Volatile vs. Interlocked vs. lock

Asked 16 years, 2 months ago Modified 2 years, 11 months ago

Viewed 174k times

785

Let's say that a class has a `public int counter` field that is accessed by multiple threads. This `int` is only incremented or decremented.

To increment this field, which approach should be used, and why?

- `lock(this.locker) this.counter++;`,
- `Interlocked.Increment(ref this.counter);`,
- Change the access modifier of `counter` to `public volatile`.

Now that I've discovered `volatile`, I've been removing many `lock` statements and the use of `Interlocked`. But is there a reason not to do this?

c# multithreading locking volatile interlocked

Share

Improve this question

Follow

edited Aug 22, 2014 at 14:31

Sam

7,398 ● 16 ● 47 ● 68

asked Sep 30, 2008 at 19:25

core
33k ● 45 ● 140 ● 195

Read the Threading in C# reference. It covers the ins and outs of your question. Each of the three have different purposes and side effects. – spoulson Sep 30, 2008 at 19:33

1 simple-talk.com/blogs/2012/01/24/… you can see the use of volitable in arrays , i don't completly understand it , but it's anouther reference to what this does. – eran otzap Sep 28, 2013 at 16:18

68 This is like saying "I've discovered that the sprinkler system is never activated, so I'm going to remove it and replace it with smoke alarms". The reason not to do this is *because it is incredibly dangerous* and *gives you almost no benefit*. If you have time to spend changing the code then **find a way to make it less multithreaded**! Don't find a way to make the multithreaded code more dangerous and easily broken! – Eric Lippert Feb 18, 2014 at 17:25

1 My house has both sprinklers *and* smoke alarms. When incrementing a counter on one thread and reading it on another it seems like you need both a lock (or an Interlocked) *and* the volatile keyword. Truth? – yoyo Nov 5, 2014 at 0:17

2 @yoyo No, you don't need both. – David Schwartz Oct 14, 2016 at 21:17

10 Answers

Sorted by: Highest score (default)

Worst (won't actually work)

1001

> Change the access modifier of `counter` to `public volatile`

As other people have mentioned, this on its own isn't actually safe at all. The point of `volatile` is that multiple threads running on multiple CPUs can and will cache data and re-order instructions.

If it is **not** `volatile`, and CPU A increments a value, then CPU B may not actually see that incremented value until some time later, which may cause problems.

If it is `volatile`, this just ensures the two CPUs see the same data at the same time. It doesn't stop them at all from interleaving their reads and write operations which is the problem you are trying to avoid.

Second Best:

> `lock(this.locker) this.counter++`;

This is safe to do (provided you remember to `lock` everywhere else that you access `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 problems as above, which is great.

The problem is, locking is slow, and if you re-use the `locker` in some other place which is not really related then you can end up blocking your other threads for no reason.

Best

> ```
> 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. It's also very fast (as MSDN says, on modern CPUs, this is often literally a single CPU instruction).

~~I'm not entirely sure however if it gets around other CPUs reordering things, or if you also need to combine volatile with the increment.~~

InterlockedNotes:

1. INTERLOCKED METHODS ARE CONCURRENTLY SAFE ON ANY NUMBER OF COREs OR CPUs.
2. Interlocked methods apply a full fence around instructions they execute, so reordering does not happen.
3. Interlocked methods **do not need or even do not support access to a volatile field**, as volatile is placed a half fence around operations on given field and interlocked is using the full fence.

Footnote: What volatile is actually good for.

As `volatile` doesn't prevent these kinds of multithreading issues, what's it for? A good example is saying you have two threads, one which always writes to a variable (say `queueLength`), and one which always reads from that same variable.

If `queueLength` is not volatile, thread A may write five times, but thread B may see those writes as being delayed (or even potentially in the wrong order).

A solution would be to lock, but you could also use volatile in this situation. This would ensure that thread B will always see the most up-to-date thing that thread A has written. Note however that this logic *only* works if you have writers who never read, and readers who never write, *and* if the thing you're writing is an atomic value. As soon as you do a single read-modify-write, you need to go to Interlocked operations or use a Lock.

