CSC2002S Parallel and Concurrent Computing

Assignment 2 Report

By: Noah Gonsenhauser, GNSNOA001

How I enforced Simulation rules:

Quit, Entrance through entrance, and exit through exit:

All of these functions were already implemented when the program was given to use

Start button:

My start button used a count down latch. The latch (named starter in the ClubSimulation class) had a count down of 1, and when the program was run, all of the Patrons (Threads) and barman would be set to await the latch to countdown. When the start button was pressed, the latch would count down by 1 and the simulation would start as the latch will be open. There is also a variable that checked if the start button had been pressed, and if so, it will disable the button’s functionality as it’s served it’s only purpose.

Pause Button:

My pause button functioned on an atomic variable (called paused) in the ClubSimulation class. The Clubgoer, Barman and ClubSimulation class were all synchronized to this variable and every time the “checkPause” function ran in any Patron / The Barman (Thread) there would be a lock applied and then a check applied to see if the Boolean had changed values. When the Pause button was pressed, the lock would be applied to the paused variable and it would be set to “true,” where all the threads would now see that the Boolean was set to true and would pause. This would also trigger the changing of the “Pause” button to a “Resume” button, changing the displayed text on said button, and when pressed would apply the lock and then change the paused variable to be “false,” and would then notify all the threads that the state of the variable had changed. The While loop that was checking the state of the button in each thread would now return “true” which would resume the running of each thread, while also changing the text of the pause button back to “Pause” and alternating functionality to pause the simulation again if pressed.

One Patron enters at a time (no crowding):

In order to prevent crowding of the entrance block, I had to make threads wait if the entrance block was occupied (using .wait().) Thus, I added a synchronized on the entrance block, block of code for both the enterClub and move functions. A while loop inside the enterClub function in the synchronized block to check if the entrance block was occupied (using entrance.occupied()), and if it was, then wait. I then put a block of code in the synchronized block in the move function that notified the entrance block if a patron moved off the entrance block in order to let in another Thread.

Limiting the amount of patrons at the club:

In order to limit club patrons, I added two thing to my already existing code: an or statement to the while statement in the “enterClub” synchronized block which simply was to check if the club was overcapacity (counter.overCapacity()) and if so, this would also result in the Thread waiting. I then also added another synchronized on entrance block of code to the “leaveClub” function, in which the entrance block would be notified if a Thread left the club and thus a new one could join so long as the entrance block wasn’t occupied.

One patron per grid block:

I synchronized the move function so that a threads would only release and return their respective blocks in an order so that two threads didn’t manage to end up on the same block.

Patrons move simultaneously:

I did this by making sure there weren’t deadlocks or race conditions that lead to patrons freezing, patrons move at their issued speeds and thus liveliness is achieved depending on club creation conditions.

Simulation Free from deadlocks:

I synchronized different blocks of code so that certain things only happened in certain orders in order to ensure there wasn’t a situation where threads were waiting for each other and couldn’t proceed.

Andre:

I created Andre to function as a Thread that moved up and down the bar. A patron would sit at a bar position and wait at their position until Andre was in the block below them, at which point Andre would notify the block above him, this would let the patron know they have their drink and can drink it and then go. I created Andre as his own Barman class with his own functions in the ClubGrid class. The clubGrid class would create the starting position of the barman to be in the middle of the grid on the x axis and one layer below the bar on the y axis. Andre would then be created with all the relevant information needed, such as an ID, currentBlock, peopleLocation, club and movingspeed. The Barman possesses the same checkPause function and startSim function as a normal thread however all he does is start, and then until the program is terminated, he checks if paused, then sleeps for a second, moves and then gives drinks. The move function calls a function in clubGrid which moves the barman by getting the barman’s current x location, and the attempting to move him in whatever direction he last moved, if that move happens to off the grid, a variable will flip and he will start to move in the other direction. He also will run the give drink function which synchronize onto the block above the one the barman is currently in, and notify the thread there. The Threads when going to get a drink will synchronize onto their current block and have a while statement to check if the barman is already below them, if not, they will wait till notified and when notified, the while condition will be true and they will get their drink.

Challenges Faced/Lessons Learnt:

In my initial implementation of the crowding prevention, before I had a working limiter to the club, my final implementation would create a race condition where my non-working limiter (which was a separate while statement) would work most of the time, but if a patron would walk onto the entrance block and then walk off of it, this would notify the waiting threads and let one in if it happened to have been let past the while loop that checked if the entrance block was empty and was just waiting to see if the club was at capacity. I then realized that I needed to combine my two while statements into and or statement, so that both conditions had to be true for a thread to get into the club. I also believed that my implementation of notifying when a thread left but a thread was on the entrance block would create the race condition of if a thread leaves the club and a thread is blocking the entrance, Threads would not try get in and would be stuck under the limit with threads waiting outside, however, due to the code notifying the threads in the queue every time they move off the entrance block, as soon as that thread moves, they would be notified and a thread would try get in again. I also learnt that one should not include too many pieces of synchronization in code, nor try to synchronize everything as this can result in unnecessary slowdown and can defeat the point of concurrency.