Programming Lab
Assignment Set - 1

Members -
Rohit Pant (160101049)
Shivam Kumar (160101066)

Problem 1: Merchandise sale for Alcheringa 2020

Q. Specify the synchronization technique that you used in the program. Explain your implementation using code snippets.

Ans. We used the **'*synchronized*'** keyword to create synchronized blocks of code to maintain consistency in the Inventory. As multiple orders run parallely on different threads there could be more than one orders running at the same time such that they are for the same merchandise. In such a case in the absence of some any synchronization the memory could be left in an inconsistent state at the end of the orders. So, we needed to ensure that only one thread may access an Inventory item at a point of time. This enforces memory consistency while allowing for parallelism too as different orders can still run on separate threads.
The synchronized keyword serves this very purpose. A synchronized block in Java is synchronized on some object. 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.

As can be seen in the below code, we implemented the solution using two such synchronized blocks. The first synchronized block ensures that only one thread updates the inventory at a time. Here the object on which the synchronization is done is the count in the inventory of the corresponding merchandise item. This also ensures that multiple orders having distinct inventory items can run this block to promote parallelism. The second synchronized block is done over the entire inventory whenever we want to print the inventory.

```
// If order is for caps
if(this.orders[i].item.equals("C")){
    // Get exclusive access on the count of caps int the inventory.
    synchronized(Inventory.caps){
        // If sufficient number of caps present in inventory, subtract from the inventory to complete the order
        if(this.orders[i].quantity <= Inventory.caps){
            Inventory.caps -= this.orders[i].quantity;
        }
        // Else the order has failed
        else success = false;
    }
    // Get exclusive access on the entire inventory.
    synchronized(inventory){
        // Print order status and inventory.
        if(success) System.out.println("Order " + this.orders[i].id + " succesful!");
        else System.out.println("Order " + this.orders[i].id + " failed!");
        inventory.print_inventory();
    }
}
```

Note that the inventory printed at the end of an order may not be the same as the inventory in reality right after that order, but same as one after multiple intervening offers. This is because different orders may enter and exit the first synchronized block before it enters the second synchronized block for some thread. In spite of this we the output will be correct nonetheless just one of a later time. To get the absolutely correct output, we would have to have a single synchronized block over the entire inventory which would decrement the corresponding entry in the inventory and then print it, but this would be against the spirit of parallelism.

Problem 2: Cold Drink Manufacturing

Q. Discuss the importance of concurrent programming here.

Ans. We used concurrent programming here as if we would have run the packing machine unit and sealing machine unit serially then if a bottle had been been processed by the sealing unit, the packing unit would have to wait till the sealing unit had finished its job. Hence, it would just increase the waiting time for both the units. Using concurrent programming, we are able to resolve this problem, as the two machines don't wait for each other if bottles are available for them to process.

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

Ans. We have used "***Semaphore***" to implement synchronization and executed threads for running the logic of the sealing & packing units simultaneously to make use of concurrency.

```
sealing.start(); // start sealing machine thread
packing.start(); // start packing machine thread
try {
    sealing.join();
} catch (InterruptedException e) {
    System.out.println(e);
}

try {
    packing.join();
} catch (InterruptedException e) {
    System.out.println(e);
}
```

If the current time is equal to wake up time of either the packing or the sealing unit, both the threads are started as we use packing.start( ) and sealing.start( ) to start execution of sealing unit and packing unit thread

and since both the threads are started so both are executed in parallel hence concurrent execution happens.

We have used four Semaphores in all.

1. unfinishedSem - Semaphore for unfinished tray
2. packingSem - Semaphore for packing unit buffer tray
3. sealingSem - Semaphore for sealing unit buffer tray
4. godownSem - Semaphore for godown

```
if (this.isPacked == true) {
    // acquire lock on godown object
    try {
        this.godownSem.acquire();
        if (this.currentBottle == 1) {
            this.godown.bottle1++;
            this.bottles.sealedbottle1++;
        } else {
            this.godown.bottle2++;
            this.bottles.sealedbottle2++;
        }
    } catch (InterruptedException exc) {
        System.out.println(exc);
    }
    this.godownSem.release();
```

