COP 4520 Spring 2019

Programming Assignment 2

Note 1:

Please, submit your work via Webcourses.

Submissions by e-mail will not be accepted.

Due date: Friday, March 8th by 11:59 PM

Late submissions are not accepted.

Note 2:

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++.

Problem 1 (40 points)

Modify the *lock-free stack* from Homework Assignment 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.*

Since you will need to update multiple memory locations in a linearizable fashion, your solution will benefit from the use of a *Descriptor Object*. To learn more about *Descriptors* and their use for multiprocessor synchronization, read the following paper:

"Lock-Free Dynamically Resizable Arrays" (the pdf version of the paper is posted on Webcourses with the name LockFreeVector.pdf).

Additional Instructions:

- Execute performance evaluation of your Stack. In your benchmark tests, vary the number of threads from 1 to 32 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.

Grading policy:

General program design and correctness: 60%

Efficiency: 20%

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

Problem 2 (60 points)

Using only single-word atomic read, write, and CompareAndSwap (CAS), design and implement an operation that performs a Restricted Double-Compare-Single-Swap (RDCSS).

We define RDCSS in the following way:

```
word_t RDCSS(word_t *a1, word_t o1, word_t *a2, word_t o2, word_t n2)
{
        r = *a2;
        if (( r == o2) && (*a1 == o1)) *a2 = n2;
        return r;
}
```

Note that *word_t* designates a type that is the size of a memory word.

RDCSS is restricted in that: a) only the location **a2** can be subject to an update, b) the memory it acts on must be partitioned into a *control section* (within which **a1** lies) and a *data section* (within which **a2** lies), and c) the function returns the value from **a2** rather than an indication of success or failure.

RDCSS should operate concurrently with 1) any access to the control section, 2) reads from the data section using a dedicated **RDCSSRead** operation, 3) other invocations of RDCSS, and 4) updates to the data section from other threads that may use regular CAS, subject to the constraint that such CAS operations may fail if an RDCSS operation is in progress.

Your task is to design and implement RDCSS and RDCSSRead. Use a Descriptor object.

Use RDCSS to support a maximum capacity of 100000 nodes in your Stack from Problem 1.

Additional Instructions:

- Execute performance evaluation of your new Stack. In your benchmark tests, vary the number of threads from 1 to 32 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.
- 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 <u>correctness,</u> <u>progress,</u> and <u>efficiency</u> of your design.

<u>Grading policy:</u>

General program design and correctness: 60%

Efficiency: 20%

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