Flying to Wonderland

The project must be done individually - no exceptions. The objective of this project is to familiarize you with the creation and execution of threads using the Thread's class methods. You should use, when necessary, the following methods to synchronize all threads: run(), start(), currentThread(), getName(), join(), yield(), sleep(time), isAlive(), getPriority(), setPriority(), interrupt(), isInterrupted().

The use of semaphores to synchronize threads is strictly DISALLOWED. Additionally, you are NOT PERMITTED to use any of the following: wait(), notify(), notifyAll(), the synchronized keyword (for methods or blocks), and any synchronized collections or synchronization tools that were not discussed in class.

You CAN, however, use the modifier **volatile** and the **AtomicInteger** and **AtomicBoolean** classes if you choose to.

Directions: Synchronize the passenger, kiosk-clerk, flight attendant, and clock threads in the context of the problem described below. The number of passengers should be entered as a command line argument. Please refer to and read carefully the project notes, tips and guidelines before starting the project.

****

Passengers of Flight CS-340 to Purell-Wonderland, NY arrive at the airport 3 hours before departure (use **sleep(random_time)** to simulate arrival). Upon their arrival, they go straight to the check-in counter to have their boarding passes printed.

Unfortunately, due to budget cuts, the airline can only maintain 2 check-in counters at this time. To avoid crowding at the check-in counters, the airline asks passengers to form lines of no more than a few passengers at each counter (determined by **counterNum**). Those not able to get on a line at a counter are asked to stand by (use **busy waiting**). To ensure fairness to those that have arrived early, passengers are helped in the order in which they arrived (**ensure FCFS among passengers**).

At each counter, there is a check-in clerk who will assist the passengers. Passengers will receive their boarding pass from the check-in clerk with a seat and zone number printed on it. The check-in clerk will generate this number and assign it to the passenger. The seat number is a random number between 1 and 30 with a corresponding zone number; passengers with seat numbers between 1 and 10 passengers are in Zone 1, passengers with seat numbers between 11 and 20 are in Zone 2, passengers with seat numbers between 21 and 30 are in Zone 3. Note that the aircraft holds only 30 passengers and is split up into 3 zones. (**Output a message to the screen with the passenger's seat and zone information**).

As soon as passengers receive their boarding pass, they rush to the security line to ensure they make it to the gate with time to spare (simulate rushing by using **getPriority()** and **setPriority()**; increase the priority of the passenger, and after a sleep of random time, set its priority back to

its default value). After all passengers receive their boarding pass, the check-in clerks are done for the day (they terminate)

Once the passengers arrive at the gate, they take a seat and wait for the flight attendant to call for passengers to board (use **busy waiting**).

A half an hour before the plane departs, the flight attendant begins to call passengers up to the door of the jet bridge. The flight attendant calls the first zone and asks the passengers to walk to the door (use **sleep to simulate**). At the door, the passengers are asked to wait in line (ensure **FCFS**) until all other passengers in the zone arrive (**busy wait**). When all have arrived, the passengers enter the plane in groups (determined by **groupNum**), so that passengers can comfortably stow their belongings and take their seats. Before boarding the plane, of course, the passengers must scan their boarding pass (use **yield() twice**). The flight attendant calls each of the remaining zones the same way.

After all zones have boarded, the flight attendant makes an announcement indicating that the door of the plane has closed. All passengers that arrive at the gate after this announcement are asked to rebook their flight and return home (**these threads terminate**). All other passengers sleep on the flight on route to their destination (use **sleep for a longer time –** long enough to be interrupted later on).

Two hours pass, and the plane prepares for landing. The flight attendant wakes up the passengers with an announcement over the loud speaker (use **interrupt()** to wake up the sleeping passengers; make sure you place a message in the catch block of the sleep method so it's clear that the passengers have been interrupted).

The plane lands and the passengers wait for the go-ahead to disembark the plane (use **busy wait**). When the flight attendant turns off the seatbelt light, passengers are asked to leave the plane in ascending order of their seat number (let's say on the plane you have Thread-3 (seat 2), Thread-4 (seat 3), Thread-2 (seat 4) and Thread-1 (seat 1); the order in which they leave is Thread-1, Thread-3, Thread-4, Thread-2; use the **isAlive( )** and **join( )** methods to accomplish this).

