Kill child process when parent process is killed

Asked 14 years, 5 months ago Modified 3 years, 6 months ago

Viewed 119k times

187

I'm creating new processes using `System.Diagnostics.Process` class from my application.

I want this processes to be killed when/if my application has crashed. But if I kill my application from Task Manager, child processes are not killed.

Is there any way to make child processes dependent on parent process?

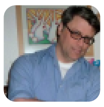
c# .net process

Share

Improve this question

Follow

edited Jan 18, 2020 at 20:50



M Katz

5,406 ●4 ●47 ●71

asked Jul 27, 2010 at 11:06

SiberianGuy

25.3k ●57 ●159 ●272

16 Answers

Sorted by: Highest score (default)

199

From this forum, credit to 'Josh'.

`Application.Quit()` and `Process.Kill()` are possible solutions, but have proven to be unreliable. When your main application dies, you are still left with child processes running. What we really want is for the child processes to die as soon as the main process dies.

The solution is to use "job objects" http://msdn.microsoft.com/en-us/library/ms682409(VS.85).aspx.

The idea is to create a "job object" for your main application, and register your child processes with the job object. If the main process dies, the OS will take care of terminating the child processes.

```
public enum JobObjectInfoType
{
    AssociateCompletionPortInformation = 7,
    BasicLimitInformation = 2,
    BasicUIRestrictions = 4,
    EndOfJobTimeInformation = 6,
    ExtendedLimitInformation = 9,
    SecurityLimitInformation = 5,
    GroupInformation = 11
}

[StructLayout(LayoutKind.Sequential)]
public struct SECURITY_ATTRIBUTES
{
    public int nLength;
    public IntPtr lpSecurityDescriptor;
    public int bInheritHandle;
```

```
}

[StructLayout(LayoutKind.Sequential)]
struct JOBOBJECT_BASIC_LIMIT_INFORMATION
{
    public Int64 PerProcessUserTimeLimit;
    public Int64 PerJobUserTimeLimit;
    public Int16 LimitFlags;
    public UInt32 MinimumWorkingSetSize;
    public UInt32 MaximumWorkingSetSize;
    public Int16 ActiveProcessLimit;
    public Int64 Affinity;
    public Int16 PriorityClass;
    public Int16 SchedulingClass;
}

[StructLayout(LayoutKind.Sequential)]
struct IO_COUNTERS
{
    public UInt64 ReadOperationCount;
    public UInt64 WriteOperationCount;
    public UInt64 OtherOperationCount;
    public UInt64 ReadTransferCount;
    public UInt64 WriteTransferCount;
    public UInt64 OtherTransferCount;
}

[StructLayout(LayoutKind.Sequential)]
struct JOBOBJECT_EXTENDED_LIMIT_INFORMATION
{
    public JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimi
    public IO_COUNTERS IoInfo;
    public UInt32 ProcessMemoryLimit;
    public UInt32 JobMemoryLimit;
    public UInt32 PeakProcessMemoryUsed;
    public UInt32 PeakJobMemoryUsed;
}

public class Job : IDisposable
{
    [DllImport("kernel32.dll", CharSet = CharSet.Unico
    static extern IntPtr CreateJobObject(object a, str

    [DllImport("kernel32.dll")]
    static extern bool SetInformationJobObject(IntPtr
infoType, IntPtr lpJobObjectInfo, uint cbJobObjectInfo

    [DllImport("kernel32.dll", SetLastError = true)]
    static extern bool AssignProcessToJobObject(IntPtr

    private IntPtr m_handle;
    private bool m_disposed = false;

    public Job()
    {
        m_handle = CreateJobObject(null, null);

        JOBOBJECT_BASIC_LIMIT_INFORMATION info = new
JOBOBJECT_BASIC_LIMIT_INFORMATION();
        info.LimitFlags = 0x2000;

        JOBOBJECT_EXTENDED_LIMIT_INFORMATION extendedI
JOBOBJECT_EXTENDED_LIMIT_INFORMATION();
        extendedInfo.BasicLimitInformation = info;

        int length =
Marshal.SizeOf(typeof(JOBOBJECT_EXTENDED_LIMIT_INFORMA
        IntPtr extendedInfoPtr = Marshal.AllocHGlobal(
        Marshal.StructureToPtr(extendedInfo, extendedI

        if (!SetInformationJobObject(m_handle,
JobObjectInfoType.ExtendedLimitInformation, extendedIn
            throw new Exception(string.Format("Unable
Error: {0}", Marshal.GetLastWin32Error()));
    }

    #region IDisposable Members

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    #endregion

    private void Dispose(bool disposing)
    {
        if (m_disposed)
            return;

        if (disposing) {}

        Close();
        m_disposed = true;
    }

    public void Close()
    {
        Win32.CloseHandle(m_handle);
        m_handle = IntPtr.Zero;
    }

    public bool AddProcess(IntPtr handle)
    {
        return AssignProcessToJobObject(m_handle, hand
    }

}
```

