What's the best way of implementing a thread-safe Dictionary?

Asked 16 years, 2 months ago Modified 4 years, 8 months ago Viewed 71k times

110

I was able to implement a thread-safe Dictionary in C# by deriving from IDictionary and defining a private SyncRoot object:

```
public class SafeDictionary<TKey, TValue>: IDictionary<TKey, TValue>
{
    private readonly object syncRoot = new object();
    private Dictionary<TKey, TValue> d = new Dictionary<TKey, TValue>();

    public object SyncRoot
    {
        get { return syncRoot; }
    }

    public void Add(TKey key, TValue value)
    {
        lock (syncRoot)
        {
            d.Add(key, value);
        }
    }

    // more IDictionary members...
}
```

I then lock on this SyncRoot object throughout my consumers (multiple threads):

Example:

```
lock (m_MySharedDictionary.SyncRoot)
{
    m_MySharedDictionary.Add(...);
}
```

I was able to make it work, but this resulted in some ugly code. My question is, is there a better, more elegant way of implementing a thread-safe Dictionary?

c# .net thread-safety collections

Share
Improve this question
Follow

edited Nov 16, 2012 at 15:30
Slim
1,748 ● 5 ● 38 ● 60

asked Oct 1, 2008 at 14:36
GP.
1,293 ● 5 ● 15 ● 20

3 What is it you find ugly about it? – Asaf R Oct 1, 2008 at 14:43

1 I think he's referring to all the lock statements he has throughout his code within the consumers of the SharedDictionary class -- he's locking in the *calling* code everytime he's accessing a method on a SharedDictionary object. – Peter Meyer Oct 1, 2008 at 14:54

Instead of using Add method try doing by assigning values ex- m_MySharedDictionary["key1"]="item1", this is thread safe. – testuser Apr 12, 2012 at 16:53

8 Answers

Sorted by: Highest score (default)

209

The .NET 4.0 class that supports concurrency is named `ConcurrentDictionary`.

Share
Improve this answer
Follow

edited Jul 19, 2016 at 9:54
Uwe Keim
40.7k ● 61 ● 185 ● 301

answered Sep 13, 2010 at 19:11
Hector Correa
26.7k ● 8 ● 60 ● 75

4 Please mark this as response, you don't need custom dictoniary if the own .Net has a solution – Alberto León Mar 14, 2013 at 10:54

28 (Remember that the other answer was written long before the existence of .NET 4.0 (released in 2010).) – Jeppe Stig Nielsen Apr 9, 2013 at 16:48

2 Unfortunately it's not a lock-free solution, so it's useless in SQL Server CLR safe assemblies. You'd need something like what's described here: cse.chalmers.se/~tsigas/papers/Lock-Free_Dictionary.pdf or perhaps this implementation: github.com/hackcraft/Ariadne – Triynko Nov 4, 2013 at 23:15

2 Really old, I know, but it is important to note that using the ConcurrentDictionary vs a Dictionary can result in significant performance losses. This is most likely the result of expensive context switching, so be certain that you need a thread-safe dictionary before using one. – outbred Mar 7, 2016 at 17:37

`ConcurrentDictionary` can be surprisingly slow in some cases. Depending on your use case some alternative `ThreadSafeDictionary` implementation might be better (disclaimer: written by me). See the bottom of the page for comparisons and recommended use cases. (NuGet) – György Kőszeg Jan 12, 2022 at 16:34

63

Attempting to synchronize internally will almost certainly be insufficient because it's at too low a level of abstraction. Say you make the `Add` and `ContainsKey` operations individually thread-safe as follows:

```
public void Add(TKey key, TValue value)
{
    lock (this.syncRoot)
    {
        this.innerDictionary.Add(key, value);
    }
}

public bool ContainsKey(TKey key)
{
    lock (this.syncRoot)
    {
        return this.innerDictionary.ContainsKey(key);
    }
}
```

Then what happens when you call this supposedly thread-safe bit of code from multiple threads? Will it always work OK?

```
if (!mySafeDictionary.ContainsKey(someKey))
{
    mySafeDictionary.Add(someKey, someValue);
}
```

The simple answer is no. At some point the `Add` method will throw an exception indicating that the key already exists in the dictionary. How can this be with a thread-safe dictionary, you might ask? Well just because each operation is thread-safe, the combination of two operations is not, as another thread could modify it between your call to `ContainsKey` and `Add`.

