CONCURRENCY

CSC2002S Assignment 2

Summary

This document regards my finding when working with multithreads and concurrency issues.

WLSGRA012 – Grant Wilson

Index

How I found problems

I followed the rules the simulation needed in a sequential order. Some of the rules were already satisfied and the simulation behaved correctly. Others did not. I struggled to see anything wrong with the simulation at first glance. I then started to fiddle with the room size and the peoples speed and I started to see problems. When I made the program run very slow with lots of people the counters would break (increment uncontrollably) and people would occupy the same block. When I made the program run very fast with many people I would find deadlocks.

Problems

The pause button

The first task was to fix the pause button. This brought about much troubles. What I wanted to do initially was find a way to pause the program. This just resulted to the main thread to pause. All *PersonMovers* were still active and running. My second goal was to create a pause Boolean. The Boolean was Atomic. I wanted it to be this way so that all threads read the value of the button correctly and no stale values were stored in cache or memory that was not updated. When the Boolean was true, the *PersonMovers* would refuse to move. This was due to an infinite while loop. Although this appeared to work, it didn't. This very naïve approach was very costly. Every thread would spin waiting for a condition to be true. So I tried to make them go to sleep when this happened and they would wake up after a period of time and recheck the Boolean. This resulted in another spin cycle, just occurring less as often. The spinning of the threads would take up almost 90% of my CPU usage!

So I created my final solution. I created an Atomic Boolean pause. Every thread would check this static Atomic variable before they move. The Boolean was read and writable so I had to think of race conditions. This pause object also acted as an intrinsic lock for the movement of the thread. If the pause value would be true, then the thread would sleep until woken up. Every thread would go to sleep if this pause value was true. They would do no work and occupy zero CPU. When the resume button is clicked, then the main thread would notifyAll() threads to wake up and continue moving. This worked very well on the behaviour of the movement of the threads. It did not stop them from pausing outside the party.

When the pause button is hit, all threads inside the floor sleep. This causes all threads who are not on the floor to carry on doing work. This resulted in the counters of people in the party to be incorrect. What I did was lock the counters in the pause button intrinsic lock. This caused all the counters to only operate if the threads were awake. So this in term caused the counters to also go to sleep (by doing absolutely no work).

The problem I had with this is knowing where to place the lock. If I placed the lock in the PartyApp class, the *PersonMover* would depend on the PartyApp to compile before it did. However the PartyApp creates all of the *PersonMover* threads. So this caused interdependencies between the two classes. I had to be clever where I kept the locks. If I kept the locks in the *PersonMover* class then there would be a *PersonMover[].length* amount of locks. I ended up storing the lock as a static lock in the *PeopleCounter* class as there is only one instance of this class.

Entrance block

Threads did not check if a block was correctly occupied. Interleaving's started to occur whenever a *PersonMover* entered another square. This caused 2 or more people to be in a block at a single time. To fix this I initially wanted to make the three methods to access a block (enter(), wait(), release()) synchronized on each other. This works very well. However, the solution that I came up with was cleaner. I realized that each block had three states. These are acquire, hold and release. I created an enum to monitor the blocks status. This enum was made private and only accessible through getters and setters. The getters and setters were synchronized on each other so that 2 threads cannot get and set the same value resulting in a race condition. A thread could only acquire a block if it was released (i.e. the blocks status is enter). Once entered the blocks status would change to wait. Bear in mind this is very clean as other method would cause threads never check only the blocks initial status. Once the thread leaves the block then it would change its status back to enter again awaiting another blocks arrival. This stopped 2 people from sharing blocks. Meaning only one person can leave the exit door, only one person can enter at the entrance and only one person can occupy a floor square at a time.

Extra Work

Blocking queue.

For additional work we had to create a queue that monitored people to enter the party in the order that they arrived. This seemed very simple to just create a queue. Many concurrent problems arose when I just used ordinary queues. I later tried to implement my own concurrent queue until one of my friends told me about java.util.concurrent data structures. I ended up using a blocking queue to add people to the queue. I created a *Queue* class that people (*PersonMover threads*) added themselves to the queue. The Queue then dequeued a *PersonMover* thread onto the entrance square and let them enter the party if the entrance was unoccupied. This worked very well as the class is completely thread safe.

I then realized that there was a mistake in my logic. If a *PersonMover* added themselves to a queue then they would need to know which queue to add themselves to. Therefore the Queue class must be compiled first. But in order for the Queue class to be compiled it needs to know what type of objects must be inside there queue(i.e. *PersonMover*). This means that the 2 classes are interdependent on each other and will not compile. I fixed this by compiling the *PersonMover* objects first. I then compiled the *Queue* class second. The *Queue* class created a queue of *PersonMover.* I then made the main PartyApp create the *PersonMover* threads and added them to the queue instead of the *PersonMover* threads adding themselves. This removed all interdependencies.

