How to avoid garbage collection in real time .NET application?

Asked 16 years, 3 months ago Modified 2 years, 3 months ago

Viewed 29k times

13

I'm writting a financial C# application which receive messages from the network, translate them into different object according to the message type and finaly apply the application business logic on them.

The point is that after the business logic is applied, I'm very sure I will never need this instance again. Rather than to wait for the garbage collector to free them, I'd like to explicitly "delete" them.

Is there a better way to do so in C#, should I use a pool of object to reuse always the same set of instance or is there a better strategy.

The goal being to avoid the garbage collection to use any CPU during a time critical process.

c# .net garbage-collection real-time finance

Share

Improve this question

Follow

edited Jan 24, 2013 at 14:14

miguel
71 ● 1 ● 2 ● 7

asked Sep 17, 2008 at 16:54

Seb
249 ● 2 ● 4 ● 10

In .Net, pooling is useful when either (a) creating a given object is slow, or (b) in a 32-bit app, to lessen memory fragmentation, of an array/list/dict etc that uses a data block > 64 KB, e.g. > 8K elements x 8 B per element, or 16K elements x 4 B per element. .Net **never relocates (moves) such large blocks**, which can lead to fragmented memory, if you request different sizes each time. Safer to allocate a large enough # elements up front, and retain them. (But clear their fields when not in use, so GC can reclaim whatever they point to.) If exceed allocated size, double the # elements.
– ToolmakerSteve Mar 22, 2014 at 20:34

In the extreme case, I have one 32-bit app (due to dependencies that don't work in 64-bit), so even though I am on a 64-bit Windows with lots of RAM, to avoid memory fragmentation I had to keep lists-of-lists, so that each sub-list fit in 64 KB, so that .Net would not allocate on "large object heap". So it could "compact" memory during GC, rather than creating large "holes" in the 4 GB address space of the app. This particular app was problematic because it included a lot of DirectX allocation from native (non-.Net) memory. The mix of non-.Net and .Net allocations led to fragmentation.
– ToolmakerSteve Mar 22, 2014 at 20:41

13 Answers

Sorted by: Highest score (default)

23

Don't delete them right away. Calling the garbage collector for each object is a bad idea. Normally you *really* don't want to mess with the garbage collector at all, and even time critical processes are just race conditions waiting to happen if they're that sensitive.

But if you know you'll have busy vs light load periods for your app, you might try a more general GC.Collect() when you reach a light period to encourage cleanup before the next busy period.

Share Improve this answer Follow

edited Feb 9, 2009 at 16:49

answered Sep 17, 2008 at 16:57

Joel Coehoorn
415k ● 114 ● 577 ● 813

8 This is fine as long as you understand that GC.Collect does not provide deterministic garbage collection. You are only telling the GC to run its cycle, in which case is determines whether to collect the memory. The call to GC.Collect does not guarantee memory collection. – JeremiahClark Oct 18, 2008 at 3:57

2 Be aware that **calling GC.Collect() too often can do more harm than good**: any objects that still have references will be moved from generation 0 to gen 1, or from gen 1 to gen 2. Any such objects will stick around longer / require more work for GC to remove. ONLY call GC.Collect() when there are NO messages still being worked on, so that there are no references to any temporary data. – ToolmakerSteve Mar 22, 2014 at 20:22

Look here: http://msdn.microsoft.com/en-us/library/bb384202.aspx

18

You can tell the garbage collector that you're doing something critical at the moment, and it will try to be nice to you.

Share Improve this answer Follow

answered Sep 17, 2008 at 16:59

Thomas
181k ● 55 ● 376 ● 501

2 Good blog entry at blogs.microsoft.co.il/blogs/sasha/archive/2008/08/10/… – Curt Hagenlocher Sep 17, 2008 at 17:11

11

You hit in yourself -- use a pool of objects and reuse those objects. The semantics of the calls to those object would need to be hidden behind a factory facade. You'll need to grow the pool in some pre-defined way. Perhaps double the size everytime it hits the limit -- a high water algorithm, or a fixed percentage. I'd really strongly advise you not to call GC.Collect().

