Name: Soliman Alnaizy

NID: so365993

Smstr: Spring 2019

Course: COP 4520

==========

Programming Assignment #3: ELIMINATION BACKOFF STACK

==========

HOW TO RUN: Compile using "javac EBOStack.java" on a Linux based terminal. I have provided a Test case in a file named "Main.java". The test case randomly calls the push() or pop() methods 10,000 times. You can use the test case I provided to demo the EBOStack.

OBJECTIVE: To design a lockfree stack that implements a backoff Elimination array to reduce contention and bottle-necking.

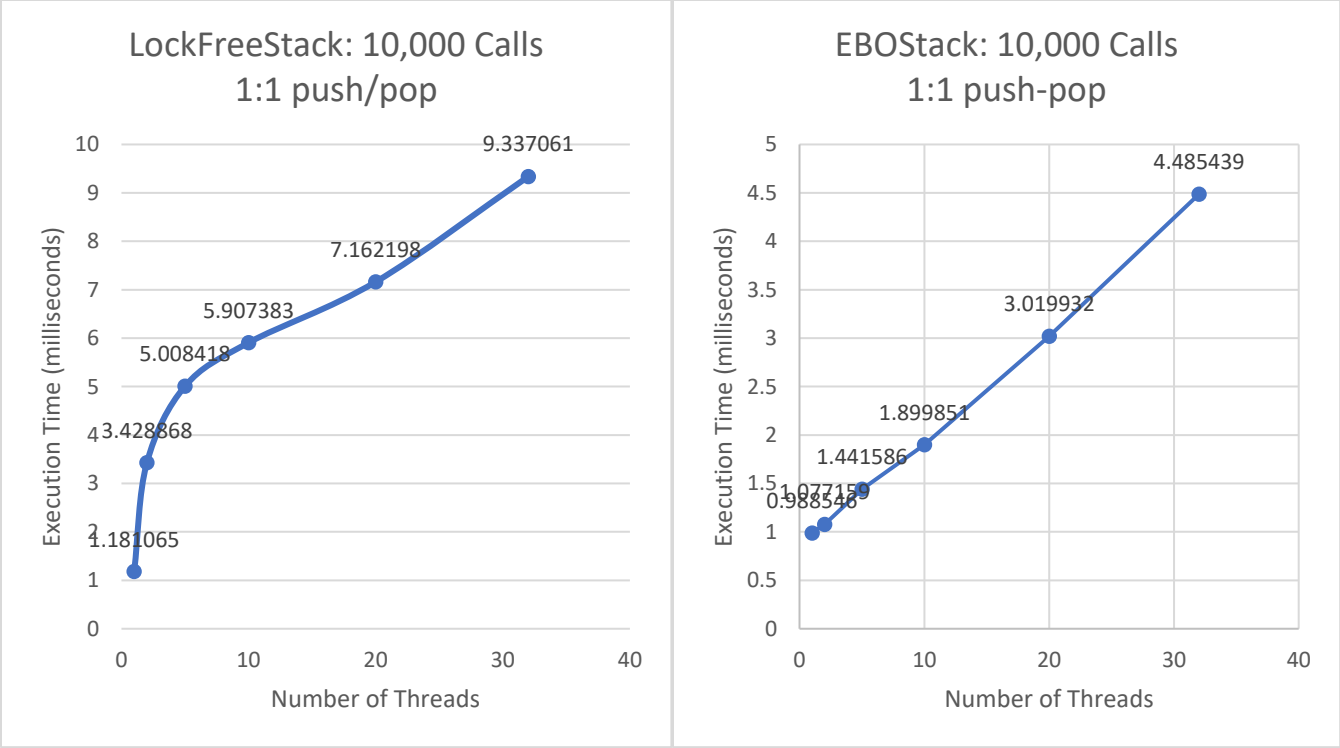
IMPLEMENTATION: When push() is called, the program tries to do a head insert onto a linked list by using a single compareAndSet() method. If the CAS fails, it means that other thread(s) is/are contending for the head node. If that happens, the thread then backs off for a certain amount of time that is decided by the ThreadLocal RangePolicy object. During that time, the thread looks for another thread that's performing a pop() method. It keeps searching until it finds another thread that's popping, or the time limit has exceeded. If it found a popping thread, it then returns null. If the time limit exceeds, it then returns the original item to the method that called. The same logic happens in a pop() method, except the pop() method is searching for a thread that's pushing a node instead. If a pop() fails, then it returns null. If it succeeds in finding a pushing thread, then it swaps values with that thread and returns that object.

