CS 431
Programming Languages Lab

Assignment 1 Report
Concurrent Programming

Group 35
Abhinav Mishra (160101005)
Nitin Kedia (160101048)

TABLE OF CONTENTS

1.MERCHANDISE SALE FOR ALCHERINGA 2020

Our implementation consists of 4 classes which are mentioned as follows:

1. Order.java: This class represents an order placed for any particular merchandise item.
2. InventoryItem.java: This class represents a particular item present in the Inventory (small T-shirt, medium T-shirt, large T-shirt or cap).
3. Inventory.java: This class represents an inventory containing different item types of merchandise.
4. MerchandiseStore.java: This class represents our store. It carries out the function of accepting and servicing orders.

1.1 CONCURRENCY & SYNCHRONIZATION TECHNIQUE AND IMPLEMENTATION (WITH CODE SNIPPETS)

For concurrency, each entered order is serviced using a separate thread, at a time 4 (equal to the number of cores available) threads are processed together. For this, we used ExecutorService available in java to create a thread pool of size 4 in the main() function of MerchandiseStore.java.

```
/* Creating an executor service to service multiple concurrent
       threads (at max equal to pool size here equal to 4) */
       ExecutorService threadPool = Executors.newFixedThreadPool(4);

       /* Iterating through the list of orders ,creating a thread
        for each order and adding it to executor service threadPool*/
       for(int i = 0; i < totalOrders; i++){
           Runnable orderProcessor = new OrderProcessor(orderList[i], inventory);
           threadPool.execute(orderProcessor);
       }

       // Shutting Down the executor service
       threadPool.shutdown();
```

Here we can see an instance of ExecutorService of pool size 4 is created with a variable name “threadPool”. For each order present in the list of orders given here stored in the array orderList a thread is created and added in the threadPool. Even though we are adding each order at once, ExecutorService ensures the number of concurrent threads never exceeds pool size. After execution of all the orders the threadPool is shutdown.

A need for synchronization arises when multiple concurrent threads try to buy the same inventory item. If synchronization is not applied this could lead to a wrong value of stocks left in the inventory for that item.

Therefore for each inventory item, we applied synchronization using the “synchronized” keyword on the processOrder() function of InventoryItem to make that block of the program synchronized that is at any time only one thread is allowed to access the particular function.

```
/* Following function processes any order for the inventory item by
   checking if the demand is less than or equal to the current stock,
   if it is so the order is successful the current stock value is updated
   else order failure is notified.
   The synchronized keyword is used to allow only one thread to access it
   at any particular time */

   public synchronized void processOrder(Order order) {
        int requiredUnits = order.requiredUnits;
        int orderIndex = order.orderNumber;

        // Checking if order can be serviced or not
        if (requiredUnits > currentStock) {
            System.out.println("Order " + orderIndex + " failed.");
            return;
        }

        // Updating the current stock value.
        currentStock = currentStock - requiredUnits;
        String overallStock = Inventory.getOverallStock();
        System.out.println("Order " + orderIndex + " successful. " + overallStock);
   }
```

(Please see the next page for Q2)

2. COLD DRINK MANUFACTURING

Our implementation consists of 8 classes which are mentioned below:

1. Bottle.java: This class represents a particular bottle by its type, state, and delivery time.
2. Godown.java: This class represents the godown of the manufacturing factory.
3. UnfinishedTray.java: This class represents the unfinished common Tray from which both packaging and sealing unit can take bottles.
4. SealingTray.java: This class represents the tray present in the sealing unit.
5. PackagingTray.java: This class represents the trays present in the packaging unit.
6. SealingUnit.java: This class represents a sealing unit of the manufacturing factory with functions for taking a new bottle from sealing tray or unfinished tray and carrying out sealing and transferring the bottle to the packaging unit tray or godown.
7. PackagingUnit.java: This class represents a packaging unit of the manufacturing factory with functions for taking a new bottle from the packaging tray or unfinished tray and carrying out packaging and transferring the bottle to the sealing unit tray or godown.
8. ColdDrinkManufacturing.java: This class represents a Cold Drink Manufacturing factory by creating the objects of required classes and taking in input the initial number of bottles and time at which the status has to be displayed.

