INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT074-3-2

CONCURRENT PROGRAMMING

APU2F2206SE / APU2F2206CS(DA) / APU2F2206CS / APD2F2206CS(DA) / APD2F2206SE / APD2F2206CS

HAND OUT DATE : 1 DECEMBER 2022

HAND IN DATE : 24 FEBRUARY 2023

WEIGHTAGE : 20%

INSTRUCTIONS TO CANDIDATES:

1. **Submit your assignment at the administrative counter.**
2. **Students are advised to underpin their answers with the use of references (cited using the Harvard Name System of Referencing).**
3. **Late submission will be awarded zero (0) unless Extenuating Circumstances (EC) are upheld.**
4. **Cases of plagiarism will be penalized.**
5. **The assignment should be bound in an appropriate style (comb bound or stapled).**
6. **Where the assignment should be submitted in both hardcopy and softcopy, the softcopy of the written assignment and source code (where appropriate) should be on a CD in an envelope / CD cover and attached to the hardcopy.**
7. **You must obtain 50% overall to pass this module.**
8. **This report is Part 2 of 2 for the assignment.**

Name : Yam Chen Xi
TP Number : TP061635
Intake Code : APD2F2206CS

Table of Contents

1.0 Achieved Requirements

The system is developed to simulate the Air Traffic Control (ATC)operations of Asia Pacific Airport. The following comprises the accomplished requirements mentioned in the assignment.

1.1 Basic Requirements

The first basic requirement of the ATC is **only one runway is utilised for landing and departing planes**. To achieve the requirement, semaphore is used to ensure only one plane can occupy the runway at a time.

Acquire() and release() are two crucial Semaphore methods for managing access to a shared resource. A permission is sought after using the Acquire() function to contact the semaphore. In the absence of any permissions, the thread will wait till one becomes available. The crucial portion of the thread's code that needs access to the shared resource can be executed after a permission has been obtained. A permission is released back to the semaphore using the Release() function. The release() function must be called by a thread once it has finished running the crucial portion of code in order to return the permission to the semaphore. Another thread will be able to get the permission it needs to use the shared resource as a result. In order to guarantee that only a specific amount of threads could utilize a common resource at any given moment, acquire() and release() methods can be used simultaneously. The most threads that can use a shared resource at once depend on the number of permits that are accessible in a semaphore (Simplilearn, 2023).

```
public class Plane extends Thread{
    private static final int NUM_RUNWAYS = 1;
    private int ID;

    // Create a Semaphore to store runway.
    // The size is fixed to 1 as there will only be 1 runway.
    public static Semaphore runway = new Semaphore(NUM_RUNWAYS);
```

Figure 1: Semaphore initialisation

The code snippet declares a constant variable NUM_RUNWAYS with a value of 1 which means that only one runway is allowed for planes to land and depart at the airport. A Semaphore object named runway is also declared which controls access to the runway. The Semaphore has a permit count of NUM_RUNWAYS, meaning that only one thread can acquire the permit and use the runway at a time.

```
//Add plane into the semaphore runway
printAvailStatus(1);
runway.acquire();

//Plane uses runway and departed
System.out.println("PLANE-" + ID + " is using the runway for departure");
Thread.sleep(randomTime);
System.out.println("\n==============================================\n"
        +"\t\t"+java.time.LocalTime.now()
        +"\n\tPLANE-" + ID + " HAS SUCCESSFULLY DEPARTED"
        +"\n==============================================\n" );

//Release plane from the semaphore runway
runway.release();
```

Figure 2: Acquire and release semaphore

Once a thread has acquired the permit calling .acquire(), it is considered "locked" and other threads must wait for it to release the permit using .release() after finish using the runway before other thread can acquire it and use the runway. This feature is also applied to one of the requirements which is to ensure there is **no planes landed on the airport ground to wait for available gate** due to the limited area at the airport.

The second requirement of the assignment is to **limit the number of airplanes present on the airport ground to a maximum of 2**. An object called an Executor executes tasks that have been submitted, often on a different thread. ThreadPoolExecutor, one of the most widely used Executor implementations, keeps a group of threads and distributes tasks to them once they are available. Then, a task is submitted for implementation using the ThreadPoolExecutor's submit method. It accepts a runnable object as an input and outputs a Future object that may be used to get the task's outcome or to stop it altogether (baeldung, 2016).

```
//To avoid having too many planes waiting
//Maximum 2 threads will be active to process tasks
ExecutorService executor = Executors.newFixedThreadPool(2);
```

Figure 3: Executor code snippet

This requirement is done by creating an executor service object named "executor" with a thread pool size of 2. This will ensure that only maximum two planes are generated when the airport has available gates to allow the planes to land, avoid having any possible collision or congestion happening at the airport.

```
//Start plane thread i
Plane plane = new Plane(i);
executor.submit(plane);
```

