PROGRAMMING LAB
ASSIGNMENT - 1
CONCURRENT PROGRAMMING

Submitted By :-
Kanika Agarwal (160101038)
Paranjay Bagga (160101050)

Problem 1: Merchandise sale for Alcheringa 2020

Synchronization Technique Used **: Semaphore**

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. Java provide **Semaphore** class that implements this mechanism.

Here, a binary semaphore is used which has only 2 states - resource is available or not available. This acts as a lock and uses **acquire()** and **release()** methods to guard the critical section.

Implementation :

We have made a new thread to run for every order with a thread-pool size of 4 signifying that max 4 threads can run parallely a time.

1. Updating counts for each merchandise types :-
   We have used synchronization(semaphore) for updating count of each merchandise type(S,M,L,C) meaning that two or more threads cannot update the respective type counts simultaneously. Only one thread is allowed to do so at a time.

```
/* Updates inventory when a small T shirt is ordred */
void updateS(Inventory inv, int orderNo, char orderType, int quantity){
    int cntSTees = 0, cntMTees = 0, cntLTees = 0, cntCaps = 0;
    try{ semS.acquire();
        if(countSTees > inv.countSTees){
            countSTees = inv.countSTees;
        }
        cntSTees = countSTees;
        cntMTees = countMTees;
        cntLTees = countLTees;
        cntCaps = countCaps;
    }
    catch(Exception e){ System.out.println("Exception has occured."); }
    int result = 0 ;
    if(quantity > countSTees) result = 1 ;
    else countSTees -= quantity ;
    inv.countSTees = countSTees;
    this.dispInventory(orderNo, result, countSTees, cntMTees, cntLTees, cntCaps) ;
    semS.release();
}
```

2. Displaying the inventory :-
   For showing the corresponding updated inventory stock alongwith too, a semaphore is used so that only 1 thread can print the output as a whole at a time.

```
/* Displays contents of inventory after acquiring a semaphore */
void dispInventory(int orderNo, int result, int cntSTees, int cntMTees, int cntLTees, int cntCaps) {
    try{ sem.acquire(); }
    catch(Exception e){ System.out.println("Exception has occured."); }
    String success ;
    if (result!=0)
        success = "failed" ;
    else
        success = "successful" ;
    System.out.println("") ;
    if(orderNo!=0)
        System.out.println("Order " + orderNo + " is " + success) ;
    System.out.println("Inventory --") ;
    System.out.println("S\tM\tL\tC") ;
    String data = String.format("%d\t%d\t%d\t%d", cntSTees, cntMTees, cntLTees, cntCaps) ;
    System.out.println(data) ;
    sem.release();
}
```

3. Getting the current inventory :-
   Also, for getting the current inventory object, synchronisation is used, so that no 2 threads access the inventory simultaneously & later get any inconsistent data.

Problem 2: Cold Drink Manufacturing

1. **Importance of Concurrent Programming** : The Manufacturing Unit has two processing units - the packaging and sealing units, which can and must be used concurrently so that both of them should incur minimum idle time (also suggested by the question statement). Without concurrency, the efficiency of the whole Unit will be severely hampered as every time, one unit would have to wait for other to finish processing before it could itself resume. Thus, then tasks would not be run parallely but serially.
2. **Handling Concurrency & Synchronization** :
   A. **Concurrency** :
      For both packaging & sealing units, a new Thread is generated every time unit (second) and processes the bottles B1/B2 as per requirement for both of them. These 2 threads run concurrently and pack/seal bottle from the various trays respectively.

```
/* two threads initialised, one each for packaging unit and sealing unit */
sUnit = new ProcessThread(false, unfinished, packB1Tray, packB2Tray, sealTray, godown, sealingUnit, packagingUnit) ;
pUnit = new ProcessThread(true, unfinished, packB1Tray, packB2Tray, sealTray, godown, sealingUnit, packagingUnit) ;
Thread tSeal = new Thread(sUnit, "Thread_Seal");
Thread tPack = new Thread(pUnit, "Thread_Pack");
```

```
/* thread either seals bottle or packages it */
public void run() {
    try {
        if(!flag) {
            sealingUnit.processBottle(unfinished, packB1Tray, packB2Tray, sealTray, godown, packagingUnit);
        }
        else{
            packagingUnit.processBottle(unfinished, packB1Tray, packB2Tray, sealTray, godown, sealingUnit);
        }
    }
    catch (final Exception e) {
        System.out.println("Exception has occurred");
    }
}
```

B. **Synchronization** :

The 2 processing units - packaging & sealing unit - access shared resources like unfinished tray, sealing tray, both the buffer trays & the godown data. Thus synchronisation is required while accessing these resources so that only 1 unit (& hence 1 thread) accesses these at a time.
Here, we have used Reentrant Locks for synchronization. Reentrant Locks are provided in Java which implement the Lock interface. The code which manipulates the shared resource is surrounded by calls to *lock()* and *unlock()* method which gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