2.1 IMPORTANCE OF CONCURRENT PROGRAMMING

The concept of concurrent programming is applicable here because for any manufacturing factory the sealing unit and processing unit keeps running concurrently only the exchange of bottles takes place between these two units.
This will minimize idle time. Also, both the units access the same unfinished tray and godown, hence synchronization has to maintained for the proper function of the program.

For any tray be it the unfinished tray, packaging tray or sealing tray it will be accessed by multiple threads during the execution of the program. For example, the unfinished tray can be accessed by the sealing unit and the packaging unit concurrently whereas in the packaging unit its tray can be accessed by the unit itself to pick up the next bottle to be packaged or by the sealing unit to transfer its sealed bottle to the packaging tray. Similar is the case for the sealing unit. Hence we used a reentrant lock for each tray to achieve synchronization.

2.2 CONCURRENCY AND SYNCHRONIZATION IN PROGRAM (WITH CODE SNIPPETS)

Reentrant Lock is used for providing synchronization in the program.

Unfinished Tray:

```
/*This function takes the bottle preference from the unit and
   returns that bottle type is present else return other bottle type
```

```
  A lock is applied to achieve synchronization so that at a time
  only one unit access the tray at time */
  public int takeBottle(int bottleType) {
      // acquire lock
      retrievalLock.lock();

      /* if requested bottle type is not available update and
      return other type after releasing the lock */
      if (!isBottleAvailable(bottleType)) {
          int otherBottleType = (bottleType + 1) % 2;

          // if no bottle is available return -1 after releasing lock
          if (!isBottleAvailable(otherBottleType)) {
              retrievalLock.unlock();
              return -1;
          }

          decrementBottleCount(otherBottleType);
          retrievalLock.unlock();
          return otherBottleType;
      }

      /* if requested bottle type is available update and
      return bottle type after releasing the lock*/
      decrementBottleCount(bottleType);
      retrievalLock.unlock();
      return bottleType;
  }
```

The lock has to be acquired at the start of the function, ensuring at a particular time only one thread/unit is able to perform the actions in the function. Before returning the allotted bottle type the lock is released which can be acquired by other unit waiting upon the lock, ensuring synchronization.

Sealing Tray:

```
  /*This function returns the bottle at the front of the queue if
  the queue is not empty else it returns -1
  A lock is applied to achieve synchronization so that at a time
  only one unit access the tray at time */
  public int takeBottle() {
      lock.lock(); // acquire lock
      // if queue empty return -1 after releasing lock
      if (isTrayEmpty()) {
            lock.unlock();
            return -1;
        }
        // else return the front element of queue
        else {
            int newBottle = queue.peek();
            queue.remove();
            lock.unlock();
            return newBottle;
        }
    }
/*This function is called to store a packaged bottle in the tray
  if the queue is not full else it returns false
  A lock is applied to achieve synchronization so that at a time
  only one unit access the tray at time */
   public boolean storeBottle(int bottleType) {
        // acquire lock
        lock.lock();

        // if tray is full return false after releasing lock
        if (isTrayFull()) {
            lock.unlock();
            return false;
        }
        // else add to queue and return true after releasing lock
        else {
            queue.add(bottleType);
            lock.unlock();
            return true;
        }
    }
```

In the case of the sealing tray, the lock has to be acquired at the start of the function, ensuring at a particular time either the sealing unit is itself able to extract a bottle from the tray or the packaging unit is able to put a packaged bottle in the tray. The lock is released before the return statement is executed

Packaging Tray:

```
  /* This function takes the bottle preference from the unit and
  returns that bottle type is present else return other bottle types
  A lock is applied to achieve synchronization so that at a time
  only one unit access the tray at a time. */
```

```
    public int takeBottle(int bottleType) {
        acquireLock();
        /* if requested bottle type is not available check if the
        the other type is available */
        if (!isBottleAvailable(bottleType)) {
            int otherBottleType = (bottleType + 1) % 2;
            // If no bottle is available return -1 after releasing lock
            if (!isBottleAvailable(otherBottleType)) {
                releaseLock();
                return -1;
            }
            /* else decrement the count of other bottle type present and
            return after releasing the lock*/
            decrementBottleCount(otherBottleType);
            releaseLock();
            return otherBottleType;
        }
        /* else decrement number of bottle type present and
            return after releasing the lock */
        decrementBottleCount(bottleType);
        releaseLock();
        return bottleType;
    }
```
Similarly, in the packaging unit, the lock has to be acquired at the start of the function, ensuring at a particular time either the packaging unit is itself able to extract a bottle from the tray or the sealing unit is able to put a sealed bottle in the tray. Again before returning the bottle, the lock is released to allow threads waiting upon it to continue.
The packaging unit also has a function for storing a bottle in the queue as the sealing unit.