Share Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot
1 ● 1

answered Sep 30, 2008 at 20:13

Orion Edwards
123k ● 66 ● 245 ● 339

33 "I'm not entirely sure ... if you also need to combine volatile with the increment." They cannot be combined AFAIK, as we

can't pass a volatile by ref. Great answer by the way.
– Hosam Aly Jan 17, 2009 at 13:07

54 Thanx much! Your footnote on "What volatile is actually good for" is what I was looking for and confirmed how I want to use volatile. – Jacques Bosch May 10, 2010 at 6:22

7 In other words, if a var is declared as volatile, the compiler will assume that the var's value will not remain the same (i.e. volatile) each time your code comes across it. So in a loop such as: while (m_Var) { }, and m_Var is set to false in another thread, the compiler won't simply check what's already in a register that was previously loaded with m_Var's value but reads the value out from m_Var again. However, it doesn't mean that not declaring volatile will cause the loop to go on infinitely - specifying volatile only guarantees that it won't if m_Var is set to false in another thread. – Zach Saw Jun 23, 2011 at 7:41

38 @Zach Saw: Under the memory model for C++, volatile is how you've described it (basically useful for device-mapped memory and not a lot else). Under the memory model for the *CLR* (this question is tagged C#) is that volatile will insert memory barriers around reads and writes to that storage location. Memory barriers (and special locked variations of some assembly instructions) are you you tell the *processor* not to reorder things, and they're fairly important...
– Orion Edwards Jul 8, 2011 at 3:39

22 @ZachSaw: A volatile field in C# prevents the C# compiler and jit compiler from making certain optimizations that would cache the value. It also makes certain guarantees about what order reads and writes may be observed to be in on multiple threads. As an implementation detail it may do so by introducing memory barriers on reads and writes. The precise semantics guaranteed are described in the specification; note that the specification does *not* guarantee that a *consistent* ordering of *all* volatile writes and reads will

be observed by *all* threads. – Eric Lippert Dec 3, 2013 at 17:22

167

EDIT: As noted in comments, these days I'm happy to use `Interlocked` for the cases of a *single variable* where it's *obviously* okay. When it gets more complicated, I'll still revert to locking...

Using `volatile` won't help when you need to increment - because the read and the write are separate instructions. Another thread could change the value after you've read but before you write back.

Personally I almost always just lock - it's easier to get right in a way which is *obviously* right than either volatility or Interlocked.Increment. As far as I'm concerned, lock-free multi-threading is for real threading experts, of which I'm not one. If Joe Duffy and his team build nice libraries which will parallelise things without as much locking as something I'd build, that's fabulous, and I'll use it in a heartbeat - but when I'm doing the threading myself, I try to keep it simple.

Share Improve this answer Follow

edited Mar 20, 2014 at 7:38
Yair Nevet
13k ● 17 ● 69 ● 109

answered Sep 30, 2008 at 19:29
Jon Skeet
1.5m ● 889 ● 9.3k ● 9.3k

21 +1 for ensuring me to forget about lock-free coding from now. – Xaqron Jan 3, 2011 at 1:51

7 lock-free codes are definitely not truly lock-free as they lock at some stage - whether at (FSB) bus or interCPU level, there's still a penalty you'd have to pay. However locking at these lower levels are generally faster so long as you don't saturate the bandwidth of where the lock occurs. – Zach Saw Jul 7, 2011 at 1:25

5 There is nothing wrong with Interlocked, it is exactly what your looking for and faster than a full lock() – Jaap Mar 22, 2012 at 20:24

5 @Jaap: Yes, these days I *would* use interlocked for a genuine single counter. I just wouldn't want to start messing around trying to work out interactions between *multiple* lock-free updates to variables. – Jon Skeet Mar 22, 2012 at 20:30

7 @ZachSaw: Your second comment says that interlocked operations "lock" at some stage; the term "lock" generally implies that one task can maintain exclusive control of a resource for an unbounded length of time; the primary advantage of lock-free programming is that it avoids the danger of resource becoming unusable as a result of the owning task getting waylaid. The bus synchronization used by the interlocked class isn't just "generally faster"--on most systems it has a bounded worst-case time, whereas locks do not. – supercat Aug 21, 2012 at 15:01

48

"`volatile`" does not replace `Interlocked.Increment`! It just makes sure that the variable is not cached, but used directly.

