Console.In.ReadLine (redirected StdIn) freezes all other threads

Asked 12 years, 1 month ago Modified 8 years, 9 months ago Viewed 2k times

4

I'm connecting a legacy application (written in COBOL) with our .NET new ones using pipes. The idea is simple: The legacy program (my ERP Menu) writes some parameters upon the stream, the .NET app reads it thru `Console.In` stream and starts a new thread opening the screen requested. Here's a snippet from the .NET side of how the idea works:

```
<STAThread(), LoaderOptimization(LoaderOptimization.MultiDomain)>
Shared Sub Main()
    If Environment.GetCommandLineArgs(1) = "PIPE"
       While True
          Dim pipeParam as String
          Try
             pipeParam = Console.In.ReadLine()
          Catch e as Exception
             Exit Sub
          End Try

          ' Deal with parameters here

          If pipeParam = "END"
             Dim newThread as New Threading.Thread(Sub()
                                                      ' Create a new AppDomain
and Loads the menu option, generally a Winforms form.
                                                   End Sub)
             newThread.Start()
          End If
       End While

    End If
End Sub
```

Everything was working fine and easy... until today. I deployed this solution in my client environment (Windows Server 2003), and it happened that none of the threads requested were being executed, except when the called process (COBOL) was being terminated (that is, the `Console.In` was being forcedly closed). From then on, all the requested winforms will start showing up and behaving as expected.

Digging this strange behaviour with logs, I found out that the threads were being normally executed until a point that a `IDictionary.ContainsKey()` statement was performed (or some other method that requires native code execution). At this point, the thread was freezing / sleeping.

If I limit the thread creation to, let say, three, it happens that every created thread hangs until the third one, that is, when `Console.In.ReadLine` is not executed anymore.

What should I try? Any advices?

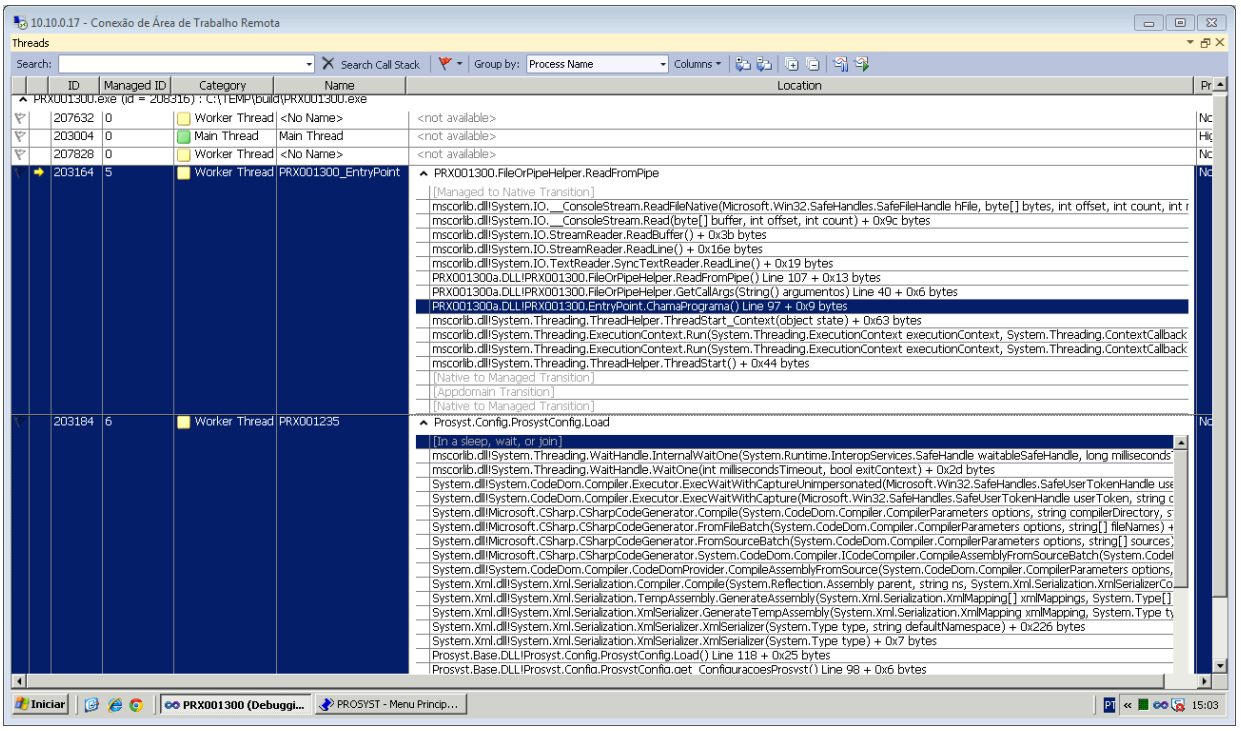
Some more info: The closest direction I've found so far was the Hans Passant's answer in this question: Interface freezes in multi-threaded c# application (the .NET SystemEvents happens to appear in my debbuger thread list, but I couldn't solve my problem with the proposed solution).

