CSC3002F: OS2

Leanne January (JNRLEA001)

Question One:

Which threads run in the program?

The Customer class extends the Thread class. These are heavyweight threads which represent the actions of a single Customer. There are as many Customer threads as there are customers specified by the command line arguments or default value.

The Inspector class extends the Thread class. This a single, heavyweight thread which checks if customers are violating social distancing limitations.

The CounterDisplay class implements the Runnable interface. This is a single, lightweight thread which updates the values showing how many customers are in the store, waiting and left the store.

The ShopView Class implements the Runnable interface. This is a single, lightweight thread which updates the animation on the simulation.

Question Two:

Which classes are shared amongst threads?

These threads share the CustomerLocation, ShopGrid, GridBlock and PeopleCounter class in the following ways:

ShopView: Uses CustomerLocation, ShopGrid and GridBlock classes.

Inspector: Uses CustomerLocation and PeopleCounter class.

CounterDisplay: Uses PeopleCounter class.

Customer: CustomerLocation, ShopGrid, GridBlock and PeopleCounter.

Question Three:

Explain the synchronization mechanisms you added to each class and why they were appropriate.

From the classes we were allowed to change:

CustomerLocation: This is the class where the locations of the customers are stored.

No changes made, all the variables that were shared and being used were already atomic and therefore protected.

PeopleCounter: This class keeps track of people inside and outside the shop.

The only change I made was in the constructor. It was not a synchronization mechanism.

GridBlock: This class represents the different blocks for the shop.

1. I added a semaphore called lockGrid which allows one thread to acquire a lock. This semaphore locks the grid block when a customer moves into it and releases the lock when the customer moves to a different block. This prevents multiple customers from accesses the same grid block.

2. I changed isOccupied to an AtomicBoolean. This was to ensure that all customer threads knew whether a grid block was occupied and prevented two threads from accessing and changing this value at the same time.

3. I also synchronized the get and release methods to ensure that only one customer could move to a particular block at a time.

ShopGrid: This class represents the shop.

I added two Semaphores:

a) lock: This semaphore allows the maximum amount of people allowed in the shop to acquire a lock. This value is either the default – 5, or specified by the user. This semaphore locks threads from entering the shop once the maximum amount of people allowed in has been reached. As threads exit, new threads can enter.

b) lockEntrance: This semaphore allows one thread to be in the entrance block at a time. This prevents multiple threads from entering the store at the same time and it prevents threads from entering the store if there is a thread in the entrance.

Question Four:

How did you ensure liveness in the code?

I ensured liveness by ensuring that the if all threads wanted to access the same critical region then progress would still be made in all threads. How I did this:

> I changed the get() method in GridBlock to return false if the block in question was occupied. This means that the move() method in ShopGrid will force the thread to stay in the same block and choose somewhere else to go on its next move. This means that if multiple threads all want to move to the same block that only one thread will be able to and the other threads would stay in their block and look for a different square to move to on its next move.

Question Five:

How did you protect against deadlock? Was this necessary?

It was necessary to protect against deadlock. I made sure not to synchronize the move() method in ShopGrid() as if it was synchronized only one customer thread could move at a time and threads have to wait for other threads to move. This means if one thread is waiting to acquire a lock, then no other thread can move.