



INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT074-3-2
CONCURRENT PROGRAMMING

APD2F2109CS(DA) / APU2F2109SE / APU2F2109CS(DA) /
APD2F2109CS / APD2F2109SE / APU2F2109CS

HAND OUT DATE: 11 MARCH 2022

HAND IN DATE: 3 JUNE 2022

WEIGHTAGE: 25%

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 submissions 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.**

Table of Contents

1.0 Basic Requirements Met.....	3
1.1 Code Snippet.....	3
1.1.1 One Runway.....	3
1.1.2 Aircrafts do not Collide	3
1.1.3 Aircraft Flow.....	4
1.1.4 Each Step Should Take Some Time.....	6
1.1.5 Congested Scenario.....	7
1.1.6 No Waiting Area	7
1.2 Explanations of Concurrency Concepts	8
2.0 Additional Requirements Met	9
2.1 Code Snippet.....	9
2.1.1 Passengers Disembarking/Embarking	9
2.1.2 Refill Supplies and Cleaning of Aircraft	10
2.1.3 Refueling of Aircraft.....	11
2.1.4 Statistics	11
2.2 Explanations of Concurrency Concepts	12
3.0 Requirements which were not Met.....	13
3.1 List of Basic Requirements	13
3.2 List of Additional Requirements	13
References.....	14

1.0 Basic Requirements Met

1.1 Code Snippet

1.1.1 One Runway

```
Semaphore mRunway = new Semaphore( permits: 1, fair: true);
```

Figure 1.1.1.1 Semaphore Implementation for Runway (Departure)

```
synchronized void landing(Plane plane)
```

Figure 1.1.1.2 Synchronization Implementation for Runway (Landing)

Both figures above show the implementation of only having one runway. With semaphore, the permits are limited to only 1, meaning that only 1 thread can access it at a time. With synchronization, landing is limited to only 1 plane at a time.

1.1.2 Aircrafts do not Collide

```
synchronized void landing(Plane plane)
```

Figure 1.1.2.1 Synchronization Implementation for Runway (Landing)

```
Semaphore mRunway = new Semaphore( permits: 1, fair: true);
```

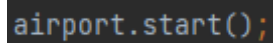
Figure 1.1.2.2 Semaphore Implementation for Runway (Departure)

```
if(airport.sGate.availablePermits() == 0)
{
    System.out.println("ATC: Plane "+planeID+", all gates are full, please standby.");
    System.out.println("Plane "+planeID+": Copy ATC, will circle around the airspace.");
}
airport.landing( plane: this);
```

Figure 1.1.2.3 Semaphore Permits Check

All figures above show the precautions so that aircrafts will not collide. Synchronization and semaphore to make sure if an aircraft is occupying the runway, no other aircraft can access it. The permit check will give out the dialogue so the plane will wait if runway/gates is full.

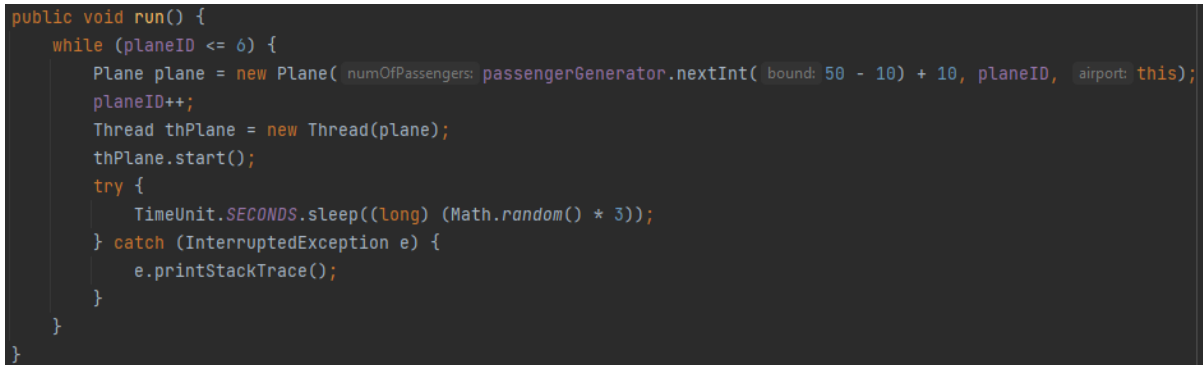
1.1.3 Aircraft Flow



```
airport.start();
```

