Report

CS431 Programming Languages Lab

Concurrent Programming Group 24

Abhinav Hinger 160101004
Daman Tekchandani 160101024

Inventory Management

1. We used **synchronized keyword** on methods which are used to place order for Small, Medium and Large and Caps. We have locks on each type of order since its **left over quantity** is the one determining whether the order will be successful or not.
So we need to be sure by it that only one thread can access the resource at a given point of time.
This synchronization is implemented in Java with a concept called **monitors**. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
In the below function, we have used synchronized on method placeOrderMedium and similarly every other type of order. So if any thread calls this function it is locked and no other thread can use this function. **Since monitor is applied on Order Type, 2 threads can call different order types that will occur concurrently.**

```
synchronized public Boolean placeOrderMedium(char merchandiseType, int quantity){
        Boolean success = false;
        // this.printInventory();

        if(this.mediumTshirtCount >= quantity){
            this.mediumTshirtCount = this.mediumTshirtCount - quantity;
            success = true;
        }
            // this.printInventory();
        return success;
    }
```

Cold Drink Factory

- Discuss the importance of concurrent programming here.

  Here it is very important to run both the packaging and sealing unit to run concurrently so that they never sit idle whenever possible. For this we create separate threads for both packaging and sealing processes.

- Using code snippets describe how you used the concurrency and synchronization in your program.

We used semaphore locks to handle synchronization between multiple threads. We locked godown and unfinished tray so that both units don't access it together.
So in our code whenever we are using status.finishedB1 or status.finishedB2 these signify our godown therefore we are putting locks while updating them so both of them don't do it together.

```
if(bottle.isSealed()){
    try {
     // acquire method
        sem.acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    if(bottle.type == 1){
        status.packagedB1++;
        status.finishedB1++;
    }
    else if(bottle.type == 2){
        status.packagedB2++;
        status.finishedB2++;
    }
    currFillPackUnit.t = 0;
    localTimePackUnit.t = 0;
    sem.release();
}
```

```
if(bottle.isPackaged()){
    try {
     // acquire method
        sem.acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    if(bottle.type == 1){
        status.sealedB1++;
        status.finishedB1++;
    }
    else if(bottle.type == 2){
        status.sealedB2++;
        status.finishedB2++;
    }
    currFillSealUnit.t = 0;
    localTimeSealUnit.t = 0;
    sem.release();
}
```

- How would the program be affected if synchronization is not used? Discuss

Then there will be a discrepancy in the count of bottles in godown and unfinished tray.

Traffic Light

1. Yes the concept of concurrency is applicable because the system requires **parallel working of all the traffic lights and cars**. For example while the traffic lights are changing their status from Green to Red, we need to change the waiting time for incoming cars. All traffic lights and car times has to be updated every second.

   For achieving this concurrent working of the system, we have used appropriate looping with relevant groups operating together in a loop.

   Here we are updating all the cars **waiting** on every Paths possible.

```
//Whether the last car has passed or not
      if(lastCarsWaitingF1.t != 0)
        lastCarsWaitingF1.t--;
      if(lastCarsWaitingF2.t != 0)
        lastCarsWaitingF2.t--;
      if(lastCarsWaitingF3.t != 0)
        lastCarsWaitingF3.t--;
      if(lastCarsWaitingT1.t != 0)
        lastCarsWaitingT1.t--;
      if(lastCarsWaitingT2.t != 0)
        lastCarsWaitingT2.t--;
      if(lastCarsWaitingT3.t != 0)
        lastCarsWaitingT3.t--;
```

   Here we update the traffic light statuses when 60 seconds are over.

```
while(true){
      if(currLight == 1 && flag){
        t1.updateStatus(1);
        t3.updateStatus(0);
      }
      else if(currLight == 2 && flag){
        t2.updateStatus(1);
        t1.updateStatus(0);
      }
      else if(currLight == 3 && flag){
        t3.updateStatus(1);
```

```
        t2.updateStatus(0);
      }
```

