Volatile and Interlocked Class

- Sandino Vargas
- CS 6030: Introduction to Multicore Programming
- Spring 2011
General Idea

- You can use Volatile or Interlocked Class operations instead of Lock if your critical section (that needs to be lock) contains simple operations like:
  - Read
  - Write
  - Additions.
Volatile

- The volatile keyword indicates that a field can be modified in the program by something such as the operating system, the hardware, or a concurrently executing thread.

- The system always reads the current value of a volatile object at the point it is requested (even if the previous instruction asked for a value from the same object)

- The value of the object is written immediately on assignment.
Using volatile ensures that one thread retrieves the most up-to-date value written by another thread.
Example

using System;
using System.Threading;
class Test
{
    public static int result;
    public static volatile bool finished;
    static void Thread2()
    {
        result = 143;
        finished = true;
    }
    static void Main()
    {
        finished = false;
        // Run Thread2() in a new thread
        new Thread(new ThreadStart(Thread2)).Start();
        // Wait for Thread2 to signal that it has a result by setting
        // finished to true.
        for (;;)
        {
            if (finished)
            {
                Console.WriteLine("result = \{0\}", result);
                return;
            }
        }
    }
}
Interlocked Class

- Provides atomic operations for variables that are shared by multiple threads
  - Add(Int32, Int32)
  - CompareExchange(Int32, Int32, Int32)
  - Decrement(Int32)
  - Exchange(Double, Double)
  - Increment(Int32)
  - Read

- This help protect against errors:
  - That can occur when the scheduler switches contexts.
Interlocked Class

- This help protect against errors:
  - That can occur when two threads are executing concurrently on separate processors.
- The Increment and Decrement methods increment or decrement a variable and store the resulting value in a single operation.
- On most computer:
  - Load a value from an instance variable into a register.
  - Increment or decrement the value.
  - Store the value in the instance variable.
//A simple method that denies reentrancy.
static bool UseResource()
{
    //0 indicates that the method is not in use.
    if(0 == Interlocked.Exchange(ref usingResource, 1))
    {
        Console.WriteLine("{0} acquired the lock", Thread.CurrentThread.Name);
        //Code to access a resource that is not thread safe would go here.
        //Simulate some work
        Thread.Sleep(500);
        Console.WriteLine("{0} exiting lock", Thread.CurrentThread.Name);
        //Release the lock
        Interlocked.Exchange(ref usingResource, 0);
        return true;
    }
    else
    {
        Console.WriteLine("   {0} was denied the lock", Thread.CurrentThread.Name);
        return false;
    }
}
Volatile vs. Interlocked

- Volatile read/write are supposed to be immune to reordering. This only means reading and writing, it does not mean any other action;
- Volatility is not forced on the CPU, i.e., hardware level (x86 uses acquire and release fences on any read/write).
- Interlocked uses atomic assembly instructions for CompareExchange (cmpxchg), Increment (inc) etc;
Volatile vs. Interlocked

- Interlocked does use a lock sometimes: a hardware lock on multi processor systems.
- Interlocked is different from volatile in that it uses a full fence, where volatile uses a half fence.
- A read following a write can be reordered when you use volatile. It can't happen with Interlocked.
Technically: there are things you can do with volatile that you cannot do with Interlocked:

- **Syntax:** you cannot write `a = b` where `a` or `b` is volatile.
- You can read a different value after you write it to a volatile variable because of reordering. You cannot do this with Interlocked. In other words: you can be less safe with volatile than you can be with Interlocked.
- **Performance:** volatile is faster than Interlocked.
Volatile vs. Interlocked

- **Semantically:** Interlocked simply provides a superset of operations and is safer to use because it applies full fencing.

- **Scope:** Declaring a variable volatile makes it volatile for every single access. It is impossible to force this behavior any other way, hence volatile cannot be replaced with Interlocked. This is needed in scenarios where other libraries, interfaces or hardware can access your variable and update it anytime, or need the most recent version.
When to use volatile?

- A good example is say you have 2 threads (A & B), one which always writes to a variable, and one which always reads from that same variable.

- A solution would be to lock, but you could also in this situation use volatile, which would ensure that thread B will always see the most up-to-date thing that thread A has written, which would be faster and result in cleaner code.
When to use Interlocked?

```csharp
lock(this.locker) this.counter++; // It prevents any other threads from executing any other code which is guarded by locker. Using locks also, prevents the multi-cpu reordering.

Interlocked.Increment(ref this.counter); // This is safe, as it effectively does the read, increment, and write in 'one hit' which can't be interrupted. Because of this it won't affect any other code and you don't need to remember to lock elsewhere either.
```
References

- http://msdn.microsoft.com
- http://stackoverflow.com/questions

END

QUESTIONS?