Another point in which synchronization is required is when a bottle is transferred to godown as at any time the concurrent threads of packaging and the sealing unit can access the godown to transfer the completed bottle it may lead to inconsistency, hence we ensure synchronization using a reentrant lock.
Godown:
```
/* This function stores the bottle in the godown by
    incrementing the bottle count in the godown.
    A lock is applied to achieve synchronization so that at a time
    only one unit access the tray at time */
    public boolean storeBottle(int bottleType) {
        storageLock.lock(); // acquire lock
        // increment corresponding bottle count
        if (bottleType == 1) {
            bottle1Count++;
        }
        else {
            bottle2Count++;
        }
        storageLock.unlock(); // release lock
        return true;
    }
```

The lock is acquired at the start of function therefore only one thread will perform the actions in the function while others will keep waiting upon them. At the very end of function after updating the bottle count lock is released allowing waiting threads to continue.

2.3 EFFECT ON PROGRAM IF SYNCHRONIZATION IS NOT USED

If synchronization is not used it can lead to erroneous program execution. For example, if no synchronization is done on the unfinished tray it may happen that a particular bottle is taken up by both the packaging and sealing unit as any of these concurrent threads can access the tray which clearly violates our problem scenario.
If synchronization is not done on the packaging or sealing tray, it may happen that both of the concurrent threads
update the bottle count in the tray which will lead to inconsistency. Similarly, if no synchronization is done for godown we can consider the case in which both concurrent threads enter the function at the same time and updates the count. In this particular case, it may happen that both updates occur concurrently and thus instead of an increment of 2 only an increment of 1 is observed.
(Please see next page for Q3)

3. AUTOMATIC TRAFFIC LIGHT SYSTEM

Our implementation consists of 5 classes which are mentioned below:

1. Vehicle.java: This class represents a particular vehicle by its number, source, destination and passage time.
2. TrafficSignal.java : This class represents a traffic signal by its number, next passage time for a vehicle under its jurisdiction
3. VehicleStatusUpdate.java: This class updates the vehicle and traffic signal status in the GUI.
4. AddVehicleWorker.java: This class adds a new vehicle request in the vehicle table
5. TrafficSystemGUI.java: This is the main class an instance of this class represents a junction and its management scenario.

3.1 IMPORTANCE OF CONCURRENT PROGRAMMING

A. Yes, Concurrency is essential for implementing this program because the GUI needs to be responsive at all times. Since the Event Dispatching Thread (EDT) should not be solely responsible for every task, we have identified two major tasks need to be done in this system and use SwingWorkers to carry them out. Main thread just spawns a worker thread, it does not get blocked on them, a worker executes its tasks in a separate thread.

   - Update remaining waiting time of the vehicles as well as the each traffic light color every second. This is done by the *VehicleStatusUpdate* worker. It loops over the current waiting times and decrements them if the vehicle is not already passed. Instead of clearing the whole table and recalculate the rows, we update the waiting time of each row for performance reasons.

```
  @Override
    protected void done() {
        updateVehicleTable();
        updateTrafficSignalTable();
        trafficSystemGUI.setInvalidDirectionLabel(false);
    }
```