In the above snippet we have applied a lock on the godown. When a bottle comes out of the sealing unit and is already packed, we first acquire lock on the godown then increment the number of bottles in the godown if we don't acquire the lock before modifying the godown values, it might happen that the packing unit adds another bottle at the same time to the godown and thus the count may get incremented incorrectly.
We didn't use semaphore while modifying number of Sealed Bottles of type 1 and type 2 because they are modified only by the sealing unit.

```
try {
    this.sealingSem.acquire();
    this.sealBuffer.sealUnitBuffer.remove();
} catch (InterruptedException exc) {
    System.out.println(exc);
}
this.sealingSem.release();
```

We acquire a semaphore before accessing the sealing unit buffer because if both the units access the buffer at the same time then it might happen that the packing unit might want to insert one bottle into the sealing unit buffer, which may be full causing the packing unit to remain idle. But if at the same time the sealing unit removes a bottle, then the packing unit doesn't need to wait for it. So these two process should not happen simultaneously hence synchronization is done using semaphore.

Q. How would the program be affected if synchronization is not used?

Ans. If synchronization is not used then there will be problems in various critical sections of our code. A shared variable which is modified by both the sealing unit and the packing unit may end up having the wrong value if synchronization is not used. To look at this problem consider the following situation.
Suppose at time t=0 we have 3 bottles of type 1 in the unfinished tray and 0 bottles of type 2. Now at t=0, both the threads running both the units will wake up and find that the unfinished tray is not empty. If synchronization is not used, both of them will decrement the current value of the number of bottles of type 1 in the unfinished tray by 1 i.e to 2. But this is clearly wrong as the expected count would be 1. But if we use synchronization in the above situation, then either the sealing or the packing unit will acquire the semaphore on the unfinished tray and only then will it begin its execution. If it is not able to acquire the semaphore, then it will wait till the unit acquiring the semaphore does not finish its job.

Problem 3: Automatic Traffic Light System

Q. Do you think the concept of concurrency is applicable while implementing the program? If yes, why? Explain using code snippets.

Ans. Yes, the concept of concurrency is applicable here. In our implementation there are two threads running at a time. One is the main thread which runs the GUI while the other is the Timer Thread generated by the TimerTask class which is scheduled to run once every every second. This is necessary to ensure that the GUI is always responsive even as the scheduled data updates go on and update the GUI in turn. Once, we begin the simulation, we still accept user input for adding more cars to the Traffic Light System. We need to keep listening for new user input on a thread. Besides this each second we compute the new status of each car and traffic light and update the GUI accordingly. We need to ensure that both these tasks can run together in parallel and communicate (using shared data) any feedback that the other may require (in our case it is the queue of the newly added cars).
One can refer to the files “TrafficSystemGUI.java” and “Worker.java” to see the code run by these two threads and their inter-communication.

```
// Begins simulation of the traffic light system
public static void run_simulation(JTable vehicleTable, JTable lightTable, Queue <Car> new_cars, ReentrantLock lock){
    Timer timer = new Timer();
    // Creating an instance of task to be scheduled
    TimerTask task = new Worker(lightSE, lightWS, lightEW, vehicleTable, lightTable, Cars, new_cars, lock);

    // Scheduling the timer instance
    timer.schedule(task, 100, 1000);
}
```

Above is a snippet of code from the Main thread. Here the main thread spawns the Timer thread in response to the clicking of the “Run Simulation” button. A task is then assigned to this thread which is then scheduled to run in intervals of one second.

Q. Did you use synchronization in the problem? If yes, explain with code snippets

Ans. As mentioned in the previous answer, our implementation of the solution has two threads. One is the main thread which runs the GUI while the other is the Timer Thread generated by the TimerTask class which is scheduled to run once every every second. Whenever a user adds a car via user input when the simulation is already running, we have to store this in a queue which will be shared by both the threads. When the timer thread is activated again, it will remove all cars added in the previous second. To control access to this shared data structure we require proper synchronization. This was achieved using "***Reentrant Locks***".
The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.
As the name says, ReentrantLock allow threads to enter into lock on a resource more than once.

```
// Acquire lock to ensure synchronization while updating the queue shared by the Main thread and the TimerThread
lock.lock();
try{
    // Add new car to queue
    new_cars.add(car);
}
catch(Exception e){
    e.printStackTrace();
}
finally{
    //  Release lock
    lock.unlock();
}
```

