Introduction to concurrent programming

Hesiquio Ballines

Properties of Concurrent Programming

Liveliness

Liveliness is the idea that work will eventually get done. In order to do this a thread must eventually make progress and finish the task at hand. In order for an implementation to follow this property the program must avoid any deadlocks that may happen between threads.

Safety

Safety in simple terms is the idea that nothing bad happens. To someone who has never done any concurrent programming this may seem easy to achieve but concurrent programming introduces a lot of possible outcomes and uncertainty. A general rule of thumb is to avoid dataraces by using locks. I will go over some examples later in this guide.

Conflicting Data Accesses

Conflicting data access is not necessarily a bad thing. There are three possible cases that can occur when there are multiple threads accessing the same data.

1. Read-Read
    a. These are fine as data will not change while multiple threads are looking at it at once
2. Write - Read
    a. These are dangerous as a write happens while important data is being read. I will go through a basic example with a common variable iteration function.
3. Write - Write
    a. Also dangerous potentially for a similar reason as a “Write-Read”

Performance

In concurrent programming it is expected that we will see a decrease in performance when multithreading is done incorrectly. Even in a simple thread safe implementation of a variable iteration function we see a huge performance decrease. In order to increase performance we have to think of the same problem in a different way and in the example we will be able to take a look on how to do that.

Concurrent Programming Showcase/Demo

Let's begin by taking a look at a simple function that iterates a single integer variable from 0 to 10,000,000. In the code below we can see the function iterateSingleVariableUnsafe() found in MultiShowcase.cpp file.

```
//Iterating a variable with no lock/mutex to make it thread-safe
void iterateSingleVariableUnsafe()
{

    while (1)
    {

        if (globalCounter < 10000 * 1000)
        {
            globalCounter++;
        }
        else
        {
            return;
        }
    }

}
```

One Thread Vs Two Threads

```
Enter an option: 1
Number of Threads: 1
Number of tests: 20
Number of iterations going over!: 0
Total Run Time(ms): 913

Enter an option: 1
Number of Threads: 2
Number of tests: 20
Number of iterations going over!: 2
Total Run Time(ms): 1650
```

By looking at these results we notice two things. The first is the number of iterations going over (the number of results that resulted in the globalCounter going over the desired limits). This violates the property of safety. We are getting an incorrect result only when more than one thread is run. This goes back to the idea of conflicting memory accesses. In this function the globalCounter is being written and read by multiple threads. This creates a datarace.

So how and why is this happening? Threads are unpredictable, context switches happen all the time and it's difficult to predict when they will happen, so let's look at a specific case. In the case where two threads exists (T1 and T2) and globalCounter is now 9,999,999, T1 makes

it to the **if(globalCounter < 10000 * 1000)** statement and passes. At this point the system switches to T2 before **globalCounter++** can execute on T1. **globalCounter** is still 9,999,999 so T2 will also pass the if statement. Now we have two threads that will execute **globalCounter++** when we only need to do it once therefore **globalCounter** will be incorrect once all the threads return from the function.

The second thing we notice is the performance. Running two threads on the same task actually decreases performance by a significant amount. This is because context switches take a certain amount of time to perform. So what is the point of running multiple threads if it is unsafe and decreases performance? Well, these two problems can be fixed but it is not as straightforward as you may think. We will first begin with an implementation that will give us the correct count every time. Let's start by looking at the code below.

```
//Iterating a single variable but making it thread safe
//Performance should be a lot worse in this case
void threadSafeIteration()
{


        while (1)
        {
            mtxLock.lock();
            if (globalCounter < 10000 * 1000)
            {
                globalCounter++;
                mtxLock.unlock();
            }
            else
            {
                mtxLock.unlock();
                return;
            }

        }
    }
```

This implementation is the same as the unsafe implementation but with added locks. Let's analyze what the locks are doing. After entering the while loop a thread may attempt to acquire a lock using the mtxLock variable defined above as **mutex mtxLock** using the **<mutex>** library. If a thread successfully acquires the lock then it may proceed with running the code until the lock is released in an unlock() statement. This allows only one thread to read and write to the global counter while the other threads wait for their turn therefore making this implementation a safe one.

This implementation also follows the lively property because we make sure that each unlock() statement will alway be called so other threads can keep doing progress. The reason we have an unlock in the if statement is because we want other threads to do work while another thread is looping back around. A more important unlock is the one in the else

statement. If this unlock were to be removed then once a thread is finished executing it would return without releasing the lock therefore we would create a deadlock and no threads would be able to advance. Now let's check the results of running this code.

```
Enter an option: 2
Number of Threads: 1
Number of tests: 20
Number of iterations going over!: 0
Total Run Time(ms): 69643

Enter an option: 2
Number of Threads: 2
Number of tests: 20
Number of iterations going over!: 0
Total Run Time(ms): 54124
```

Now we see that there are no errors so we have safety but the performance is incredibly hurt by adding locks to our implementation. At this point it is better to just run everything on one thread. But why does this happen if there are multiple threads working to tackle this task? The reason is because there is a ton of work wasted when threads are waiting for a lock. In addition just acquiring and releasing the lock can be an expensive task. We have to be smart about how we design our multithreaded programs. In this final part of the demo we will look at something we can do to improve our performance.