Which means to write this type of scenario correctly you need a lock *outside* the dictionary, e.g.

```
lock (mySafeDictionary)
{
    if (!mySafeDictionary.ContainsKey(someKey))
    {
        mySafeDictionary.Add(someKey, someValue);
    }
}
```

But now, seeing as you're having to write externally locking code, you're mixing up internal and external synchronisation, which always leads to problems such as unclear code and deadlocks. So ultimately you're probably better to either:

1. Use a normal `Dictionary<TKey, TValue>` and synchronize externally, enclosing the compound operations on it, or
2. Write a new thread-safe wrapper with a different interface (i.e. not `IDictionary<T>`) that combines the operations such as an `AddIfNotContained`

method so you never need to combine operations from it.

(I tend to go with #1 myself)

Share Improve this answer Follow

answered Dec 30, 2008 at 0:07
Greg Beech
136k ●45 ●209 ●250

10 It's worth pointing out that .NET 4.0 will include a whole bunch of thread-safe containers such as collections and dictionaries, which have a different interface to the standard collection (i.e. they're doing option 2 above for you). – Greg Beech May 9, 2009 at 10:15

1 It's also worth noting that the granularity offered by even crude locking will often be sufficient for a single-writer multiple-readers approach if one designs a suitable enumerator which lets the underlying class know when it's disposed (methods which would want to write the dictionary while an undisposed enumerator exists should replace the dictionary with a copy). – supercat Aug 3, 2012 at 19:52

43

As Peter said, you can encapsulate all of the thread safety inside the class. You will need to be careful with any events you expose or add, making sure that they get invoked outside of any locks.

```
public class SafeDictionary<TKey, TValue>: IDictionary<TKey, TValue>
{
    private readonly object syncRoot = new object();
    private Dictionary<TKey, TValue> d = new Dictionary<TKey, TValue>();

    public void Add(TKey key, TValue value)
    {
        lock (syncRoot)
        {
            d.Add(key, value);
        }
        OnItemAdded(EventArgs.Empty);
    }

    public event EventHandler ItemAdded;

    protected virtual void OnItemAdded(EventArgs e)
    {
        EventHandler handler = ItemAdded;
        if (handler != null)
            handler(this, e);
    }

    // more IDictionary members...
}
```

Edit: The MSDN docs point out that enumerating is inherently not thread safe. That can be one reason for exposing a synchronization object outside your class. Another way to approach that would be to provide some methods for performing an action on all members and lock around the enumerating of the members. The problem with this is that you don't know if the action passed to that function calls some member of your dictionary (that would result in a deadlock). Exposing the synchronization object allows the consumer to make those decisions and doesn't hide the deadlock inside your class.

Share
Improve this answer
Follow

edited Oct 1, 2008 at 15:04

answered Oct 1, 2008 at 14:49
fryguybob
4,410 ● 2 ● 30 ● 36

@fryguybob: Enumeration was actually the only reason why I was exposing the synchronization object. By convention, I would perform a lock on that object only when I'm enumerating through the collection. – GP. Oct 1, 2008 at 15:49

1 If your dictionary isn't too large you can enumerate on a copy and have that built in to the class. – fryguybob Oct 1, 2008 at 17:06

2 My dictionary isn't too large, and I think that did the trick. What I did was make a new public method called CopyForEnum() which returns new instance of a Dictionary with copies of the private dictionary. This method was then called for enumarations, and the SyncRoot was removed. Thanks! – GP. Oct 1, 2008 at 17:29

12 This is also not an inherently threadsafe class since dictionary operations tend to be granular. A little logic along the lines of if (dict.Contains(whatever)) { dict.Remove(whatever); dict.Add(whatever, newval); } is assuredly a race condition waiting to happen. – plinth Dec 30, 2008 at 0:48

6

You shouldn't publish your private lock object through a property. The lock object should exist privately for the sole purpose of acting as a rendezvous point.

If performance proves to be poor using the standard lock then Wintellect's Power Threading collection of locks can be very useful.

Share Improve this answer Follow

answered Oct 1, 2008 at 14:51
Jonathan Webb
1,573 ● 1 ● 17 ● 26

5

There are several problems with implementation method you are describing.