   - For every incoming vehicle assign next passage time and display it in a new row. This is done by a *AddVehicleWorker* which is invoked for every vehicle entered by the user. Since the system supports batch inputs, multiple *AddVehicleWorker* threads can run concurrently.

```
     @Override
    protected Vehicle doInBackground() throws Exception {
        // Generate vehicle object by using validated use input
        if (!isFlowValid(sourceDirection, destinationDirection)) {
            return null;
        }

        int newVehicleId = trafficSystemGUI.getNewVehicleId();
        int trafficLightNumber = getTrafficSignalNumber(sourceDirection,
destinationDirection);
        // Allot passing time for this vehicle and calc waiting time from that
        int passageTime = trafficSystemGUI.getNextPassageTime(trafficLightNumber);
        int waitingTime = passageTime - trafficSystemGUI.currentTime;
        String vehicleStatus = getVehicleStatus(passageTime);

        Vehicle newVehicle = new Vehicle(newVehicleId, sourceDirection,
destinationDirection, vehicleStatus, waitingTime);
        return newVehicle;
    }
    @Override
    protected void done() {
        try {
            Vehicle newVehicle = get();
            if (newVehicle == null) {
                // invalid user input
                trafficSystemGUI.setInvalidDirectionLabel(true);
                return;
            }
            // get synchronized access to the vehicle table data and
            // append this new vehicle
            trafficSystemGUI.acquireSemaphore();
            trafficSystemGUI.vehicleModel.addRow(newVehicle.getVehicleStatus());
            trafficSystemGUI.setInvalidDirectionLabel(false);
            trafficSystemGUI.releaseSemaphore();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
```

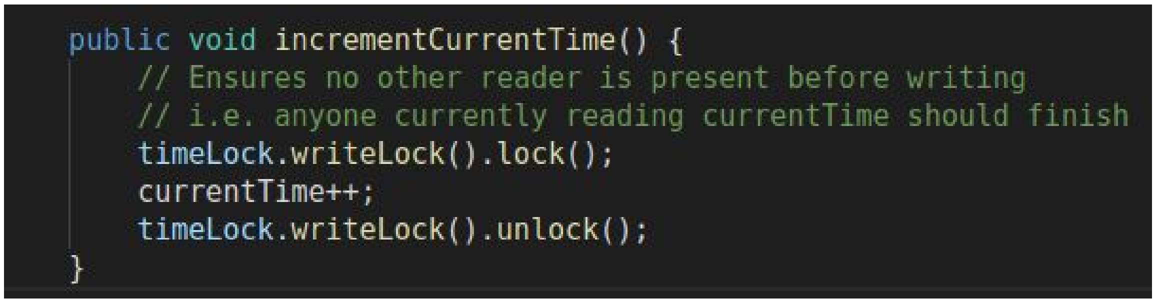
3.2 SYNCHRONIZATION USED IN PROGRAM (WITH CODE SNIPPETS)

B. In the previous part we explained that concurrent workers are used in the system. We highlight three synchronisation constructs employed to prevent inconsistent access to shared data below:

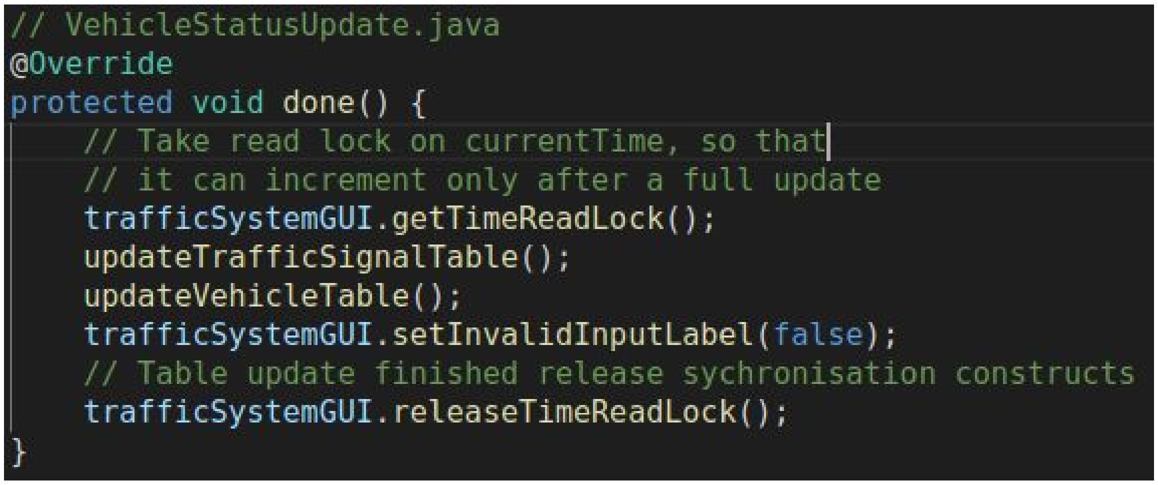
- *currentTime*: At the heart of the system is the currentTime member variable of the uppermost level class TrafficSystemGUI. This is increment after each second using a Timer. This variable is **read** for the following purposes:
    i. After each increment, a worker updates the waiting times and traffic light color. This worker must finish before the next increment.

