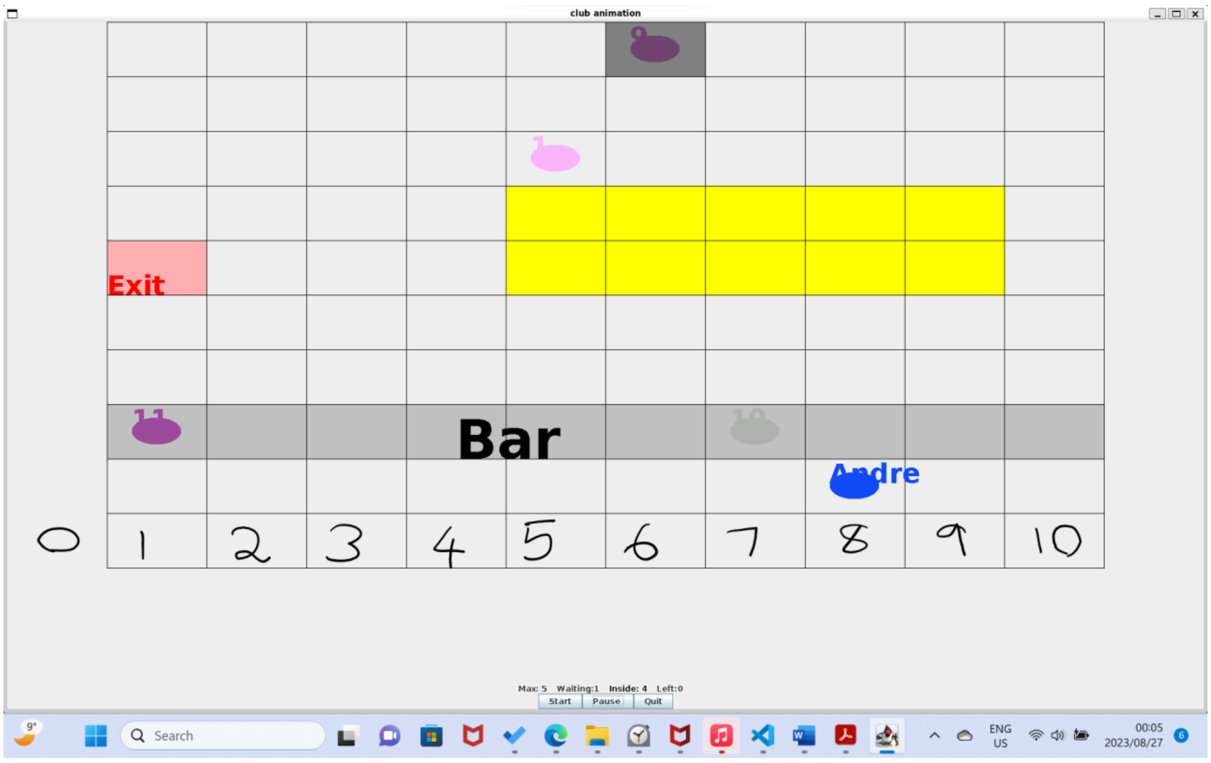
The enterClub() method in clubGrid synchronizes on the entrance object – the entrance being a gridblock. The method checks that there is no one on the entrance block and that the club is not at max capacity, if both conditions are false, only then can the clubgoer enter. This ensures that threads wait while the club is at max capacity.

Andre the barman:

The way Andre works is that he patrols up and down the row below the bar, checking each block adjacent to him. This uses a while loop that calls the patrol() method on each iteration. The workings of the customer in getDrink() is that when the customer is on a block that is the bar and they are thirsty(a Boolean flag), a lock is placed on the block and if they are thirsty they are set to wait().
Andre then checks whether the block is occupied – i.e. somebody is waiting to be served, he sleeps for a while (preparing the drink) and then calls .notify() on the block where the customer is on. This allows the customer to leave after they've been served.

