Programming Lab (CS431)

Assignment 1
Concurrency

Apurva N Saraogi (160101013)
Divyansh Sharma (160101026)

Q1.

As we don't need synchronization in each and every line of the code base, and only require in some part, we have used the **synchronized block** technique to minimize the operations in the critical section of the code.

```
synchronized (lock[type]){
    // Check if given order is available in the inventory.
    if (isAvailable(details)) {
        // It is available. So decrease the quantity from the inventory followed by showing message
        // order successful and print the inventory.
        qty[type] -= details.getQuantity();
        printInventoryWithMessage(String.format("Order %d successful.", details.getId()));
    } else {
        // It is not available.
        // Shows a message informing order failed and print the inventory.
        printInventoryWithMessage(String.format("Order %d failed.", details.getId()));
    }
}
```

First, the input is taken from the file mentioned in *Constants.java* from which the input is read and threads for each order are created. After creation inventory is initialized followed by starting all the threads. Each thread calls a *transact* function of the inventory passing its order details. The *transact* function has the synchronized block whose snipped is above.

```
for (int i = 0; i < totalOrders; i++) {
    // Read the next line contains the data in the format "id type qty" so will
    // split it and create a new order object and add it to the respective thread.
    String[] details = (reader.readLine().trim()).split("\\s+");
    threads[i] = new Serve();
    threads[i].details = new Order(Integer.parseInt(details[0]), details[1].charAt(0),
            Integer.parseInt(details[2]));
}

// Create the inventory.
Inventory inventory = new Inventory(quantity);

// Initially print the inventory details for the first user to see it.
inventory.printInventoryWithMessage(null);

// Start all the created threads.
for (int i = 0; i < totalOrders; i++) {
    threads[i].start();
}
```

Q2.a) Since the packager and sealer are working independently and parallelly, it is important to give them independent threads for execution as it would be difficult to represent their working in sequential code.
For reference, consider the case when the packager takes input from completely filled tray 1 and at the same time sealer wants to push a bottle into tray 1 of the packager.
Hence 4 threads are made to cater to concurrency

```
// thread for taking packaged bottles to sealer.
PackagerToSealerThread t1 = new PackagerToSealerThread(packager, sealer, time, finished);
// thread for taking sealed bottles to packager.
SealerToPackagerThread t2 = new SealerToPackagerThread(packager, sealer, time, finished);
```

```
// thread for taking input into packager from tray.
TrayToPackagerThread t3 = new TrayToPackagerThread(packager, time, unfinished);
// thread for taking input into sealer from tray.
TrayToSealerThread t4 = new TrayToSealerThread(sealer, time, unfinished);
```

Also to satisfy the order of operation, there is a sleep method to let the first two threads complete first.

```
// sleeping to let the threads complete.
try {
    Thread.sleep(100);
} catch (InterruptedException ex) {
    Thread.currentThread().interrupt();
}
```

b) Semaphores are used as synchronization technique to prevent the case when both package and sealer try to take bottle from unfinished tray at same time or try to dump bottle into finished tray at same time.
Semaphores are introduced in unfinished tray and finished tray.

```
// semaphore lock for unfinished tray.
Semaphore sema = new Semaphore(1);
```

```
try {
    sema.acquire();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

c) If synchronization is not implemented, very elementary problems will arise. Consider the case when there is only one bottle left in the unfinished tray and both packager and sealer attempt to take it at same time. Without synchronization they will be able to get a copy of the bottle each. But we need only one unit to have access to the bottle. Semaphore of size 1 enables only 1 thread to have access to the section of code.

Q3.a)

Concurrency plays an important role in the implemented application. Take an example, consider a huge number of cars are present, so updating time for each of the car in User Interface from the main thread will take a lot of time. Instead of making Main Thread wait for completing the update of entire tables, we create and dispatch threads that are responsible for updating the data for each car, as well as for the traffic signals. The global timer of the program runs on a separate thread waking again at intervals of 1 second. Concurrency helps in putting off the load at other threads and saving time on the current thread which is necessary when something can be done in background while is done in foreground.