Looking at the constructor ...

```
JOBOBJECT_BASIC_LIMIT_INFORMATION info = new
JOBOBJECT_BASIC_LIMIT_INFORMATION();
info.LimitFlags = 0x2000;
```

The key here is to setup the job object properly. In the constructor I'm setting the "limits" to 0x2000, which is the numeric value for `JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE`.

MSDN defines this flag as:

> Causes all processes associated with the job to terminate when the last handle to the job is

> closed.

Once this class is setup...you just have to register each child process with the job. For example:

```
[DllImport("user32.dll", SetLastError = true)]
public static extern uint GetWindowThreadProcessId(Int
lpdwProcessId);

Excel.Application app = new Excel.ApplicationClass();

uint pid = 0;
Win32.GetWindowThreadProcessId(new IntPtr(app.Hwnd), o
 job.AddProcess(Process.GetProcessById((int)pid).Handl
```

Share Improve this answer Follow

edited Nov 4, 2013 at 11:37

Mariusz Jamro
31.6k ● 25 ● 125 ● 167

answered Jan 11, 2011 at 12:02

Matt Howells
41.3k ● 20 ● 85 ● 104

5 I would add a link to CloseHandle – Austin Salonen Mar 16, 2011 at 20:34

7 Unfortunately, I was unable to run this in 64bit mode. Here I posted a working example, which is based on this.
– Alexander Yezutov Feb 6, 2012 at 17:55

6 For 64 bit mode apps -> stackoverflow.com/a/5976162 For Vista/Win7 issue -> social.msdn.microsoft.com/forums/en-US/windowssecurity/thread/… – hB0 Aug 23, 2012 at 9:46

4 Ok, MSDN says: The JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE flag requires

use of a JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure. – SerG Mar 19, 2014 at 10:17

3 I fixed the compilation error and the memory leak. I also replaced all the p/invoke signatures and native structs with a simple package reference to `PInvoke.Win32`. You can see the final result here: gist.github.com/AArnott/2609636d2f2369495abe76e8a01446a4 – Andrew Arnott Jul 13, 2018 at 16:37

95

This answer started with @Matt Howells' excellent answer plus others (see links in the code below). Improvements:

- Supports 32-bit and 64-bit.
- Fixes some problems in @Matt Howells' answer:
  1. The small memory leak of `extendedInfoPtr`
  2. The 'Win32' compile error, and
  3. A stack-unbalanced exception I got in the call to `CreateJobObject` (using Windows 10, Visual Studio 2015, 32-bit).
- Names the Job, so if you use SysInternals, for example, you can easily find it.
- Has a somewhat simpler API and less code.

Here's how to use this code:

```
// Get a Process object somehow.
Process process = Process.Start(exePath, args);
```

```
// Add the Process to ChildProcessTracker.
ChildProcessTracker.AddProcess(process);
```

To support Windows 7 requires:

- A simple app.manifest change as @adam smith describes.
- Registry settings to be added if you are using Visual Studio.