We have implemented these locks for accessing each of the trays and the godown.
Unfinished Tray : getB1ListSize(), getB2ListSize(), popB1Bottle(), popB2Bottle();
Buffer Trays (2 Buffer Trays & 1 Sealer Tray) : getListSize(), pushBottle(), popBottle();
Godown : incB1Cnt(), incB2Cnt();

```
/* pushes the Bottle b to the current Buffer Tray list */
void pushBottle(Bottle b){

    reentLock.lock();
    try{ this.tray.add(b); }
    catch (Exception e) { System.out.println("Exception has occurred"); }
    finally { reentLock.unlock(); }

    return;
}
```

```
/*
   if both Packaging & Sealing Unit have just completed the processing,
   then let the Sealing Unit to process first and update the respective trays.
   Following the completion of this thread only, the Packaging Unit thread too could process
*/
if(packagingUnit.timeLeft == 1 && sealingUnit.timeLeft == 1){
    tSeal.start();
    tSeal.join();
    tPack.start();
    tPack.join();
}
/* Else let both the threads to process simultaneously */
else{
    tSeal.start();
    tPack.start();
    tSeal.join();
    tPack.join();
}
```

Also, as shown above, if the case arises where both the sealing & packaging unit have just processed the bottle at the same time, then, since the Sealer is slower to process than Packager, let the sealer process first. Explanation is given in the following point.
Also, here, for every time unit iteration, we have waited for both the threads to complete their execution before moving to the next iteration.

3. **Importance of Synchronization** :

As said earlier, synchronization is important for accessing shared resources so that there is no problem of data inconsistency.

a. Accessing & Modifying Data of Unfinished Tray :-

If both Sealing and Packaging units try to access the bottles from the unfinished tray at the same

time, then one thread will win the race condition and gain access earlier and the other thread(or unit) may get redundant data since the first thread may be updating the tray size(by popping out a value) which is actually changed as of that time.

b. Accessing & Modifying Data of Sealing & Buffer Trays :-
   As above, similar case occurs when both the units are accessing & changing the tray sizes at the same time as it may happen that the size read by one was just modified by the other one & thus it has read the previous wrong value which may cause problems related to data inconsistency.
c. Modifying the Godown Counts :-
   Also, while updating the bottle B1 & B2 counts in the godown, without synchronization it may happen that only of the unit's updated bottle count retains while the other update gets overwritten by it, thus reducing the no. of updates at that instant from 2 to 1.
d. Finishing processing the bottle simultaneously :-
   We know that Sealing unit takes more time to process a bottle than the other. Hence, there is very less possibility that the Sealing Tray will be empty just at the instant the sealing unit has finished processing the current bottle.
   Based on this observation, for cases when both the units have just finished processing the bottles at an instant, we let the sealing unit thread to first complete its execution. If this isn't done, then both the threads will execute simultaneously. Now, if synchronization is not applied, consider the cases where both the units have just processed the last bottles simultaneously :-
   i. Sealing & Buffer Trays are Empty :
      Now, on the basis of which thread runs earlier, the next bottle inputs to the resp. Units will be different & hence will produce inconsistent results.
   ii. Sealing Tray is Full :
      Now, again if the Packaging Unit completes earlier, then it couldn't send the bottle to the sealing tray since it is full & hence it has to remain idle for the next 1 time unit(second). Only when the Sealing Unit runs earlier, then only none of them can remain idle for that iteration.
e. Waiting for threads to die :-
   We wait for both of the threads to complete their execution before we start the execution of new threads for the next iteration. Without ensuring this, previous threads may hamper the working of the threads of the current iteration which will again result in data inconsistency.

Problem 3: Automatic Traffic Light System

1. **Concurrency** :
   Yes, the concept of concurrency is definitely applicable here since we make 6 Threads, 1 for each traffic direction & reverse, which run every second & are concurrently used to update the remaining time for each vehicle in their respective queue acc. to the source & destination direction. Also, we have made one more thread for adding new vehicle entry in the corresponding table after considering its moving direction.

If we hadn't used different threads & worked with only a single thread, then each vehicle data would be updated serially in the table which would further lead to delay in time & cause issues with overall time management & hence inconsistent data in system.
Hence, for managing all directions separately & parallely, we have used different threads to introduce the much needed concurrency.

2. **Synchronization** :
   Synchronization Technique Used **: Synchronised** Keyword (Implicit Monitor)
   Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. 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. This synchronization is implemented in Java with a concept called monitors.

   a. **Accessing & Updating Direction Queues** :
      We have 2 types of threads - first (for each direction) & second (only one for the current vehicle added). First type of threads run every second & update the remaining time values of each vehicle currently in the table for each direction queue while the second type only updates these values at the time of addition of a new vehicle.
      Since both these threads may access the shared queue contents simultaneously, thus we need to apply synchronization on these shared resources so that only one thread can access them at a particular time & thus there is no inconsistency in data values.