Incrementing a variable requires actually three operations:

1. read
2. increment
3. write

`Interlocked.Increment` performs all three parts as a single atomic operation.

Share Improve this answer Follow

edited Sep 7, 2009 at 16:11

answered Sep 30, 2008 at 19:32

Michael Damatov

15.6k ● 10 ● 48 ● 71

---

5 Said another way, Interlocked changes are full-fenced and as such are atomic. Volatile members are only partially-fenced and as such are not guarenteed to be thread-safe.
– JoeGeeky Dec 4, 2011 at 19:58

---

2 Actually, `volatile` does *not* make sure the variable is not cached. It just puts restrictions on how it can be cached. For example, it can still be cached in things the CPU's L2 cache because they're made coherent in hardware. It can still be prefected. Writes can still be posted to cache, and so on.

(Which I think was what Zach was getting at.)
– David Schwartz Dec 4, 2015 at 12:36

45

Either lock or interlocked increment is what you are looking for.

Volatile is definitely not what you're after - it simply tells the compiler to treat the variable as always changing even if the current code path allows the compiler to optimize a read from memory otherwise.

e.g.

```
while (m_Var)
{ }
```

if m_Var is set to false in another thread but it's not declared as volatile, the compiler is free to make it an infinite loop (but doesn't mean it always will) by making it check against a CPU register (e.g. EAX because that was what m_Var was fetched into from the very beginning) instead of issuing another read to the memory location of m_Var (this may be cached - we don't know and don't care and that's the point of cache coherency of x86/x64). All the posts earlier by others who mentioned instruction reordering simply show they don't understand x86/x64 architectures. Volatile does *not* issue read/write barriers as implied by the earlier posts saying 'it prevents reordering'. In fact, thanks again to MESI protocol, we are guaranteed the result we read is always the same across

CPUs regardless of whether the actual results have been retired to physical memory or simply reside in the local CPU's cache. I won't go too far into the details of this but rest assured that if this goes wrong, Intel/AMD would likely issue a processor recall! This also means that we do not have to care about out of order execution etc. Results are always guaranteed to retire in order - otherwise we are stuffed!

With Interlocked Increment, the processor needs to go out, fetch the value from the address given, then increment and write it back -- all that while having exclusive ownership of the entire cache line (lock xadd) to make sure no other processors can modify its value.

With volatile, you'll still end up with just 1 instruction (assuming the JIT is efficient as it should) - inc dword ptr [m_Var]. However, the processor (cpuA) doesn't ask for exclusive ownership of the cache line while doing all it did with the interlocked version. As you can imagine, this means other processors could write an updated value back to m_Var after it's been read by cpuA. So instead of now having incremented the value twice, you end up with just once.

Hope this clears up the issue.

For more info, see 'Understand the Impact of Low-Lock Techniques in Multithreaded Apps' - http://msdn.microsoft.com/en-au/magazine/cc163715.aspx p.s. What prompted this very late reply? All the replies were so blatantly incorrect (especially the one marked as answer) in their explanation I just had to clear it up for anyone else reading this. *shrugs*

p.p.s. I'm assuming that the target is x86/x64 and not IA64 (it has a different memory model). Note that Microsoft's ECMA specs is screwed up in that it specifies the weakest memory model instead of the strongest one (it's always better to specify against the strongest memory model so it is consistent across platforms - otherwise code that would run 24-7 on x86/x64 may not run at all on IA64 although Intel has implemented similarly strong memory model for IA64) - Microsoft admitted this themselves - http://blogs.msdn.com/b/cbrumme/archive/2003/05/17/51445.aspx.

