ReaderWriterLockSlim.EnterUpgradeableReadLock() Always A Deadlock?

Asked 10 years, 10 months ago Modified 7 years, 8 months ago Viewed 9k times

9

I'm very familiar with `ReaderWriterLockSlim` but tried my hand at implementing `EnterUpgradeableReadLock()` recently in a class... Soon after I realized that this is almost certainly a guaranteed deadlock when 2 or more threads run the code:

```
Thread A --> enter upgradeable read lock
Thread B --> enter upgradeable read lock
Thread A --> tries to enter write lock, blocks for B to leave read
Thread B --> tries to enter write lock, blocks for A to leave read
Thread A --> waiting for B to exit read lock
Thread B --> waiting for A to exit read lock
```

What am I missing here?

EDIT

Added code example of my scenario. The `Run()` method would be called by 2 or more threads concurrently.

```
public class Deadlocker
{
    private readonly ReaderWriterLockSlim _lock = new
ReaderWriterLockSlim(LockRecursionPolicy.SupportsRecursion);

    public void Run()
    {
        _lock.EnterUpgradeableReadLock();
        try
        {
            _lock.EnterWriteLock();
            try
            {
                // Do something
            }
            finally
            {
                _lock.ExitWriteLock();
            }
        }
        finally
        {
            _lock.ExitUpgradeableReadLock();
        }
    }
}
```

c# multithreading thread-safety readerwriterlockslim

Share

Improve this question

Follow

edited Jan 28, 2014 at 16:27

asked Jan 28, 2014 at 16:17

Haney

34.6k ● 9 ● 60 ● 70

can you post some simple code that replicates the issue? – Gusdor Jan 28, 2014 at 16:23

@Gusdor sure, updating – Haney Jan 28, 2014 at 16:25

"Only one thread can enter upgradeable mode at any given time." - msdn.microsoft.com/en-us/library/… – Jon B Jan 28, 2014 at 16:30

@JonB then what's the point of it over a Write lock? Why does it exist? – Haney Jan 28, 2014 at 16:32

It seems to still have value when you have a single upgradable thread. If you have multiple threads that may need to write, then it's not the way to go. You could enter a read, exit, then enter a write. Then one of two threads would win the race. – Jon B Jan 28, 2014 at 16:33

3 Answers

Sorted by: Highest score (default)

40

A long time after the OP, but I don't agree with the currently accepted answer.

The statement `Thread B --> enter upgradeable read lock` is incorrect. From the docs

> Only one thread can be in upgradeable mode at any time

And in response to your comments: it is intended for a very different usage to a Read-Write pattern.

TL;DR. Upgradeable mode is useful:

- if a writer must check a shared resource before writing to it, and (optionally) needs to avoid race conditions with other writers;
- and it should not stop readers *until it is 100 % sure it must write* to the shared resource;
- and *it is quite likely a writer will decide it should not write* to the shared resource once it has performed the check.

Or, in pseudocode, where this:

```
// no other writers or upgradeables allowed in here => no race conditions
EnterUpgradeableLock();
if (isWriteRequired()) { EnterWriteLock(); DoWrite(); ExitWriteLock(); }
ExitUpgradeableLock();
```

gives "better performance" ÷ than this:

```
EnterWriteLock(); if (isWriteRequired()) { DoWrite(); } ExitWriteLock();
```

It should be used with care if the exclusive lock sections take a very long time due to it's use of SpinLock.

A similar lock construct

The Upgradeable lock is surprisingly similar to a SQL server SIX lock (Shared with Intent to go eXclusive) †.

- To rewrite the statement above in these terms, an Upgradeable lock says "a writer Intends to write to a resource, but wants to Share it with other readers while it [double]checks a condition to see if it should eXclusively lock and perform the write" ‡.

Without the existence of an Intent lock, you must perform the "should I make this change" check inside an eXclusive lock, which can hurt concurrency.

Why can't you share Upgradeable?

If the Upgradeable lock was shareable with other Upgradeable locks it would be possible to have a race condition with other Upgradeable lock owners. You would therefore require yet another check once inside the Write lock, removing the benefits of doing the check without preventing other reads in the first place.

