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

COMP 346 – Fall 2017
Programming Assignment 1

Due Date **By 11:59pm Friday November 3, 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

Valid stack state:

```
[a][b][c][d][e][$]...
↑           ↑
+--bottom   +--top
```

Invalid stack state:

```
[b][a][c][$][d][$]...
↑           ↑
+--bottom   +--top
```

Multiple Producer and Consumer threads operate on the shared stack, and perform *Push* and *Pop* operations respectively. Details of these operations are elaborated in the following tasks. In addition, there is another thread of execution called CharStackProber, which periodically prints out the contents of the stack; this is for checking the validity of the stack. As a difference from assignment 1, you will be using semaphore(s) for mutually exclusive access to the shared stack.

The source code for this assignment is supplied separately. You will need to separate the source code into multiple files. You will also need to fill in the missing code. You have to complete the following tasks as part of this assignment:

Organize: (i) Organize the given code into multiple source files. (ii) Understand the code. (iii) Ensure that you are working on a native thread environment.

Task 1: Refer to the Semaphore class supplied and answer the following: according to the classical definition of a semaphore, it can be initialized only to a non-negative value. If the value of a semaphore becomes negative (via *Wait()* operation), then its absolute value indicates the number of processes/threads on its wait queue. Does the given semaphore class implementation satisfy these requirements? If not, modify the semaphore class implementation accordingly. Submit the modified code.

Task 2: Refer to the *StackManager* class supplied. Inside the *main()* routine, it starts two *Producer* threads, two *Consumer* threads, and a *CharStackProber* thread which periodically prints out the stack state. Note that all five threads access the same stack, and hence mutual exclusion is mandatory to maintain validity of the stack at all times. You will use the modified Semaphore class (Task 1) to implement the mutual exclusion of the shared data. Following are the specific details of the task:

(i) Each *Producer* thread goes in a loop three times, and each time in the loop it first checks for the character on the top of the stack (without removing it), and then pushes the next higher character into the stack. For instance, if the current top of the stack is 'd' then it pushes 'e'. Fill in the missing code for the *Producer* class. Note that your code must be able to handle the necessary exceptions (e.g., stack full).

(ii) Each *Consumer* thread goes in a loop three times, and each time in the loop it pops out the item from the top of the stack. Fill in the missing code for the *Consumer* class, taking proper care for exceptions.

(iii) The *CharStackProber* thread goes in a loop six times and each time in the loop, it prints out the contents of the stack in the format shown in the following example. For example: if the current contents of the stack are 'a', 'b', 'c', then it prints out in the format "Stack S = ([a],[b],[c],[$],[$],[$],[$],[$],[$],[$])". Fill in the missing code. Note that your output must conform to this format so that Task 3 in the following can be addressed.

Submit the complete program.

Task 3: Complete the following:

(i) Run the completed program three times, and dump the output of run i to a file named: Output_i.txt (i = 1..3).

(ii) We are also interested in specifically checking the output generated by *CharStackProber*. To achieve this, you run the program three more times, however using the following command: *java StackManager | grep 'Stack S'* (Note: this command is to be issued from the shell command prompt on a Linux environment; for equivalent commands on your Java development toolkit, consult the manual and/or talk to your TAs). Dump the output of each run i to the file named: Grep_i.txt (i = 1..3).

Submit the outputs of the above.

Task 4: You will bring in some extra synchronization to the completed code as follows: the two *Producer* threads, which run concurrently, must first complete pushing characters into the stack, only after which the two *Consumer* threads can concurrently access the stack. There is no restriction on the *CharStackProber* thread and so its code remains unchanged. You must not alter any code of the *main()* routine of the *StackManager* class. The required synchronization must be achieved through the use of Semaphores (the modified Semaphore class in Task 1), and mutual exclusion of the stack must be preserved at all times. Use minimum possible synchronization to achieve the result. Submit the modified program.

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 to the correct directory of **Lab Assignment 1** using moodle. Assignments are uploaded to the wrong directory 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/