Here we update the car statuses: Waiting, Crossing or Passed

```
//Decrease the time on all cars
     for(int i=0;i<carList.size();i++){
       if(carList.get(i).timeLeft !=0)
         carList.get(i).timeLeft--;
       if(carList.get(i).timeLeft <= 6)
         carList.get(i).status = 1;
       if(carList.get(i).timeLeft == 0)
         carList.get(i).status = 2;
     }
```

Hence in each cycle as the global time increases by 1, every car and traffic lights time parameter gets increased or decreased by 1 hence keeping all concurrently functioning wrt the system.

2. Yes. Synchronisation was needed in this problem because when a car is in the process of crossing , No other car from its own path can cross even if they arrived after fraction of a second. To implement this, we used a basic approach where we just increased the waiting time of the incoming car by 6 to account for this constraint in our system.

   In the below snipped, there are 2 cases:

   A. Light is Green:

      1. If the next car can pass through the crossing then update the last car waiting + 6, since it only needs to wait 6 more seconds
      2. If the next car cannot pass(i.e time left not 6 seconds) then it will go after 120 seconds. But we add 6 seconds also to account for the locked crossing.
      3. If already another car added with large waiting time, just add 6 seconds to the waiting time of the last car.

   B. Light is Red
      1. It is the first car at the Red Crossing, just update according to time left
      2. There is already another car, then just add +6 to account for car not passing crossing while another car in front is crossing.

```
public static int getRemainingTime(Car car,TrafficLight t,Integ
lastCarsWaiting){
```

```
    System.out.println("traffic light :"+t.id+" status: "+t.status+"
lastcarwait: "+lastCarsWaiting.t);
    if(t.status == 1){
      if(t.time >=(lastCarsWaiting.t+6)){
        lastCarsWaiting.t += 6;
        return lastCarsWaiting.t;
      }
      else if(t.time >= lastCarsWaiting.t){
        lastCarsWaiting.t = t.time+126;
        return lastCarsWaiting.t;
      }
      else{
        lastCarsWaiting.t+=6;
        return lastCarsWaiting.t;
      }
    }
    else{
      if(lastCarsWaiting.t == 0){
        lastCarsWaiting.t = t.time+6;
        return lastCarsWaiting.t;
      }
      else{
        lastCarsWaiting.t += 6;
        return lastCarsWaiting.t;
      }
    }
  }
```

3. If there are 4 crossings , then the number of paths will increase. From each direction, a car can now go to 3 other directions out of which 2 will be controlled by traffic light and 1 will be free to go.

Need to return 12 directions instead of 6

sE means source direction is East. We will need sN and dN to account for the new direction. Each direction will have 3 options.

```
public int AddCarButtonClicked(java.awt.event.MouseEvent evt)
{//GEN-FIRST:event_jButton1MouseClicked

        if(sE.isSelected() && dW.isSelected()) return 0;
        if(sE.isSelected() && dS.isSelected()) return 1;
        if(sE.isSelected() && dN.isSelected()) return 2;
```

```
        ....
        return -1;

    }
```

Waiting time will be 180 because of 4 traffic lights. Decrease times of all 4 together.

```
else if(t4.isGreen()){
        t1.decreTime();
        t2.decreTime();
        t3.decreTime();
     t4.decreTime();
        flag = false;
        if(t3.isTimeZero()){
          currLight = 1;
          flag = true;
        }
      }
```

One thing that will change is that cars coming towards a particular direction will have **cars coming from more than 1 direction**. For eg. When Traffic light of North is Green, cars will come from East and South. If we assume that cars have different lanes for different destinations, we may also have to account for crowding at the destination directions. So a locking system across the lanes will be created to account for proper passing of vehicles.

126 → 186 if car cannot cross

```
else if(t.time >= lastCarsWaiting.t){
        lastCarsWaiting.t = t.time+186;
        return lastCarsWaiting.t;
      }
```