Figure 4: Executing plane thread using executor

Then create a new plane thread and submits it to the executor service using the "submit" method. This means that the plane will run as a separate thread within the thread pool managed by the executor service.

```
---------- PLANE-1: CLEANING AIRCRAFT CABIN PROCESS ----------
PLANE-1 Cabin Cleaner Crew reporting for work
          Allocating fuel truck...
         [Fuel Truck Available]: 1

---------- PLANE-1: PASSENGER EMBARKMENT PROCESS ----------
PLANE-1 Passenger-1 is embarking

---------- PLANE-1: REFUELLING AIRCRAFT PROCESS ----------
PLANE-1 Aircraft Refueller reporting for work
PLANE-2 Passenger-3 is disembarking
```

Figure 5: Tasks given to planes upon landing

The third requirement is to **allow the planes to carry out respective tasks**. Which are arrive on the runway, go to designated gate, dock there, disembark passengers, replenish supplies and fuel, embark new passengers, undock, coast to runway, and depart.

```
if (isEmergency == true){
    emergencyLanding();
}else{
    normalLanding();
}
```

Figure 6: Calling tasks method

The tasks are done in sequence by calling normalLanding() method and emergencyLanding() method.

```
//Airplane uses runway
System.out.println("\n********** PLANE-" + ID + " ANNOUNCEMENT: RUNWAY AVAILABLE **********");
Thread.sleep(randomTime);
System.out.println("["+ java.time.LocalTime.now() + "]" + " PLANE-" + ID + " is using the runway

//Add plane into the semaphore gates
printAvailStatus(2);
gates.acquire();
System.out.println("\n********** PLANE-" + ID + " ANNOUNCEMENT: GATE AVAILABLE **********");
System.out.println("PLANE-" + ID + " has been assigned to gate-" + gateNumber);
System.out.println("PLANE-" + ID + " has successfully docked at the gate-" + gateNumber);

//Passengers disembarked from plane
System.out.println("\n---------- PLANE-" + ID + ": PASSENGER DISEMBARKMENT PROCESS ----------");
for (int i=1; i<=(numDisembark); i++){
    //Passenger disembark in a random generated
    System.out.println("PLANE-" + ID + " Passenger-" + i + " is disembarking");
    Thread.sleep(random.nextInt(150) + 30);
};
System.out.println("========== PLANE-" + ID + " ALL PASSENGERS HAVE DISEMBARKED ==========");
```

Figure 7: normalLanding method code snippet

The method starts by generating a random time for the plane to use the runway, and a random number of passengers to disembark and board the plane. It then assigns the plane to a gate. Next, the method releases the plane from the runway semaphore. The plane then announces that it is using the runway. After landing, the plane acquires the semaphore for its assigned gate, announces that it is available, and docks at the gate. The passengers disembark, and the cabin crew cleans the cabin. Then, the aircraft refueler acquires the fuel truck semaphore, refuels the plane, and releases the truck. Finally, the passengers board the plane, and the plane departs. Lastly, releases the plane from the gates and runway semaphores.

The fourth requirement is to **have time gap between each step**. By generating random time to utilise .sleep() method on threads, this can ensure that the thread temporary sleeps for a random duration of time from 0.3 to 0.5 seconds before proceeding to the next task.

```
//Randomly sleep the thread in between 0.3 and 0.5 seonds
int randomTime = random.nextInt(200) + 300;
```

Figure 8: randomTime initialisation

```
//Aircraft Refueller refuel aircraft
//Display fuel truck availability
printAvailStatus(3);
System.out.println("\n---------- PLANE-" + ID + ": REFUELLING AIRCRAFT PROCESS ----------");
System.out.println("PLANE-" + ID + " Aircraft Refueller reporting for work");
Thread.sleep(randomTime);
fuelTruck.acquire();
System.out.println("PLANE-" + ID + " Aircraft Refueller is using the fuel truck");
Thread.sleep(randomTime);
System.out.println("PLANE-" + ID + " Aircraft Refueller is fueling the plane");
Thread.sleep(randomTime);
System.out.println("PLANE-" + ID + " Aircraft Refueller is returning the fueling truck");
Thread.sleep(randomTime);
System.out.println("========== PLANE-" + ID + " AIRCRAFT DONE REFUELLING ==========");
fuelTruck.release();
```

Figure 9: Random thread sleep duration in between tasks

1.2 Additional Requirements

The first additional requirement is that **passenger disembark**, **cleaning aircraft, refiling aircraft and refuelling aircraft should happen concurrently**.

```
---------- PLANE-5: PASSENGER EMBARKMENT PROCESS ----------
PLANE-5 Passenger-1 is embarking

---------- PLANE-5: CLEANING AIRCRAFT CABIN PROCESS ----------
PLANE-5 Cabin Cleaner Crew reporting for work

---------- PLANE-5: REFUELLING AIRCRAFT PROCESS ----------
PLANE-5 Aircraft Refueller reporting for work
PLANE-3 Passenger-10 is embarking
```

Figure 10: Implementing tasks concurrently

These processes are able to happen concurrently by creating threads to be run in the method. The process will first unload all the passengers, then clean the cabin and refuel the plane. Simultaneously, embark new passengers to be onboard. These will allow the ATC system to be more efficient and having less waiting time.

```
//Cabin crew cleaner clean the aircraft cabin
CleanCabin t1 = new CleanCabin(ID, randomTime);
t1.start();

//Aircraft Refueller refuel aircraft
//Display fuel truck availability
printAvailStatus(3);
Refuel t2 = new Refuel(ID, randomTime);
t2.start();
```

