Assignment 1 Programming Lab

Report by Group 8

Sahib Khan 160101057
Saurabh Bazari 160101061

Problem 1: Merchandise sale for Alcheringa 2020

1. **Synchronization Technique used**

We have used synchronized blocks that are inbuilt in java for synchronization.
Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized with some objects. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

What we have done is we have 8 threads that will process the orders. All the orders are distributed among these 8 threads equally and then these threads will run parallelly processing these orders. Each thread will process all orders assigned to it sequentially.
Now if two threads are processing orders of the same object we need synchronization while accessing the inventory.We have not used one thread for each order as if the number of orders is very large say 1000 then the program will crash due to 1000 threads being run.

As can be seen from the snap above we have written all the code for executing caps order is written under the synchronized block which is synchronized on lockCap object. Thus when one thread is executing an order for caps other threads will wait for the first thread to complete. Only when a first thread comes executes all the code written in synchronized blocks and comes out of it the other thread can run.

```
public void executeCaps(Order order){
    synchronized(inventory.lockCap){
        int orderSize=order.getOrderQuantity();
        int currentSize = inventory.getCaps();

        if(orderSize > currentSize){
public void executeSmallTShirt(Order order){
    synchronized(inventory.lockS){
        int orderSize=order.getOrderQuantity();
        int currentSize = inventory.getSmallTShirt();
        if(orderSize > currentSize){

public void executeMediumTShirt(Order order){
    synchronized(inventory.lockM){
        int orderSize=order.getOrderQuantity();
        int currentSize = inventory.getMediumTShirt();
        if(orderSize > currentSize){

public void executeLargeTShirt(Order order){
    synchronized(inventory.lockL){
        int orderSize=order.getOrderQuantity();
        int currentSize = inventory.getLargeTShirt();
        if(orderSize > currentSize){
```

Problem 2: Cold Drink Manufacturing

1. **Importance of concurrent programming here.**

Concurrency is the key to solving this problem. In the context of this problem, we will discuss it here.
Concurrency here means that both units are running parallelly. Suppose if we run have run them sequentially that is first the packing unit runs packs a bottle then sealing unit takes the packed unit and seals it. Then this cycle is repeated. In this case, both of them waste a lot of time waiting for the other. Instead, concurrency is used in this problem, so that both the units run side by side. Thus the waiting time and efficiency are greatly improved. In this approach, we must ensure that both the units access the buffers in a synchronous way.

```
if(nextWakeUpTimePacking==currentSecond){
  // Start packing unit by checking current time
  packingUnitThread.start();
  packingUnitThread.join();
}
if(nextWakeUpTimeSealing==currentSecond){
  // Start sealing unit by checking current time
  sealingUnitThread.start();
  sealingUnitThread.join();
}
```

2. **Concurrency and synchronization in our program**

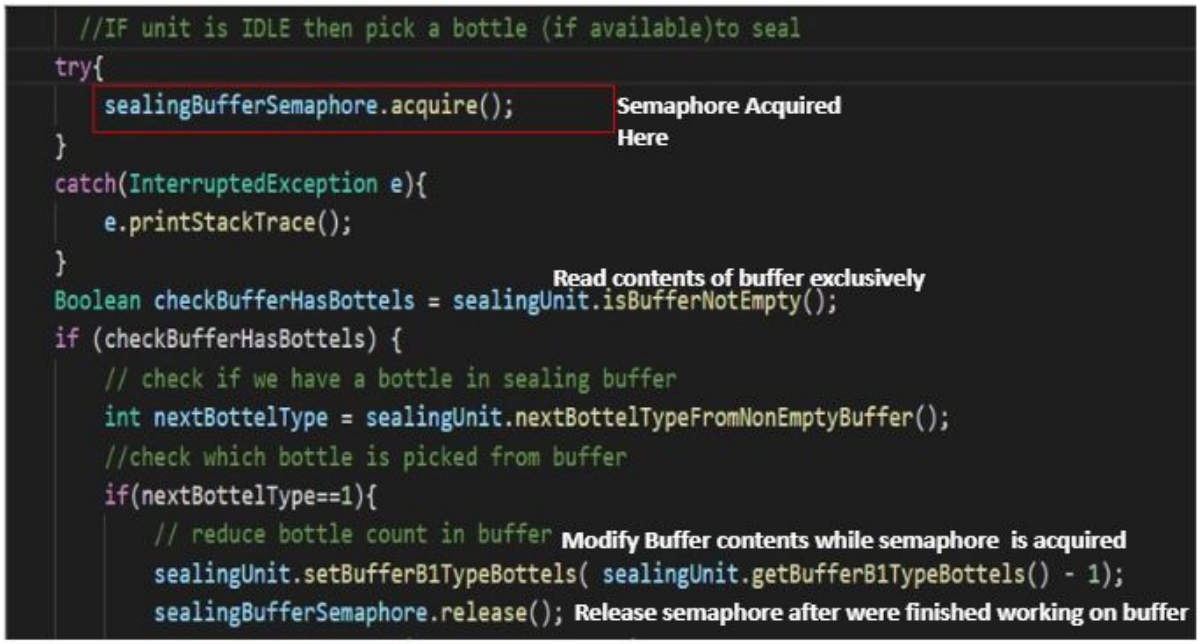
In our program, we have two threads one each for sealing unit and packing unit respectively. Whenever a unit needs to access a buffer or modify the buffers it must acquire a semaphore for that buffer.
Thus in our code, we have 4 semaphores.