Room Limit

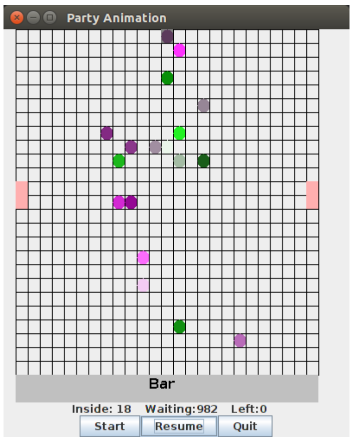
The room limit was a very easy problem to solve. I saw that the counter that monitored the amount of people in the room could only increment if someone enters the room (using the entrance block). I made a global static variable *RoomLimit*. Every thread had to check this value before entering the room. If the counter was less then the room limit and the entrance block was unoccupied then the thread may enter the room else it would wait. The blocking queue would also wait for this thread to enter the room before it dequeued another thread. The *RoomLimit* variable is never written too and therefore can not encounter a race condition.

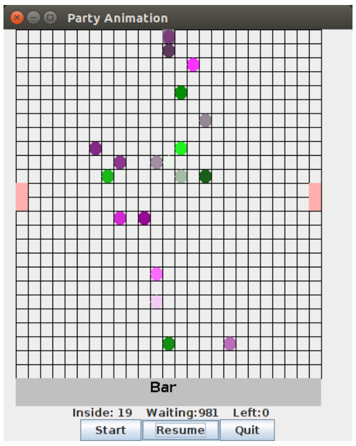
Inaccessible pause button until game started.

I also found a small bug in the program. A user may use the pause button before the simulation started. This would lock all threads before they are created. All I did was make the pause button accessible only when the simulation started.

Unrealistic pause and play

One issue I encountered was that my program would pause perfectly but when it would resume, all threads would move at exactly the same time. This is caused by the fact that when the game pauses, all slow threads (threads that had to wait for longer periods before moving) catch up to the faster threads for a single statement of code. Then all threads move at the same time before moving at their different speeds again. The bottom pictures show this flaw. They are taken one frame apart from each other. Every block has moved one square. This is not right.





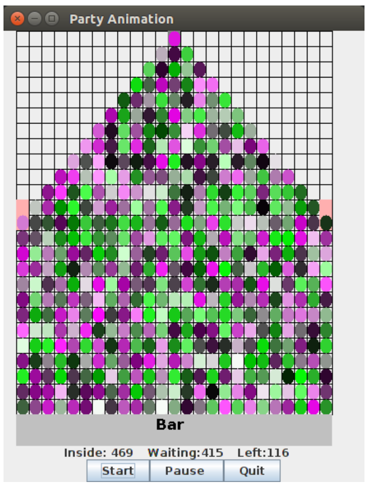
To fix this I made all the threads wait the time they would normally wait before moving. This made it appear that they had been moving at a constant speed the entire time.

Refreshments block up.

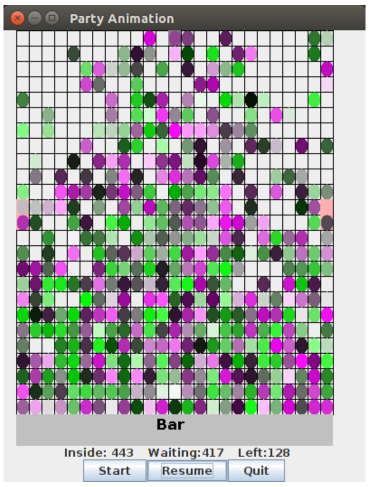
I made an improvement to simulate a smooth flow of people around the party. If you make many people come into the room at a fast pace it causes the room to clog up. People are trying to get to their refreshment so they are constantly moving downward. Other people have had their refreshment but are trying to leave. There are too many people in the room so the top row of people are all trying to get refreshment and blocking everyone from leaving the party.

This is also because so people are on the exit doors and are still "thirsty" so they do not leave. We have a deadlock. People who are thirsty are trying to wait for the people below them to release their block while people who want to go home are waiting for people above them to release their block. In term the whole party freezes and no one moves.

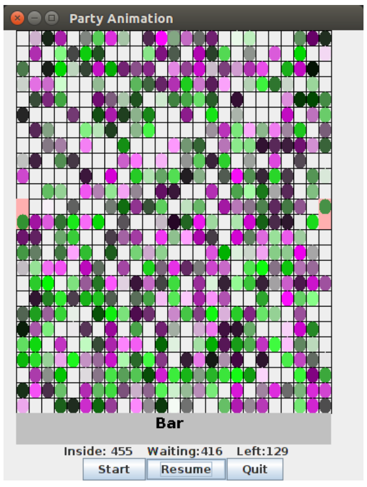
To fix this I made the people check the room size and the people inside before they made their decision like in a real situation. If there are too many people in the room then the person will not head for refreshments. I made the person wonder around the room if the room is 70% full. Then when people start leaving then they will go and get a drink. This causes the clog up of people to be released and simulates a real party. The bottom pictures show the release of traffic when the room is full.



Inside: 469 Waiting:415 Left:116



Inside: 443 Waiting:417 Left:128



Inside: 455 Waiting:416 Left:129