```
      // Schedule it to run at every 1 second with delay of 0 seconds.
    timer.schedule(new TimerTask() {
        @Override
        public void run() {
            // increment the current time.
            currentTime++;
            // Update next time for the current signal.
            updateNextTime(currentSignal());
            // Refresh the user interface by updating all values in the tables.
            UserInterface.reDraw();
        }}, 0, 1000);

// This method is invoked every one second, creates threads and dispatches them.
public static void reDraw() {
    // If the window is not yet initialized there is nothing to redraw.
    if (!isInitialized) return;
    // Updating Car Data. It only updates those which are either in Waiting or Crossing state.
    for (int i = 0; i < Main.cars.size(); i++) {
        if (Main.isPassed.get(i)) continue;
        final int rowNumber = i;
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                updateCarData(rowNumber);
            }
        });
    }
    // Update the traffic signals.
    for (int i = 0; i < Main.signals.length; i++) {
        final int rowNumber = i;
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                updateTrafficData(rowNumber);
            }
        });
    }
}
```

b)

Synchronization is necessary in this problem to make sure that no two cars are crossing on the same path together no matter how small difference is there in their arrival times. To handle this we have a function that computes when a car arrives what is the time it is going to leave. The function for it is displayed below. There are two possible cases i.e., the car can go in this turn or it has to wait for the next time the light goes green which are handled by the code. There are two more similar functions like this in the main code base.

```
// Updates the next time once a car arrives.
public static void updateNextTimeCar(int signal) {
    // So overhere we check if a new car comes at this instant will it be able to
    // cross or will it have to wait for two cycles.
    // As mentioned earlier next is when the road will be free. So we will check
    // whether Current Signal Time + Crossing Time + Crossing Time is in the same
    // signal if yes
    // then increment by 1 otherwise we need to set the next time to the time when
    // the current signal will be green again.
    if (next[signal] % Constants.TIME_PER_SIGNAL + Constants.CROSSING_TIME *
    (Constants.NUMBER_OF_SIGNALS - 1) <= Constants.TIME_PER_SIGNAL) {
        next[signal] += Constants.CROSSING_TIME;
    } else {
        next[signal] = (next[signal] / Constants.TIME_PER_SIGNAL + Constants.NUMBER_OF_SIGNALS)
        * Constants.TIME_PER_SIGNAL;
    }
}
```

c)

When it is a 4-way crossroad the car going from W to N, N to E, E to S and S to W won't need to wait for signals applying similar logic as of this question, while the all cars going on a different path from these will have to wait for signals. Now when T1 is Green cars will go from South to East as well as South to North. Similarly for other signals. There are not much changes to be done in the code for making it adapt to 4 signals. The suggested changes are written in the comments of the snippet.

```
    // Constants.java increase this to 4
    public static final int NUMBER_OF_SIGNALS = 3;

    // Main.java add Constants.TIME_PER_SIGNAL * 3 as the 4th element and add one more zero at the end
    public static int next[] = { 0, Constants.TIME_PER_SIGNAL, Constants.TIME_PER_SIGNAL * 2 , 0,0,0};
    // Array containing the objects for each signal. Add a new signal T4.
    public static TrafficSignal signals[] = { new TrafficSignal("T1"), new TrafficSignal("T2"), new
    TrafficSignal("T3") };
```

```
// User Interface.java. We need to add North to both source and destiantion radio buttons.
sourceRadioButton1 = new JRadioButton("South");
sourceRadioButton2 = new JRadioButton("West");
sourceRadioButton3 = new JRadioButton("East");

// And according to a selection of buttons add a new car. The first variable in addNewCar method
// is type, it must be 0 for all cars waiting for signal T1, similarly for other 3.
if (sourceRadioButton1.isSelected() && destinationRadioButton3.isSelected()) {
    addEntry(Main.addNewCar(0, "South", "East"));
} else if (sourceRadioButton1.isSelected() && destinationRadioButton4.isSelected()) {
    addEntry(Main.addNewCar(0, "South", "North"));
} else if (sourceRadioButton2.isSelected() && destinationRadioButton1.isSelected()) {
    addEntry(Main.addNewCar(1, "West", "South"));
...
```