ii. Suppose a new vehicle comes into the system, the value of currentTime at that time should be used for alloting passage time for this vehicle. In between this allotment, the currentTime

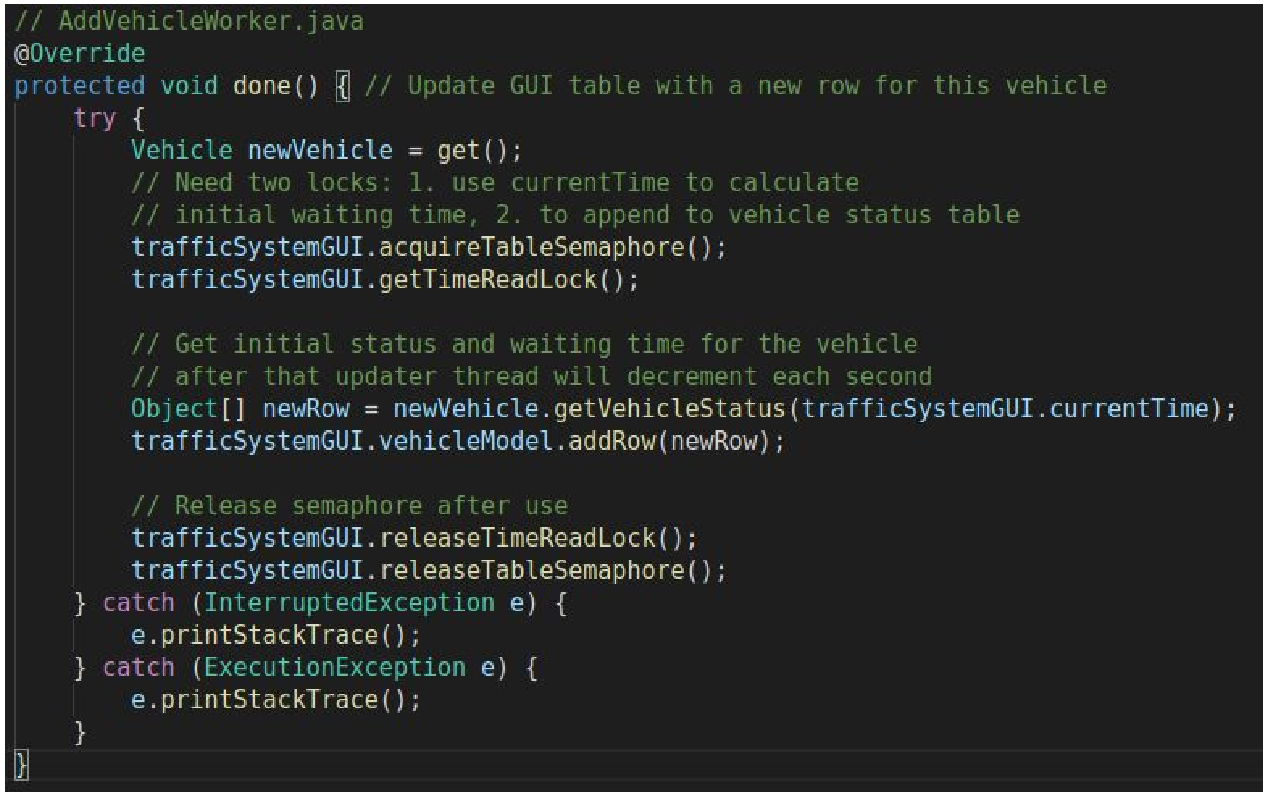
should not change.
Thus currentTime is protected by a Reader's Writer's lock. Only one writer is present (the incrementer). *ReadWriteLock* ensures that when a write is happening no other reader or writer is present. Also, multiple readers can be present simultaneously because currentTime is used for multiple calculations and using a simple lock would decrease concurrency. In both cases i and ii, acquiring read locks prevents currentTime++ from happening till they are done.



```
    public void incrementCurrentTime() {
        // Ensures no other reader is present before writing
        // i.e. anyone currently reading currentTime should finish
        timeLock.writeLock().lock();
        currentTime++;
        timeLock.writeLock().unlock();
    }
```

