Karim Rhoualem
Student #26603157

Due: Sunday, November 1, 2020
Section: FK-X

Programming Assignment #2 – Lab Report

The program is comprised of four classes: BlockManager, BlockStack, BaseThread, and Semaphore. They all work together to recreate the barrier synchronization process amongst concurrently running threads. Semaphores are used to do so and to ensure that each class' critical section is accessed atomically (meaning that no other thread can access their atomic section until they are signalled to do so). The program begins by creating several threads of each class. As the threads are generated in a random order, each of them must complete their phase 1 before moving on to their phase 2 (note that each of these phases simply print statistical information about the state of the stack). The order of phase 2 must be completed in increasing order according to the number of the thread ID. Prior to reaching their phase 2, their critical sections (where the stack manipulation occurs) must be completed atomically, and as such, require that semaphores be used for synchronization as well.

The BlockStack class is the class that encompasses the main functionality associated with the program's stack. It defines the stack and its associated pop(), push(), getters, and observational methods to manipulate and read the stack. The BaseThread class defines the functionality of both phases, as well as the turnTestAndSet() method that is used to determine the order of execution of the phase 2's (increasing order according to each thread's ID). The Semaphore class recreates the functionality of ordinary semaphores by defining the functionality of the wait() and signal() methods. Additionally, semaphore class also defines methods to increment and read counters that are used to determine when all the threads have completed execution of phase 1 and phase 2.

For task 1, a stack access counter variable is defined in the BlockStack class. This counter displays the final count indicating the number of times that the stack was accessed along the course of program execution. At program termination, the counter almost always displays an access count value of 138. Additionally, for task 1, console messages are printed to display successful stack pushes and pops, as well as correct representation of stack contents.

For task 2, proper object-oriented programming concepts are implemented via the utilization of encapsulation and data hiding. The push and pop methods, in all scenarios, handle the situations where the stack is empty or full. Rather than using exceptions that may hinder program performance, conditional statements are used to do nothing to the stack when it is either full or empty. In those scenarios, the program just displays the current contents of the stack and moves on to the next critical section.

For task 3, a semaphore called 'mutex' is wrapped around the critical sections of the AcquireBlock, ReleaseBlock, and CharStackProber classes that are found within the parent BlockManager class. Their critical sections access the stack and there require semaphore synchronization to ensure that accurate stack information is always available.

For task 4, phase 1 of every thread is assured to complete before any phase 2 executes. This is done by using two semaphores, s1 and s2. The program is modelled after the following depiction of barrier synchronization:

```
semaphore s1 = -n + 2, s2 = 0;
process P1      process P2   ...    process Pn
  <phase I>       <phase I>  ...      <phase I>
  P (s1)          V (s1)     ...      V (s1)
  V (s2)          P (s2)     ...      P (s2)
                  V (s2)     ...      V (s2)
  <phase II>      <phase II> ...      <phase II>
```

Process 1 corresponds to the AcquireBlock class, and the other two processes are ReleaseBlock and CharStackProber. The program has 10 total threads (3 AcquireBlock threads, 3 RleaseBlock threads, and 4 CharStackProber threads). The semaphore s1 value is originally set to -3, that way once the Release Block and CharStackProber threads (a total of 7) have executed, s1 has a value of 4, which allows every thread of the AcquireBlock class to pass their s1.wait() statements. This approach is coupled with the use of semaphore s2 (originally given a value of 0). The ReleaseBlock and CharStackProber threads will have to wait until each AcquireBlock class executes before being able to access phase 2. This works because the AcquireBlock threads will be the last ones to complete phase 1.

For task 5, the turnTestAndSet() method, defined within the BaseThread class, implements the logic in the AcquireBlock, ReleaseBlock, and CharStackProber classes that determines the order in which each phase 2 will execute. Each phase 2 will execute in increasing order of their thread ID value. The turnTestAndSet() method handles this by blocking its critical section with a new binary semaphore, and then checking the thread ID against an siTurn variable the is progressively incremented to keep track of which thread's turn it is to execute phase 2 (this count starts at a value of 1). This way, each phase 2 progressively executes according to the thread ID count. Along the way, messages are displayed in the console when a thread tries to run before it should, and when it is actually time for a thread to run its phase 2. Finally, a message is displayed when all the threads have completed their phase 2, and then the program terminates.