COP 4520 Spring 2020

Programming Assignment 2

Notes:

Please, submit your work via Webcourses.

Submissions by e-mail will not be accepted.

Due date: Monday, February 10th by 11:59 PM

Late submissions are not accepted.

This assignment is individual.

You can use a programming language of your choice for this assignment.

If you do not have a preference for a programming language, I would recommend C++.

Grading policy:

General program design and correctness: 50%

Efficiency: 30%

Documentation including statements and proof of correctness, efficiency, and experimental evaluation: 20%

Additional Instructions:

Cheating in any form will not be tolerated.

In addition to being parallel, your design should also be efficient in terms of execution time and memory use.

Problem 1 (60 points)

You are given the partial pseudocode for a concurrent stack class using locks (please see next page). In this design, *head* points to the node at the top of the stack, and *numOps* refers to the number of operations that have occurred during a given execution. The Node class contains a generic value and a pointer to the next node in the stack. The constructor for the stack class allocates a lock and sets the head to NULL, indicating the stack is empty.

The push operation first acquires the lock. If this is successful, a node *n* is allocated with the given value *x. n*'s next pointer is then directed at the current head of the stack. Afterwards, n is made the new head of the stack and *numOps* is incremented.

The pop operation is similar to the push operation. First, it acquires the lock and checks if the stack is empty. If the stack is empty, it releases the lock and returns *null*. Otherwise, it retrieves the node from the top of the stack, changes *head* to *n.next*, and *numOps* is incremented.

Your assignment is as follows:

1. Remove the lock element, and modify the push and pop operations using atomic operations to transform the stack into a lock-free stack. Briefly discuss the correctness and progress properties of your modified lock-free stack.

2. Observe the variables in the stack class and decide which of them should be declared **volatile (or atomic)**. Consider the atomics library functionalities of the programming language of your choice. Find out whether an atomic variable is by default volatile or not. Please explain. Discuss what could go wrong if you run your program without declaring these variables **atomic** or **volatile.** Be careful not to declare variables atomic/volatile if this is not necessary as this would lead to significant performance loss.

Submission Instructions:

Please submit a compressed folder containing your implementation and a brief ReadMe file. Include a main program that spawns 4 concurrent threads accessing the stack. Each thread should execute 150,000 operations and each time it should randomly choose whether to call push or pop. You may want to prepopulate the stack with a certain number of elements, say 50,000. Pre-allocate all of the stack's nodes and if you are using a language that does not provide garbage collection, make sure to collect all deleted nodes in a designated list.

Include a brief report with your submission providing a brief discussion on your use of atomic variables and a summary of your implementation as well as an explanation on the correctness and progress properties of your implementation.

```
1: public class Stack<T>{
2:     Lock lock
3:     Node* head
4:     int numOps = 0
5:
6:     class Node {
7:         T val
8:         Node* next
9:
10:        Node(T _val) {
11:            val = _val
           }
       }
12:    public Stack() {
13:        lock = new ReentrantLock()
14:        head = NULL
15:    }
16:    public bool push(T x) {
17:        lock.lock()
18:        Node *n = new Node(x)
19:        n.next = head
20:        head = n
21:        numOps++
22:        lock.unlock()
23:        return true
24:    }
25:    public T pop() {
26:        lock.lock()
27:        if head == NULL then
28:            lock.unlock()
29:            return NULL
30:        Node *n = head
31:        head = n.next
32:        numOps++
33:        lock.unlock()
34:        return n.val
35:    }
36:    public int getNumOps() {
37:        return numOps
38:    }
39: }
```

Problem 2 (40 points)

Modify the *lock-free stack* from Problem 1 so that it supports a *size()* method that returns the number of elements stored in the container.

To do this correctly, you will have to introduce a *size* field in the Stack class and then modify the *push()* and *pop()* methods accordingly.

Your implementation of the Stack must be *linearizable* and *lock-free.*

Additional Instructions:

- Execute performance evaluation of your Stack. In your benchmark tests, vary the number of threads from 1 to 4 and produce graphs where you map the total execution time on the y-axis and the number of threads on the x-axis. Produce at least 3 different graphs representing different ratios of the invocation of *push*, *pop*, and *size*.
- In your benchmark tests, the total execution time should begin prior to spawning threads and end after all threads complete. All nodes and random bits should be generated before the total execution time begins.
- To avoid "polluting" your results with the overhead of memory management (the standard *malloc* and *free* do not scale well and are not thread-safe), you should have each thread pre-allocate its own supply of nodes, which it can keep on a private list when they are not in the stack.
- For implementing your concurrent queues, you may want to make use of the <atomic> library in C++, or <stdatomic.h> in C.
- Provide a ReadMe file with instructions explaining how to compile and run your program.
- Provide a brief summary of your approach and an informal statement reasoning about the correctness, progress, and efficiency of your design.