

Re: How can I make an efficient First-Come-First-Serve locking mechani

Source: <http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.general/2004-06/2199.html>

From: Niki Estner (niki.estner_at_cube.net)

Date: 06/22/04

Date: Tue, 22 Jun 2004 22:47:20 +0200

The docs about mutexes say: (actually win32 mutexes, but I'd guess this also applies to .net-mutexes on a win32 system)

"Threads that are waiting for ownership of a mutex are placed in a first in, first out (FIFO) queue. Therefore, the first thread to wait on the mutex will be the first to receive ownership of the mutex, regardless of thread priority. However, kernel-mode APCs and events that suspend a thread will cause the system to remove the thread from the queue. When the thread resumes its wait for the mutex, it is placed at the end of the queue."

Thus you cannot be 100% sure your threads will **always** come in fifo order, however your worst-case scenario does seem pretty implausible. Also, the priority of threads that have been waiting for too long will be increased by the OS, so they will (after a second or so) be first anyway.

The same docs say that this ordering cannot be assumed for critical sections; I **think** Monitor.Enter internally uses critical sections, but I really don't know.

Last but not least, a little suggestion: Maybe, if your Write method is that expensive (I suppose a database or file), and so many threads access, you should somehow buffer it? Waiting threads reduce throughput, a few extra instructions might make everything a lot faster?

Niki

"Chris Tanger" <Chris.Tanger@discussions.microsoft.com> wrote in news:AF4C6CAA-EDB7-4806-A2AC-B825C9F21F0C@microsoft.com...
> *I am creating a class that has a method "Write" that I wish to make threadsafe. The method must block calling threads until the task performed in write is complete. Only 1 thread at a time can perform the task within "Write". 1-10 different threads may call "Write" simultaneously and continuously, some in a greedy manner. That is to say that some of the threads calling "Write" will take all they can get, while other threads may only call "Write" once in a while.*
> *I have considered using a waitHandle, Monitor, or a C# lock statement*

however I have heard that these thread concurrency items will not guarantee first-come-first-serve to threads. This could cause the once in a while threads to get starved by the greedy threads. For example: Say that thread "a", "b", and "c" are greedy and they will each call "Write" within an endless loop. Say that thread "d" is not greedy and simply needs to send a message every 5 seconds. Say we have the following code.

```
>  
> public void Write()  
> {  
> Monitor.Enter(commonlockingobject)  
> Thread.Sleep(500); // Simulate work  
> Montor.Exit(commonlockingobject)  
> }  
>  
> Assume that all threads are started in the order a, b, c, d and of course  
call "Write" immediately.  
>  
> So it is a given that thread a acquires the Monitor immediately and  
the remaining threads wait. Who will acquire the lock next? As I  
understand, there is no way to know for sure. It will probably be "b", but  
there are no guarantees.  
> Considering that there are no guarantees let us say that once "a"  
exits the Monitor "b" acquires the lock next. "a" calls "Write" again as  
soon as its call returns and of course "a" is blocked by Monitor.Enter. Now  
let us say that once thread "b" call Monitor.Exit thread "a" acquires the  
Monitor again (why not there are no guarantees?). Then thread "b" calls  
"Write" again and is again blocked by Monitor.Enter. So say thread "b"  
acquires the Monitor next, and then thread "a", and then "b" again, and so  
on. So thread "c" and "d" never get called. This is not probably but as  
far as I understand this could happen. Even if thread "d" wasn't able to  
acquire the Monitor for 20 seconds that would be unacceptable. Thread "d"  
should be able to acquire the Monitor after 1.5 seconds in this scenario and  
if the probability of each thread acquiring the lock is even thread "d"  
should acquire the lock on average about every 0.75 seconds.
```

```
>  
> OK so you see my problem, now how to get around this problem.  
Assuming that Monitor acquisition is not first-come-first-serve I must do  
something differently, but it must still be as efficient as possible. One  
of my thoughts is code similar to the following:
```

```
>  
> Queue q = new Queue();  
>  
> public void Write()  
> {  
> // I could of course simply create a stack of size N where N is the  
> // max potential number of calling threads and put AutoResetEvents  
> // on the stack when they are not being used and pop them off the  
stack  
> // while they are being used, that would avoid all this object  
creation  
> // but would then require a synchronized stack wrapper which might
```

slow things down

> *// even more*

> *AutoResetEvent are = new AutoResetEvent(false);*

>

> *lock(q)*

> {

> *if(q.Count > 0)*

> *q.Enqueue(are);*

> *else*

> *are.Set();*

> }

>

> *are.WaitOne();*

>

> *lock(q)*

> {

> *if(q.Count > 0)*

> *((AutoResetEvent)q.Dequeue()).Set();*

> }

>

> }

>

>

> *Two questions:*

> *1) Does anyone have proof that Monitor.Enter is or is NOT*

first-come-first-serve? I asked a similar question before in a newsgroup and got an MVP reply as well as some others, but no one could provide a reference to a reliable source of documentation on the subject.

> *2) If Monitor.Enter is NOT first-come-fist-serve does anyone have a way of doing this in a more efficient manner than the code above? I am pretty certain the answer is yes, I would very much like to see the superior or better still ultimate solution to this problem.*

>

>

>

> *Thanks,*

>

> *-Chris Tanger*

>