Issues and challenges:

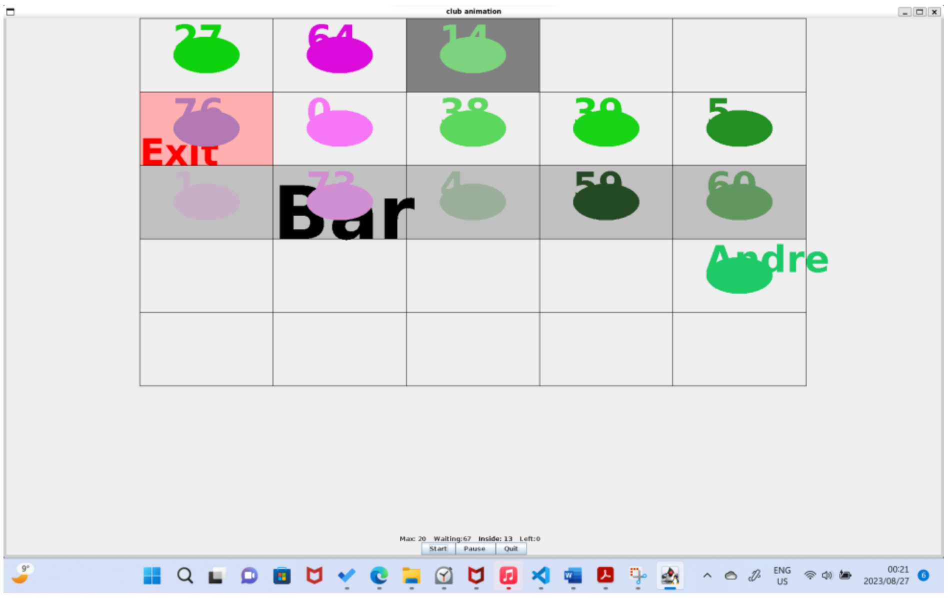
Challenges in implementation arrived at the start of implementing Andre the barman. Andre's coordinate system had issues where, so outside the grid on the display was index 0 for Andre and the last block on the right inside the grid was index 10 for a grid that was 10 by 10 squares.



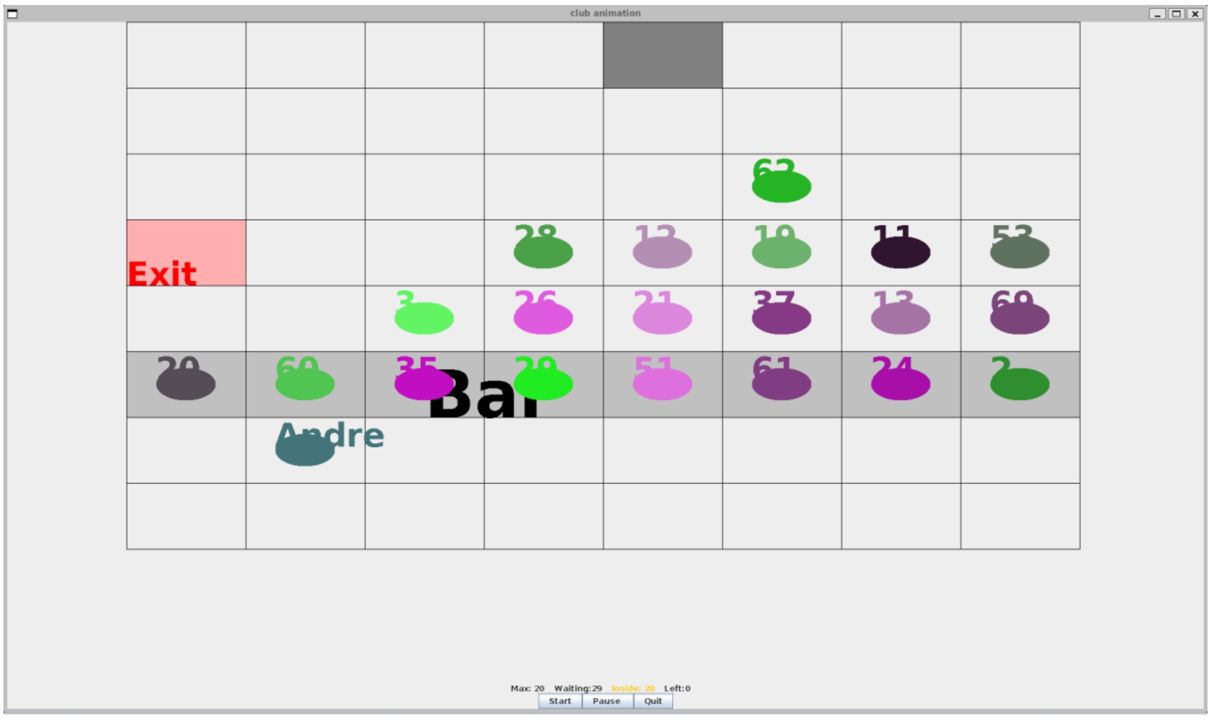
This error caused another error - index out of range when Andre attempted to move to the furthest block right on the grid. The array used for the gridBlocks was lengthened by one for x-values to ensure that Andre does not get an index out of range error. The amount of gridBlocks initialized was also increased by another column. Andre moves correctly in the grid as while he patrols he checks if the block he is moving to is 1 on his coordinate system (which is the furthest block to the left in the grid). He checks if the x-coordinate of the next block he is moving to is the same value as ClubGrid.getMaxX().

