How scalable is System.Threading.Timer?

Asked 16 years, 3 months ago Modified 11 years, 3 months ago

Viewed 10k times

26

I'm writing an app that will need to make use of `Timer`s, but potentially very many of them. How scalable is the `System.Threading.Timer` class? The documentation merely say it's "lightweight", but doesn't explain further. Do these timers get sucked into a single thread (or very small threadpool) that processes all the callbacks on behalf of a `Timer`, or does each `Timer` have its own thread?

I guess another way to rephrase the question is: How is `System.Threading.Timer` implemented?

c# **.net** **multithreading** **timer**

Share

Improve this question

Follow

edited Sep 5, 2013 at 16:16

asked Aug 28, 2008 at 3:01

Ben Collins

20.7k ● 18 ● 128 ● 190

4 Answers

Sorted by: Highest score (default)

29

I say this in response to a lot of questions: Don't forget that the (managed) source code to the framework is available. You can use this tool to get it all: http://www.codeplex.com/NetMassDownloader

Unfortunately, in this specific case, a lot of the implementation is in native code, so you don't get to look at it...

They definitely use pool threads rather than a thread-per-timer, though.

The standard way to implement a big collection of timers (which is how the kernel does it internally, and I would suspect is indirectly how your big collection of Timers ends up) is to maintain the list sorted by time-until-expiry - so the system only ever has to worry about checking the next timer which is going to expire, not the whole list.

Roughly, this gives O(log n) for starting a timer and O(1) for processing running timers.

Edit: Just been looking in Jeff Richter's book. He says (of Threading.Timer) that it uses a single thread for all Timer objects, this thread knows when the next timer (i.e. as above) is due and calls ThreadPool.QueueUserWorkItem for the callbacks as appropriate. This has the effect that if you don't finish servicing one callback on a timer before the next is due, that your callback will reenter on another

pool thread. So in summary I doubt you'll see a big problem with having lots of timers, but you might suffer thread pool exhaustion if large numbers of them are firing at the same timer and/or their callbacks are slow-running.

Share Improve this answer Follow

edited Aug 28, 2008 at 19:55

answered Aug 28, 2008 at 8:12

Will Dean

39.5k • 11 • 92 • 118

A Priority Queue would probably be more efficient than a sorted list unless all the timers are added in bulk at the beginning, then sorted, and no more are added later. – RAL Feb 25, 2010 at 4:39

Sure - 'a list sorted by time-until-expiry' could certainly be a kind of priority queue - I didn't mean to imply 'a list that has had a Sort operation run across it' – Will Dean Feb 26, 2010 at 9:23

1 I just spent some time going over the code in the sscli. Note that the .NET ThreadPool has changed immensely since Rotor was released, so it is entirely possible that the System.Threading.Timer has also changed immensely. In fact timers were badly broken in .NET 1.1 and only fixed to be security safe and exception safe in .NET 2.0 Anyway, in Rotor the timers are kept in a linked list and fired by a dedicated Timer firing thread. There is one timer firing thread for the entire runtime (even across multiple app domains). – Michael Graczyk Jun 29, 2012 at 8:58

2 The thread just traverses the list of timers and calls (the native) QueueUserWorkItem for all timers that have expired. The operations of inserting new timers and deleting old ones

are performed as APC calls into the timer thread (msdn.microsoft.com/en-us/library/windows/desktop/…) In summary, the implementation is designed to be simple and safe (now), but not hugely performant. That said, It should be plenty fast enough for most situations, but if you are trying to create 10000 timers a second, you are probably going to run into trouble. – Michael Graczyk Jun 29, 2012 at 9:02

1 Actually after doing a little test it appears that timer traversal is still O(log n), but even still I was able to get timers to keep pace on my machine at ~20,000 per second.
– Michael Graczyk Jun 29, 2012 at 9:17

7

I think you might want to rethink your design (that is, if you have control over the design yourself). If you're using so many timers that this is actually a concern for you, there's clearly some potential for consolidation there.

Here's a good article from MSDN Magazine from a few years ago that compares the three available timer classes, and gives some insight into their implementations:

http://msdn.microsoft.com/en-us/magazine/cc164015.aspx

Share Improve this answer

Follow

answered Aug 28, 2008 at 7:23

TheSmurf
15.6k ● 3 ● 41 ● 48

5

> Consolidate them. Create a timer service and ask that for the timers. It will only need to keep 1 active timer (for the next due call)...

For this to be an improvement over just creating lots of Threading.Timer objects, you have to assume that it isn't exactly what Threading.Timer is already doing internally. I'd be interested to know how you came to that conclusion (I haven't disassembled the native bits of the framework, so you could well be right).

Share Improve this answer

Follow

answered Aug 28, 2008 at 9:34

Will Dean

39.5k ● 11 ● 92 ● 118

0

^^ as DannySmurf says : Consolidate them. Create a timer service and ask that for the timers. It will only need to keep 1 active timer (for the next due call) and a history of all the timer requests and recalculate this on AddTimer() / RemoveTimer().

Share Improve this answer

Follow

answered Aug 28, 2008 at 8:46

Quibblesome

25.4k ● 10 ● 62 ● 104