Threadsafe Events

14 years ago (Jun 19, 2009) • 6 Comments

Disclaimer: *This blog entry only deals with the common case of instance events; static events are ignored. Furthermore, the contents of this blog entry are 100% my own opinion. However, it is the opinion of someone who has specialized in multithreading for 13 years.*

When writing components in a multithreaded world, one question that commonly crops up is, "how do I make my events threadsafe?" The asker is usually concerned with threadsafe subscription and unsubscription, but threadsafe raising must also be taken into consideration.

The Wrong Solution #1, from the C# Language Specification

The C# language authors attempted to make event subscription and unsubscription threadsafe by default. To do so, they allow (but do not require) locking on `this`, which is generally considered bad. This code:

```
public event MyEventHandler MyEvent;
```

Logically becomes this code:

```
private MyEventHandler __MyEvent;
public event MyEventHandler MyEvent
{
    add
    {
        lock (this)
        {
            this.__MyEvent += value;
        }
    }

    remove
    {
        lock (this)
        {
            this.__MyEvent -= value;
        }
    }
}
```

Chris Burrows, a Microsoft developer on the C# compiler team, explains why this is bad in his blog post Field-like Events Considered Harmful (https://docs.microsoft.com/en-us/archive/blogs/cburrows/field-like-events-considered-harmful?WT.mc_id=DT-MVP-5000058). His blog post covers the reasoning thoroughly, so it won't be repeated here.

👉 **Minor rant:** The Java language fell into the same trap; see **Practical API Design's Java Monitor page (http://wiki.apidesign.org/wiki/Java_Monitor)**. Why is it that some language designers believe they can declaratively solve multithreading problems? If the solution was really as simple as that, then why haven't other people already figured it out? Multithreaded programming has consumed some of the brightest minds for decades, and it's *hard*. Language designers can't make multithreading complexities go away by sprinkling some magical fairy powder, even if they name the powder `MethodImplOptions.Synchronized`. In fact, most of the time they're just making it worse.

Pretend for a minute that locking on `this` is OK. It actually *would* work, after all; it just raises the likelihood of unexpected deadlocks. It's also possible that a future C# compiler may lock on a super-secret private field instead of `this`. However, even if the *implementation* is OK, the *design* is still flawed. The problem becomes clear when one ponders how to raise the event in a threadsafe manner.

This is the standard, simple, and logical event-raising code:

```
if (this.MyEvent != null)
{
    this.MyEvent(this, args);
}
```

If there are multiple threads subscribing to and unsubscribing from an event, then the built-in field-like event locking works *only* for subscribing and unsubscribing. The event raising code exposes a problem: if another thread unsubscribes from the event after the `if` statement but before the event is raised, then this code may result in a NullReferenceException!

So, it turns out that "thread-safe" events weren't really thread-safe. Moving on…

The Wrong Solution #2, from the Framework Design Guidelines and MSDN

One solution to the problem described above is to make a copy of the event delegate before testing it. The event raising code becomes:

```
MyEventHandler myEvent = this.MyEvent;
if (myEvent != null)
{
    myEvent(this, args);
}
```

This is the solution used by [MSDN examples]https://docs.microsoft.com/en-us/dotnet/csharp /programming-guide/events/how-to-publish-events-that-conform-to-net-framework-guidelines?WT.mc_id=DT-MVP-5000058) and recommended by the semi-standard Framework Design Guidelines (https://www.amazon.com/gp/product/0321545613?ie=UTF8&tag=stepheclearys-20& linkCode=as2&camp=1789&creative=390957&creativeASIN=0321545613) (my 2nd edition has it on page 157).

This solution is simple, obvious, and wrong. [By the way, I'm not dissing Framework Design Guidelines. They have lots of good advice, and I don't mean to be critical of the book in general. They're just mistaken in this particular recommendataion.]

Programmers without a strong background in multithreaded programming may not immediately detect why this solution is wrong. Delegates are immutable reference types, so the local variable copy is atomic; no problem there. The problem exists in the memory model: it is possible that an out-of-date value for the delegate field is held in one processor's cache. Without going into a painful level of detail, in order to ensure that one is reading the current value of a non-volatile field, one must either issue a memory barrier or wrap the copy operation within a lock (and it must be the same lock acquired by the event add/remove methods).