In my case, I didn't need to support Windows 7, so I have a simple check at the top of the static constructor below.

```
/// <summary>
/// Allows processes to be automatically killed if thi
unexpectedly quits.
/// This feature requires Windows 8 or greater. On Win
</summary>
/// <remarks>References:
///  https://stackoverflow.com/a/4657392/386091
///  https://stackoverflow.com/a/9164742/386091 </rema
public static class ChildProcessTracker
{
    /// <summary>
    /// Add the process to be tracked. If our current
child processes
    /// that we are tracking will be automatically kil
process terminates
    /// first, that's fine, too.</summary>
    /// <param name="process"></param>
    public static void AddProcess(Process process)
    {
        if (s_jobHandle != IntPtr.Zero)
        {
            bool success = AssignProcessToJobObject(s_
process.Handle);
            if (!success && !process.HasExited)
                throw new Win32Exception();
        }
    }

    static ChildProcessTracker()
    {
        // This feature requires Windows 8 or later. T
requires
        //  registry settings to be added if you are u
an
        //  app.manifest change.
        //  https://stackoverflow.com/a/4232259/386091
        //  https://stackoverflow.com/a/9507862/386091
        if (Environment.OSVersion.Version < new Versio
            return;

        // The job name is optional (and can be null)
diagnostics.
        //  If it's not null, it has to be unique. Use
command-line
        //  utility: handle -a ChildProcessTracker
        string jobName = "ChildProcessTracker" +
Process.GetCurrentProcess().Id;
        s_jobHandle = CreateJobObject(IntPtr.Zero, job

        var info = new JOBOBJECT_BASIC_LIMIT_INFORMATI

        // This is the key flag. When our process is k
automatically
        //  close the job handle, and when that happen
processes to
        //  be killed, too.
        info.LimitFlags = JOBOBJECTLIMIT.JOB_OBJECT_LI

        var extendedInfo = new JOBOBJECT_EXTENDED_LIMI
        extendedInfo.BasicLimitInformation = info;

        int length =
Marshal.SizeOf(typeof(JOBOBJECT_EXTENDED_LIMIT_INFORMA
        IntPtr extendedInfoPtr = Marshal.AllocHGlobal(
        try
        {
            Marshal.StructureToPtr(extendedInfo, exten

            if (!SetInformationJobObject(s_jobHandle,
JobObjectInfoType.ExtendedLimitInformation,
                extendedInfoPtr, (uint)length))
            {
                throw new Win32Exception();
            }
        }
        finally
        {
            Marshal.FreeHGlobal(extendedInfoPtr);
        }
    }

    [DllImport("kernel32.dll", CharSet = CharSet.Unico
    static extern IntPtr CreateJobObject(IntPtr lpJobA

    [DllImport("kernel32.dll")]
    static extern bool SetInformationJobObject(IntPtr
infoType,
        IntPtr lpJobObjectInfo, uint cbJobObjectInfoLe

    [DllImport("kernel32.dll", SetLastError = true)]
    static extern bool AssignProcessToJobObject(IntPtr

    // Windows will automatically close any open job h
terminates.
    //  This can be verified by using SysInternals' Ha
job handle
    //  is closed, the child processes will be killed.
    private static readonly IntPtr s_jobHandle;
}

public enum JobObjectInfoType
{
    AssociateCompletionPortInformation = 7,
    BasicLimitInformation = 2,
    BasicUIRestrictions = 4,
    EndOfJobTimeInformation = 6,
    ExtendedLimitInformation = 9,
    SecurityLimitInformation = 5,
    GroupInformation = 11
}

[StructLayout(LayoutKind.Sequential)]
public struct JOBOBJECT_BASIC_LIMIT_INFORMATION
{
    public Int64 PerProcessUserTimeLimit;
    public Int64 PerJobUserTimeLimit;
    public JOBOBJECTLIMIT LimitFlags;
    public UIntPtr MinimumWorkingSetSize;
    public UIntPtr MaximumWorkingSetSize;
    public UInt32 ActiveProcessLimit;
    public Int64 Affinity;
    public UInt32 PriorityClass;
    public UInt32 SchedulingClass;
}

[Flags]
public enum JOBOBJECTLIMIT : uint
{
    JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE = 0x2000
}

[StructLayout(LayoutKind.Sequential)]
public struct IO_COUNTERS
{
    public UInt64 ReadOperationCount;
    public UInt64 WriteOperationCount;
    public UInt64 OtherOperationCount;
    public UInt64 ReadTransferCount;
    public UInt64 WriteTransferCount;
    public UInt64 OtherTransferCount;
}

[StructLayout(LayoutKind.Sequential)]
public struct JOBOBJECT_EXTENDED_LIMIT_INFORMATION
{
    public JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimi
    public IO_COUNTERS IoInfo;
    public UIntPtr ProcessMemoryLimit;
    public UIntPtr JobMemoryLimit;
    public UIntPtr PeakProcessMemoryUsed;
    public UIntPtr PeakJobMemoryUsed;
}
```

I carefully tested both the 32-bit and 64-bit versions of the structs by programmatically comparing the managed and native versions to each other (the overall size as well as the offsets for each member).

I've tested this code on Windows 7, 8, and 10.

