cs340 – Spring 2017
Project 2

Exam Day
(synchronization of threads using semaphores)

Due Date: May 20th

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)

As semaphore constructor use ONLY: Semaphore(int permits)
Creates a Semaphore with the given number of permits and nonfair fairness setting.

As methods use ONLY: acquire(), release(); You can also use:

getQueueLength()
Returns an estimate of the number of threads waiting to acquire.

hasQueuedThreads()
Queries whether any threads are 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 (beside the operations on semaphores).

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.

You should keep the concurrency of the threads as high as possible, however the access to shared structures has to be done in a Mutual Exclusive fashion, using a mutex semaphore.

Many of the activities can be simulated using the sleep(of a random time) method.

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.

I strongly suggest you to try a pseudo-code first and trace it in order to see if your implementation will be correct.

If you don't know to do a Java implementation, write a pseudo-code solution for up to 38 points. The pseudo-code must be documented and not follow any specific programming language.

Students come to school (simulated by sleep of random time) and ***wait*** for the instructor to arrive and open the door. Each student should attempt to take two exams. After two attempts (successful or not) student will group in groups of size **group_size**.

Once the instructor arrives (sleep of random time), he allows students to enter the classroom and **wait** until is time for the exam to start. Students enter the classroom up to the classroom's *capacity* and eventually **wait** for the Instructor to handout the exam.
The last student to fit in the classroom will let the instructor know that the exam needs to start.
If the classroom is full or if the student didn't make it by the time the exam starts, we consider that the student missed the exam.

The Instructor hands out the exam (signal) to students in the order in which the student entered the classroom. Next students work on the exam and **wait** for the Instructor to signal them when exam ended.

Throughout the duration of the exam, the Instructor sleeps (of fixed amount of time). When he wakes up, he allows all the students to leave the classroom.

Next, the instructor and students (who took the exam) will take a break (sleep of random time) and get ready for the next exam. NOTE: The students who missed the exam are already waiting to enter the classroom for the next exam.

At the end of the day, the exam grades for each student should be displayed (exam1, exam2). You can assign grades by generating random numbers between 10 and 100. If a student missed an exam, his grade will be 0.

Once the grades are posted, students will groups of **groups_size** and next they will leave**.** Give a display of the name of students grouping together. The instructor waits for all students to leave. The last student to leave will let the teacher that is time to go home. The instructor will terminate after all students leave.
The main thread will terminate once the instructor leaves as well.

Default values**: number of students: 14**
class capacity: 8
group_size : 3

Choose appropriate amount of time(s) that will agree with the content of the story. I haven't written the code for this project yet, but from the experience of grading previous semester's projects, a project should take somewhere between 45 seconds and at most 1 minute and ½, to run and complete.

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. Main class is run by the main thread. The other threads must be manually specified 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. **Don't create packages**.

4. The program asks you to create different types of threads. There is more than one instance of the thread. No manual specification of each thread's activity is allowed (e.g. no Student2.getInClssroom()).

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 to initialize time at the beginning of the **main method**, so that it will be unique to all threads.

6. There should be printout messages indicating the execution interleaving. Whenever you want to print something from that thread use: msg("some message about what action is simulated");

7. NAME YOUR THREADS or the above lines that were added would mean nothing.
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. **No** use of **wait( ), notify( ) or notifyAll( )** are allowed.

11. **FCFS should be implemented in a queue or other data structure**.

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

13. Command line arguments must be implemented to allow changes to the **number students, capacity**, **group_size.**

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

Tips:
-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:
In 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_p2

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**, Create directory structure for files and also Compress the contents of the file should be checked.
-Press Finish

Upload the project on Blackboard.

The project must be done individually, not any other sources. No plagiarism, No cheating. Read the academic integrity list one more time.