Figure 1.1.3.1 Starting Point

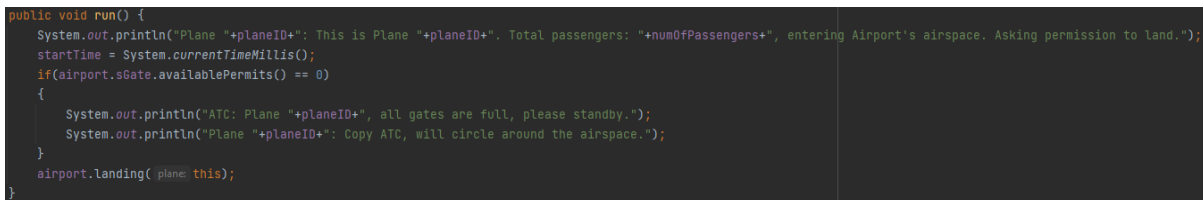
The airport flow starts from the main function, where airport thread is started.



```
public void run() {  
    while (planeID <= 6) {  
        Plane plane = new Plane( numOfPassengers: passengerGenerator.nextInt( bound: 50 - 10) + 10, planeID, airport: this);  
        planeID++;  
        Thread thPlane = new Thread(plane);  
        thPlane.start();  
        try {  
            TimeUnit.SECONDS.sleep((long) (Math.random() * 3));  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Figure 1.1.3.2 Airport Thread Run (Plane Generator)

Within the airport thread, there is a plane generator to create 6 aircrafts, then run it after creation. The number of passengers is randomized between 10-50 passengers per aircraft. The generator will generate a plane between 0-3 seconds randomly.



```
public void run() {  
    System.out.println("Plane "+planeID+": This is Plane "+planeID+". Total passengers: "+numOfPassengers+", entering Airport's airspace. Asking permission to land.");  
    startTime = System.currentTimeMillis();  
    if(airport.sGate.availablePermits() == 0)  
    {  
        System.out.println("ATC: Plane "+planeID+", all gates are full, please standby.");  
        System.out.println("Plane "+planeID+": Copy ATC, will circle around the airspace.");  
    }  
    airport.landing( plane: this);  
}
```

Figure 1.1.3.3 Plane Thread Run

Within the plane thread, it will input that the plane is approaching the airspace. If all gates are occupied (0 available semaphore permit), the plane will circle around the airspace above airport until a permit is available.