When the load on your pool gets low enough you could shrink the pool and that will eventually trigger a garbage collection -- let the CLR worry about it.

Share Improve this answer Follow

answered Sep 17, 2008 at 17:35

Phil Bennett
4,839 ● 4 ● 31 ● 28

7

Attempting to second-guess the garbage collector is generally a very bad idea. On Windows, the garbage collector is a generational one and can be relied upon to be pretty efficient. There are some noted exceptions to this general rule - the most common being the occurrence of a one-time event that you know for a fact will have caused a lot of old objects to die - once objects are promoted to Gen2 (the longest lived) they tend to hang around.

In the case you mention, you sound as though you are generating a number of short-lived objects - these will result in Gen0 collections. These happen relatively often anyway, and are the most efficient. You could avoid them by having a reusable pool of objects, if you prefer, but it is best to ascertain for certain if GC is a performance problem before taking such action - the CLR profiler is the tool for doing this.

It should be noted that the garbage collector is different on different .NET frameworks - on the compact framework (which runs on the Xbox 360 and on mobile platforms) it is a non-generational GC and as such you must be much more careful about what garbage your program generates.

Share Improve this answer Follow

answered Sep 17, 2008 at 17:30

NM.

818 ● 6 ● 11

6

Forcing a GC.Collect() is generally a bad idea, leave the GC to do what it does best. It sounds like the best solution would be to use a pool of objects that you can grow if necessary - I've used this pattern successfully.

This way you avoid not only the garbage collection but the regular allocation cost as well.

Finally, are you sure that the GC is causing you a problem? You should probably measure and prove this before implementing any perf-saving solutions - you may be causing yourself unnecessary work!

Share Improve this answer Follow

answered Sep 17, 2008 at 17:02

Simon Steele
11.6k ● 4 ● 47 ● 67

5

> "The goal being to avoid the garbage collection to use any CPU during a time critical process"

Q: If by time critical, you mean you're listening to some esoteric piece of hardware, and you can't afford to miss the interrupt?

A: If so then C# isn't the language to use, you want Assembler, C or C++ for that.

Q: If by time Critical, you mean while there are lots of messages in the pipe, and you don't want to let the Garbage collector slow things down?

A: If so you are worrying needlessly. By the sounds of things your objects are very short lived, this means the garbage collector will recycle them very efficiently, without any apparent lag in performance.

However, the only way to know for sure is test it, set it up to run overnight processing a constant stream of test messages, I'll be stunned if you your performance stats can spot when the GC kicks in (and even if you can spot it, I'll be even more surprised if it actually matters).

Share Improve this answer Follow

edited Sep 9, 2022 at 16:01
StayOnTarget
12.9k ● 11 ● 59 ● 105

answered Feb 9, 2009 at 17:09
Binary Worrier
51.7k ● 20 ● 142 ● 186

3

Get a good understanding and feel on how the Garbage Collector behaves, and you will understand why what you are thinking of here is not recommended. unless you really like the CLR to spend time rearranging objects in memory *alot*.

- http://msdn.microsoft.com/en-us/magazine/bb985010.aspx
- http://msdn.microsoft.com/en-us/magazine/bb985011.aspx

Share Improve this answer Follow

answered Sep 17, 2008 at 17:10

icelava
9,857 ● 7 ● 53 ● 75

3

How intensive is the app? I wrote an app that captures 3 sound cards (Managed DirectX, 44.1KHz, Stereo, 16-bit), in 8KB blocks, and sends 2 of the 3 streams to another computer via TCP/IP. The UI renders an audio level meter and (smooth) scrolling title/artist for each of the 3 channels. This runs on PCs with XP, 1.8GHz, 512MB, etc. The App uses about 5% of the CPU.

I stayed clear of manually calling GC methods. But I did have to tune a few things that were wasteful. I used RedGate's Ant profiler to hone in on the wasteful portions. An awesome tool!

I wanted to use a pool of pre-allocated byte arrays, but the managed DX Assembly allocates byte buffers internally, then returns that to the App. It turned out that I didn't have to.