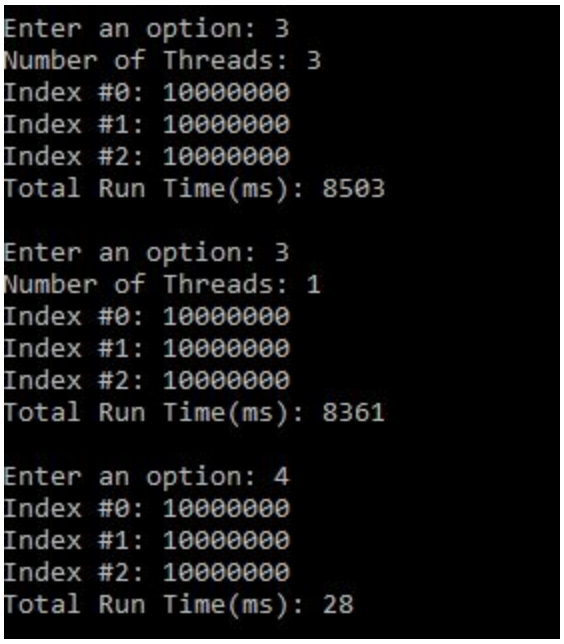
Something that we want to do is to try to reduce the number of conflicting accesses. Below there are two functions that are actually very similar.

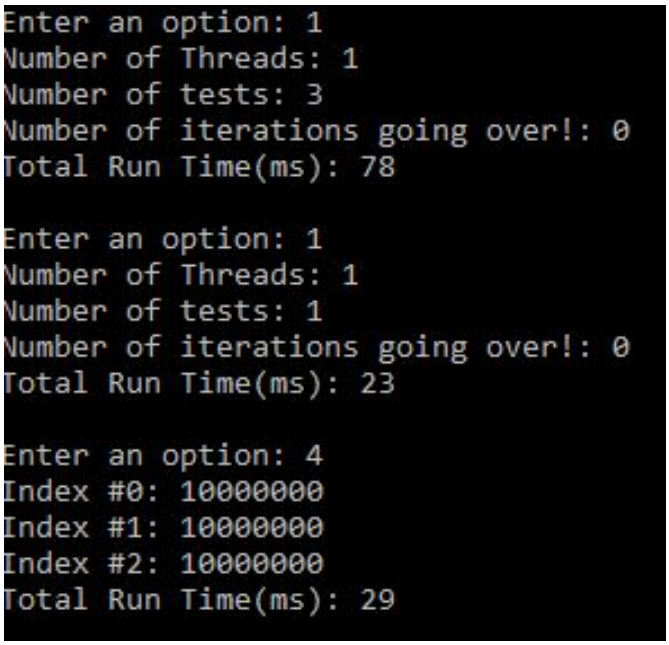
```
//Iterating an array using a thread safe implementation
//It iterates three different array sections using a set number of threads
void threadSafeArray(int index)
{

    while (1)
    {

        mtxLock.lock();
        if (commonArray[index] < 10000 * 1000)
        {
            commonArray[index]++;
            mtxLock.unlock();
        }
        else
        {
            mtxLock.unlock();
            return;
        }


    }
}
```

```
void optimizedThreadSafe(int index)
{
    nodeArray[index].nodeLock.lock();
    while (1)
    {


        if (nodeArray[index].value < 10000 * 1000)
        {
            nodeArray[index].value++;

        }
        else
        {
            nodeArray[index].nodeLock.unlock();
            return;
        }


    }
}
```

Both of these functions use an array that each contain 3 elements. The threadSafeArray function uses a simple integer array. The optimizedThreadSafe function uses an array of nodes where each node contains a mutex object and an integer value. By doing this we can lock

individual indexes in the array rather than locking the whole array. Here we sacrifice a little bit of memory but what we gain is a huge increase in performance by allowing different threads to work on different indexes at the same time. In addition, this remains safe because in the case that another thread is trying to read the same index it will be locked out of reading it as long as it attempts to acquire a lock beforehand. Below we can see the performance before and after the optimization. As you can see by removing conflicting accesses we can greatly improve performance even against running one thread with no locks.





Final Words

It is very difficult to develop proper multithreaded programs but this is a good starting point. Using the concepts we have learned we can further expand potential implementations. One includes using read/write locks that allows for locks to be acquired when a read is happening but only one thread can acquire it whenever there is a write. Thank you and good luck!