1. Semaphore unfinishedTraySemaphore;
2. Semaphore packagingBufferSemaphore;
3. Semaphore sealingBufferSemaphore;
4. Semaphore godownSemaphore;

Each Semaphore is allowed at max one member to acquire it and others trying to acquire it are blocked. Then they are unblocked when the first member releases the semaphore.

```
try{
    sealingBufferSemaphore.acquire();
}
catch(InterruptedException e){
    e.printStackTrace();
}
Boolean checkBufferHasBottels = sealingUnit.isBufferNotEmpty();
if (checkBufferHasBottels) {
```

As can be seen from the snap, before calling the function isBufferNotEmpty of sealing unit class, we acquire the sealing buffer semaphore so that we ensure only this thread is reading the buffer value and buffer contents are not being modified simultaneously.



This example shows that synchronization maintained throughout the code by always acquiring the semaphore whenever we want to work on a buffer.

For concurrent programming, what we do is, we maintain a variable currentSecond. Then each Thread (for each unit) is ran when the currentSecond is equal to its wakeupTime. When we wake up a thread it performs some work say pick a bottle for processing and sets it next wakeup time accordingly (for example if a packing unit picks bottles it sets it to wake up time as currentSecond+3). Then we recompute currentSecond as min of wakeupTime of both. This way we have simulated the process for both the units.

3. Without Synchronization Failure

In this problem, synchronization is needed for each buffer i.e. packing buffer, sealing buffer, and unfinished tray. The program will fail if synchronization is not used. Whenever a bottle is picked or inserted in each of the above, some technique must be used to ensure mutual exclusion preventing race conditions.

Let us take a case.
Buffer: Sealing Buffer
No of bottles in buffer =2;
Now at some time t
The packing unit inserts a bottle in the buffer.
Packing unit executes :

```
Bottles = getSealingBufferBottles();
Bottles;
setSealingBufferBottles(Bottles);
```

At the same time, the sealing unit picks a bottle from the buffer to seal.
Sealing unit executes

```
Bottles = getSealingBufferBottles();
if(bottles>0){
        Bottles--;
        setSealingBufferBottles(Bottles);
}
```

If this code is executed simultaneously then a final number of bottles in sealing buffer is dependent on race condition.
Suppose both reads at the same time then

Sealing unit read bottles=2;
Packing unit reads bottles=2;
Packing unit sets buffer bottles = bottles+1 = 3;
Sealing units sets buffer bottles = bottles-1 = 1;

So the final result will be 1 bottle in sealing buffer, which is incorrect, it should have been 2.
Thus synchronization is essential for our code**.**

Problem 3: Automatic Traffic Light System

1. **Concurrency in the problem**

In this problem, we needed concurrency to update traffic light, update each vehicle waiting time and new vehicle addition parallelly.

In real-time input, user gives any number of vehicles at any instance of time. So, simultaneously we have to update traffic lights and other vehicles waiting time. Because they are independent of a new vehicle.

In the input object, there is an Action Listener that listens for an event to occur. In our case, it is even where the submit button is pressed. When this event occurs this function reads the input (number of cars to be added in each lane) and insert cars accordingly into the vehicle list.

Traffic light controller thread controls traffic lights color and remaining time independently from other threads.
Whenever a new vehicle adds in the vehicle list, its waiting time is calculated by lane and traffic light status and timer thread update vehicle list and GUI every second.

```
Input input = new Input(jFrame);
start=true;
int forContinue=0;
// wait for first input then start the clock
while(start){
}
// start all threads
System.out.println("start");
trafficLightControllerThread.start();
timerThread.start();
```

2. **Synchronization in this problem**

In this problem synchronization is needed while accessing the vehicle list. For each lane when a new car is entered it must be added to the vehicle list. The table showing the waiting time for each car also needs to access this list to update waiting time of vehicles that are already present and also for the new vehicles which enter the system. Thus this two access must be exclusive. We have implemented this using Lock. Whenever the vehicle list need to be read or modified we first acquire the lock and perform our task and then subsequently release the lock.