EFFICIENCY: Theoretically, an EBOStack is more efficient that a plain lock free stack since the backoff mechanism reduces contention on the top node of the stack. However, in practice, we are still trying to find the ideal minimum and maximum backoff time

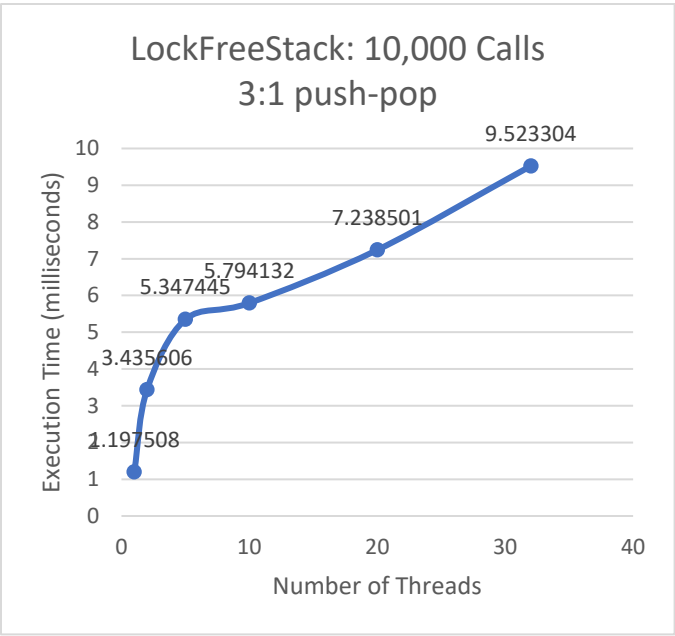
CORRECTNESS: A stack is a FIFO structure. We can guarantee that items that are pushed on the stack first and popped first. Both the push() and pop() methods use Java's atomic compareAndSet() method.

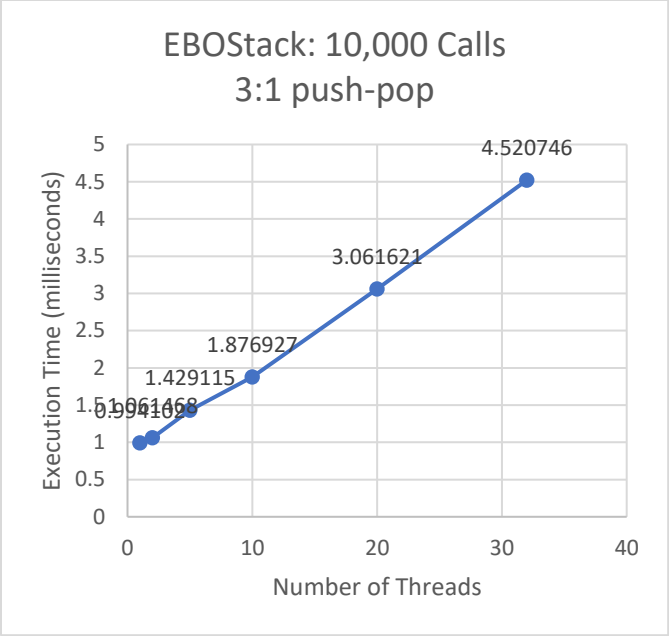
If the CAS succeeds a push() method will perform a head insertion onto the linked list, and in the case of a pop(), the stack will perform a head deletion. In the case where the CAS fails, it then takes the node to the Elimination array and looks for an opposite thread to cancel out. We can safely discard the threads that are pushing and popping since they technically aren't on the stack yet. Preserving the overall FIFO nature of the stack.

