GC contains lots of pinned objects after a while

Asked 16 years ago Modified 15 years, 10 months ago Viewed 2k times

4

I have a strange phenomenon while continuously instantiating a com-wrapper and then letting the GC collect it (not forced).

I'm testing this on .net cf on WinCE x86. Monitoring the performance with .net Compact framework remote monitor. Native memory is tracked with Windows CE Remote performance monitor from the platform builder toolkit.

During the first 1000 created instances every counter in perfmon seems ok:

- GC heap goes up and down but the average remains the same
- Pinned objects is 0
- native memory keeps the same average
- ...

However, after those 1000 (approximately) the Pinned object counter goes up and never goes down in count ever again. The memory usage stays the same however.

I don't know what conclusion to pull from this information... Is this a bug in the counters, is this a bug in my software?

[EDIT]

I do notice that the Pinned objects counter starts to go up as soon the total bytes in use after GC stabilises as does the Objects not moved by compactor counter.

The graphic of the counters http://files.stormenet.be/gc_pinnedobj.jpg

[/EDIT]

Here's the involved code:

```
private void pButton6_Click(object sender, EventArgs e) {
    if (_running) {
        _running = false;
        return;
    }
    _loopcount = 0;
    _running = true;

    Thread d = new Thread(new ThreadStart(LoopRun));
    d.Start();
}
```

```
    private void LoopRun() {
        while (_running) {
            CreateInstances();
            _loopcount++;
            RefreshLabel();
        }
    }

    void CreateInstances() {
        List<Ppb.Drawing.Image> list = new List<Ppb.Drawing.Image>();
        for (int i = 0; i < 10; i++) {
            Ppb.Drawing.Image g = resourcesObj.someBitmap;
            list.Add(g);
        }

    }
```

The Image object contains an AlphaImage:

```
    public sealed class AlphaImage : IDisposable {
    IImage _image;
    Size _size;
    IntPtr _bufferPtr;

    public static AlphaImage CreateFromBuffer(byte[] buffer, long size) {
        AlphaImage instance = new AlphaImage();
        IImage img;
        instance._bufferPtr = Marshal.AllocHGlobal((int)size);
        Marshal.Copy(buffer, 0, instance._bufferPtr, (int)size);
        GetIImagingFactory().CreateImageFromBuffer(instance._bufferPtr,
(uint)size, BufferDisposalFlag.BufferDisposalFlagGlobalFree, out img);
        instance.SetImage(img);
        return instance;
    }

    void SetImage(IImage image) {
        _image = image;
        ImageInfo imgInfo;
        _image.GetImageInfo(out imgInfo);
        _size = new Size((int)imgInfo.Width, (int)imgInfo.Height);
    }

    ~AlphaImage() {
        Dispose();
    }

    #region IDisposable Members

    public void Dispose() {
        Marshal.FinalReleaseComObject(_image);
    }
}
```

c# memory com compact-framework garbage-collection

Share

Improve this question

Follow

edited Feb 11, 2009 at 20:58

Scott Dorman

42.5k ● 12 ● 81 ● 112

asked Dec 17, 2008 at 10:52

Stormenet

26.4k ● 9 ● 56 ● 68

2 Answers

Sorted by: Highest score (default)

4

Well, there's a bug in your code in that you're creating a lot of IDisposable instances and never calling Dispose on them. I'd hope that the finalizers would eventually kick in, but they shouldn't really be necessary. In your production code, do you dispose of everything appropriately - and if not, is there some reason why you can't?

If you put some logging in the AlphaImage finalizer (detecting AppDomain unloading and application shutdown and not logging in those cases!) does it show the finalizer being called?

EDIT: One potential problem which probably *isn't* biting you, but may be worth fixing anyway - if the call to CreateImageFromBuffer fails for whatever reason, you still own the memory created by AllocHGlobal, and that will currently be leaked. I suspect that's not the problem or it would be blowing up more spectacularly, but it's worth thinking about.

Share

Improve this answer

Follow

edited Dec 17, 2008 at 11:49

answered Dec 17, 2008 at 10:59

Jon Skeet

1.5m ● 889 ● 9.3k ● 9.3k

> The finalizer is being called (if it wouldn't I would be out of memory in a matter of seconds). Actually, there is an Image class wrapper around the AlphaImage class that calls the Dispose of the AlphaImage. – Stormenet Dec 17, 2008 at 11:38

> Still, you shouldn't count on the finalizer - that's what IDisposable is for. You should use the using (image) { ... } construct to make sure the Dispose method is called as soon as possible. Also, use GC.SuppressFinalize() in Dispose() so that the finalizer will not be called for a disposed object. – configurator Dec 17, 2008 at 13:00

> @configurator: True about the suppressFinalize, I added that. However, since it's an Image, much like an usual bitmap, it is used in usercontrols so you can't use the using construct, so I have to rely on the finalizer. – Stormenet Dec 17, 2008 at 13:16

> UserControls should have a Dispose tree going down from top control to bottom. That what the dispose method automatically generated in the designer.cs is for in part. This should call your user control which should dispose any children that contain resources. – Quibblesome Dec 23, 2008 at 13:38

I doubt it's a bug in RPM. What we don't have here is any insight into the Ppb.Drawing stuff. The place I see for a potential problem is the GetIImagingFactory call. What

3

does it do? It's probably just a singleton getter, but it's something I'd chase.

I also see an AllochHGlobal, but nowhere do I see that allocation getting freed. For now that's where I'd focus.

Share Improve this answer Follow

answered Dec 17, 2008 at 12:19

ctacke

67.2k • 20 • 98 • 155

GetIImagingFactory is indeed a singleton getter. The Allocated memory from AllocHGlobal is freed by the com object itself. In the CreateImageFromBuffer call I secify it should free the memory from HGlobal memory. – Stormenet Dec 17, 2008 at 12:46

It was the Allocation that caused the pinned objects. The com object did free the memory, but the .net clr still thought it was in use, thus causing a leak. – Stormenet Sep 16, 2011 at 12:30