

COP 6616 Fall 2019

Programming Assignment 1

Notes:

Please, submit your work via Webcourses.

Submissions by e-mail will not be accepted.

Due date: Wednesday, September 11th 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 (30 points)

Your non-technical manager assigns you the task to find all primes between 1 and 10^8 . The assumption is that your company is going to use a parallel machine that supports eight concurrent threads. Thus, in your design you should plan to spawn 8 threads that will perform the necessary computation. Your boss does not have a strong technical background but she is a reasonable person. Therefore, she expects to see that the work is distributed such that the computational execution time is approximately equivalent among the threads. Finally, you need to provide a brief summary of your approach and an informal statement reasoning about the correctness and efficiency of your design. Provide a summary of the experimental evaluation of your approach. Remember, that your company cannot afford a supercomputer and rents a machine by the minute, so the longer your program takes, the more it costs. Feel free to use any programming language of your choice *that supports multi-threading* as long as you provide a ReadMe file with instructions for your manager explaining how to compile and run your program from the command prompt.

Required Output:

Please print the following output to a file named primes.txt:

<execution time> <total number of primes found> <sum of all primes found>
<top ten maximum primes, listed in order from lowest to highest>

Notes on Output:

1. Zero and one are neither prime nor composite, so please don't include them in the total number of primes found and the sum of all primes found.
2. The execution time should start prior to spawning the threads and end after all threads complete.

Problem 2 (70 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. 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:        }
13:    }
14:    public Stack() {
15:        lock = new ReentrantLock()
16:        head = NULL
17:    }
18:    public bool push(T x) {
19:        lock.lock()
20:        Node *n = new Node(x)
21:        n.next = head
22:        head = n
23:        numOps++
24:        lock.unlock()
25:        return true
26:    }
27:    public T pop() {
28:        lock.lock()
29:        if head == NULL then
30:            lock.unlock()
31:            return NULL
32:        Node *n = head
33:        head = n.next
34:        numOps++
35:        lock.unlock()
36:        return n.val
37:    }
38:    public int getNumOps() {
39:        return numOps
40:    }
41: }
```
