

Re: When is "volatile" used instead of "lock" ?

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2007-06/msg02497.html>

- *From:* "Willy Denoyette [MVP]" <willy.denoyette@xxxxxxxxxx>
 - *Date:* Thu, 14 Jun 2007 23:30:04 +0200
-

"Jon Skeet [C# MVP]" <skeet@xxxxxxxx> wrote in message
<news:MPG.20dbba8120b2e63e220@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>

Peter Ritchie [C# MVP] <PRSoCo@xxxxxxxxxxxxxxxxxxxx> wrote:

- > Acquiring a lock has acquire semantics, and releasing a lock has
- > release semantics. You don't need any volatility if all access to any
- > particular item of shared data is always made having acquired a certain
- > lock.

...which only applies to reference types. Most of this discussion has been revolving around value types (by virtue of Interlocked.Increment), for which "lock" cannot not apply. e.g. you can't switch from using lock on a member to using Interlocked.Increment on that member, one works with references and the other with value types (specifically Int32 and Int64). This is what raised my concern.

It's not a case of using a lock on a particular value – taking the lock out creates a memory barrier beyond which *no* reads can pass, not just reads on the locked expression.

- > It certainly *is* documented. ECMA 335, section 12.6.5:
- >
- > <quote>
- > Acquiring a lock (System.Threading.Monitor.Enter or entering a
- > synchronized method) shall implicitly
- > perform a volatile read operation, and releasing a lock
- > (System.Threading.Monitor.Exit or leaving a
- > synchronized method) shall implicitly perform a volatile write
- > operation.
- > </quote>

...still doesn't document anything about the members/variables within the locked block (please read my example). That quote applies only to the reference used as the parameter for the lock.

Re: When is "volatile" used instead of "lock" ?

There can be no lock acquire semantics for value members. Suggesting "locking appropriately" cannot apply here and can be misconstrued by some people by creating something like "lock(myLocker){intMember = SomeMethod();}" which does not do the same thing as making intMember volatile, increases overhead needlessly, and still leaves a potential bug.

No, it *doesn't* leave a bug – you've misunderstood the effect of lock having acquire semantics.

>> volatile and lock should be used in conjunction, one is not a >> replacement
>> for the other.
>
> If you lock appropriately, you never need to use volatile.

Even if the discussion hasn't been about value types, a dangerous statement; because it could only apply to reference types (i.e. if myObject is wrapped with lock(myObject) in every thread, yes I don't need to declare it with volatile—but that's probably not why I'm using lock). In the context of reference types, volatile only applies to the pointer (reference) not anything within the object it references. Reference assignment is atomic, there's no need to use lock to guard that sort of thing. You use lock to guard a non-atomic invariant, volatile has nothing to do with that—it has to do with the optimization (ordering, caching) of pointer/value reads and writes.

Atomicity and volatility are very different things, and shouldn't be confused.

Locks do more than just guarding non-atomic invariants though – they have the acquire/release semantics which make volatility unnecessary.

To be absolutely clear on this, if I have:

```
int someValue;  
object myLock;  
  
...  
  
lock (myLock)  
{  
int x = someValue;  
someValue = x+1;  
}
```

then the read of someValue *cannot* be from a cache – it *must* occur

Re: When is "volatile" used instead of "lock" ?

after the lock has been taken out. Likewise before the lock is released, the write back to someValue **must** have been made effectively flushed (it can't occur later than the release in the logical memory model).

Actually on modern processors (others aren't supported anyway, unless you are running W98 on a 80386) , the read and writes will come/go from/to the cache (L1, L2 ..), the cache coherency protocol will guarantee consistency across the cache lines holding the variable has changed. That way, the "software" has a uniform view of what is called the "memory" irrespective the number of HW threads (not talking about NUMA here!).

Here's how that's guaranteed by the spec:

"Acquiring a lock (System.Threading.Monitor.Enter or entering a synchronized method) shall implicitly perform a volatile read operation"

and

"A volatile read has acquire semantics meaning that the read is guaranteed to occur prior to any references to memory that occur after the read instruction in the CIL instruction sequence."

That means that the volatile read due to the lock is guaranteed to occur prior to the "reference to memory" (reading someValue) which occurs later in the CIL instruction sequence.

The same thing happens the other way round for releasing the lock.

Calling Monitor.Enter/Monitor.Exit is a pretty heavy-weight means of ensuring acquire semantics; at least 5 times slower if volatile is all you need.

But still fast enough for almost everything I've ever needed to do, and I find it a lot easier to reason about a single way of doing things than having multiple ways for multiple situations. Just a personal preference – but it definitely **is** safe, without ever needing to declare anything volatile.

Probably one of the reasons why I've never seen a volatile modifier on a field in the FCL. And to repeat myself, volatile is not a guarantee against re-ordering and write buffering by CPU's implementing a weak memory model, like the IA64. Volatile serves only one thing, that is, prevent optimizations like re-registering and re-ordering as there would be done by the JIT compiler.

Re: When is "volatile" used instead of "lock" ?

Re: When is "volatile" used instead of "lock" ?

Willy.

.