Other challenges include all the clubgoers swarming the bar and creating a situation wherein no clubgoer can leave the bar since all the blocks around them are occupied. Additionally, when the grid size was made to be small for a large number of clubgoers with a large club capacity, clubgoers were put in a situation where they could not move to the next block as another thread would hold the lock for that block.

For example:



Since no clubgoer can move to an occupied grid block this creates a situation wherein nobody in the club can move.

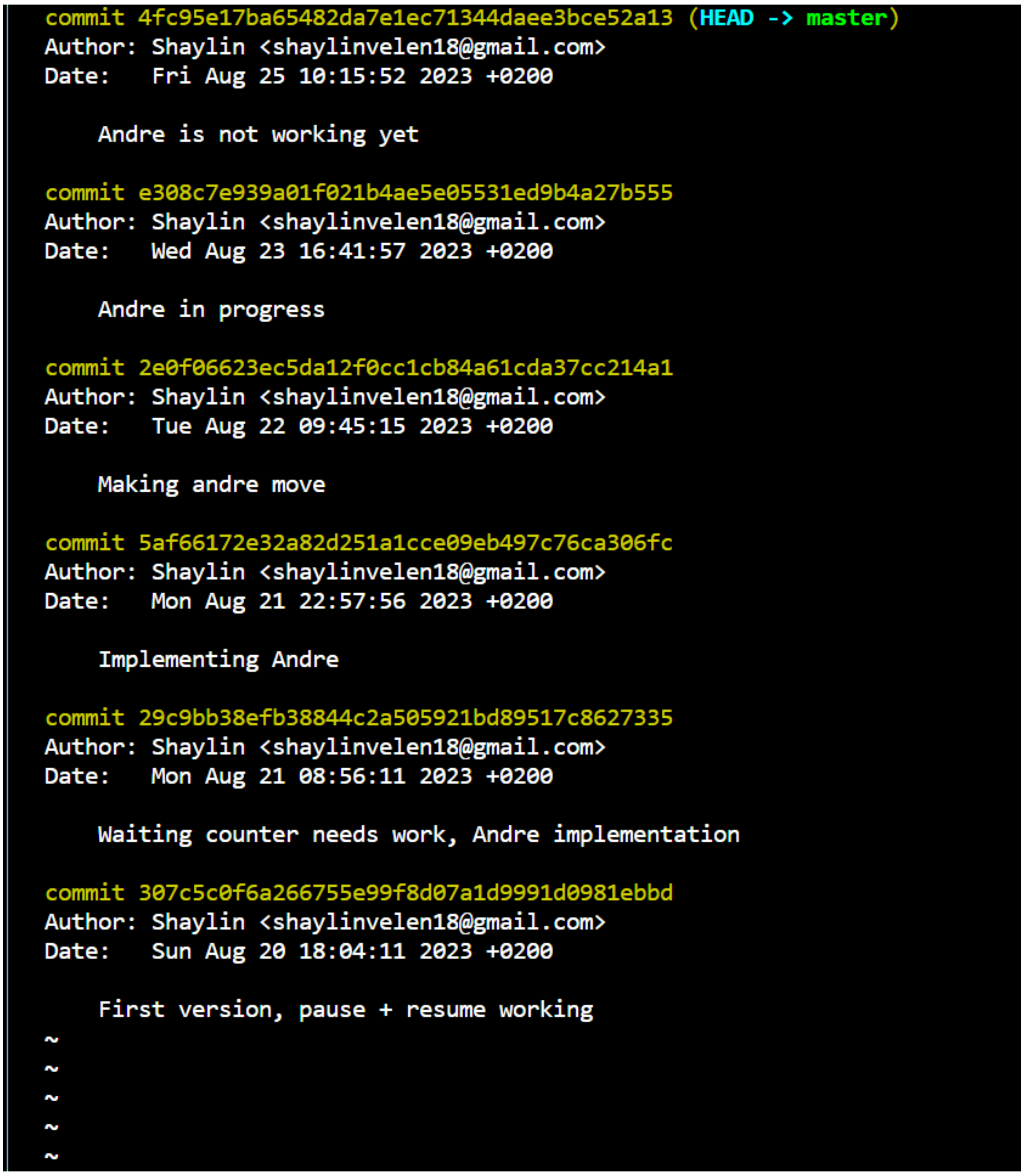


Threads wait at the bar until they are served, causing a cluster of clubgoers in the bar area to the point where there is no space for the people who have been served to leave.

Lessons learned:

In concurrent computing, thread safety is prioritized to ensure that the program behaves accurately and in a deterministic fashion. Deadlocks were discovered to be problematic as the program is stuck in a state where there is a lack of liveliness and hence a programmer should ensure that the program is free of deadlocks. Once you have ensured thread safety, to prevent deadlocks one should only synchronize blocks of the program that need to be synchronized to refrain from a