Multi-Threaded Design and Testing

BY EVAN OLDS
CPT S 422

Atomic Actions

- An atomic action is something that happens "all at once", and always <u>behaves</u> as though it cannot be divided into multiple steps
- Identifying atomic actions is key in writing and testing multithreaded code

Atomic Actions

- Nothing really guaranteed to be atomic by the C++ language specification
- In C# and Java assignment (of most types) is atomic
- Consider an integer value in memory shared by multiple threads. Assignment on this integer happens "all at once"; if both threads are assigning to the same integer concurrently, it won't copy 16 bits from the thread A assignment and 16 from the thread B assignment. It will get all 32-bits from one of the two threads.
- Also note what is meant by assignment here:
  - a register to memory move
  - a memory register move
- Copying one value to another in code is a memory copy, not an assignment:
  - int x = 6, y = 7;   x = y;

Non-Atomic Actions

- Things that aren’t atomic
- Think: read, modify, write. If the action needs to do this, then it’s (almost always) non-atomic.
- Example: addition.
  - x+=6
  - Need to read x, into a register (from memory), add 6 to it, and write it back.
  - Other actions (including assignments and write backs) can happen in between any of those 3 steps
- One caveat…

Atomic/Non-Atomic Actions Warning

- Warning: despite some actions *typically* being non-atomic, like the addition example from the previous slide, frameworks/SDKs and/or hardware features exist to do some of these actions atomically
- So take an increment action:
  - Without any details about the programming language and hardware, you should assume this is non-atmoic (read, increment, write back)
  - BUT, there exists atomic increment operations
  - There's an instruction on x86 architecture to do an atomic increment
  - Java has AtomicInteger class and other frameworks may have similar things
- In other words no action in programming is inherently atomic or non-atomic; atomicity can vary across language, frameworks, hardware architecture, etc.

Dealing With Multithreaded Code

- Given the definitions on previous slides, what types of problems can arise in multithreaded code?
- Does atomic imply thread-safe?
- What mechanisms exist to deal with such problems?

Dealing With Multithreaded Code

- Critical section (Wikipedia): “In concurrent programming, a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.”
- Locks, mutex, critical sections
  - Can provide thread safety
  - Can also ruin your parallel execution

Implementing Safe Concurrent Code

- First the software engineering side of it
  - Consider what algorithms / data structures may assist in implementation and see if you even need locks or critical sections
- Language side of it
  - C# has **lock** keyword: lock (o) { /* code block */ }
  - Makes sure that only one thread is in that block per object o

Scenario 1: Thread-Safe Producer / Consumer

- If you were asked to implement a thread-safe queue, how would you do it?
- Let’s make this one simple: only thread A will only ever enqueue, thread B will only ever dequeue
- One thread produces and the other consumes

Scenario 1: Thread-Safe Producer / Consumer

- If the consumer dequeues when the queue is empty, it just returns (doesn't have to block until the queue has an item)
- Producer must always successfully enqueue
- State of the internal queue must not be corrupted because of the two threads adding and removing concurrently
- NO LOCKS, MUTEXS, ETC. CAN BE USED!

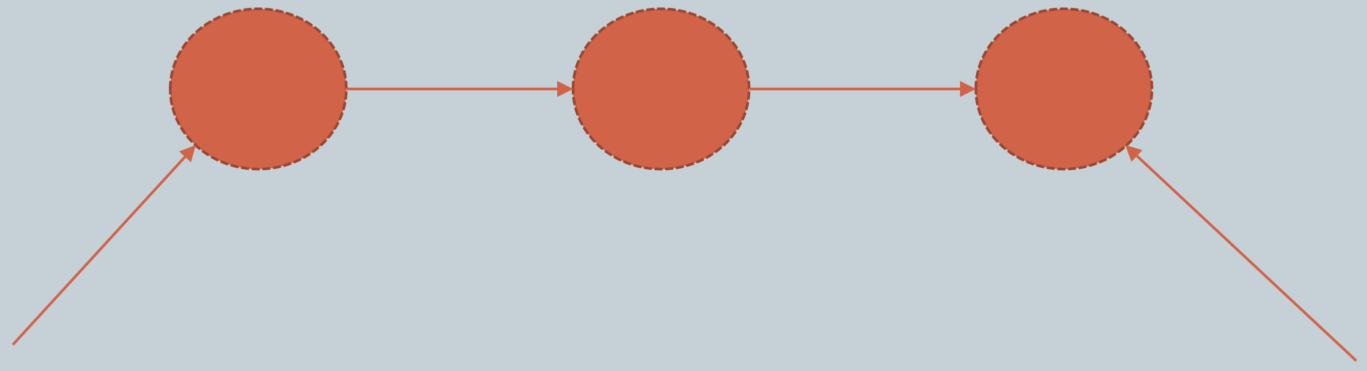
Scenario 1: Thread-Safe Producer / Consumer

In addition, the queue implementation:

- must not deadlock
- must not just keep all values in memory and never actually remove them
- must be O(1) for enqueue and dequeue (how?)
- must not have excessive busy-waiting (using 100% of CPU in a loop while waiting for another thread to complete some action).