Figure 11: Executing task threads

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        EMERGENCY LANDING REQUIRED
              05:22:54.390
        PLANE-3 REQUESTING TO LAND
            **FUEL SHORTAGE**
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Figure 12: Displaying emergency landing plane

```
[05:22:55.673] PLANE-3 is using the runway for landing
            No gate available...
##### PLANE-3 IS WAITING FOR AVAILABLE GATE #####
```

Figure 13: Emergency landing approved

The second additional requirement is to **create a congested scenario** where there are two planes landed on the airport and occupied two gates. Then, the third airplane required emergency landing due to fuel shortage.

```
//Get start time
long startTime = System.currentTimeMillis();
if (i == 3){
    Executors.newFixedThreadPool(3);
    //Assign emergency plane through parameter
    gates = new Gate(i,true,startTime);
    gates.setPriority(Thread.MAX_PRIORITY);
}
else{
    Executors.newFixedThreadPool(2);
    //Assign plane
    gates = new Gate(i,false,startTime);
    plane.setPriority(Thread.MIN_PRIORITY);
}
```

Figure 14: Allowing emergency landing code snippet

This requirement is fulfilled by setting thread pool size using executors. It will only specifically allow airplane that has emergency to land on the airport successfully and proceed with its tasks.

```
------------- Statistics -------------
              Passenger
             PLANE-1: 22
             PLANE-2: 33
             PLANE-3: 37
             PLANE-4: 17
             PLANE-5: 33
             PLANE-6: 27
           Total        : 169
          Average       : 28

                ----
            Runtime(second)
             PLANE-1: 7.12s
             PLANE-2: 10.18s
             PLANE-3: 25.26s
             PLANE-4: 8.2s
             PLANE-5: 13.09s
             PLANE-6: 15.94s
           Total        : 79.79
          Average       : 13.3s
          Minimum       : 7.12s
          Maximum       : 25.26s
```

Figure 15: Statistics report on console

The last additional requirement is to **create a statistics report** on the number of passengers and waiting time of a plane.

To get the number of passengers, getPassenger() method with the parameter ID and passenger amount is used to get embark passenger amount of each flight and store it into an integer array. Then, by using getStatisticalReport() method, it will calculate the total amount of embarked passengers and the average number of passengers of each flight. Lastly, display the passenger information on console.

```
static int[] getPassengers(int ID,int passenger) {
    //Store into passengers array
    passengers[ID-1] = passenger;
    return passengers;
```

Figure 16: getPassenger method

```
static void getStatisticalReport() {
    //Calculate number of passengers
    int sumPass = 0;
    for (int i = 0; i < passengers.length; i++) {
        sumPass += passengers[i];
    }
    int averagePass = sumPass/passengers.length;
```

Figure 17: Number of passengers calculation code snippet

To get the runtime of the flights, getRunTime() method is used. It required to pass in parameter such as plane ID and startTime of the flight. Next, by using getStatisticalReport() method, it will calculate the total, average, min and max runtime of the planes. Lastly, display the runtime information on console.

```
static double[] getRunTime(int ID,long startTime) {
    //Calculate runtime and convert to seconds
    long endTime = System.currentTimeMillis();
    long runtimeInMillis = endTime - startTime;
    double runTime = runtimeInMillis / 1000.0;
    //Store into runtimes array
    runtimes[ID-1] = runTime;
    return runtimes;
}
```

Figure 18: getRunTime method

```
DecimalFormat df = new DecimalFormat("#.##");
double sum = 0;
double min = Double.MAX_VALUE;
double max = Double.MIN_VALUE;
for (int i = 0; i < runtimes.length; i++) {
    sum += runtimes[i];
    if (runtimes[i] < min) {
        min = runtimes[i];
    }
    if (runtimes[i] > max) {
        max = runtimes[i];
    }
}
double average = sum / runtimes.length;
```

Figure 19: Runtime calculation code snippet

2.0 Unfulfilled Requirements

All requirements are met, hence, there are no unfulfilled requirements.

References

baeldung. (2016). *A Guide to the Java ExecutorService | Baeldung*. Baeldung.com. https://www.baeldung.com/java-executor-service-tutorial

javatpoint. (2011). *Java Thread setPriority() Method with Examples*. Javatpoint. https://www.javatpoint.com/java-thread-setpriority-method

Oracle. (n.d.-a). *Random (Java Platform SE 8)*. Oracle.com. https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

Oracle. (n.d.-b). *Thread (Java Platform SE 7)*. Oracle.com. https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

ravi.geek24. (2022). *Java Threads*. GeeksforGeeks. https://www.geeksforgeeks.org/java-threads/

Simplilearn. (2023). *What is Semaphore in Java? A Complete Guide | Simplilearn*. Simplilearn. https://www.simplilearn.com/what-is-semaphore-in-java-uses-article

W3Schools. (2020). *Java Methods*. W3schools. https://www.w3schools.com/java/java_methods.asp