A code snippet showing the same is given below.

```
vehicleListLock.lock();
vehicleTableData[0]=Integer.toString(vehicle.getVehicleNumber());
vehicleTableData[1]=Character.toString(vehicle.getSourceDirectionSymbol());
vehicleTableData[2]=Character.toString(vehicle.getDestinationDirectionSymbol());
vehicleList.add(vehicle);
vehicleTableModel.addRow(vehicleTableData);
vehicleListLock.unlock();
```

Asa show in this snap vehicleListLock is a reentrant lock that is part of the concurrency library in JAVA. It provides mutual exclusion in our code.

3. **four-way junction instead T-shaped junction**

In a 4 way junction, there will be 4 traffic lights.
For a car in a lane, there are 3 lane which it can go to.
Out of 3 one lane will have free flowing traffic as earlier(car can go there regardless of traffic light)
For the other two lanes cars will go sequentially, and each car takes 6 seconds to cross(during which time car behind it has to wait)
We just need to do minor changes in our code, mainly creating a new lane, and our code runs fine for 4 lane case.

TrafficLight.java
No changes

Vehicle.java
No changes

ThreadTimer.java
No changes

RoundButton.java
No changes

TrafficLightController.java
In this instead of having 3 trafficLight instances we created one more TrafficLlight for North Lane

TrafficLight trafficLightForNorth
TrafficLight trafficLightForSouth;
TrafficLight trafficLightForWest;
TrafficLight trafficLightForEast;

```
Constructor :-
Add button for all traffic light, set colour and status for each traffic light;
run(){
        while(true){
                Set colour for each light
                For loop with 60 iteration{
                        Update time for each traffic light;
                        Thread.sleep for 1 sec;
                        Decrement traffic light time by 1;
                }
                Update colour for each light
        }
}
```

TimerTaskHelper.java

In function run we needs to function call for new lane added to update its values after second.
Below is the code given.

```
Main.southLane.setLastVehicleRemaingTime(Main.southLane.getLastVehicleRemaingTime()-1
);
Main.westLane.setLastVehicleRemaingTime(Main.westLane.getLastVehicleRemaingTime()-1 );
Main.eastLane.setLastVehicleRemaingTime(Main.eastLane.getLastVehicleRemaingTime()-1 );
Main.northLane.setLastVehicleRemaingTime(Main.northLane.getLastVehicleRemaingTime()-1
);
```

Lane.java

In this file we just need to change the variable waitingTimeforAgainGreen from 120 to 180
Because in round robin fashion every traffic lights turns green after 60*3 =180 seconds.

```
  private int waitingTimeforAgainGreen =180;
```

Main.java

```
TrafficLight trafficLightForSouth = new TrafficLight(120,"RED");
TrafficLight trafficLightForWest = new TrafficLight(60,"RED");
TrafficLight trafficLightForEast = new TrafficLight(60,"GREEN");
TrafficLight trafficLightForNorth = new TrafficLight(180,"RED");

southLane = new Lane('S',trafficLightForSouth,vehicleListLock);
```

```
westLane = new Lane('W',trafficLightForWest,vehicleListLock);
eastLane = new Lane('E',trafficLightForEast,vehicleListLock);
northLane = new Lane('N',trafficLightForEast,vehicleListLock);

TrafficLightControllerThread trafficLightControllerThread = new TrafficLightControllerThread(
    trafficLightForSouth,trafficLightForWest,trafficLightForEast,TrafficLight
trafficLightForNorth,jFrame);

 public void inputFromGUI(char sourceDirectionSymbol,char destinatoinDirectionSymbol, int
numberOfVehicle){
        if(free flow){
                Create vehicle object;
                Set as Passes!!
                Add to vehicle List
        }
        else{
                Call destination lane method insertIntoQueue()
        }
}
```

Input.java

In a new case, a car in a lane can go to 3 different direction, so there can be 12 different possibilities for car to go. So we need to take 12 inputs.

```
rowData [] ={"S","W","0"};
      inputTableModel.addRow(rowData);
rowData [] ={"S","N","0"};
      inputTableModel.addRow(rowData);
rowData [] ={"N","E","0"};
      inputTableModel.addRow(rowData);
rowData [] ={"N","W","0"};
      inputTableModel.addRow(rowData);
rowData [] ={"N","S","0"};
      inputTableModel.addRow(rowData);
rowData [] ={"W","N","0"};
      inputTableModel.addRow(rowData);
rowData [] ={"W,"E","0"};
      inputTableModel.addRow(rowData);
rowData [] ={"W","S","0"};
      inputTableModel.addRow(rowData);
rowData [] ={"E","N","0"};
      inputTableModel.addRow(rowData);
rowData [] ={"E","S","0"};
```

```
        inputTableModel.addRow(rowData);
rowData [] ={"E","W","0"};
        inputTableModel.addRow(rowData);
```