Share Improve this answer Follow

answered Oct 18, 2008 at 3:25

GregUzelac
472 ● 2 ● 6

Completely non-astroturf response, but I'm reading this question because we're facing a similar issue to the original poster's, and one of the first things mentioned was Ants.
– Marc Bollinger Nov 6, 2008 at 22:17

An 8kB buffer for 44.1kHz stereo 16-bit sample streams makes for about 45ms delay. That's far from realtime. You'd have a lot more trouble getting below 5ms unless you stick to non-allocating native programming. – Johan Boulé Feb 19, 2018 at 1:11

2

If it is absolutely time critical then you should use a deterministic platform like C/C++. Even calling GC.Collect() will generate CPU cycles.

Your question starts off with the suggestion that you want to save memory but getting rid of objects. This is a space critical optimization. You need to decide what you really want because the GC is better at optimizing this situation than a human.

Share Improve this answer Follow

answered Sep 17, 2008 at 17:07

Brian Lyttle
14.6k ● 15 ● 69 ● 106

2

From the sound of it, it seems like you're talking about deterministic finalization (destructors in C++), which doesn't exist in C#. The closest thing that you will find in C# is the Disposable pattern. Basically you implement the **IDisposable** interface.

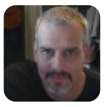
The basic pattern is this:

```
public class MyClass: IDisposable
{
    private bool _disposed;
    public void Dispose()
    {
        Dispose( true );
        GC.SuppressFinalize( this );
    }

    protected virtual void Dispose( bool disposing )
    {
        if( _disposed )
            return;

        if( disposing )
        {
            // Dispose managed resources here
        }

        _disposed = true;
    }
}
```

Share Improve this answer

Follow

answered Sep 17, 2008 at 21:43



Eddie Velasquez
573 ● 6 ● 12

Is this dangerous if your class has other reference types as properties? Will they still be left behind for garbage collection or dealt with immediately when disposed? – jocull Apr 2, 2014 at 13:22

---

1 @jocull `Dispose` actually has nothing to do with garbage collection. The best you can do there is deterministically dispose of *un*managed resources (of course, you should also have a finalizer in that case), or set some internal references to `null` (thus making them possibly eligible for collection). The only part that's deterministic is the unmanaged resources. Managed memory is *always* managed - this prevents you from accidentally deleting something that still has references, for example. Even the stack isn't *guaranteed*

to be there. It's an implementation detail, not part of the spec.
– Luaan Mar 30, 2015 at 12:45

1

You could have a limited amount of instances of each type in a pool, and reuse the already done with instances. The size of the pool would depend on the amount of messages you'll be processing.

Share Improve this answer Follow

answered Sep 17, 2008 at 17:02

Vinko Vrsalovic
340k ● 55 ● 340 ● 373

1

Instead of creating a new instance of an object every time you get a message, why don't you reuse objects that have already been used? This way you won't be fighting against the garbage collector and your heap memory won't be getting fragmented.**

For each message type, you can create a pool to hold the instances that are not in use. Whenever you receive a network message, you look at the message type, pull a waiting instance out of the appropriate pool and apply your business logic. After that, you put that instance of the message object back into it's pool.

You will most likely want to "lazy load" your pool with instances so your code scales easily. Therefore, your pool class will need to detect when a null instance has been pulled and fill it up before handing it out. Then when the calling code puts it back in the pool it's a real instance.

** "Object pooling is a pattern to use that allows objects to be reused rather than allocated and deallocated, which helps to prevent heap fragmentation as well as costly GC compactions."

http://geekswithblogs.net/robp/archive/2008/08/07/speedy-c-part-2-optimizing-memory-allocations---pooling-and.aspx

Share Improve this answer Follow

edited Jul 30, 2022 at 14:06
Glorfindel
22.6k ● 13 ● 89 ● 116

answered Sep 17, 2008 at 17:15
Wayne Bloss
5,540 ● 7 ● 52 ● 85

0

In theory the GC shouldn't run if your CPU is under heavy load or unless it really needs to. But if you have to, you may want to just keep all of your objects in memory, perhaps a singleton instance, and never clean them up unless you're ready. That's probably the only way to guarantee when the GC runs.

Share Improve this answer Follow

answered Sep 17, 2008 at 17:06
Ryan Rife
