Threads and Resource Contention

Class: CS315 – Distributed Scalable Computing
Whitworth University, Instructor: Scott Griffith
with heavy reference from Kent Jones

Last Modified: 9/19/2018

Part 1: Is vector Thread Safe? (30pts)

We have discussed shortly what is meant by **thread safe**. To elaborate, there is an assumption that every operation you put in a thread will work the way you want it to. This is not true! Take for example using cout to display information to the screen, when using threads you are not guaranteed that a full cout insertion will execute without another cout running over it. This would mean that cout is NOT thread safe. Every library, construct, operation, API etc. should be evaluated for thread safe-ness before throwing into a concurrent program.

In our discussion, Jude brought up the thread safeness of the **vector** construct. I was a little unsure on the spot, I didn't want to miss-inform from ignorance. I started formulating a piece of sample code that shows either way the thread safe-ness of vector.

I then realized this is a great skill for a concurrent programmer! You get to do this too!

I want you to put together a program that shows if **vector is thread safe or not.**

I would suggest referring to the reference for vector (http://www.cplusplus.com/reference/vector/vector/). It is worth pointing out that some parts of vector might be thread safe, and other might not be. I specifically want to know if the following methods are thread safe:

push_back()
pop_back()
operator[]
insert()
swap()

Be careful as you go about doing this. Make sure you know what you are testing when you are testing. Don't just try random things.

Question: Explain your findings! Walk me through your reasoning behind your code structure and why it proves one way or another the thread safe-ness of vector (10 pts)

Part 2: Mutual Exclusion V.1 (25pts)

There are lots of ways to play with mutual exclusion, but one of the canonical examples is with a bank account.

You are going to develop a program that utilizes a **BankAccount class** with two primary thread safe methods to perform banking operations. The first one will perform deposits, the second will perform withdrawals. The account value can never go below zero, so if a withdrawal would take more than the account has, it waits for a deposit to be made before the withdrawal is made.

Required elements of your BankAccount class:
Variables: int accountID; float balance;
Methods: void Withdrawal(float amount), void Deposit(float amount), Print()

Print will show the current value of the account, including accountID. Withdrawal will subtract an amount, blocking until there is enough in the account to withdrawal. Deposit will add value to the account, regardless of if there is a pending withdrawal.

Your class and implementation **must be thread safe.** I would suggest utilizing a mutex to protect the critical code section of both withdrawing and depositing.

You code should spawn a minimum of two threads, one that withdrawals funds from the account and one that deposits smaller amounts. Your deposit amount should be smaller than your withdrawal amount. These threads should be made up of for loops that perform at least 20 operations each.

Show me that your BankAccount is thread safe and that the withdrawal/deposit methods are accurate.

Hint: The thread initializer really does not like passing variables by reference. You could overcome this by using global variables (not as ideal) or lambda function (more ideal). Remember #include <mutex> mutex gu; gu.lock(); gu.unlock();

Hint: You may find it easier to force interactions by forcing delays in your loops to cause the threads to stall in between operations. I might suggest taking a look at: http://www.cplusplus.com/reference/thread/this_thread/sleep_for/

Hint: mutex-es cannot be copied. This means you may want to use pointers to your class, not assignments.

Question: What was your mutex strategy? What are your critical sections? (5pts)

Part 3: Mutual Exclusion V.2 (25pts)

Building off the code started above, you are going to make a new method for your BankAccount. You are going to implement a **PriorityWithdrawal (float)** method. This priority will take precedence over other withdrawals that may be pending, still waiting for the funds to be available.

Same as above, show that your new method is thread safe. For this part you should have a minimum of three threads that will each have a loop utilizing Withdrawal, Deposit and Priority Withdrawal.

Question: Are you sure your code will not dead-lock? Why? How many mutex structures are you using? (5pts)

Part 4: Condition Variables (20pts)

I know this section is going to feel like duplicating work, but stick with me for a bit.

To demonstrate Condition Variable use, you are going to create two different threaded functions. The first function is going to take an array of int and shuffle them. You are going to need to use a randomizer (I kind of like the one we used in LM1, I will keep that code in this module for reference, LM2_part4_random.cpp). The other function will take that array and sort it. These two functions must coordinate with each other using condition variables. The sorting algorithm will only start to process once the shuffler has completed, and vice versa. Complete 5 cycles of shuffling and sorting.

You are going to need to develop a function to check your sort, you can also use this function to check to see if it was randomized.

As the problem is stated, you should not need to use thread safe variables for the array part, as the condition variable will protect a race condition on the array. Your solution should be a general solution for arrays of n size. You can make this a user input value if you would like.

You can use whatever sorting algorithm you see fit, but you have to implement it yourself.

Bonus: Thread your sorting algorithm. We are going to do this later in class as an example of concurrency, but you can take a stab at it now. Your top level sorting thread can spawn other threads to aid in sorting the array quickly. For this, make sure to be using thread safe operations. (Extra 10pts)