This use of GC.SuppressFinalize() doesn't feel right

Asked 11 years, 9 months ago Modified 11 years, 6 months ago Viewed 3k times

10

I have been having some issues with utilizing a vendor library where occasionally an entity calculated by the library would be null when it should always have valid data in it.

The functioning code (after debugging the issue with the vendor) is roughly as follows:

```
    Task.Factory.StartNew(() => ValidateCalibration(pelRectRaw2Ds,
crspFeatures, Calibration.Raw2DFromPhys3Ds));

    .....

    private void ValidateCalibration(List<Rectangle> pelRectRaw2Ds,
List<List<3DCrspFeaturesCollection>> crspFeatures, List<3DCameraCalibration>
getRaw2DFromPhys3Ds)
    {
        var calibrationValidator = new 3DCameraCalibrationValidator();

        // This is required according to vendor otherwise
validationResultsUsingRecomputedExtrinsics is occasionally null after
preforming the validation
        GC.SuppressFinalize(calibrationValidator);

        3DCameraCalibrationValidationResult
validationResultUsingOriginalCalibrations;
        3DCameraCalibrationValidationResult
validationResultsUsingRecomputedExtrinsics;
        calibrationValidator.Execute(pelRectRaw2Ds, crspFeatures,
getRaw2DFromPhys3Ds, out validationResultUsingOriginalCalibrations, out
validationResultsUsingRecomputedExtrinsics);

        Calibration.CalibrationValidations.Add(new CalibrationValidation
            {
                Timestamp = DateTime.Now,
                UserName = Globals.InspectionSystemObject.CurrentUserName,
                ValidationResultUsingOriginalCalibrations =
validationResultUsingOriginalCalibrations,
                ValidationResultsUsingRecomputedExtrinsics =
validationResultsUsingRecomputedExtrinsics
            });
    }
```

The validation process is a fairly time consuming operation so I hand it off to a Task. The problem I had was that originally I did not have the call to GC.SuppressFinalize(calibrationValidator) and when the application was run from a Release build, then the out parameter validationResultsUsingRecomputedExtrinsics would be null. If I ran the application from a Debug build (either with or without the

Debugger attached) then validationResultsUsingRecomputedExtrinsics would contain valid data.

I don't fully understand what GC.SuppressFinalize() has done in this situation, or how it has fixed the problem. Everything I can find regarding GC.SuppressFinalize() is that it is used when implementing IDisposable. I can't find any use of it in "standard" code.

How/why does the addition of the call to GC.SuppressFinalize(calibrationValidator) fix this problem?

I understand that without intimate knowledge of the internals of the vendor library, it might not be possible to know for sure, but any insight would help.

The application is compiled with VS2012, targeting .NET 4.0. That vendor library requires that the useLegacyV2RuntimeActivationPolicy="true" option is specified in app.config.

This is the justification I received from the vendor:

> The SuppressFinalize command makes sure that the garbage collector will not clean something up “early”. It seems like for some reason your application was sometimes having the garbage collector get a bit zealous and clean up the object before you were truly done with it; it is almost certainly scope related and possibly due to the multi-threading causing confusion on the scope of the calibrationValidator. Below is the response I got from Engineering.
>
> Because the variable was created in the local scope, and that function runs in the background thread, Garbage Collection runs in the main thread, and it seems that the Garbage collection is not smart enough in handling multi-thread situations. Sometimes, it just releases it too early (internal execution of validator not finished yet, and still needs this variable).

c# .net garbage-collection

Share

Improve this question

Follow

edited Mar 22, 2013 at 2:15

asked Mar 22, 2013 at 0:05

Dave Nay

559 ● 5 ● 23

2 Looks like a code smell to me. – Bernard Mar 22, 2013 at 0:14

1 @Bernard I agree...hence the reason I am looking for clarification. – Dave Nay Mar 22, 2013 at 0:16

If this is more appropriate for StackOverflow, that's fine (I'm not going to duplicate it, please migrate it) – Dave Nay Mar 22, 2013 at 0:17

I think this is most likely caused by a bug in the finalizer of `3DCameraCalibrationValidator` (because the authors didn't expect that the finalizer could run while `Execute()` is executing). Using `SuppressFinalize()` works around that bug. – svick Mar 22, 2013 at 1:15

3 Answers

Sorted by: Highest score (default)

18

This is in all likelihood a hack to solve a premature garbage collection problem. Not uncommon with unmanaged code, typical in camera applications. It is not a healthy hack, good odds that this will cause a resource leak because the finalizer doesn't execute. Wrappers for unmanaged code almost always have something to do in the finalizer, it is very common that they need to release unmanaged memory.

