University of Cape Town

Operating Systems:

Assignment 2: Synchronization

Sihle Calana

6-21-2020

Contents

Introduction

This report is about an assignment in which I had to correct the code for a multithreaded Java simulation of customers entering a shop under Covid19 pandemic level three regulations, using synchronization mechanisms to ensure that it operates within the specified synchronization constraints.

Which Threads Run In The Program?

In this simulation, threads behave like customers shopping for an item and then proceeding towards the checkout before exiting the store. There are multiple restrictions on these "customers" (threads) due to the level 3 restrictions such as keeping a safe distance from other "customers" and the number of customers allowed inside the store.

Which Classes Are Shared Amongst Threads?

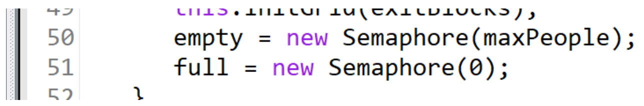
Since there are multiple customers in the store at a time, ShopGrid is one of the shared class between threads but only between a certain number of threads (more specifically a number below or equal to the maximum store occupancy). ShopGrid is class representing the shop as a grid of (another class called) GridBlocks.

Each customer (that has been allowed to enter) has a particular location in the store, at any given point in time, determined by the Gridblock they are on. To add onto this, no other customer may be on the same gridblock as another customer as it violates the Covid rules. This means that each thread will have an object of GridBlock class and CustomerLocation class. Gridblock is a class representing the blocks for the shop and CustomerLocation is a class where locations of each customer are stored.

The last class that is shared by each and every thread is the PeopleCounter class. This is a class to keep track of the number of customers inside, outside and those who have left the shop.

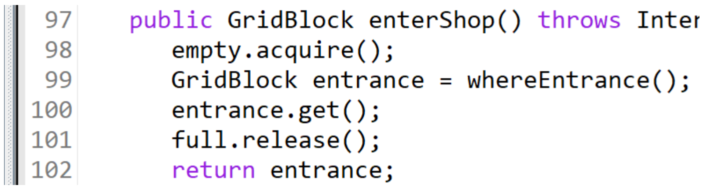
Synchronization Methods Use

ShopGrid: I noticed that threads are both consumers and producers. enterShop() acts as a producer method (increasing the number of customers in the store) and leaveShop() acts as a consumer method (decreasing the number of customers in the store). The issue is that we shouldn't allow threads to access ShopGrid if the shop is full, so I used 3 semaphores to solve the classic producer-consumer synchronization problem. Empty initialized to the stores max occupancy as the store is initially vacant and full initialized to zero as there aren't any customers in the store to begin with.



```
        this.initGrid(exitBlocks);
        empty = new Semaphore(maxPeople);
        full = new Semaphore(0);
    }
```

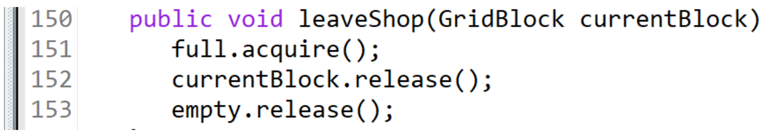
For every thread accessing ShopGrid – upon a customer entering the store, it acquires the empty semaphore lock (empty basically acts like a multiplex), thus decrementing the number of threads that may enter the store. Once the entrance grid block is retrieved the full semaphore is release, thus incrementing how much closer the store is to getting full.



```
public GridBlock enterShop() throws Inter
   empty.acquire();
   GridBlock entrance = whereEntrance();
   entrance.get();
   full.release();
   return entrance;
```

If the store is full, empty will have a semaphore value of 0. Any other thread that tries to access the store will have to wait until another customer already inside leaves the store which will call release on empty (and acquire on full), therefore incrementing empty by one (and decrementing full). This lets the first thread in the waiting queue acquire the empty semaphore and enter the store.