The passengers disperse after the flight and go off to their respective destinations (threads terminate). The flight attendant cleans the aircraft and is the last to leave after all the passengers. (thread terminates).

TO REITERATE, you are not permitted to use monitors, semaphores, collections or any other synchronization tool if they were not discussed in class. You can, however, use the **volatile** modifier and the classes **AtomicInteger** and **AtomicBoolean**.

A few things to note:

1. In order to keep track of time, there needs to be a clock thread. The clock thread signals the flight attendant when it's time to start the boarding process and disembark the plane.

The clock will sleep for a fixed amount of time before and in between these two events. After all passengers have disembarked, the clock will announce that it is terminating and then terminate.

2. Make it very clear which passenger thread departs the plane and what their seat number is. A message indicating departure might look like this:

   Passenger-1: is in seat 6 and departs the plane.

3. Initial values:

   numPassengers = 30
   groupNum = 4
   counterNum = 3

Guidelines:

1. Do not submit any code that does not compile and run. If there are parts of the code that contain bugs, comment it out and leave the code in. A program that does not compile nor run will not be graded.

2. Closely follow all the requirements of the project's description.

3. The main method is contained in the main thread. All other thread classes must be manually created by either implementing the Runnable interface or extending the Thread class. **Separate the classes into separate files (do not leave all the classes in one file, create a class for each type of thread). DO NOT create packages.**

4. The project asks that you create different types of threads. There is more than one instance of a thread. **No manual specification of each thread's activity is allowed (e.g. no Passenger5.goThroughTheDoor())**

5. Add the following lines to all the threads you make:

```
public static long time = System.currentTimeMillis();

public void msg(String m) {
    System.out.println("["+(System.currentTimeMillis()-time)+"] "+getName()+": "+m);
}
```

It is recommended that you initialize the time at the beginning of the main method, so that it is unique to all threads.

6. There should be output messages that describe how the threads are executing. Whenever

you want to print something from a thread use: msg("some message about what action is simulated");

7. NAME YOUR THREADS. Here's how the constructors could look like (you may use any variant of this as long as each thread is unique and distinguishable):

```
// Default constructor
public RandomThread(int id) {
    setName("RandomThread-" + id);
}
```

8. Design an OOP program. All thread-related tasks must be specified in its respective classes, no class body should be empty.

9. thread.sleep() is not busy wait. while (expr) {..} is busy wait.

10. "Synchronized" is not a FCFS implementation. The “synchronized” keyword in Java allows a lock on the method, any thread that accesses the lock first will control that block of code; it is used to enforce mutual exclusion on the critical section. **FCFS should be implemented in a queue or other data structure**.

11. DO NOT USE System.exit(0); the threads are supposed to terminate naturally by running to the end of their run methods.

12. A command line argument must be implemented to allow changes to the **nPassenger** variable.

13. Javadoc is not required. Proper basic commenting explaining the flow of the program, self-explanatory variable names, correct whitespace and indentations are required.

A helpful tip:
-If you run into some synchronization issues, and don't know which thread or threads are causing it, press F11 which will run the program in debug mode. You will clearly see the thread names in the debug perspective.

Setting up project/Submission:
For those that use Eclipse:
Name your project as follows: LASTNAME_FIRSTNAME_CSXXX_PY, where LASTNAME is your last name, FIRSTNAME is your first name, XXX is your course, and Y is the current project number.

For example: Doe_John_CS340_p1

To submit:
-Right click on your project and click export
-Click on General (expand it)
-Select Archive File
-Select your project (make sure that .classpath and .project are also selected)
-Click Browse, select where you want to save it to and name it as
LASTNAME_FIRSTNAME_CSXXX_PY
-Select Save in zip format (.zip)
-Press Finish

PLEASE UPLOAD YOUR FILE ON BLACKBOARD ON THE CORRESPONDING COLUMN.

The project must be done individually with no use of other sources including Internet.

No plagiarism, no cheating.