```
synchronized void landing(Plane plane)
{
    try {
        sGate.acquire();
        mRunway.acquire();
        if(gateAIndicator)
        {
            System.out.println("ATC: Plane "+plane.planeID+", permission granted. Please land and head to gate A");
            System.out.println("Plane "+plane.planeID+": Affirmative ATC. Landing now.");
            plane.endTime = System.currentTimeMillis();
            plane.calculateTime();
            totalTime += plane.elapsedTime;
            totalPassenger += plane.numOfPassengers;
            if(plane.elapsedTime > maxTime)
            {
                maxTime = plane.elapsedTime;
            }
            if(plane.elapsedTime < minTime)
            {
                minTime = plane.elapsedTime;
            }
            plane.gateName = 'A';
            gateAIndicator = false;
            currentThread().sleep(1500);
            System.out.println("Plane "+plane.planeID+": has landed. Docking at Gate "+plane.gateName+".");
            mRunway.release();
            Gate gateA = new Gate( "name: 'A'", available: true, airport: this, fuelTruck);
            gateA.setPlane(plane);
            gateA.start();
        }
        else if(gateBIndicator)
        {
            System.out.println("ATC: Plane "+plane.planeID+", permission granted. Please land and head to gate B");
            System.out.println("Plane "+plane.planeID+": Affirmative ATC. Landing now.");
            plane.endTime = System.currentTimeMillis();
            plane.calculateTime();
            totalTime += plane.elapsedTime;
            totalPassenger += plane.numOfPassengers;
            if(plane.elapsedTime > maxTime)
            {
                maxTime = plane.elapsedTime;
            }
            if(plane.elapsedTime < minTime)
            {
                minTime = plane.elapsedTime;
            }
            plane.gateName = 'B';
            gateBIndicator = false;
            currentThread().sleep(1500);
            System.out.println("Plane "+plane.planeID+": has landed. Docking at Gate "+plane.gateName+".");
            mRunway.release();
            Gate gateB = new Gate( "name: 'B'", available: true, airport: this, fuelTruck);
            gateB.setPlane(plane);
            gateB.start();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Figure 1.1.3.4 Landing and Docking to Gate

If a gate is empty, the plane will land. The plane will be designated to an empty gate. After the plane has docked, it will start the processes that will be explained in the additional requirements part.

```
void depart(Plane plane)
{
    System.out.println("Plane "+plane.planeID+": Plane "+plane.planeID+" will depart from Gate "+plane.gateName+" Asking ATC for approval.");
    try {
        mRunway.acquire();
        System.out.println("ATC: Plane "+plane.planeID+" Departure approved, Godspeed.");
        System.out.println("Plane "+plane.planeID+": Taking off.");
        currentThread().sleep(1500);
        System.out.println("Plane "+plane.planeID+": is airborne! See you around ATC. Plane "+plane.planeID+" out.");
        System.out.println("ATC: See you around, Plane "+plane.planeID+" ATC out.");
        countPlane++;
        System.out.println("ATC: Total departed plane: "+countPlane);
        if(plane.gateName == 'A')
        {
            gateAIndicator = true;
            System.out.println("ATC: Gate A is now available.");
        }
        else
        {
            gateBIndicator = true;
            System.out.println("ATC: Gate B is now available.");
        }
        sGate.release();
        mRunway.release();
        if(countPlane == 6)
        {
            System.out.println("ATC: All aircraft have departed, generating report: ");
            System.out.println("ATC: Gate A is empty: "+gateAIndicator+".");
            System.out.println("ATC: Gate B is empty: "+gateBIndicator+".");
            System.out.println("ATC: Minimum wait time: "+minTime+" seconds.");
            System.out.println("ATC: Maximum wait time: "+maxTime+" seconds.");
            System.out.println("ATC: Average wait time: "+totalTime/countPlane+" seconds.");
            System.out.println("ATC: Number of plane served: "+countPlane+" planes.");
            System.out.println("ATC: Number of passenger served: "+totalPassenger+" passengers.");
            System.out.println("ATC: Report Done. ATC going dark. Good Night.");
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Figure 1.1.3.5 Departure

After the plane has done all processes, it will start for departure process. After the connection between ATC and a plane is disconnected, the previously occupied gate and the runway will be available to use.

1.1.4 Each Step Should Take Some Time

```
currentThread().sleep(1500);
```

Figure 1.1.4.1 Sleep() function

Using sleep method, it will put the thread process to halt, giving an illusion of a step being done.

1.1.5 Congested Scenario

```
Plane plane = new Plane( numOfPassengers: passengerGenerator.nextInt( bound: 50 - 10) + 10, planeID, airport: this);
planeID++;
Thread thPlane = new Thread(plane);
thPlane.start();
try {
    TimeUnit.SECONDS.sleep((long) (Math.random() * 3));
```

Figure 1.1.5.1 Plane Generator Delay Between each Generation

With the generation period between 0-3 seconds and the whole process for a plane can take up to 8 seconds, there will be some congested scenario where the gates are full.

1.1.6 No Waiting Area

