Cs340 – SUMMER 2012 Project 2

You are asked to synchronize the threads of the following story using semaphores and operations on semaphores. Do NOT use busy waiting. Any wait for students and teacher should be implemented using wait on semaphores.

Note: the projects must be done individually. No exceptions.

Collaboration will be treated as cheating.

The initial regular weight for project 2 (as a programming project) was 7%. You have the following choices:
1. You can submit a pseudo-code implementation; in that case the project will weight 6% instead of 7 % (you loose 1%).
Due Date: Wed., Aug.15th.

Or
2. You can submit a java-code implementation; in that case the project will weight 9% instead of 7% (you get 2% EC).
Due Date: Sun, Aug. 19th.

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

1. Pseudo-code implementation

You are asked to synchronize the threads of the following 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.

Defaults values:	**capacity**	**15**
	#Students	**11**
	#questions_A	**4**
	#questions_B	**3**

Submission:
Submit a printout of your solution to me or leave it for me in the department office.
Email a copy of it to: fluture@cs.qc.cuny.edu

2. java-code implementation

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

DO NOT use synchronized methods (beside the operations on semaphores).

Do NOT use wait( ), notify( ) or notifyAll( ) as monitor methods. Use only the semaphore class and its methods.

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 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.

The number of threads should be read as command line arguments.

Defaults values:	**capacity**	**15**
	#Students	**11**
	#questions_A	**4**
	#questions_B	**3**

Submission Requirements similar to project1.

Online Office Hours

Students come to school and **wait** in front of the computer lab until it is time for the lab to open. Once the lab is open, students enter the lab to the **capacity** of the lab. If the capacity is reached no additional students can enter the lab. Students beyond the lab capacity will terminate their execution.

There are two types of questions that the student might have. The type and number of questions that each student will be allowed to ask is determined randomly.
Type A: the student can email the teacher up to **#questions_A**. These questions do not need an immediate answer from the teacher.
Type B: the student will have a chat session with the teacher. During a chat session, the student is allowed to ask at most **#questions_B.**

Once the student enters the lab, he will take some time to think about his questionA. *(use* ***sleep(random time)****).* Each question should contain a timestamp (*use* ***age()***) and the name of the student (*use* ***getName()***).

Type A questions: As soon as the student determines his questionA, he will send it to the teacher. All of the questions sent by students will be answered by the teacher **in the order in which they have been sent.**

After the student is done sending his typeA questions, if he wants to have a chat with the teacher, he needs to **wait** until the chat session starts and it is his turn to chat. Once it is his turn to chat, he will send a question to the teacher and **wait** for an answer. If he has fewer than **#questions_B** he needs to let the teacher know when it is his very last question (when his chat session ends).

After his chatting session, the student will browse the Internet **waiting** for the online office hour to end.

Next the students will leave the lab in descending order of their name. For example if there are 4 student threads named from T0 to T3. T0 will **wait** for T1, T1 will **wait** for T2, T2 will **wait** for T3 and T3 after some sleep of random time will terminate.

The teacher arrives at his office before the online office hour begins. He spends some time answering typeA questions, but might not be able to answer all of them. Once the time for the online chatting session arrives, he will inform a student **wait**ing to chat that he is online and can take questions. He chats with students.

When done chatting, if there is any remaining time until office hours ends, the teacher checks if there are any new or unanswered typeA questions and answers them.
Once the office hour ends, the teacher will terminate as well. The timer will keep track of the times and, if necessary, signal the correct threads.

Synchronize the *student* threads, *teacher* thread and *timer* thread following the conditions of the story. Instead of creating a timer thread, you can chose to have the main thread taking track of the specific times. It is your choice.

For example a possible time scale can be (you might consider additional time intervals):

Teacher's arrival time
Start of online office hour
start online chatting session
end online chatting session
end online office hour

Use appropriate System.out.println() statements in the program. Also make use of the **age()** method provided to keep track of the Threads age and action time.
There are no real messages exchanged; it is just a simulation.