At issue is that the calibrationValidator object can be garbage collected while the unmanaged code is running. Having another thread in your program makes this likely since that other thread can be allocating objects and trigger a GC. This is very easy to miss by the owner of the code while testing, either by never having tested it while using multiple threads or just not getting lucky enough to trigger a GC at the wrong time.

The proper fix on your end is to ensure that the jitter marks the object in use past the call so that the garbage collector won't collect it. You do so by adding `GC.KeepAlive(calibrationValidator)` after the `Execute()` call.

Share
Improve this answer
Follow

edited Jun 18, 2013 at 18:11

answered Mar 22, 2013 at 3:24
Hans Passant
940k ● 148 ● 1.7k ● 2.6k

Thank you Hans. I should have stated in my original question that this being a resource leak was an additional concern. – Dave Nay Mar 22, 2013 at 11:55

3

When it comes to understanding `IDisposable`, `GC.SuppressFinalize`, and finalizers in C#, I don't think a better explanation exists than the following article.

DG Update: Dispose, Finalization, and Resource Management

> Alright! Here it is: the revised “Dispose, Finalization, and Resource Management” Design Guideline entry. I mentioned this work previously here and here. At ~25 printed pages, it's not what I would consider to be a minor update. Took me much longer than anticipated, but I'm happy with the result. I got to work with and received good amounts of feedback from HSutter, BrianGru, CBrumme, Jeff Richter, and a couple other folks on it... Good fun.

Key Concept for this question:

It is *so obvious* that `GC.SuppressFinalize()` should only be called on `this` that the article doesn't even mention that directly. It does however mention the practice of wrapping finalizable objects to isolate them from a public API in order to ensure that external code is not able to call `GC.SuppressFinalize()` on those resources (see the following quote). Whoever designed the library described in the original question has no grasp on the way finalization in .NET works.

Quoted from the blog article:

> Even in the absence of one of the rare situations noted above, a finalizable object with a publicly accessible reference could have its finalization suppressed by any arbitrary untrusted caller. Specifically, they can call GC.SuppressFinalize on you and prevent finalization from occurring altogether, including critical finalization. A good mitigation strategy to deal with this is to wrap critical resources in a non-public instance that has a finalizer. So long as you do not leak this to callers, they will not be able to suppress finalization. If you migrate to using SafeHandle in your class and never expose it outside your class, you can guarantee finalization of your resources (with the caveats mentioned above and assuming a correct SafeHandle implementation).

Share

Improve this answer

Follow

edited Mar 22, 2013 at 1:59

answered Mar 22, 2013 at 1:52

Sam Harwell

99.7k ● 22 ● 214 ● 282

3 "Whoever designed the library described in the original question has no grasp on the way finalization in .NET works." I would believe this. They are primarily a C++ company and I am certain that their .NET library is only a wrapper to their C++ library. I have added their response to my question, which seems to support the idea that this is a poorly implemented library. – Dave Nay Mar 22, 2013 at 2:17

@DaveNay Feel free to send them my contact information (tunnelvisionlabs.com). We've helped more than one team get their finalization details in order, especially when managed/native interop is at play. :) – Sam Harwell Mar 22, 2013 at 2:24

2 They should use `GC.KeepAlive (this)` at the end of that Execute function to prevent `this` being garbage-collected and finalized too early. As a workaround, you can use `GC.KeepAlive (calibrationValidator)` after the Execute call. – Anton Tykhyy Mar 22, 2013 at 2:52

1

There are some mentions of multithreading or native code being the cause of this issue. But the same thing can happen in a purely managed and mostly single-threaded program.

Consider the following program:

```
using System;

class Program
{
    private static void Main()
    {
        var outer = new Outer();
        Console.WriteLine(outer.GetValue() == null);
    }
}

class Outer
{
    private Inner m_inner = new Inner();

    public object GetValue()
    {
        return m_inner.GetValue();
    }

    ~Outer()
    {
        m_inner.Dispose();
    }
}

class Inner
{
    private object m_value = new object();

    public object GetValue()
    {
        GC.Collect();
        GC.WaitForPendingFinalizers();
        return m_value;
    }

    public void Dispose()
    {
        m_value = null;
    }
}
```

Here, while `outer.GetValue()` is being called, `outer` will be garbage collected and finalized (at least in Release mode). The finalizer nulls out the field of the `Inner` object, which means `GetValue()` will return `null`.

In real code, you most likely wouldn't have the `GC` calls there. Instead you would create some managed object, which (non-deterministically) causes the garbage collector to run.

(I said this code is mostly single-threaded. In fact, the finalizer will run on another thread, but because of the call to `WaitForPendingFinalizers()`, it's almost as if it ran on the main thread.)

Share Improve this answer Follow

answered Mar 22, 2013 at 18:50

svick
244k ● 53 ● 403 ● 526