In short, this solution does prevent the NullReferenceException race condition; but it introduces *another* race condition in its place (raising an unsubscribed event handler).

The Wrong Solution #3, from Jon Skeet

[OK, let me say this first: Jon Skeet is an awesome programmer. I highly recommend his book C# in Depth (https://www.amazon.com/C-Depth-Jon-Skeet-dp-1617294535/dp/1617294535?&linkCode=ll1& tag=stepheclearys-20&linkId=487eed7f1fec8cc1ea832efc1c165998&language=en_US&ref_=as_li_ss_tl) to *anyone and everyone using C#* (I own the first edition and will buy the 2nd as soon as it comes out; he's writing it now and I'm so excited!). I follow his blog. I highly respect the man, and I can't believe my first mention of him on my blog is in a negative light… However, he did come up with a wrong solution for thread-safe events. To give him credit, though, he ended his paper recommending the right solution!]

Jon Skeet has a great treatment of this subject in his paper Delegates and Events (https://csharpindepth.com/Articles/Events) (you may wish to skip to the section titled "Thread-safe events"). He covers everything that I've described above, but then proceeds on to propose another wrong solution. He dislikes the memory barrier solution (as do I), and attempts to solve it by wrapping the copy operation within the lock. As Jon points out, the event add/remove methods *may* lock `this` or they could lock something else (remember, a future C# compiler may choose to lock on a super-secret private field instead). So, the default add/remove methods have to be replaced with ones that perform an explicit lock, as such:

```
private object myEventLock = new object();
private MyEventHandler myEvent;
public MyEventHandler MyEvent
{
    add
    {
        lock (this.myEventLock)
        {
            this.myEvent += value;
        }
    }

    remove
    {
        lock (this.myEventLock)
        {
            this.myEvent -= value;
        }
    }
}

protected virtual OnMyEvent(MyEventArgs args)
{
    MyEventHandler localMyEvent;
    lock (this.myEventLock)
    {
        localMyEvent = this.myEvent;
    }

    if (localMyEvent != null)
    {
        localMyEvent(this, args);
    }
}
```

That's a fair amount of code for a single event! Some people have even written helper objects to reduce the amount of code. Before jumping on that bandwagon, though, remember that this solution is also wrong.

There is *still* a race condition.

Specifically, it is possible that the value of `myEvent` is modified after it has been read into `localMyEvent` but before it is raised. This can result in an unsubscribed handler being invoked, which could be problematic. So, this solution does solve the last solution's problem (with the memory model and processor cache), but it turns out there was an underlying race condition anyway (this same problem does affect the other two solutions above, too).

The Wrong Solution #4, from Nobody (but just in case you were thinking about it!)

A natural response is to extend the `lock` statement in Jon's code to include raising the event. That does prevent the race condition problem from all the solutions described above, but it introduces a more serious problem.

If this solution is used, then an event handler cannot wait on another thread that is attempting to subscribe or unsubscribe a handler to the same event. In other words, it's the original "unexpected deadlock" story (the same reason why locking on `this` is bad). Jon does make a note of this in Delegates and Events (http://www.yoda.arachsys.com/csharp/events.html).

To my knowledge, no one has proposed this as a solution. In general, the community seems to favor solutions that fail "loudly" (with exceptions) instead of failing "silently" (with a deadlock).

Why All Solutions are Wrong, by Stephen Cleary (that's me!)

"Callbacks" (usually events in C#) have always been problematic for multithreaded programming. This is because a good rule of thumb for component design is: **Do your best to allow the event handler to do *anything.*** Since "communicate with another thread that is attempting to take any lock" is one example of "anything", a natural corollary of this rule is: **Never hold locks during callbacks.**

This is the reasoning behind why locking on exposed objects (such as `this`) is considered bad practice (see MSDN: lock Statement (http://msdn.microsoft.com/en-us/library /c5kehkcz.aspx?WT.mc_id=DT-MVP-5000058)). Holding that lock while raising an event (such as solution 4 does) makes the bad practice even worse.

To review, all the solutions above fail in one of two situations.

Solutions 1-3 above all fail the same use case:

- Thread A will raise the event.
- Thread B subscribes a handler to the event. The handler code depends on a resource.
- Thread A begins to raise the event. Immediately before the delegate is invoked, Thread A is preempted by Thread B.
- Thread B no longer needs the event notification, so it unsubscribes the handler from the event and disposes of the resource.
- Thread A proceeds to raise the event (that has been unsubscribed). The handler code depends on a resource that has now been disposed.