Leaving the store works similarly to entering the store, just vice versa as the order of acquiring and/or releasing different semaphore changes:

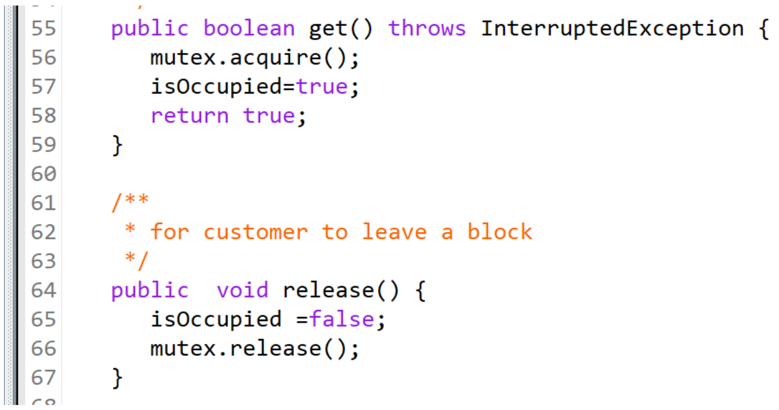


```
public void leaveShop(GridBlock currentBlock)
   full.acquire();
   currentBlock.release();
   empty.release();
```

I chose to use Semaphores instead of monitors as it made regulating the number of threads active inside the shop a lot easier.

Another way I ensure safe synchronization is by leaving the release of the block previously owned by a thread to last when moving (see the picture under Liveness heading). Doing this ensured that any other customer wanting to get this block doesn't do so until the new block is returned to the moving customer. This was especially helpful for customers browsing on the entrance block and moving to another grid block while another customer that's outside wants to get in.

GridBlock: As stated before customers need to social distance meaning only one gridblock may be occupied by a thread. As the enterShop() method from ShopGrid stands, multiple threads will have access to the entrance grid block (while the store is below maximum capacity). To avoid this, within the get method of the GridBlock class I made it such that a mutex (initialised to 1) is acquired. Considering get() is called whenever a customer moves to a block, this ensures no more than one customer is on a grid block in that particular location.



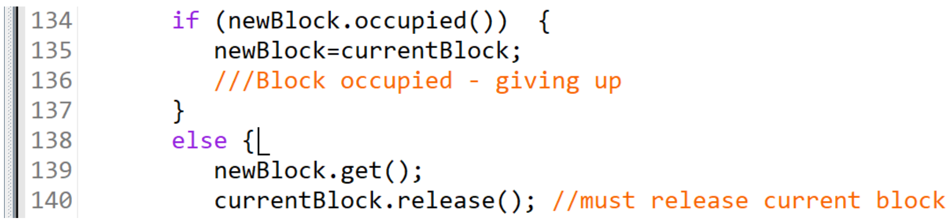
```
public boolean get() throws InterruptedException {
   mutex.acquire();
   isOccupied=true;
   return true;
}

/**
 * for customer to leave a block
 */
public  void release() {
   isOccupied =false;
   mutex.release();
}
```

Remember synchronize acquires the lock with '{' and releases it with '}'. The reason I chose not to just synchronize the get() method (use monitors) is because if a customer gets a particular grid block and exits the get() method, another customer wanting the same grid block may also go through the get() method and acquire the same grid block. Notice that when a customer leaves a grid block and calls release, the mutex is also released allowing other customers to enter that specific grid block. The reason the mutex was initialised to one in the first place was to ensure mutual exclusion.

PeopleCounter: This class is a lot like the CustomerLocation class Prof Kuttel provided for this assignment. Both are access and set by numerous threads, with the only exception being that PeopleCounter class is used by customers (using interchangeably with threads) waiting outside, shopping and those who've already left. For this reason I decided to make all the integer variables Atomic (aside from maxPeople which is final). This ensures that the whatever actions done by the atomic variable is uninterrupted, thus avoiding data races.

Liveness

Liveness is ensured for threads active inside the store as see in the ShopGrid class move method. If a grid block a customer wishes to occupy is taken instead of waiting for it to be available, the customer will choose to stay at its current gridblock instead.



```
      if (newBlock.occupied())  {
         newBlock=currentBlock;
         ///Block occupied - giving up
      }
      else {
         newBlock.get();
         currentBlock.release(); //must release current block
```

Another way in which I ensured liveness is through the use of atomicIntegers in the PeopleCounter class. When updating an atomic variable value no other threads are suspended and these other threads may proceed to do other work.

The only time a thread is suspended is when it is waiting to get into the store and of course terminated when it exits, otherwise threads are always active. Avoiding the use of monitors also helped ensure liveness as excessive use of monitors over large portions of code risks making the code sequential.

Deadlock Protection

Having read the way I implemented the entering and exiting the shop methods, you can see that any lock a thread acquired upon entering is released upon exiting (eg. The empty muliplex). Similar thing applies for moving between grid blocks. Acquire grid block mutex (for mutual exclusion) upon arrival and release grid block mutex upon departure. In both scenarios, the thread waiting for the lock to be release will eventually get it.