UPDATED NEWS

I could solve this issue by waiting the sub-thread to finish load the `Form`. This "Is Ready" signal is passed through `AppDomain.SetData()` and `AppDomain.GetData()`. Somehow, after the form creation the sub-thread doesn't freezes anymore when the main one goes on `Console.ReadLine`. Though the problem is solved, I'm intrigued with this. I'm trying to reproduce this in a "simple as possible" test case.

Some More Details

- The entry-point .exe is compiled to 32-bit. All other libraries are 'AnyCpu'. The issue happens running on both 32-bits (my client) and 64-bits (my development) machines (both windows Server 2003).
- Updated `Sub Main()` attributes in the above snippet.
- Tried to put the `Console.ReadLine` in a worker thread. Didn't solved (see image below).
- The COBOL application won't freeze, because it is executed in a separate OS Process. The pipe happens to be my IPC approach. The COBOL application in this case only writes parameters and don't wait for response.
- The stack trace is in the image below (the thread `PRX001235` is deserializing an xml config file before connecting to the database and before effectively loading the form - in this case, seems to be still in managed code - sometimes the `PRX001235` thread will freeze in native code when trying to connect to the database):



.net vb.net multithreading pipe

Share

Improve this question

Follow

edited May 23, 2017 at 10:28

Community Bot

1 • 1

asked Nov 18, 2012 at 3:23

J.Hudler

1,258 • 9 • 28

Not as an solution, but a workaround: can your COBOL app write to a txt file, which you can read from your .NET app? This should avoid the freezing, but maybe slower than the Commandline Solution. – Christian Sauer Nov 20, 2012 at 10:27

Hi @ChristianSauer. I'm really considering this solution you suggested. The main reason I'm insisting with pipes, is that I can terminate the slave application (.NET) when the stream is closed (that is, the COBOL application is also terminated). With text files I lose this "tracking" benefit. – J.Hudler Nov 20, 2012 at 10:52

I don't know what the problem is, but could you try changing to an extra thread that just handles the `While ... ReadLine` loop, with IPC to retrieve the parameters etc.? It would be interesting to see what hangs in this scenario. – Mark Hurd Nov 20, 2012 at 14:05

Also in your `' Deal with parameters here` and `' Start new thread. Generally showing a Winforms form.` I presume you have variables that are "lifted" into the closure for the `Sub()`. Is there any chance you have a race condition or a deadlock in locks you've added to avoid them? – Mark Hurd Nov 20, 2012 at 14:27

1 @ChristianSauer: And as I understand it, it is not the OP's problem; in fact I believe that part is working fine. The problem is `ReadLine` is blocking within the Win32API in such a way as all Managed threads are stuck, not just the one that should be blocking. – Mark Hurd Nov 21, 2012 at 0:59

3 Answers

Sorted by: Highest score (default)

4

First off, you are doing something *very* unusual, so unusual outcomes are not unexpected. Secondly, what you are doing is strictly *verboten* in the Windows SDK docs. Which dictates that a thread that created a single-threaded apartment is never allowed to make a blocking call.

Clearly your worker threads are blocked on a lock somewhere inside the bowels of the operating system, a lock taken by the main thread of your program. It would help a lot to see the call stack of one of those blocked threads to identify the possible lock. That requires enabling unmanaged code debugging so you can see the unmanaged stack frames and enabling the Microsoft Symbol Server so you get debugging symbols for the Windows code and can make sense out of the stack trace.

Not having one to look at, I'll have to speculate:

- Console.ReadLine() itself takes an internal lock that makes the console safe to use from multiple threads, serializing calls to Read(). Any thread that uses a Console method can hit that same lock. This is not likely the culprit, unlikely that these worker threads are also using the console.
- The strictly *verboten* angle is associated with the relevance of apartment threading, a COM feature. An STA is enabled by the [STAThread] attribute on your program's Main() method, or a Thread.SetApartmentState() call. An STA is required to use components that are fundamentally thread-unsafe, like windows, the clipboard, drag and drop, the shell dialogs like OpenFileDialog and many other COM coclasses, some of which you might not recognize since they were wrapped by .NET classes. An STA apartment ensures that these components are used in a thread-safe manner, a call to a method of such a component from a worker thread is automatically marshaled to the thread that created it. The exact equivalent to Control.Invoke(). Such marshaling requires that the thread pumps a message loop to dispatch the call on the right thread. And your main thread is **not** pumping when it is blocked on the Console.ReadLine() call. The worker thread will be stalled on the call until the main thread starts pumping again. Very high odds for deadlock, albeit that you don't actually deadlock completely because the ReadLine() call ultimately completes. Notable is that the CLR avoids these kind of deadlocks, it pumps a message loop when you use the *lock* keyword, call WaitOne() on a synchronization object or call Thread.Join(), the common kind of blocking calls in .NET programming. It however doesn't do this for Console.ReadLine(), at least until the .NET 4.0 workaround as shown by Mark.
- A highly speculative one, triggered by your observation that you avoid the problem by waiting for the form to be created. You might have this problem on a 64-bit operating system and your EXE project has the Platform target setting set