Share Improve this answer Follow

edited Jul 7, 2011 at 11:50

answered Jun 23, 2011 at 15:08

Zach Saw
4,378 ● 4 ● 35 ● 49

---

4 Interesting. Can you reference this? I'd happily vote this up, but posting with some aggressive language 3 years after a highly voted answer that is consistent with the resources I've read is going to require a bit more tangible proof.
– Steven Evers Jul 7, 2011 at 3:28

2 Why anyone would want to prevent the CPU from caching is beyond me. The whole real estate (definitely not negligible in size and cost) dedicated to perform cache coherency is completely wasted if that's the case... Unless you require no cache coherency, such as a graphics card, PCI device etc, you wouldn't set a cache line to write-through. – Zach Saw Jul 7, 2011 at 4:29

4 Yes, everything you say is if not 100% at least 99% on the mark. This site is (mostly) pretty useful when you are in the rush of development at work but unfortunately the accuracy of the answers corresponding to the (game of) votes is not there. So basically in stackoverflow you can get a feeling of what is the popular understanding of the readers not what it really is. Sometimes the top answers are just pure gibberish - myths of kind. And unfortunately this is what breeds into the folks who come across the read while solving the problem. It's understandable though, nobody can know everything. – user1416420 Dec 14, 2012 at 8:03

1 The problem with this answer, and your comments everywhere on this question, is that it's exclusive to x86, when the question was not. Knowing about the underlying hardware memory model is useful at times, but doesn't replace knowledge of the CLR memory model. For example, just because a memory barrier is implicit on x86 does not mean that the CLR memory model doesn't require memory barriers for `volatile` (more than C++ `volatile`). .NET code runs on a half a dozen architectures, and C++ far more than that. – Ben Voigt Feb 10, 2013 at 16:19

1 @BenVoigt I could go on and answer about all the architectures .NET runs on, but that would take a few pages and is definitely not suitable for SO. It is far better to educate people based on the most widely used .NET underlying hardware mem-model than one that is arbitrary. And with my comments 'everywhere', I was correcting the mistakes people were making in assuming flushing / invalidating the cache

etc. They made assumptions about the underlying hardware without specifying which hardware. – Zach Saw Feb 10, 2013 at 22:39

18

Interlocked functions do not lock. They are atomic, meaning that they can complete without the possibility of a context switch during increment. So there is no chance of deadlock or wait.

I would say that you should always prefer it to a lock and increment.

Volatile is useful if you need writes in one thread to be read in another, and if you want the optimizer to not reorder operations on a variable (because things are happening in another thread that the optimizer doesn't know about). It's an orthogonal choice to how you increment.

This is a really good article if you want to read more about lock-free code, and the right way to approach writing it

http://www.ddj.com/hpc-high-performance-computing/210604448

Share Improve this answer Follow

edited Sep 30, 2008 at 19:33

answered Sep 30, 2008 at 19:27

Lou Franco
89k ● 14 ● 136 ● 198

13

lock(...) works, but may block a thread, and could cause deadlock if other code is using the same locks in an incompatible way.

Interlocked.* is the correct way to do it ... much less overhead as modern CPUs support this as a primitive.

volatile on its own is not correct. A thread attempting to retrieve and then write back a modified value could still conflict with another thread doing the same.

Share Improve this answer Follow

answered Sep 30, 2008 at 19:32

Rob Walker
47.4k ● 15 ● 100 ● 137

10

I did some test to see how the theory actually works: kennethxu.blogspot.com/2009/05/interlocked-vs-monitor-performance.html. My test was more focused on CompareExchnage but the result for Increment is similar. Interlocked is not necessary faster in multi-cpu environment. Here is the test result for Increment on a 2 years old 16 CPU server. Bare in mind that the test also involves the safe read after increase, which is typical in real world.

```
D:\>InterlockVsMonitor.exe 16
Using 16 threads:
          InterlockAtomic.RunIncrement         (ns):
```

```
Minimal,   8409 Maxmial
    MonitorVolatileAtomic.RunIncrement         (ns):
Minimal,   7243 Maxmial

D:\>InterlockVsMonitor.exe 4
Using 4 threads:
          InterlockAtomic.RunIncrement         (ns):
Minimal,   4321 Maxmial
    MonitorVolatileAtomic.RunIncrement         (ns):
Minimal,   1018 Maxmial
```