   b. **Displaying Updated Tables** :
      Again, since we don't want to display inconsistent remaining time of vehicles in the tables, thus the 2 types of threads stated above, should again run parallely in synchronization over these shared resources and thus value should be displayed by only one thread at a time.

```
// Adds incoming vehicle to queue and updates its remaining time in queue and its status as 'Pass' or 'Wait'
void addV(Vehicle v, Time time, long prevRemTime, long vLightNo){
    synchronized (this) {
    try {
            long addTime = 0;
            long curTotal = 54;
            long cycle = 180;
            long incr = 0;
            long comp = 0;
            long cur_time = time.cur_time;
            long tLightNo;
            if (time.cur_time < 60)
                tLightNo = 1;
            else if (time.cur_time < 120){...}
            else{...}

            if (vLightNo == 0) {...} else {...}

            if (v.remTime == 0)
                v.updateStatus("Pass");
            else
                v.updateStatus("Wait");

            vehicles.add(v);
        }
    catch(Exception e){
            System.out.println("Exception has occurred in addV");
        }
    }
}
```

```java
// Removes first vehicle from queue and updates its status as 'Pass'
Vehicle removeV(){
    synchronized (this) {
    try {
            updateLastVTime( b: true);

            Vehicle v = vehicles.remove( i: 0);
            lastCarNo = v.vNo;
            v.updateStatus("Pass");
        }
    catch(Exception e){
            System.out.println("Exception has occurred in removeV");
        }
    }
    return v;
}
```

```java
// updates remaining time in queue for each vehicle and removes a vehicle if its remaining time reac
void updateRemainTime(){
    synchronized (this) {
    try {
            for (Vehicle v : vehicles)
                v.remTime--;
            if ((vehicles.size() > 0) && (vehicles.get(0).remTime <= 0))
                removeV();
        }
    catch(Exception e){
            System.out.println("Exception has occurred in updateRemainTime");
        }
    }
}
```

```java
// updates lastVTime such that the vehicles in the queue wait lastVTime more seconds for the last vehicle that left to pass comp
void updateLastVTime(boolean b){
       synchronized(this){
       try {

           if (!b) {
               lastVTime--;
               if (lastVTime < 0) {
                   lastVTime = 0;
                   lastCarNo = -1;
               }
           } else lastVTime = 6;
       } catch (Exception e) {
           System.out.println("Exception has occurred in updateLastVTime");
       }
   }
}
```

3. **Pseudo-Code** :

Most of the implementation would remain the same except few changes. Since there are four roads now, every vehicle has 3 ways to go after the junction, out of which one is free-pass and the other two are controlled by the traffic light. Thus, for every green light, there are two routes open for the queue opposite to the light, and four other routes that are always open. Hence, the three red lights lead to six blocked routes (two from each queue opposite a red light).

Our traffic light turn green in a round-robin manner.

We maintain 12 queues of vehicles for every source-destination combination.
We first insert any incoming vehicle in a queue as follows.

Source	Destination	Inserted in Queue	Controlling Traffic Light
S	W	QSW	--
S	N	QSN	T1
S	E	QSE	T1
W	N	QWN	--
W	E	QWE	T2
W	S	QWS	T2
N	E	QNE	--
N	S	QNS	T3
N	W	QNW	T3
E	S	QES	--
E	W	QEW	T4
E	N	QEN	T4

.
Thus,

Vehicle coming from S and going to W is inserted to QSW.
Vehicle coming from S and going to N is inserted to QSN.
Vehicle coming from S and going to E is inserted to QSE.

When T1 is red, vehicles in Q1 coming from S can freely go only to W.
When T1 is green, vehicles in Q1 coming from S can go in any direction, ie, W, N, S.

In our original implementation for three roads, we have used two variables *cycle* and *incr* which are going to change in this new implementation for four roads.

cycle = 240; // instead of 180 in previous implementation

incr += 180; // instead of incrementing by 120 as in previous code

```
// Adds incoming vehicle to queue and updates its remaining time in queue and its status as 'Pass' or 'Wait'
void addV(Vehicle v, Time time, long prevRemTime, long vLightNo){
    synchronized (this) {
    try {
            long addTime, curTotal, cycle, incr, comp, cur_time, tLightNo;
            addTime = 0; curTotal = 54; cur_time = time.cur_time;
            cycle = 240;                                    // instead of 180 in previous implementation
            incr = 0;
            if (time.cur_time < 60) {...}
            else if (time.cur_time < 120){...}
            else{...}

            if (vLightNo == 0) {...} else {
                long lightDiff = vLightNo - tLightNo;
                if (lightDiff < 0){...}
                switch ((int) ((lightDiff) % 3)) {...}
                if (vehicles.isEmpty()) {...} else {
                    if (curTotal - 6 - cur_time - prevRemTime + addTime >= 0)
                        v.remTime = prevRemTime + 6 ;
                    else {
                        while (true) {
                            comp = prevRemTime - cycle - incr + cur_time;
                            if (curTotal - 6 - comp >= 0) {...}
                            incr += 180;                // instead of incrementing by 120 as in previous code
                        }
                    }
                }
            }
            if (v.remTime == 0)
                v.updateStatus("Pass");
            else
                v.updateStatus("Wait");
            vehicles.add(v);
        }
    catch(Exception e){...}
    }
}
```