Solution 4 fails this use case:

- Thread A will raise the event.

- Thread B subscribes a handler to the event. The handler code communicates with Thread C.
- Thread A begins to raise the event. Immediately before the delegate is invoked, Thread A is preempted by Thread C.
- Thread C subscribes a handler to the event. Thread C blocks.
- Thread A proceeds to raise the event. The handler code cannot communicate with Thread C because it is blocked.

A general-purpose “thread-safe event” solution does not exist - at least, not using the synchronization primitives we currently have at our disposal. The implementation *must* either have a race condition or deadlock possibility. A lock can prevent contention (solving the race condition), but only if it is held during the raising of the event (possibly causing deadlock). Alternatively, an unadorned raised event does not have the possibility of a deadlock, but loses the guarantees of the lock (causing a race condition).

A *general-purpose* solution does not exist, but it *is* possible to solve the problem for a specific event by imposing special requirements on the user. Some of the solutions above may work if one places restrictions on the event handlers.

Solution 3 (and solution 2 in Microsoft’s current implementations) works if the event handler is coded to handle the situation where it is invoked after it has unsubscribed from the event. It is not difficult to write a handler this way; asynchronous callback contexts (/2009/04/asynchronous-callback-contexts.html) would help with the implementation. The drawback is that each event handler must include multithread-awareness code, which complicates the method.

Solution 4 may also be made to work if the event handler does not block on a thread that subscribes to or unsubscribes from that same event. For simplicity, APIs that take this route often just state that event handlers may not block. The drawback is that this can be difficult to guarantee, since many objects hide their locking logic from their callers.

Conclusion

A general-purpose solution does not exist, and all other solutions have serious drawbacks (placing severe restrictions on the actions available to the event handler).

For this reason, I recommend the same approach that Jon Skeet ends up recommending at the end of Delegates and Events (http://www.yoda.arachsys.com/csharp/events.html): “don’t do that”, i.e., don’t use events in a multithreaded fashion. If an event exists on an object, then only one thread should be able to subscribe to or unsubscribe from that event, and it’s the same thread that will raise the event.

One nice side effect of this approach is that the code becomes much simpler:

```
public event MyEventHandler MyEvent;

protected virtual OnMyEvent(MyEventArgs args)
{
    if (this.MyEvent != null)
    {
        this.MyEvent(this, args);
    }
}
```

Efficiency freaks can go one step further and explicitly implement the backing field, add handler, and remove handler. By removing the default locking (which is useless), the code is more explicit but also more efficient:

```
private MyEventHandler myEvent;
public event MyEventHandler MyEvent
{
    add
    {
        this.myEvent += value;
    }

    remove
    {
        this.myEvent -= value;
    }
}

protected virtual OnMyEvent(MyEventArgs args)
{
    if (this.myEvent != null)
    {
        this.myEvent(this, args);
    }
}
```

Another side effect is that this type of event handling forces one towards Event-Based Asynchronous Programming (http://msdn.microsoft.com/en-us/library/hkasytyf.aspx?WT.mc_id=DT-MVP-5000058) (or something very similar to it). EBAP is a logical conclusion for asynchronous object design, yielding maximal reusability. EBAP is also more consistent with regards to normal object concurrency restrictions: “Public static members of this type are thread safe. Any instance members are not guaranteed to be thread safe.” Events that can only be accessed by one thread follow this common pattern; the event, as an instance member, is not guaranteed to be thread safe.

A third side effect takes longer to realize: more correct communication among threads. Instead of various threads directly subscribing to events (which would be run on another thread anyway), one must implement some form of thread communication. This forces the programmer to more clearly state the requirements from each thread’s perspective, and this in turn results in less buggy multithreading code. Usually, more appropriate ways for thread communciation are found. The event subscription model is naturally discarded as a thread communication method (due to its inherent unsuitability) in favor of much more proven design patterns. This will eventually result in more correct multithreading code, though the process requires a minor redesign.

A Final Note

As of this writing, it is still popular to promote solution 2 (copy the delegate before raising the event). However, I strongly discourage this practice; it makes the code more obscure, and provides a false sense of security because it *does not solve the problem!* It is far better to simply not have "thread-safe events".