1. You shouldn't ever expose your synchronization object. Doing so will open up yourself to a consumer grabbing the object and taking a lock on it and then you're toast.
2. You're implementing a non-thread safe interface with a thread safe class. IMHO this will cost you down the road

Personally, I've found the best way to implement a thread safe class is via immutability. It really reduces the number of problems you can run into with thread safety. Check out Eric Lippert's Blog for more details.

Share

Improve this answer

Follow

edited Mar 27, 2020 at 18:10

E. van Putten
673 ● 8 ● 20

answered Oct 1, 2008 at 15:13

JaredPar
753k ● 151 ● 1.3k ● 1.5k

3

You don't need to lock the SyncRoot property in your consumer objects. The lock you have within the methods of the dictionary is sufficient.

To Elaborate: What ends up happening is that your dictionary is locked for a longer period of time than is necessary.

What happens in your case is the following:

Say thread A acquires the lock on SyncRoot *before* the call to m_mySharedDictionary.Add. Thread B then attempts to acquire the lock but is blocked. In fact, all other threads are blocked. Thread A is allowed to call into the Add method. At the lock statement within the Add method, thread A is allowed to obtain the lock again because it already owns it. Upon exiting the lock context within the method and then outside the method, thread A has released all locks allowing other threads to continue.

You can simply allow any consumer to call into the Add method as the lock statement within your SharedDictionary class Add method will have the same effect. At this point in time, you have redundant locking. You would only lock on SyncRoot outside of one of the dictionary methods if you had to perform two operations on the dictionary object that needed to be guaranteed to occur consecutively.

Share

Improve this answer

Follow

edited Jun 19, 2012 at 8:44

El Ronnoco
11.9k ● 5 ● 39 ● 67

answered Oct 1, 2008 at 14:42

Peter Meyer
26.1k ● 1 ● 35 ● 53

1 Not true...if you do two operations after another that are internally thread-safe, that does not mean that the overall code block is thread safe. For instance: if(!myDict.ContainsKey(someKey)) { myDict.Add(someKey, someValue); } would not be threadsafe, even it ContainsKey and Add are threadsafe – Tim Sep 2, 2011 at 18:35

Your point is correct, but is out of context with my answer and the question. If you look at the question, it doesn't talk about calling ContainsKey, nor does my answer. My answer refers to acquiring a lock on SyncRoot which is shown in the example in the original question. Within the context of the lock statement one or more thread-safe operations would indeed execute safely. – Peter Meyer Sep 6, 2011 at 11:19

I guess if all he is ever doing is adding to the dictionary, but since he has "// more IDictionary members...", I assume at some point he is also going to want to read back data from the dictionary. If that is the case, then there needs to be some externally accessible locking mechanism. It doesn't matter if it is the SyncRoot in the dictionary itself or another object used solely for locking, but without such a scheme, the overall code will not be thread-safe. – Tim Sep 6, 2011 at 18:25

The external locking mechanism is as he shows in his example in the question: lock (m_MySharedDictionary.SyncRoot) { m_MySharedDictionary.Add(...); } -- it would be perfectly safe to do the following: lock (m_MySharedDictionary.SyncRoot) { if (!m_MySharedDictionary.Contains(...)) { m_MySharedDictionary.Add(...); } } In other words, the external locking mechanism is the lock statement that operates on the public property SyncRoot. – Peter Meyer Sep 7, 2011 at 1:43

0

Just a thought why not recreate the dictionary? If reading is a multitude of writing then locking will synchronize all requests.

example

```
    private static readonly object Lock = new object();
    private static Dictionary<string, string> _dict = new Dictionary<string,
string>();

    private string Fetch(string key)
    {
        lock (Lock)
        {
            string returnValue;
            if (_dict.TryGetValue(key, out returnValue))
                return returnValue;

            returnValue = "find the new value";
            _dict = new Dictionary<string, string>(_dict) { { key, returnValue
} };

            return returnValue;
        }
    }

    public string GetValue(key)
    {
        string returnValue;

        return _dict.TryGetValue(key, out returnValue)? returnValue :
Fetch(key);
    }
```

Share Improve this answer Follow

answered Nov 16, 2012 at 15:27

verbedr
2,232 ● 2 ● 19 ● 18

-6

[Collections And Synchronization]

Share Improve this answer Follow

answered Oct 1, 2008 at 14:52

MagicKat
9,811 ● 7 ● 34 ● 43

9 -1 I've voted this down because a) it's just a link with no explanation and b) it's just a link with no explanation! – El Ronnoco Jun 19, 2012 at 8:45