Share Improve this answer Follow

edited Jul 9, 2018 at 15:58

answered May 4, 2016 at 18:07

Ron
1,988 ● 22 ● 25

---

what about closing job handle ?? – Frank Q. Nov 2, 2016 at 23:18

@FrankQ. It's important to let Windows close s_jobHandle for us when our process terminates, because our process may terminate unexpectedly (like by crashing or if the user uses Task Manager). See my comment on s_jobHandle's. – Ron Nov 10, 2016 at 5:33

It is working for me. May I ask what's your opinion on using the JOB_OBJECT_LIMIT_BREAKAWAY_OK flag? learn.microsoft.com/en-us/windows/desktop/api/winnt/… – Yiping Oct 22, 2018 at 6:06

---

1 @yiping It sounds like CREATE_BREAKAWAY_FROM_JOB allows your child process to spawn a process that can outlive your original process. That's a different requirement than what the OP asked for. Instead, if you can make your original process spawn that long-lived process (and not use ChildProcessTracker) that would be simpler. – Ron Oct 23, 2018 at 18:20

@Ron Can you add an overload, that accepts just the process handle and not the whole process? – Jannik Apr 9, 2019 at 8:40

48

This post is intended as an extension to @Matt Howells' answer, specifically for those who run into problems with using Job Objects under **Vista or Win7**, especially if you get an access denied error ('5') when calling AssignProcessToJobObject.

tl;dr

To ensure compatibility with Vista and Win7, add the following manifest to the .NET parent process:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
  <v3:trustInfo xmlns:v3="urn:schemas-microsoft-com:as
    <v3:security>
      <v3:requestedPrivileges>
        <v3:requestedExecutionLevel level="asInvoker"
      </v3:requestedPrivileges>
    </v3:security>
  </v3:trustInfo>
  <compatibility xmlns="urn:schemas-microsoft-com:comp
    <!-- We specify these, in addition to the UAC abov
Compatibility Assistant in Vista and Win7 -->
    <!-- We try to avoid PCA so we can use Windows Job
    <!-- See https://stackoverflow.com/questions/33429
when-parent-process-is-killed -->

    <application>
      <!--The ID below indicates application support f
      <supportedOS Id="{e2011457-1546-43c5-a5fe-008dee
      <!--The ID below indicates application support f
      <supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a24402
    </application>
  </compatibility>
</assembly>
```

Note that when you add new manifest in Visual Studio 2012 it will contain the above snippet already so you do not need to copy it from hear. It will also include a node for Windows 8.

full explanation

Your job association will fail with an access denied error if the process you're starting is already associated with another job. Enter Program Compatibility Assistant, which, starting in Windows Vista, will assign all kinds of processes to its own jobs.

In Vista you can mark your application to be excluded from PCA by simply including an application manifest. Visual Studio seems to do this for .NET apps automatically, so you're fine there.

A simple manifest no longer cuts it in Win7. [1] There, you have to specifically specify that you're compatible with Win7 with the tag in your manifest. [2]

This led me to worry about Windows 8. Will I have to change my manifest once again? Apparently there's a break in the clouds, as Windows 8 now allows a process to belong to multiple jobs. [3] So I haven't tested it yet, but I imagine that this madness will be over now if you simply include a manifest with the supportedOS information.

Tip 1: If you're developing a .NET app with Visual Studio, as I was, here [4] are some nice instructions on how to customize your application manifest.

Tip 2: Be careful with launching your application from Visual Studio. I found that, after adding the appropriate manifest, I still had problems with PCA when launching from Visual Studio, even if I used Start without Debugging. Launching my application from Explorer worked, however. After manually adding devenv for exclusion from PCA using the registry, starting applications that used Job Objects from VS started working as well. [5]

Tip 3: If you ever want to know if PCA is your problem, try launching your application from the command line, or copy the program to a network drive and run it from there. PCA is automatically disabled in those contexts.

[1] http://blogs.msdn.com/b/cjacks/archive/2009/06/18/pca-changes-for-windows-7-how-to-tell-us-you-are-not-an-installer-take-2-because-we-changed-the-rules-on-you.aspx

[2] http://ayende.com/blog/4360/how-to-opt-out-of-program-compatibility-assistant

[3] http://msdn.microsoft.com/en-us/library/windows/desktop/ms681949(v=vs.85).aspx: "A process can be associated with more than one job in Windows 8"

[4] How can I embed an application manifest into an application using VS2008?

[5] How to stop the Visual Studio debugger starting my process in a job object?

Share Improve this answer Follow

edited May 23, 2017 at 12:26
Community Bot
1 ● 1

answered Feb 29, 2012 at 22:45
adam smith
764 ● 8 ● 16

3 These are great additions to the topic, thank you! I made use of every aspect of this answer, including the links.
– Johnny Kauffman Aug 11, 2014 at 17:51

17

Here's an alternative that may work for some when you have control of the code the child process runs. The benefit of this approach is it doesn't require any native Windows calls.

The basic idea is to redirect the child's standard input to a stream whose other end is connected to the parent, and use that stream to detect when the parent has gone away. When you use `System.Diagnostics.Process` to start the child, it's easy to ensure its standard input is redirected:

