COMP 346 – Fall 2020
Programming Assignment 2

Due: 11:59 PM – October 30, 2020

Mutual Exclusion and Barrier Synchronization

1. Objectives

The objective of this assignment is to allow you to learn how to implement critical sections and barrier synchronization of concurrent threads using semaphores.

2. Preparation

2.2 Source Code

There are four files to be distributed with the assignment. A soft copy of the code is available to download from the course website.

2.2.1 File Checklist

Files distributed with the assignment requirements:

```
common/BaseThread.java
common/Semaphore.java
BlockManager.java
BlockStack.java
```

2.3 BlockManager and BlockStack

A version of the `BlockManager` class is supplied. The `BlockStack` will have to be modified in accordance with tasks 1 and 2. You will need to ensure that your code matches the method naming in these files. Specifically, in `BlockStack`:

- The method to get `iTop` is to called `getITop()`; //change code
- The method to get `iSize` is called `getISize()`; //change code
- The method to get the stack access counter is to called `getAccessCounter()`;
- A new utility method is expected to be there, called `isEmpty()`, which returns true if the stack is empty; false otherwise. The definition of the method could be as following:

```
public boolean isEmpty()
{
      return (this.iTop == -1);
}
```

The exception handling in `BlockManager` is done in a general way, so it should theoretically cover all your implementations. However, you must ensure that your exceptions (created in task 2) work with the `BlockManager` class.

3. Background

We are going utilize the `BaseThread` data member – `siTurn` – to indicate the thread ID (TID) of the thread that is allowed to proceed to <phase II> (details below). There are four other methods in the `BaseThread`. Two of them are `phase1()` and `phase2()`, which barely do anything useful. However, they indicate the phase # and the state of the currently executing thread. Another method is `turnTestAndSet()`. This method tests both the turn and the TID for equality, and increments/decrements the turn if they are equal. The method returns `true` if the turn has changed; `false` otherwise. This method is intended to be used primarily in the last task.

➔ Note: This is a very good method to mess things up!

• The Semaphore is a class that implements the semaphore operations `Signal()/V()` and `Wait()/P()` using Java's synchronization monitor primitives, such as, `synchronized`, `wait()`, and `notify()`. Objects of this class are going to be used to once again bring operations into the right order in this hostile world of synchronization.

• The Synchronization Quest is based on the idea, where you have to synchronize all the threads according to some criteria. These criteria are:

1. PHASE I of every thread must be done before any of them may begin PHASE II.
2. PHASE II must be executed in the ascending order (...4-5-6…) of their TID.

4. Tasks

The following tasks are given. You must make sure that you place clear comments for every task that involves coding changes that you have made. This will be considered in the grading process.

Task 1: Writing Some Java Code, Part I

Declare an integer "stack access counter" variable in the BlockStack class. Have it incremented by 1 every time the stack is accessed (i.e. via push(), pop(), pick(), or getAt() methods). Print indication messages for successful push and pop operations. Print '*' instead of '$' for each empty position in the stack (modify given code).

When the main thread terminates, print out the counter's value. Submit the modified code and the output.

Task 2: Writing Some More Java Code, Part II

The BlockStack class has somewhat bogus implementation, no checking for boundaries, etc... Most of the class is also not quite correctly coded from the good object-oriented practices point of view like data hiding, encapsulation, etc.

1. Make the `iSize`, `iTop`, `acStack`, and possibly your stack access counter private and create methods to retrieve their values. Do appropriate changes in the main code.

2. Modify the push() operation of the BlockStack class to handle the case when the stack is empty (last element was popped). Calling push() on empty stack should place an 'a' on top.
3. Implement boundaries, empty/full stack checks and alike using the Java's exception handling mechanism. **Declare your own exception**, or a set of exceptions. Make appropriate changes in the main code to catch those exceptions. Print "Empty Stack !!!" message in case of empty stack and "Full Stack !!!" message in case of full stack.
NOTE: If do you catch ArrayIndexOutOfBoundsException it's a good thing, but it's not your own exception :-)

Task 3: Atomicity Bugs Hunt with Mutex
Compile and run the Java app given to you as is, and look at the output. After possibly getting scared of what you have seen, you will have to correct things. Yet, before you do so, make execution of the critical sections atomic. Use the mutex semaphore for that purpose.

Task 4: The Right Order, Take I
In this task you have to ensure that the PHASE I of every thread is completed before PHASE II of any thread has a chance to commence. Each time a thread is finished print a message that thread-*i* has finished PHASE I. Finally, print a message indicates that all the threads finished PHASE I. You still need to do so using semaphore operations. Submit the output and a context diff to the original sources.

Task 5: The Right Order, Take II
The second synchronization requirement on top of the one of "Take I" is that all 10 threads must start their PHASE II in order of their TID, i.e. 1, 2, 3, 4 ... The second semaphore, s2, and `turnTestAndSet()` method are provided to you to help with this. Each time a thread attempts to start but fails because of incorrect order, please print a message that thread-*i* has attempted but waiting for its turn to finish PHASE II. Finally, print a message indicates that all the threads finished PHASE II. Submit the output the modified source code.

---

5 Implementation Notes

• You may NOT use the `synchronized` keyword. Use the `Semaphore` objects provided to you as your primary weapon.

• Refer to the Synchronization Tutorials for barrier synchronization examples.

• It's up to you to determine the initial values of the semaphores as long as your solution provides correct mutual exclusion and synchronization without deadlock, starvation, etc.

---

6 Deliverables

The assignment can be done in a group of two students. The deliverable consists of a well-commented code and at least on page report specifying the high-level description of the code (description of the 5 tasks that you have done and the rationale behind your synchronization methods). This assignment will take 7% of your total mark for programming assignments. Also for this assignment 85% of the mark is dedicated to your code and 15% to your report.

Submission instructions:

-Copy the java files for each task in one folder, name the folders task 1, task 2, ... Then copy all the folders along with your report file in a different folder and put your student id_(assignment number) as the folder name.

-Zip the folder and Submit it via Moodle.

-If you are working in pair put the student id of both students as the name of the Zip file.

Finally, please notice that a demo, for about 10 minutes, is required with the marker. Both team members must be present for the demo. Failing to demo your assignment will result in zero mark regardless of your submission.

---

7 Grading Scheme

Grading Scheme:

T#	Mark
1	/1
2	/1
3	/2
4	/2
5	/2.5
Report	/1.5

Total: 10

8 References

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