```
if(airport.sGate.availablePermits() == 0)
{
    System.out.println("ATC: Plane "+planeID+", all gates are full, please standby.");
    System.out.println("Plane "+planeID+": Copy ATC, will circle around the airspace.");
}
airport.landing( plane: this);
```

Figure 1.1.6.1 If Gate is Full, Plane will be Circling Around

As there is no waiting area, the plane will circle around above the airport if the gates are full..

1.2 Explanations of Concurrency Concepts

For the basic requirements, the usages of semaphore and synchronization is used to make sure the process go smoothly as intended.

Semaphore is a thread synchronization method in Java. It is included in `java.util.concurrent` package. It uses a counter to controls access of threads to a shared resource (GeeksforGeeks, 2018). In this project, it is used to control the flow of runway and gates.

Synchronization is a thread controlling method to control multiple threads' access to a shared resources/process. It is used where multiple threads want to access one same process (javaTpoint, n.d.). in this project basic requirement part, it is used to control the landing process.

2.0 Additional Requirements Met

2.1 Code Snippet

```
@Override
public void run() {
    try {
        c = cleaning.cleaningProcess();
        b = passengers.passengersProcess();
        cleaning.start();
        passengers.start();
        fuelTruck.useFuelTruck(plane);
        cleaning.join();
        passengers.join();

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Plane "+plane.planeID+": All process done. Final check.");
    try {
        currentThread().sleep( millis: 1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Plane "+plane.planeID+": Final check done. Ready to Depart.");
    airport.depart(plane);
}
```

Figure 2.1.1 Gate Thread Process

For the additional requirements, the whole process is split into 3 parts. After all parts of the process has been completed, then the plane will do a final check before starting the departure process.

2.1.1 Passengers Disembarking/Embarking

```
passengers.start();
```

Figure 2.1.1.1 Passengers Thread Start

```
public void run() {
    System.out.println("Plane "+plane.planeID+": Unloading Passengers Start.");
    for(int i = 1; i <= plane.numOfPassengers; i++)
    {
        System.out.println("Plane "+plane.planeID+": Passengers " + i + " left the plane.");
        try {
            currentThread().sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println("Plane "+plane.planeID+": Unloading Passengers Finished!");
    try {
        currentThread().sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Plane "+plane.planeID+": Loading Passengers Start.");
    for(int i = 1; i <= plane.numOfPassengers; i++)
    {
        System.out.println("Plane "+plane.planeID+": Passengers " + i + " entered the plane.");
        try {
            currentThread().sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println("Plane "+plane.planeID+": Loading Passengers Finished!");
}
```

Figure 2.1.1.2 Passengers Thread Process

For passengers disembarking/embarking process, passengers thread is used. The passengers inside the plane will start to go out through the gate. After all passengers has disembark, the passengers from the boarding room will enter the plane.

2.1.2 Refill Supplies and Cleaning of Aircraft

```
cleaning.start();
```

Figure 2.1.2.1 Cleaning Thread Start

```
@Override
public void run() {
    System.out.println("Plane "+plane.planeID+": Cleaning, Resupplying, and Maintenance Start.");
    try {
        currentThread().sleep(4000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Plane "+plane.planeID+": Cleaning, Resupplying, and Maintenance Done!");
}
```

Figure 2.1.2.2 Cleaning Thread Process

For plane cleaning, resupplying, and maintenance process, Cleaning thread is used. The process will be started, then after 4 seconds, the process will be done.

2.1.3 Refueling of Aircraft

```
public class FuelTruck {  
  
    synchronized void useFuelTruck(Plane plane) throws InterruptedException  
    {  
        System.out.println("Plane "+plane.planeID+": Refuelling Start.");  
        currentThread().sleep( millis: 500);  
        System.out.println("Plane "+plane.planeID+": Refuelling...");  
        currentThread().sleep( millis: 2000);  
        System.out.println("Plane "+plane.planeID+": Refuelling Done!");  
    }  
}
```

Figure 2.1.3.1 Refueling Process

For plane refuelling process, FuelTruck class will be used. The useFuelTruck function will be used, where the refuelling start, took 2.5 seconds, then refuelling will be done.

2.1.4 Statistics