```
Process childProcess = new Process();
childProcess.StartInfo = new ProcessStartInfo("pathToC
childProcess.StartInfo.RedirectStandardInput = true;
childProcess.StartInfo.CreateNoWindow = true; // no se
black console window which the user can't input into
```

And then, on the child process, take advantage of the fact that `Read`s from the standard input stream will always return with at least 1 byte until the stream is closed, when they will start returning 0 bytes. An outline of the way I ended up doing this is below; my way also uses a message pump to keep the main thread available for things other than watching standard in, but this general approach could be used without message pumps too.

```
using System;
using System.IO;
using System.Threading;
using System.Windows.Forms;

static int Main()
{
    Application.Run(new MyApplicationContext());
    return 0;
}

public class MyApplicationContext : ApplicationContext
{
    private SynchronizationContext _mainThreadMessageQ
    private Stream _stdInput;

    public MyApplicationContext()
    {
        _stdInput = Console.OpenStandardInput();

        // feel free to use a better way to post to th
if you know one ;)
        System.Windows.Forms.Timer handoffToMessageLoo
System.Windows.Forms.Timer();
        handoffToMessageLoopTimer.Interval = 1;
        handoffToMessageLoopTimer.Tick += new EventHan
PostMessageLoopInitialization(handoffToMessageLoopTime
        handoffToMessageLoopTimer.Start();
    }

    private void PostMessageLoopInitialization(System.
    {
        if (_mainThreadMessageQueue == null)
        {
            t.Stop();
            _mainThreadMessageQueue = SynchronizationC
        }

        // constantly monitor standard input on a back
        // signal the main thread when stuff happens.
        BeginMonitoringStdIn(null);

        // start up your application's real work here
    }

    private void BeginMonitoringStdIn(object state)
    {
        if (SynchronizationContext.Current == _mainThr
        {
            // we're already running on the main threa
            var buffer = new byte[128];

            _stdInput.BeginRead(buffer, 0, buffer.Leng
                {
                    int amtRead = _stdInput.EndRead(as

                    if (amtRead == 0)
                    {
                        _mainThreadMessageQueue.Post(n
SendOrPostCallback(ApplicationTeardown), null);
                    }
                    else
                    {
                        BeginMonitoringStdIn(null);
                    }
                }, null);
        }
        else
        {
            // not invoked from the main thread - disp
method on the main thread and return
```

```
                _mainThreadMessageQueue.Post(new
SendOrPostCallback(BeginMonitoringStdIn), null);
            }
        }

        private void ApplicationTeardown(object state)
        {
            // tear down your application gracefully here
            _stdInput.Close();

            this.ExitThread();
        }
}
```

Caveats to this approach:

1. the actual child .exe that is launched must be a console application so it remains attached to stdin/out/err. As in the above example, I easily adapted my existing application that used a message pump (but didn't show a GUI) by just creating a tiny console project that referenced the existing project, instantiating my application context and calling `Application.Run()` inside the `Main` method of the console .exe.
2. Technically, this merely *signals* the child process when the parent exits, so it will work whether the parent process exited normally or crashed, but its still up to the child processes to perform its own shutdown. This may or may not be what you want...

Share Improve this answer Follow

answered Nov 28, 2012 at 21:46

mbaynton
431 ● 4 ● 4

15

There is another relevant method, easy and effective, to finish child processes on program termination. You can implement and **attach a debugger** to them from the parent; when the parent process ends, child processes will be killed by the OS. It can go both ways attaching a debugger to the parent from the child (note that you can only attach one debugger at a time). You can find more info on the subject here.

Here you have an utility class that launches a new process and attaches a debugger to it. It has been adapted from this post by Roger Knapp. The only requirement is that both processes need to share the same bitness. You cannot debug a 32bit process from a 64bit process or vice versa.