Scenario 1: Thread-Safe Producer / Consumer

Linked-list implementation of queue



Dequeue from front

Enqueue in back

(see code demo and work out in class)

Scenario 2: Sleep Sort

- There's a "clever" (but highly impractical) sorting algorithm that you probably never saw in Cpt S 223
- Suppose the task is to get an array of bytes and DISPLAY them in sorted order (not actually sort them within the array)
- So the code:

```
byte[] nums = new byte[]{7, 1, 4, 2, 7, 3, 9};
SleepSort(nums);
```

- Would display on screen: 1 2 3 4 7 7 9

Scenario 2: Sleep Sort

- The logic in sleep sort is to loop through the input array and spawn a new thread/task for each number X. This task sleeps for X seconds, then displays X.

```
void SleepSort(byte[] nums)
{
 foreach (byte X in nums)
 {
  DoOnNewThread(delegate() { Thread.Sleep(X * 1000); Console.Write(X + “ ”) });
 }
}
```

- It would work because you always wait longer for bigger values, so they are implicitly displayed after smaller values
- It’s key that each sleep happens in parallel

Scenario 2: Sleep Sort

- If the “DoOnNewThread” function from the previous slide creates a new thread and then runs the task, will all the task run “roughly” in parallel?
- In other words, what does it cost to create a new thread?

Scenario 2: Sleep Sort

- We want all threads available at the exact moment we request them (ideally)
- Enter concept: Thread Pool
  - A collection of already created threads sitting idle and ready to do work
  - Logically we use it somewhat like this when we have some task to execute: ThreadPool.RunTask(task)
  - If the pool has a thread available immediately then it runs the task immediately on that thread
  - Otherwise it blocks until a thread becomes available, then runs the task on that thread

Scenario 2: Sleep Sort

- .NET Framework has a ThreadPool class
  - QueueUserWorkItem member function
  - “Queues a method for execution. The method executes when a thread pool thread becomes available”
- Potential problem: that ThreadPool class isn’t designed to make a thread available ASAP when QueueUserWorkItem is called
  - Why?

Scenario 2: Sleep Sort

- Potential problem: that ThreadPool class isn’t designed to make a thread available ASAP when QueueUserWorkItem is called
  - Why? Because it’s designed to keep an optimal number of threads running
  - Not too few, not too little
- If we want the guarantee that we can always have X threads immediately available, we’ll want our own thread class
  - Create X threads in constructor of our class
  - Have each one sleep in an idle state (how?)
  - When a task needs to be executed, wake up a thread, run the task, then put the thread back to sleep

Creating a Thread Pool

- Consider putting tasks in a BlockingCollection

```
// In constructor of some class that needs a thread pool:
var coll = new BlockingCollection<MyTask>();
for i = 0 to desired_num_threads
    Thread t = new Thread
    t.StartAt(ThreadWorkFunc, coll)

// ThreadWorkFunc logic:
void ThreadWorkFunc(BlockingCollection coll) {
  while (true) {
    coll.Take().Execute();
  }
}
```

Testing Multithreaded Code

- Homework assignment will cover a few threading concepts
  - The producer/consumer problem in the homework will help you understand how to tackle some concurrency problems without locks
  - The sleep sorter with thread pool is more about a threading-oriented design pattern and using .NET framework elements
- Testing, as usual, requires knowledge of various ways multithreaded data-structures can be implemented
- But testing multithreaded code can be quite difficult even with this understanding

Testing Multithreaded Code

Consider the following scenario

- An app processes two things in parallel with two threads: A and B
- Both thread A and thread B need to access a shared resource S
- Unit test is made
  - Does thousands of operations over a > 10 second span
  - Validates integrity of resource S at the end
- Suppose everything is fine with the unit test. Does that mean that the concurrent code for the app is safe?
  - How confident would you feel about the results of this unit test

Testing Multithreaded Code

Suppose everything is fine with the unit test. Does that mean that the concurrent code for the app is safe?

Not necessarily. You want tests that ensure actual concurrent access of the shared resource. Threads A and B using the resource "every so often" is not as good as almost constant access from both threads, which increases likelihood of concurrent access. Keep this in mind when writing unit tests for multithreaded code.