



Concordia University  
Department of Computer Science and Software Engineering  
Comp 5461 – Operating Systems  
Summer 2025

Programming Assignment 2  
Submission through Moodle  
**Due Date:** 22 July 2025 @ 23:59.

## **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:

BaseThread.java  
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 called `getITop()`; //change code
- The method to get `iSize` is called `getISize()`; //change code
- The method to get the stack access counter is 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 to utilize the `BaseThread` data member – `siTurn` – to indicate the thread ID (TID) of the thread that is allowed to proceed <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 bring operations once again 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 criterias 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 indicating that all the threads finished PHASE II. Submit the output of 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. Although it is discouraged to initialize a semaphore to a negative value in the classical definition (and you should not do so in any solutions related to the theory component of the course), you may do so in this assignment provided you can justify your choice (efficiency, overhead, semantics, etc.).

## 6. Deliverables

**IMPORTANT:** You are allowed to work on a team of 2 students at most (including yourself!). You and your teammate must be in the same section. 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 **Programming Assignment 2** using Moodle. Assignments 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 12345678 would submit a zip file named *a1\_12345678.zip*. If you work on a team and your IDs are 12345678 and 34567890, you would submit a zip file named *a1\_12345678\_34567890.zip*.

Submit your assignment electronically via Moodle based on the instruction given as indicated above. **Please see course outline for submission rules, as well as for the required demo of the assignment.** A working copy of the code and a sample output should be submitted for the tasks that require them. A text file with answers to the different tasks should be provided. 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.** This is your proof of submission, which you may need should a submission problem arises.

For every task, submit the complete output and a context diff to the original code. Thus, a file list to submit might look like this:

BlockManager1.java  
BlockStack1.java  
Output1.txt

BlockManager2.java  
BlockStack2.java  
Output2.txt

BlockManager3.java  
BlockStack3.java  
Output3.txt

BlockManager4.java  
BlockStack4.java  
Output4.txt

BlockManager5.java  
BlockStack5.java  
Output5.txt

Archive these files into a zip file called `pa2.zip` and submit it electronically using Moodle system. If you work on a team, only one of the members should submit the assignment. Do not submit two copies.

## 7. Grading Scheme

Grading Scheme:

=====		
T#	MX	MK
-----		
1	/1	
2	/1	
3	/2	
4	/3	
5	/3	
-----		

Total:

(T# - task number, MX - max (out of), MK - your mark)

## 8. References

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