Example

If we view all lock wait/entry/exit events as sequential, and the work inside a lock as parallel, then we can write a scenario in "Marble" form (`e` enter; `w` wait; `x` exit; `cr` check resource; `mr` mutate resource; `R` Shared/Read; `U` Intent/Upgradeable; `W` eXclusive/Write):

```
1--eU--cr--wW----eW--mr--xWxU--------------
2------eR----xR----------------eR--xR------
3--------eR----xR-------------------------
4----wU----------------------eU--cr--xU----
```

In words: T1 enters the Upgradeable/Intent lock. T4 waits for the Upgradeable/Intent lock. T2 and T3 enter read locks. T1 meanwhile checks the resource, wins the race and waits for an eXclusive/Write lock. T2&T3 exit their locks. T1 enters the eXclusive/Write lock and makes the change. T4 enters the Upgradeable/Intent lock, doesn't need to make it's change and exits, without blocking T2 which does another read in the meantime.

In 8 bullet points...

The Upgradeable lock is:

1. used by any Writer;
2. who is likely to check first and then decide not to perform the Write for any reason (losing a race condition, or in a Getsert pattern);
3. and who should not block Readers until it knows it must perform the Write;
4. whereupon it will take out an exclusive lock and do so.

Upgradeable is not required if one of the following apply (including but not limited to):

5. Contention rates between readers and writers who `writelock-check-nowrite-exit` are approximately zero (the write-condition check is super fast) - i.e. an Upgradeable construct doesn't help Reader throughput;
6. The probability of having a writer which Writes once in a Write lock is ~1 because either:
    - `ReadLock-Check-WriteLock-DoubleCheck` is so fast it only causes race losers once per trillion writes;
    - all changes are unique (all changes must happen, races can't exist); *or*
    - the "last change wins" (all changes still must happen, even though they aren't unique)

It is also not required if a `lock(...){...}` is more appropriate, i.e.:

7. the probability of overlapping read and/or write windows is low (locks can be as much about preventing very rare events as protecting highly likely events, not to speak of simple Memory Barrier requirements)
8. All your lock acquisitions are Upgradeable or Write, never Read ('duh')

[÷] *Where "performance" is up to you to define*

[†] *If you view the lock object as the table, and the protected resources as the resources lower in the hierarchy, this analogy approximately holds*

‡ *The initial check in a Read lock would be optional, the check within the Upgradeable lock is mandatory, therefore it can be used in a single or double check pattern.*

Share Improve this answer Follow

answered Oct 26, 2014 at 21:18

Andy Brown
19.1k ● 3 ● 54 ● 62

1 Fantastic answer! – lucounu Sep 24, 2016 at 23:37

2

I recommend avoiding EnterUpgradeableReadLock(). Just use EnterWriteLock() instead. I know that seems inefficient, the upgradeable read lock is almost as bad as a write lock anyway.

- http://ayende.com/blog/4349/using-readerwriterlockslims-enterupgradeablereadlock
- http://joeduffyblog.com/2007/02/07/introducing-the-new-readerwriterlockslim-in-orcas
- http://ayende.com/blog/4349/using-readerwriterlockslims-enterupgradeablereadlock
- Is ReaderWriterLockSlim.EnterUpgradeableReadLock() essentially the same as Monitor.Enter()?

Share Improve this answer Follow

edited May 23, 2017 at 11:47

Community Bot
1 ● 1

answered Jan 28, 2014 at 16:26

Moby Disk
3,851 ● 1 ● 21 ● 41

1 I think I see. It's a third "kind" of lock that respects the N reads but only allows a single thread to enter the upgradeable at any time... Good for when a single thread does the writes I guess, but still not exactly different than a Read...Write pattern? – Haney Jan 28, 2014 at 16:42

Not sure why I never marked this the answer. Thanks! – Haney Mar 16, 2014 at 17:43

-1

You have an error in your example

```
private readonly ReaderWriterLockSlim _lock = new
ReaderWriterLockSlim(LockRecursionPolicy.SupportsRecursion);
```

it should be

```
private static readonly ReaderWriterLockSlim _lock = new
ReaderWriterLockSlim(LockRecursionPolicy.SupportsRecursion);
```

Now in your code everytime a class is instantianed it is creating new instance of ReaderWriterLockSlim which is unable to lock anything because every single thread has it's own instance of it. Making it static will force all threads to use one instance which will work as it should

Share Improve this answer Follow

answered Apr 25, 2017 at 4:21

vortex

107 ● 12

1 That's not necessarily correct. If I held a static or singleton reference to this class in another, it would operate as intended. – Haney Apr 25, 2017 at 14:02

1 The OP's code deadlocking is the best proof that multiple (wannabe) writers are using the same lock instance, proving your proving him wrong wrong. – Evgeniy Berezovsky Nov 9, 2019 at 3:41