```
public class ProcessRunner
{
    // see http://csharptest.net/1051/managed-anti-deb
users-from-attaching-a-debugger/
    // see https://stackoverflow.com/a/24012744/298275

    public Process ChildProcess { get; set; }

    public bool StartProcess(string fileName)
    {
        var processStartInfo = new ProcessStartInfo(fi
        {
            UseShellExecute = false,
            WindowStyle = ProcessWindowStyle.Normal,
            ErrorDialog = false
        };

        this.ChildProcess = Process.Start(processStart
        if (ChildProcess == null)
```

```
                return false;

            new Thread(NullDebugger) {IsBackground = true}
            return true;
        }

        private void NullDebugger(object arg)
        {
            // Attach to the process we provided the threa
            if (DebugActiveProcess((int) arg))
            {
                var debugEvent = new DEBUG_EVENT {bytes =
                while (!this.ChildProcess.HasExited)
                {
                    if (WaitForDebugEvent(out debugEvent,
                    {
                        // return DBG_CONTINUE for all eve
type
                        var continueFlag = DBG_CONTINUE;
                        if (debugEvent.dwDebugEventCode ==
DebugEventType.EXCEPTION_DEBUG_EVENT)
                            continueFlag = DBG_EXCEPTION_N
                        ContinueDebugEvent(debugEvent.dwPr
debugEvent.dwThreadId, continueFlag);
                    }
                }
            }
            else
            {
                //we were not able to attach the debugger
                //do the processes have the same bitness?
                //throw ApplicationException("Unable to at
child? // Send Event? // Ignore?
            }
        }

        #region "API imports"

        private const int DBG_CONTINUE = 0x00010002;
        private const int DBG_EXCEPTION_NOT_HANDLED = unch

        private enum DebugEventType : int
        {
            CREATE_PROCESS_DEBUG_EVENT = 3,
            //Reports a create-process debugging event. Th
    u.CreateProcessInfo specifies a CREATE_PROCESS_DEBUG_I
            CREATE_THREAD_DEBUG_EVENT = 2,
            //Reports a create-thread debugging event. The
    specifies a CREATE_THREAD_DEBUG_INFO structure.
            EXCEPTION_DEBUG_EVENT = 1,
            //Reports an exception debugging event. The va
    specifies an EXCEPTION_DEBUG_INFO structure.
            EXIT_PROCESS_DEBUG_EVENT = 5,
            //Reports an exit-process debugging event. The
    specifies an EXIT_PROCESS_DEBUG_INFO structure.
            EXIT_THREAD_DEBUG_EVENT = 4,
            //Reports an exit-thread debugging event. The
    specifies an EXIT_THREAD_DEBUG_INFO structure.
            LOAD_DLL_DEBUG_EVENT = 6,
            //Reports a load-dynamic-link-library (DLL) de
    of u.LoadDll specifies a LOAD_DLL_DEBUG_INFO structure
            OUTPUT_DEBUG_STRING_EVENT = 8,
            //Reports an output-debugging-string debugging
    u.DebugString specifies an OUTPUT_DEBUG_STRING_INFO st
            RIP_EVENT = 9,
            //Reports a RIP-debugging event (system debugg
    u.RipInfo specifies a RIP_INFO structure.
            UNLOAD_DLL_DEBUG_EVENT = 7,
            //Reports an unload-DLL debugging event. The v
    specifies an UNLOAD_DLL_DEBUG_INFO structure.
        }

         [StructLayout(LayoutKind.Sequential)]
        private struct DEBUG_EVENT
        {
            [MarshalAs(UnmanagedType.I4)] public DebugEven
            public int dwProcessId;
            public int dwThreadId;
            [MarshalAs(UnmanagedType.ByValArray, SizeConst
    bytes;
        }

         [DllImport("Kernel32.dll", SetLastError = true)]
        private static extern bool DebugActiveProcess(int

         [DllImport("Kernel32.dll", SetLastError = true)]
        private static extern bool WaitForDebugEvent([Out]
    lpDebugEvent, int dwMilliseconds);
```

```
    [DllImport("Kernel32.dll", SetLastError = true)]
    private static extern bool ContinueDebugEvent(int
dwThreadId, int dwContinueStatus);

    [DllImport("Kernel32.dll", SetLastError = true)]
    public static extern bool IsDebuggerPresent();

    #endregion
}
```

Usage:

```
    new ProcessRunner().StartProcess("c:\\Windows\\sys
```

Share Improve this answer Follow

edited Dec 4, 2020 at 15:00

answered Jun 3, 2014 at 10:25

Marco Regueira
1,088 ● 8 ● 17

13

One way is to pass PID of parent process to the child. The child will periodically poll if the process with the specified pid exists or not. If not it will just quit.

You can also use Process.WaitForExit method in child method to be notified when the parent process ends but it might not work in case of Task Manager.

Share Improve this answer Follow

answered Jul 27, 2010 at 11:09

Giorgi
30.8k ● 13 ● 91 ● 127

1 How can I pass parent PID to child? Is there any system solution? I can't modify child process binaries. – SiberianGuy Jul 27, 2010 at 11:17

1 Well, if you can't modify child process you can not use my solution even if you pass PID to it. – Giorgi Jul 27, 2010 at 11:19

@Idsa You can pass it via the command line: `Process.Start(string fileName, string arguments)` – Steve Mar 17, 2012 at 5:29

2 Rather than poll you can hook into the Exit event on the process class. – RichardOD Sep 4, 2012 at 21:08

Tried it but parent process not always exits if it has childs alive (at least in my case Cobol->NET). Easy to check watching the process hierarchy in Sysinternals ProcessExplorer. – Ivan Ferrer Villa Oct 23, 2012 at 12:03

12

I was looking for a solution to this problem that did not require unmanaged code. I was also not able to use standard input/output redirection because it was a Windows Forms application.

My solution was to create a named pipe in the parent process and then connect the child process to the same pipe. If the parent process exits then the pipe becomes broken and the child can detect this.

Below is an example using two console applications:

Parent

```
private const string PipeName = "471450d6-70db-49dc-94

public static void Main(string[] args)
{
    Console.WriteLine("Main program running");

    using (NamedPipeServerStream pipe = new NamedPipeS
PipeDirection.Out))
    {
        Process.Start("child.exe");

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Child

```
private const string PipeName = "471450d6-70db-49dc-94
as parent

public static void Main(string[] args)
{
    Console.WriteLine("Child process running");

    using (NamedPipeClientStream pipe = new NamedPipeC
PipeName, PipeDirection.In))
    {
        pipe.Connect();
        pipe.BeginRead(new byte[1], 0, 1, PipeBrokenCa

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

private static void PipeBrokenCallback(IAsyncResult ar
{
    // the pipe was closed (parent process died), so e

    try
    {
        NamedPipeClientStream pipe = (NamedPipeClientS
        pipe.EndRead(ar);
    }
    catch (IOException) { }

    Environment.Exit(1);
}
```

Share Improve this answer

Follow

answered Jun 8, 2017 at 9:40

Alsty
847 ● 10 ● 23

6

Use **event handlers** to make hooks on a few exit scenarios:

```
var process = Process.Start("program.exe");
AppDomain.CurrentDomain.DomainUnload += (s, e) => { pr
process.WaitForExit(); };
AppDomain.CurrentDomain.ProcessExit += (s, e) => { pro
process.WaitForExit(); };
AppDomain.CurrentDomain.UnhandledException += (s, e) =
process.WaitForExit(); };
```

Share Improve this answer

Follow

answered Jul 24, 2017 at 17:24

Justin D. Harris
2,265 ● 3 ● 27 ● 38

2 So simple yet so effective. – uncommon_name May 15, 2020 at 20:48

6 These events don't seem to get called when the Parent Process is Killed from Task Manager
– Arundale Ramanathan Sep 7, 2020 at 8:23

3

Yet another addition to the abundant richness of solutions proposed so far....

The problem with many of them is that they rely upon the parent and child process to shut down in an orderly manner, which isn't always true when development is underway. I found that my child process was often being orphaned whenever I terminated the parent process in the debugger, which required me to kill the orphaned process(es) with Task Manager in order to rebuild my solution.

The solution: Pass the parent process ID in on the commandline (or even less invasive, in the environment variables) of the child process.

In the parent process, the process ID is available as:

```
Process.CurrentProcess().Id;
```

In the child process:

```
Process parentProcess = Process.GetProcessById(parentP
parentProcess.Exited += (s, e) =>
{
    // clean up what you can.
    this.Dispose();
    // maybe log an error
    ....

    // And terminate with prejudice!
    //(since something has already gone terribly wrong
    Process.GetCurrentProcess().Kill();
};
```

I am of two minds as to whether this is acceptable practice in production code. On the one hand, this should never happen. But on the other hand, it may mean the difference between restarting a process, and rebooting a production server. And what should never happen often does.

And it sure is useful while debugging orderly shutdown problems.

