CSCI 340, FL21 – p2
Instructor: Simina Fluture, PhD

340 Project 2 – FALL 21
The project must be done individually. No exceptions.

You have the following choices:
1. You can submit a pseudo-code implementation:
Due Date: December 10 (no late submission allowed)

OR

2. You can submit a java-code implementation; in that case the project weight will be 7% instead of 6% (you get 1% EC on project 2's weight).
I wish I could give you more but too many students are plagiarizing.
Due Date: Tuesday, December 14 (no late submission allowed)

DO NOT SUBMIT BOTH TYPES OF IMPLEMENTATION: EITHER YOU SUBMIT THE PSEUDO-CODE, OR YOU SUBMIT THE JAVA-CODE.

Directions: You are asked to synchronize the threads of the following story using semaphores and operations on semaphores. Do NOT use busy waiting or any other synchronization tools besides the methods (specified below) of the semaphore class. Please refer to and read the project notes carefully, tips and guidelines before starting the project.

Plagiarism is not accepted. Receiving and/or giving answers and code from/to other students enrolled in this or a previous semester, or a third source including the Internet is academic dishonesty subject to standard University policies.

Election Day

Voters are anxious to vote. Once arrived at the designated voting place (simulated by sleep of random time) they will have to **wait** until one of the available ***ID_checker(s)*** will check their ID.

Once done with the ID check, voters can enter the voting kiosk line to fill out a ballot form with their information. There are *num_k* kiosks. The voter will pick the kiosk with the shortest line and **wait** until they get in front so they can enter their information (simulate the wait using **sleep for a long time)**. A ***kiosk_helper*** will control all this movement. He will inform the next voter when a kiosk becomes available and that he/she can move on. When a voter is done completing his/her ballot (and usually it takes some time – simulate by sleep of random time), he/she will let the helper know and the helper will **signal** the next voter on that specific line so that he/she can move on.

Finally, the last step. There are *num_sm* scanning machines. Voters will group in groups of size num_sm and **wait** until the scanning_helper will let them know they can move on. The scanning helper will signal them as soon as all the machines are available.

Once they have their ballot (simulate by sleep of random time) scanned, the voter leaves the scanning room and the building. The last voter to leave the building will signal the helpers that it is time for them to go home too.

CSCI 340, FL21 – p2
Instructor: Simina Fluture, PhD

1. Pseudo-code implementation

You are asked to synchronize the threads of the story using semaphores and operations on semaphores. Do NOT use busy waiting.

Use pseudo-code similar to the one used in class (NOT java pseudo-code).
Mention the operations that can be simulated by a fixed or random sleep_time.
Your documentation should be clear and extensive.
Explain the reason for each semaphore that you used in your implementation. Explain each semaphore type and initialization.
Discuss the possible flows of your implementation. Deadlock is not allowed.

2. Java-code implementation

Using Java programming, synchronize the threads, in the context of the problem. Closely follow the implementation requirements. The synchronization should be implemented through Java semaphores and operations on semaphores (acquire and release)

Keep in mind that semaphores are shared variables (should be declared as static)

For semaphore constructors, use ONLY:

Semaphore(int permits, boolean fair)
Creates a Semaphore with the given number of permits and the given fairness setting.

In methods use ONLY: acquire(), release();

You can also use:

getQueueLength()

Returns an estimate of the number of threads waiting to acquire.

DO NOT USE ANY OF THE OTHER METHODS of the semaphore's class, besides the ones mentioned above.
Any **wait** must be implemented using P(semaphores) (acquire).
Any shared variable must be protected by a mutex semaphore such that Mutual Exclusion is implemented.
Document your project and explain the purpose and the initialization of each semaphore.

DO NOT use synchronized methods or blocks, Do NOT use wait( ), notify( ) or notifyAll( ) as monitor methods. Whenever a synchronization issue can be resolved use semaphores and not a different type of implementation (no collections, no more atomic classes)
Use appropriate System.out.println() statements to reflect the time of each particular action done by a specific thread. This is necessary for us to observe how the synchronization is working.
Submission similar to project1. Name your project: YourLastname_Firstname_CS340_p2
Upload it on Blackboard.

Default values:
num_voters = 20
num_ID_checkers = 3
num_k = 2
num_sm = 4

- 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.

- 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.

- 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())

- 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.

  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");

- 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);
}
```

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

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

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

Setting up project/Submission:

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.

PLEASE ZIP ONLY THE JAVA FILES (SOURCE FILES). PLEASE UPLOAD YOUR FILE ON BLACKBOARD IN THE CORRESPONDING COLUMN.