Share Improve this answer Follow

edited Aug 21, 2012 at 12:11

answered May 24, 2009 at 23:12

Kenneth Xu
1,514 ● 15 ● 15

The code sample you tested was soooo trivial though - it really doesn't make much sense testing it that way! The best would be to understand what the different methods are actually doing and use the appropriate one based on the usage scenario you have. – Zach Saw Jun 23, 2011 at 15:10

@Zach, the how discussion here was about the scenario of increasing a counter in a thread safe manner. What other usage scenario was in your mind or how would you test it? Thanks for the comment BTW. – Kenneth Xu Jun 27, 2011 at 21:05

Point is, it's an artificial test. You're not going to hammer the same location that often in any real world scenario. If you are, then well you're bottlenecked by the FSB (as shown in your server boxes). Anyway, look at my reply on your blog. – Zach Saw Jul 7, 2011 at 1:18

3 Looking it back again. If the true bottleneck is with FSB, the the monitor implementation should observe the same bottleneck. The real difference is that Interlocked is doing busy wait and retry which becomes a real issue with high performance counting. At least I hope my comment raise the attention that Interlocked is not always the right choice for counting. The fact the people are looking at alternatives well explained it. You need a long adder gee.cs.oswego.edu/dl/jsr166/dist/jsr166edocs/jsr166e/… – Kenneth Xu Oct 6, 2013 at 16:01

9

I second Jon Skeet's answer and want to add the following links for everyone who want to know more about "volatile" and Interlocked:

Atomicity, volatility and immutability are different, part one - (Eric Lippert's Fabulous Adventures In Coding)

Atomicity, volatility and immutability are different, part two

Atomicity, volatility and immutability are different, part three

Sayonara Volatile - (Wayback Machine snapshot of Joe Duffy's Weblog as it appeared in 2012)

Share Improve this answer

Follow

edited Jul 27, 2016 at 17:54

Brian Rogers
130k ● 31 ● 310 ● 310

answered Apr 29, 2012 at 22:04

zihotki
5,191 ● 23 ● 26

4

I would like to add to mentioned in the other answers the difference between `volatile`, `Interlocked`, and `lock`:

[The volatile keyword can be applied to fields of these types]:

- Reference types.
- Pointer types (in an unsafe context). Note that although the pointer itself can be volatile, the object that it points to cannot. In other words, you cannot declare a "pointer" to be "volatile".
- Simple types such as `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, and `bool`.
- An enum type with one of the following base types: `byte`, `sbyte`, `short`, ushort, `int`, or `uint`.
- Generic type parameters known to be reference types.
- `IntPtr` and `UIntPtr`.

Other types, including `double` and `long`, cannot be marked "volatile" because reads and writes to fields of those types cannot be guaranteed to be atomic. To protect multi-threaded access to those types of fields, use the `Interlocked` class members or protect access using the `lock` statement.

Share Improve this answer

Follow

edited Mar 22, 2019 at 21:15

T.S.

19.3k ● 11 ● 67 ● 93

answered Dec 2, 2018 at 15:51

V. S.

1,244 ● 15 ● 18

2

I'm just here to point out the mistake about volatile in Orion Edwards' answer.

He said:

> "If it is volatile, this just ensures the two CPUs see the same data at the same time."

It's wrong. In microsoft' doc about volatile, mentioned:

> "On a multiprocessor system, a volatile read operation does not guarantee to obtain the latest value written to that memory location by any processor. Similarly, a volatile write operation does not guarantee that the value written would be immediately visible to other processors."

Share Improve this answer

Follow

answered Jan 21, 2022 at 7:39

j HeSansi

21 ● 1

This does not provide an answer to the question. Once you have sufficient reputation you will be able to comment on any post; instead, provide answers that don't require clarification from the asker. - From Review – sjakobi Jan 28, 2022 at 1:02