state where two objects do not wait on receiving a lock from each other. In summary interleaving of threads can be controlled by using latches, wait() calls and spins so that threads only run when the programmer needs them to run.

Git usage log:



```
commit 4fc95e17ba65482da7e1ec71344daee3bce52a13 (HEAD -> master)
Author: Shaylin <shaylinvelen18@gmail.com>
Date:   Fri Aug 25 10:15:52 2023 +0200

    Andre is not working yet

commit e308c7e939a01f021b4ae5e05531ed9b4a27b555
Author: Shaylin <shaylinvelen18@gmail.com>
Date:   Wed Aug 23 16:41:57 2023 +0200

    Andre in progress

commit 2e0f06623ec5da12f0cc1cb84a61cda37cc214a1
Author: Shaylin <shaylinvelen18@gmail.com>
Date:   Tue Aug 22 09:45:15 2023 +0200

    Making andre move

commit 5af66172e32a82d251a1cce09eb497c76ca306fc
Author: Shaylin <shaylinvelen18@gmail.com>
Date:   Mon Aug 21 22:57:56 2023 +0200

    Implementing Andre

commit 29c9bb38efb38844c2a505921bd89517c8627335
Author: Shaylin <shaylinvelen18@gmail.com>
Date:   Mon Aug 21 08:56:11 2023 +0200

    Waiting counter needs work, Andre implementation

commit 307c5c0f6a266755e99f8d07a1d9991d0981ebbd
Author: Shaylin <shaylinvelen18@gmail.com>
Date:   Sun Aug 20 18:04:11 2023 +0200

    First version, pause + resume working
~
~
~
~
~
```