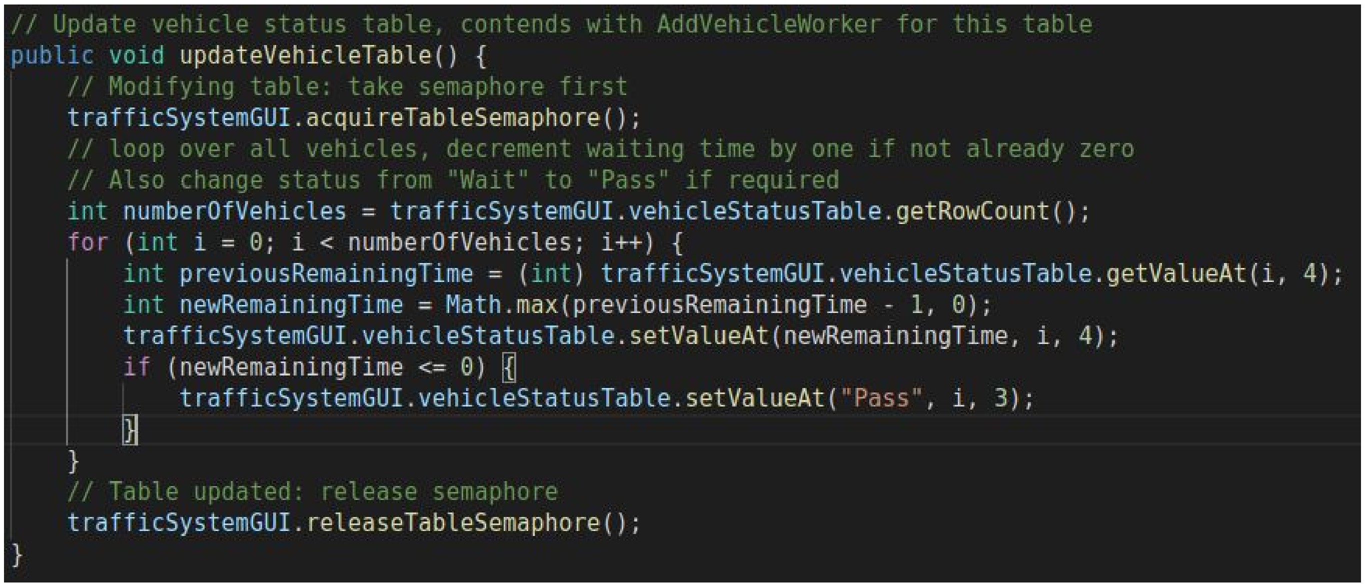


```
// VehicleStatusUpdate.java
@Override
protected void done() {
    // Take read lock on currentTime, so that
    // it can increment only after a full update
    trafficSystemGUI.getTimeReadLock();
    updateTrafficSignalTable();
    updateVehicleTable();
    trafficSystemGUI.setInvalidInputLabel(false);
    // Table update finished release sychronisation constructs
    trafficSystemGUI.releaseTimeReadLock();
}
```