Above is a snippet from the code of the main thread. Whenever the "Add car" button is clicked after the simulation has begun the above code is run. The lock is acquired, the car object is added to the shared queue and finally the lock is released.

```
// Acquire lock to ensure synchronization while updating the queue shared by the Main thread and the TimerThread
lock.lock();
try{
    // If any new cars have arrived in the last second, then update the cars arrived till now
    if(new_cars.size() != 0){
        while(new_cars.size() != 0){
            // Get the front of the queue and update array of arrived cars
            Car car = new_cars.peek();
            car.arrivalTime = this.currentTime;
            this.cars.add(car);

            // Remove car from queue
            new_cars.remove();
        }
    }
}
catch(Exception e){
    e.printStackTrace();
}
finally{
    // Unlock the lock
    lock.unlock();
}
```

Above is a snippet from the code of the Timer Thread. Each second it runs a method ***getNewCars()*** to fetch new cars entered by the user, if any.
It acquires the lock and checks if any new car has arrived in the previous second. If this is the case then it removes elements from the queue until it becomes empty.

Q. If instead of T-junction, if it would have been a four way junction (i.e. another way towards old SAC), will there be any change in the problem implementation? Write the pseudocode for the same.

Ans. If the problem were changed to be a 4-way junction, the total number of source-destination direction pairs will increase from 6 to 12. We would need 4 traffic signals to govern the junction. From each source direction there can be 3 different destination direction. Out of these 1 will still be free to pass for all the source directions. The remaining two will be then governed by their traffic light. We can implement the waiting scheme using two queues for the two possible destination directions. It would then depend on the problem specification if car will have to wait for 6s for the currently passing car to pass even if its destination direction is same. The cyclic time of the traffic light system will increase from 180s to 240s. Some

minor changes to the calculation of car departure times to conform with the new system will also be needed (for example, increasing the backoff time from 120 to 180 if a car misses the current Green signal).

Following is the code for the modified problem. The lines that had to be altered have been highlighted.

```
// Sets the departure time for a vehicle
   void set_departure_time(int activeLight, int currentTime){
      boolean check = false;

      // If car is destined for a direction not governed by a traffic light
      if(light == null){
         // Allow car to pass immediately
         departureTime = 0;
         status = "Passing";
         check = true;
         return;
      }

      // If there are no cars currently waiting at the light
      if(light.waitingCars == 0) {
         // If signal currently active is different from the car's signal
         if(activeLight != light.lightID){
            // Compute the offset to add to waiting time
            int difference;
            if(activeLight > light.lightID){
               difference = 4 + light.lightID - activeLight;
            }
            else difference = light.lightID - activeLight;
            // Schedule departure at the time when the light next becomes Green
            departureTime = difference*60 - currentTime;
         }
         else{
            // Find how much time has elapsed since the last car has run to check it if it has
passed
            if(currentTime-light.previousRunTime > 6){
               // If another car can be scheduled before the signal becomes Red, allow it to pass
               if(currentTime < 54){
                  light.previousRunTime = currentTime;
                  departureTime = -1;
```

```
                this.status = "Passing";
                check = true;
            }
            // Else schedule it to pass when the signal becomes Green again
            else{
                light.previousRunTime = (240-currentTime)%60;
                departureTime = 240-currentTime;
            }
        }
        // If another car is still passing
        else{
            // If another car can be scheduled before the signal becomes Red, schedule it to
pass when the
            // passing car has passed
            if(light.previousRunTime <= 48){
                departureTime = 6 - (currentTime - light.previousRunTime);
            }
            // Else schedule it to leave when the signal next becomes Green
            else{
                departureTime = 240 - currentTime;
            }
        }
    }
    // Update the last waiting car time
    light.lastWaitingTime = departureTime + 6;
    if(!check) light.waitingCars++;
  }
  else {
    int waitingTime = light.lastWaitingTime;
    // If last waiting car won't allow for another car to pass before the signal turns "Red"
    // schedule departure when the light next turns Green
    if(light.lastWaitingTime%60 > 54 || light.lastWaitingTime > 54){
        waitingTime += 240 - light.lastWaitingTime%60;
    }
    else if (light.lastWaitingTime != 0 && light.lastWaitingTime%60 == 0) {
        waitingTime += 180;
    }
    // Set departure time and update last waiting car time to the time when this car would
have passed
    departureTime = waitingTime;
    light.lastWaitingTime = departureTime + 6;
    light.waitingCars++;
  }
```