```
if(countPlane == 6)  
{  
    System.out.println("ATC: All aircraft have departed, generating report: ");  
    System.out.println("ATC: Gate A is empty: "+gateAIndicator+".");  
    System.out.println("ATC: Gate B is empty: "+gateBIndicator+".");  
    System.out.println("ATC: Minimum wait time: "+minTime+" seconds.");  
    System.out.println("ATC: Maximum wait time: "+maxTime+" seconds.");  
    System.out.println("ATC: Average wait time: "+totalTime/countPlane+" seconds.");  
    System.out.println("ATC: Number of plane served: "+countPlane+" planes.");  
    System.out.println("ATC: Number of passenger served: "+totalPassenger+" passengers.");  
    System.out.println("ATC: Report Done. ATC going dark. Good Night.");  
}
```

Figure 2.1.4.1 Statistics Generation Process

After 6 planes has departure from the airport, today's performance will be printed as statistics as shown above. After all statistics has been done, ATC will shut down.

2.2 Explanations of Concurrency Concepts

For the additional requirements, the usages synchronization and join method is used to make sure the process go as requested.

Synchronization is a thread controlling method to control multiple threads' access to a shared resources/process. It is used where multiple threads want to access one same process (javaTpoint, n.d.). in this project additional requirement part, it is used to make sure that the refuelling process is exclusive to one plane at a time, since there is only 1 fuel truck.

Join method in Java allowed one thread to wait until another thread completes its execution (GeeksforGeeks, 2021). In this project, it is used to make sure all processes have been done before starting the departure part (refer to figure 2.1.1).

3.0 Requirements which were not Met

3.1 List of Basic Requirements

From all the snippets above, **all basic requirements as stated below has been fulfilled:**

- There is only 1 runway for all planes to land and depart.
- Ensure that the aircraft does not collide with another aircraft on the runway or gates
- Once an aircraft obtains permission to land, it should land on the runway, coast to the assigned gate, dock to the gate, allow passengers to disembark, refill supplies and fuel, receive new passengers, undock, coast to the assigned runway and take-off.
- Each step should take some time.
- A congested scenario should be simulated where planes are waiting to land while the 2 gates are occupied.
- As the airport is small, there is no waiting area on the ground for the planes to wait for a gate to become available

3.2 List of Additional Requirements

From all the snippets above, **all additional requirements as stated below has been fulfilled:**

These events should happen concurrently:

- Passengers disembarking/embarking
- Refill supplies and cleaning of aircraft

As there is only 1 refueling truck, this event should happen exclusively:

- Refueling of aircraft

The Statistics

At the end of the simulation, i.e., when all planes have left the airport, the ATC manager should do some sanity checks of the airport and print out some statistics on the run. The result of the sanity checks must be printed. You must:

- Check that all gates are indeed empty.
- Print out statistics on
- Maximum/Average/Minimum waiting time for a plane.
- Number of planes served/Passengers boarded.

References

GeeksforGeeks. (2018, December 10). *Semaphore in Java*. Retrieved May 31, 2022, from <https://www.geeksforgeeks.org/semaphore-in-java/>

GeeksforGeeks. (2021, February 17). *Joining Threads in Java*. Retrieved May 31, 2022, from <https://www.geeksforgeeks.org/joining-threads-in-java/>

javaTpoint. (n.d.). *Synchronization in Java - javatpoint*. Wwww.Javatpoint.Com. Retrieved May 31, 2022, from <https://www.javatpoint.com/synchronization-in-java#:~:text=Java%20Synchronized%20Method&text=Synchronized%20method%20is%20used%20to,the%20thread%20completes%20its%20task.>