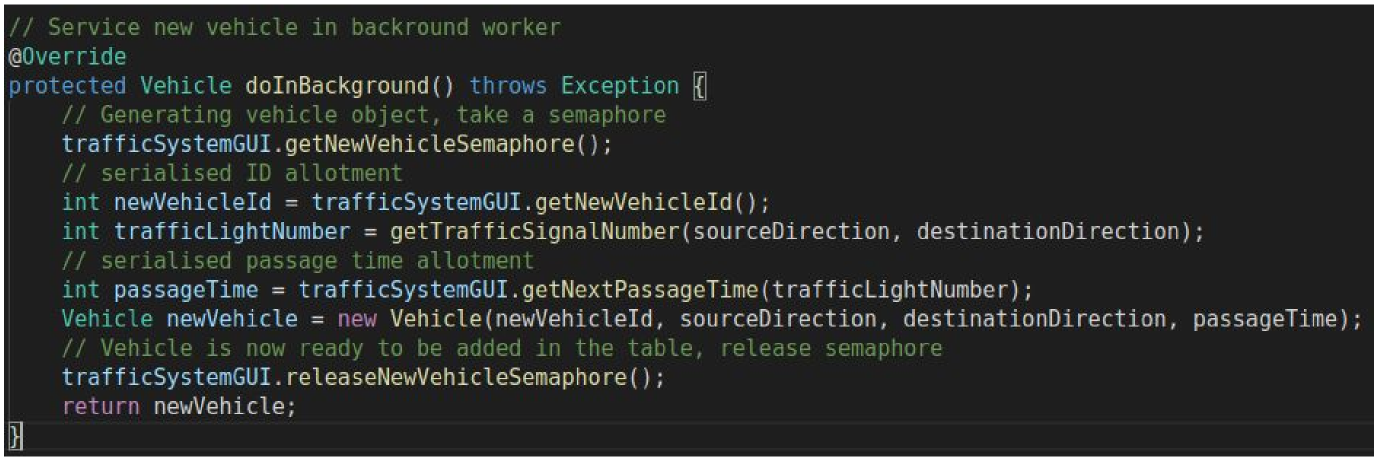
```java
// AddVehicleWorker.java
@Override
protected void done() { // Update GUI table with a new row for this vehicle
    try {
        Vehicle newVehicle = get();
        // Need two locks: 1. use currentTime to calculate
        // initial waiting time, 2. to append to vehicle status table
        trafficSystemGUI.acquireTableSemaphore();
        trafficSystemGUI.getTimeReadLock();

        // Get initial status and waiting time for the vehicle
        // after that updater thread will decrement each second
        Object[] newRow = newVehicle.getVehicleStatus(trafficSystemGUI.currentTime);
        trafficSystemGUI.vehicleModel.addRow(newRow);

        // Release semaphore after use
        trafficSystemGUI.releaseTimeReadLock();
        trafficSystemGUI.releaseTableSemaphore();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
```

- *vehicleStatusTable:* vehicleStatusUpdate worker and AddVehicleWorker(s) both modify this table. So, we use a semaphore (with permit = 1) so that only one of them can access the table in a given time.



```java
// Update vehicle status table, contends with AddVehicleWorker for this table
public void updateVehicleTable() {
    // Modifying table: take semaphore first
    trafficSystemGUI.acquireTableSemaphore();
    // loop over all vehicles, decrement waiting time by one if not already zero
    // Also change status from "Wait" to "Pass" if required
    int numberOfVehicles = trafficSystemGUI.vehicleStatusTable.getRowCount();
    for (int i = 0; i < numberOfVehicles; i++) {
        int previousRemainingTime = (int) trafficSystemGUI.vehicleStatusTable.getValueAt(i, 4);
        int newRemainingTime = Math.max(previousRemainingTime - 1, 0);
        trafficSystemGUI.vehicleStatusTable.setValueAt(newRemainingTime, i, 4);
        if (newRemainingTime <= 0) {
            trafficSystemGUI.vehicleStatusTable.setValueAt("Pass", i, 3);
        }
    }
    // Table updated: release semaphore
    trafficSystemGUI.releaseTableSemaphore();
}
```

- Allotment of *nextPassageTime* and *newVehicleId:* Request to add another vehicle may come even when one vehicle is currently being added to the system. There should be no contention in such cases. If these two vehicles map to the same traffic light, the former must be alloted passing time before the other. Thus multiple *AddVehicleWorkers* need to get a *newVehicleSemaphore* first then carry out the Id and time allotment.



```
// Service new vehicle in backround worker
@Override
protected Vehicle doInBackground() throws Exception {
    // Generating vehicle object, take a semaphore
    trafficSystemGUI.getNewVehicleSemaphore();
    // serialised ID allotment
    int newVehicleId = trafficSystemGUI.getNewVehicleId();
    int trafficLightNumber = getTrafficSignalNumber(sourceDirection, destinationDirection);
    // serialised passage time allotment
    int passageTime = trafficSystemGUI.getNextPassageTime(trafficLightNumber);
    Vehicle newVehicle = new Vehicle(newVehicleId, sourceDirection, destinationDirection, passageTime);
    // Vehicle is now ready to be added in the table, release semaphore
    trafficSystemGUI.releaseNewVehicleSemaphore();
    return newVehicle;
}
```

