Concordia University Computer Science & Software Engineering
Faculty of Engineering and Computer Science

COMP 346 – Fall 2017
Programming Assignment 3

Due Date	**By 11:59pm Friday November 24, 2017**
Format	Programming assignments must be typed and submitted online to Moodle system. **Scanned Hand-written assignments will be discarded**.
Late Submission:	none accepted

Purpose: In this assignment, you will be using semaphores to guard concurrency and avoid race conditions involving a shared data structure. The shared data structure is a stack of characters, with top and bottom pointers indicating the top and the bottom of the stack respectively. The stack is in a valid state as long as the characters in the stack are in an increasing order or the stack is empty. Such a valid state is shown in the following diagram.

CEAB/CIPS Attributes: Design/Problem analysis/Communication Skills

Total Marks: 50

Objectives: This programming assignment is an extension to the classical problem of synchronization – the Dining Philosophers problem. You are going to solve it using the Monitor construct built on top of Java's synchronization primitives. The extension here is that sometimes philosophers would like to talk, but only one (any) philosopher can be talking at a time while he/she is not eating. Moreover, deadlocks and starvations are possible and need to be handled.

The source code for this assignment is supplied separately in a zip file. You will need to fill in the missing code. You have to complete the following **4** tasks as part of this assignment:

<u>**Note**</u>: All code written by you should be well commented. This will be part of the grading scheme.

Task 1. The Philosopher Class

Complete implementation of the Philosopher class, i.e., all its methods, according to the comments in the code. Specifically, *eat()*, *think()*, *talk(),* and *run()* methods have to be fully implemented. Some hints are provided within the code. Your added code must be well commented.

Task 2. The Monitor

Implement the Monitor class for the problem. Make sure that it is correct; **both deadlock- and starvation-free (**<u>Note</u>: this is an important criterion for the grading scheme**)**; uses either Java's synchronization primitives such as wait()/notify()/notifyAll() or uses Java utility *lock* and condition variables; and **does not use** any Semaphore objects. Implement the four methods of the Monitor class; specifically, *pickUp()*, *putDown()*, *requestTalk()*, and *endTalk()*. Add as many member variables and methods to meet the following specifications:

1. A philosopher is allowed to pick up the chopsticks if they are both available. It has to be atomic so that no deadlock is possible. Refer to the related discussion in your textbook.
2. Starvation is to be handled.

3. If a given philosopher has decided to make a statement, he/she can do so only if no one else is talking at the moment. The philosopher wishing to make the statement first makes a request to talk; and has to wait if someone else is talking. When talking is finished then others are notifies by *endTalk*.

Task 3. Variable Number of Philosophers

Make the program accept the number of philosophers as a command line argument, and spawn exactly that many number of philosophers instead of the default specified in code. If there is no command line argument, the given default should be used. If the argument is not a positive integer, report an error to the user and print the usage information as in the example below:

You can use Integer.parseInt() method to extract an int value from a character string. Test your implementation with a varied number of philosophers. Submit your output from "*make regression*" (refer to the included Makefile for a Linux environment); for any other environment you are using, consult the user's manual for equivalence.

Task 4. Starvation

Briefly explain in a few sentences how your implementation handles starvation. This will facilitate the marking process.

Source Code

There is a file provided with the assignment. You will need this file to work on the assignment. A soft copy of the needed code is available for download from moodle.

File Checklist:

- A2source.java

Deliverables

IMPORTANT: You are allowed to work on a team of 2 students at most (including yourself!). Any teams of 3 or more students will result in 0 marks for all team members. If your work on a team, ONLY one copy of the assignment is to be submitted for both members. You must make sure that you upload the assignment properly using moodle. Assignments are uploaded incorrectly will be discarded and no resubmission will be allowed.

Naming convention for uploaded file: Create one zip file, containing all needed files for your assignment using the following naming convention:
The zip file should be called *a#_studentID*, where # is the number of the assignment *studentID* is your student ID(s) number. For example, for the first assignment, student 1234567 would submit

a zip file named *a1_1234567.zip.* If you work on a team and your IDs are 1234567 and 4567890, you would submit a zip file named *a1_1234567_4567890.zip.*

Submit your assignment electronically via moodle. **Please see course outline for submission rules and format, as well as for the required demo of the assignment.** Working copy of the program(s) and sample outputs should be submitted for the tasks that require them. A text or pdf file with written answers, if any, to tasks 1 to 4 should be provided. Archive all files with an archiving and compressing utility, such as zip, into a single file. Put it all in a file layout as explained below, archive it with any archiving and compressing utility, such as WinZip, WinRAR, tar, gzip, bzip2, or others. **You must keep a record of your submission confirmation from moodle.** This is your proof of submission, which you may need should a submission problem arises.

Grading Scheme

Grading Scheme:

T#	
1	10
2	20(10,5,5)
3	10
4	10

Total:50

References

API: http://java.sun.com/j2se/1.3/docs/api/