to "x86" instead of "AnyCPU". In that case, your program runs in the WOW64 emulation layer, the layer that allows a 64-bit operating system to execute 32-bit programs. The Windows window manager is a 64-bit sub-system, calls it makes to send notifications to a 32-bit window go through a thunk that switches bit-ness. There's a problem with the Form.Load event, triggered when the window manager delivers the WM_SHOWWINDOW message. It puts the emulation layer in a difficult state because that travels the 64-bit to 32-bit boundary several times. This is also the source of a very nasty exception swallowing problem, documented in this answer. The odds are very high that this code holds an emulator lock at the time the Load event fires. So calling Console.ReadLine() in the Load event is likely to be very troublesome, I'd expect this lock to be passed by any 32-bit worker thread as well when they make an api call. Highly speculative, but easy to test if you can change the Platform target to AnyCPU.

Not so sure if it is worth chasing the cause of this problem, given that the solution is so simple. Just don't call Console.ReadLine() on your main thread. Call it on a worker thread instead. This also stops your UI from freezing when the COBOL program is unresponsive. Do note however that you now must marshal yourself with Control.Begin/Invoke() if whatever you receive also updates your UI.

Share

Improve this answer

Follow

edited May 23, 2017 at 10:28

Community Bot

1 • 1

answered Nov 22, 2012 at 15:49

Hans Passant

940k • 148 • 1.7k • 2.6k

Thank you for such a detailed and thorough answer. To reduce your speculative assumptions, I'm adding more details in my question. I'm also going to try to put the native stack trace here as soon as possible. – J.Hudler Nov 22, 2012 at 16:50

I'll stick with the 3rd bullet. Use sgen.exe to pre-compile XML serializers. – Hans Passant Nov 23, 2012 at 16:54

Set as right answer for the educational explanation - this made me meditate about my code design and clarified what the issue was. – J.Hudler Nov 26, 2012 at 10:57

2

Comparing .NET 3.5/2.0 and .NET 4.0 with Reflector: .NET 4.0 always explicitly calls `__ConsoleStream.WaitForAvailableConsoleInput(hFile)` in `__ConsoleStream.ReadFileNative` and I can find no equivalent call in .NET 3.5/2.0.

The `WaitForAvailableConsoleInput` `InternalCall` checks for a pipe, and if so avoids waiting on it if it's got data available or if it's closed.

+50

Just summarising what I understand is the current state of this issue: It can be worked around by ensuring the other threads (AppDomains) are pumping messages before allowing the main thread to continue to the `ReadLine` call.

I think that can be a (nearly) final answer, as it means there are Windows messages going on during the server's version of `ReadLine` and that's enough for me to know the pumping is needed. This comes down to Windows documentation needing to mention where Windows messaging is needed/used.

Share

Improve this answer

Follow

edited Mar 14, 2016 at 4:29

answered Nov 21, 2012 at 2:19

Mark Hurd

10.9k ● 10 ● 73 ● 105

Just tried to switch to 3.5 the two involved libraries, but the error list turned to be not that small. I'm also trying to port this situation to a test case and paste here, but there may be some "hidden" operations involved that make this to happen - simple code as the snippet doesn't make it happen. (anyways, migrating to 3.5 is not an option for me). – J.Hudler Nov 22, 2012 at 10:53

FYI I saw the same thing when wanting to allow Clipboard interaction from the DotLisp REPL connected to a normal Console window (in WinXP as well as Win2k3). I have a background thread that continually calls `(System.Threading.Thread:CurrentThread.Join 0)` `(System.Windows.Forms.Application:DoEvents)` `(System.GC:WaitForPendingFinalizers) (System.Threading.Thread:Sleep 55)`. – Mark Hurd Nov 23, 2012 at 0:54

As now I got the moral of the story, I'm awarding you the bounties for your assistance from the beginning. – J.Hudler Nov 26, 2012 at 10:51

0

have you tried setting these threads to background threads by setting Thread.IsBackground = true

Share Improve this answer Follow

answered Nov 21, 2012 at 10:34

Ibrahim

351 ● 4 ● 15

1 Just tried your approach, but no success. Also, a cannot have the sub-thread as background because I want the opened forms to keep running even if the Menu (main thread) is closed. – J.Hudler Nov 22, 2012 at 11:03

as I understood your app is a console app and runs forms, if I am correct, you may consider running these forms out of process .. instead of threads .. to do this you develop your forms as its own application and find a way to pass the parameters to your form, pipe or socket ... etc. I know this will change your design a bit .. but I assume it is a problem with .Net being a COM object, not that only I believe the .Net thread is not like c++ CreateThread .. I am not sure of this but you might dig it up – Ibrahim Nov 24, 2012 at 15:38