3.3 PSEUDOCODE FOR 4-WAY JUNCTION

C. If there would have been a four-way junction instead of T-junction following distinctions will be observed:

- Since the north direction is also added the number of conflicting directions pairs will increase significantly. Current only 3 pairs South-East, West-South and East-West requires synchronization but after adding the North direction the number of conflicting pairs will increase to 8 with only South-East, West-North, North-East and East-South not causing any conflict.
- Therefore we will require 4 traffic signal lights each corresponding to one source direction. When a traffic signal light is green only vehicles with source direction the same as the green traffic signal (there will be two such flows) and those vehicles with source-destination pair among the 4 pairs that do not cause conflict will be allowed to move.
- The time duration of the entire cycle will increase from 180 to 240 and therefore the formula for calculation of next passage time will also change by replacing duration of the cycle from 180 to 240.
- The infrastructure for updating the tables every second and manage incoming vehicles will remain the same.

Pseudocode (We highlight the classes and changes to made in them):

```
TrafficSystemGUI
    // In the constructor 4 traffic signal objects will be created each for one direction
    T1 = new TrafficSignal(2);       // For South Direction
    T2 = new TrafficSignal(4);       // West Direction
    T3 = new TrafficSignal(3);       // East Direction
    T4 = new TrafficSignal(1);       // North Direction
    // 12 input text fields for batch input will be required to take vehicles from all 12 flows
```

AddVehicleWorker

// Maps each vehicle to its traffic light (none is allotted for the non-conflicting 4 directions)

```
int getTrafficSignalNumber(source, destination) {
    // source and destination are directions
    // ensure source != destination
    switch (source) {
        case (South):
            if (destination is North or East) return T1;
            if (destination is West) can directly pass;
        case (West):
            if (destination is East or South) return T2;
            if (destination is North) can directly pass;
        case (East):
            if (destination is North or West) return T3;
            if (destination is South) can directly pass;
        case (North):
            if (destination is West or South) return T4;
            if (destination is East) can directly pass
        default:
            break;
    }
}
```

TableUpdater

```
void updateTrafficLightStatusTable() {
    currentCountDown = currentTime % 240;
    activeLight = currentCountDown / 60;
    for each trafficLight in system {
        if (trafficLight == activeLight) {
            color = green;
            remainingTime = (60 * activeLight) - currentCountDown;
        }
        else {
            color = red;
            remainingTime = none;
        }
        // also set these values in the table
    }
}
```

TrafficSignal

```
//Function to allot passage time for a vehicle
int getNextPassageTime(currentTime) {
     trafficLightDuration = 60;  // Time allotted to a single traffic light
     cycleDuration = 240;       // Number of Traffic Lights(4) * 60
     if  (currentTime > nextPassageTime) {
          // Time has passed new passage time needs to be calculated
          if (current Time lies in the time frame of this traffic signal
     and there is more than 6 sec left) {
               nextPassageTime = currentTime;
          }
          else {
               // next passage time will be start time for this traffic light in
     next cycle
               currentCycleNumber = currentTime / cycleDuration;
               nextPassageTime = (currentCycleNumber + 1) * cycleDuration +
               (trafficLightNumber - 1) * trafficLightDuration;
          }

          allottedTime = nextPassageTime;
          UPDATE_NEXT_PASSAGE_TIME();
          return allottedTime;
          }
     }

/* Update nextPassageTime which will be allotted to new vehicle which comes with
same source direction */
void updateNextPassageTime() {
          currentPassageTime = nextPassageTime;
          nextPassageTime  = currentPassageTime + 6;

          if (nextPassageTime does not lie in the time frame of this traffic
signal or time left is less than 6 sec) {
               // next passage time will be start time for this traffic light in
     next cycle
               currentCycleNumber = currentPassageTime / cycleDuration;
               nextPassageTime = (currentCycleNumber + 1) * cycleDuration +
               (trafficLightNumber - 1) * trafficLightDuration;
          }
          return;
     }
```