Share Improve this answer Follow

edited Jun 18, 2021 at 1:31

answered Jun 22, 2020 at 19:11

Robin Davies
7,797 ● 1 ● 39 ● 52

2 It's `Process.GetCurrentProcess().Id` not `Process.CurrentProcess.Id` – Mariusz Jamro May 23, 2021 at 11:43

1 You also need to enable events with `parentProcess.EnableRaisingEvents = true;` – Axiom255 May 11, 2022 at 7:03

1

I see two options:

1. If you know exactly what child process could be started and you are sure they are only started from your main process, then you could consider simply searching for them by name and kill them.
2. Iterate through all processes and kill every process that has your process as a parent (I guess you need to kill the child processes first). Here is explained how you can get the parent process id.

edited May 23, 2017 at 12:34
Community Bot
1 ● 1

answered Jul 27, 2010 at 12:19
Stefan Egli
17k ● 3 ● 55 ● 78

1

I've made a child process management library where the parent process and the child process are monitored due a bidirectional WCF pipe. If either the child process terminates or the parent process terminates each other is notified. There is also a debugger helper available which automatically attaches the VS debugger to the started child process

Project site:

http://www.crawler-lib.net/child-processes

NuGet Packages:

https://www.nuget.org/packages/ChildProcesses
https://www.nuget.org/packages/ChildProcesses.VisualStudioDebug/

Share Improve this answer Follow

answered Apr 8, 2015 at 23:15

Thomas Maierhofer
2,681 ● 18 ● 33

1

Just my 2018 version. Use it aside your Main() method.

```
    using System.Management;
    using System.Diagnostics;

    ...

    // Called when the Main Window is closed
    protected override void OnClosed(EventArgs EventAr
    {
        string query = "Select * From Win32_Process Wh
Process.GetCurrentProcess().Id;
        ManagementObjectSearcher searcher = new
ManagementObjectSearcher(query);
        ManagementObjectCollection processList = searc
        foreach (var obj in processList)
        {
            object data = obj.Properties["processid"].
            if (data != null)
            {
                // retrieve the process
                var childId = Convert.ToInt32(data);
                var childProcess = Process.GetProcessB

                // ensure the current process is still
                if (childProcess != null) childProcess
            }
        }
```

```
            Environment.Exit(0);
        }
```

Share Improve this answer

Follow

answered Oct 28, 2018 at 13:39

Laynor
1,533 ● 10 ● 21

7 I doubt this gets executed when the parent process is being **killed**. – springy76 Oct 30, 2018 at 9:11

1

Solution that worked for me:

When creating process add tag `process.EnableRaisingEvents = true;` :

```
csc = new Process();
csc.StartInfo.UseShellExecute = false;
csc.StartInfo.CreateNoWindow = true;
csc.StartInfo.FileName = Path.Combine(HLib.path_dataFo
csc.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
csc.StartInfo.ErrorDialog = false;
csc.StartInfo.RedirectStandardInput = true;
csc.StartInfo.RedirectStandardOutput = true;
csc.EnableRaisingEvents = true;
csc.Start();
```

Share Improve this answer

Follow

answered Jul 26, 2020 at 7:19

Whitebrim
394 ● 3 ● 10

call job.AddProcess better to do after start of the process:

0

```
prc.Start();
job.AddProcess(prc.Handle);
```

When calling AddProcess before the terminate, child processes are not killed. (Windows 7 SP1)

```
private void KillProcess(Process proc)
{
    var job = new Job();
    job.AddProcess(proc.Handle);
    job.Close();
}
```

Share Improve this answer Follow

answered Aug 7, 2013 at 11:52

Alexey
31 ● 2

Calling job.AddProcess after start of the process would kill the purpose of using this object. – SerG Mar 19, 2014 at 10:02

0

I had the same problem. I was creating child processes that never got killed if my main app crashed. I had to destroy manually child processes when debugging. I found that there was no need to make the children somewhat depend on parent. In my main, I added a try catch to do a CleanUp() of child processes on exit.

```
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(
        try
        {
            Application.Run(new frmMonitorSensors());
        }
        catch(Exception ex)
        {
            CleanUp();
            ErrorLogging.Add(ex.ToString());
        }
    }

    static private void CleanUp()
    {
        List<string> processesToKill = new List<string
"Process2" };
        foreach (string toKill in processesToKill)
        {
            Process[] processes = Process.GetProcesses
            foreach (Process p in processes)
            {
                p.Kill();
            }
        }
    }
```

Share Improve this answer

Follow

answered Aug 26, 2020 at 0:53

Daniel Silverwood

1

0

If you can catch situation, when your processes tree should be killed, since .NET 5.0 (.NET Core 3.0) you could use Process.Kill(bool entireProcessTree)

Share Improve this answer

Follow

answered Jan 14, 2021 at 10:24

user1234567

4,291 ● 3 ● 22 ● 27