PROGRESS: The threads are non-blocking. In the case of an empty stack, the thread doesn't spin indefinitely until another thread pushes onto the stack. Instead, an EmptyStackException is thrown after a certain time of spinning and the thread then continues normally. The EBOStack is also deadlock free because no locks are used. It is also starvation free because all threads are non-blocking, and eventually all threads will gain access to the stack. Below are a few graphs that show the average time it takes for a program of a m threads to make n method calls. The first graph shows a Time by Number of Threads graph. The method calls were 50% push() methods and 50% pop() methods. The second graph is pushing and popping randomly at a 3:1 ratio respectively. The last graph is pushing and popping 10,000 times randomly at a 1:3 ratio. Each graph has the execution time of a simple LockFreeStack to serve as reference.

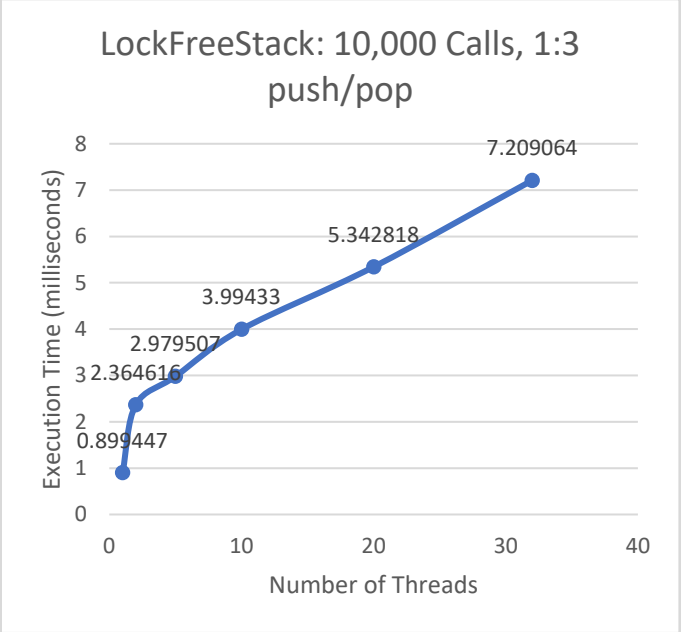


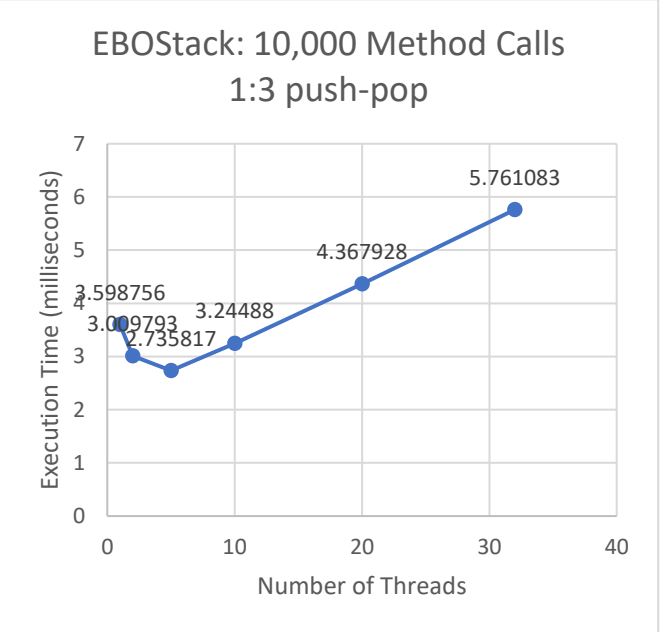
The graphs above show that the EBOStack performs about 50% faster than the normal LockFreeStack in nearly all cases where the ratio of push() and pop() methods are nearly the same.





When the number of push() calls are greater than the pop() calls, performance is nearly identical to the graphs above.





This is the interesting situation. The LockFreeStack performs better when there is a high number of pop() calls relative to push() calls and low number of threads. However, the performance of the EBOStack starts to improve significantly when we start going higher than 5 threads. This is probably because the elimination array is doing a fantastic job at matching pending popping threads with pushing threads. Reducing contention and bottlenecking at the top of the stack.