Microsoft®

Production Debugging for .NET Framework Applications

patterns & practices
proven practices for predictable results

Version 1.0

Contents

1

Introduction to Production Debugging for .NET Framework Applications

Every developer or support engineer debugs applications at some stage in his or her career, yet debugging is often viewed as an arcane and difficult topic. While it is often difficult to determine why an application is hanging, leaking memory, or crashing, there are techniques you can apply to make the debugging process more productive and efficient. The *Production Debugging for .NET Framework Applications* guide aims to equip you with the mindset, tools, and techniques that will help you successfully identify and resolve common application debugging.

The techniques presented here are not simplified classroom exercises. They are based on the experience of both Microsoft customers and Microsoft internal development teams. These techniques represent proven, best-practice approaches to debugging a variety of issues.

This guide presents walkthroughs of three scenarios that ASP.NET applications may encounter when running in production environments: memory consumption, contention (also known as "deadlock"), and unexpected server crashes.

These scenarios concentrate on debugging Microsoft® .NET framework applications in a production environment, focusing on low-level details such as thread states and memory allocations. However, the thought processes and techniques discussed will help you understand debugging on a much broader scale. This document helps you learn how to think about debugging in a general sense, and equips you with the mindset to successfully approach unknown and unforeseen debugging situations in the future.

Although the example scenarios target ASP.NET and the Visual C#™ development tool, the thought processes and techniques discussed are common across all .NET technologies and languages, and problems encountered are common themes. The scenarios also introduce the tools that Microsoft provides for the Microsoft Windows® operating system and .NET debugging, including the use of WinDbg, Core Debugger (CDB), and debugger extension DLLs for displaying managed call stacks and object data.

This chapter provides an introduction to the Production Debugging for .NET Framework Applications guide, defines the scope of the chapters, and provides background material. It also introduces the system requirements for the walkthroughs, and gives instructions for installing the required software.

Production Debugging vs. Development Debugging

The primary goal of debugging a system is to isolate and determine the root cause of a performance, configuration, or abnormal error condition that impedes normal system operation. Generally speaking, this goal holds true whether you are debugging in a development environment or in a production environment. However, there are crucial differences between the two environments.

When debugging in a production environment, determining the root cause may be less important than simply getting the system into an operational state.

Time constraints and business cases also differ between development and production environments. Debugging within a production environment is usually done within much tighter time constraints, given that the end user or customer may be losing revenue as a consequence of a non-operational system.

The walkthroughs presented in this guide demonstrate many non-invasive techniques that can be used in a production debugging environment.

Environment

In a production environment, the system experiences live operational loads and timing situations, which may be difficult or impossible to reproduce in a development environment. In addition, scenarios can arise that have not been anticipated during system design. Timing-sensitive or load-sensitive scenarios that lead to abnormal behavior can be difficult to reproduce or debug successfully, even during live operation in a production environment.

Because physical access to a production system is often restricted, offline or remote-access techniques, such as Terminal Services sessions and remote debugging sessions, are required.

Tools

In a development environment, you usually have access to an integrated development environment (IDE), source code, and debug versions of libraries and symbols. In a production environment, the IDE and source code are often not present, and you may only have access to release versions of libraries and symbols. Note that release versions of symbols are an invaluable resource, but are often not built during a normal product development lifecycle. Symbols are not as necessary when debugging .NET code, but they are still beneficial. This is because, while the .NET metadata contains the function names and parameter types, symbols provide source file names, line numbers, and local variable names.

For more information on symbols and how they are used, see article Q121366, "PDB and DBG Files – What They Are and How They Work," in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q121366.*

The Debugging Tools for Windows toolkit contains a wealth of tools that are helpful to many debugging approaches and processes. The toolkit is small and non-intrusive and can usually be installed in a development or production environment pending end user or customer approval. The rest of this guide describes many of the tools that this toolkit contains.

You can download the latest version of the Debugging Tools for Windows toolkit from the Microsoft Web site at *http://www.microsoft.com/ddk/debugging.*

Actions

In a development environment, you often have the luxury of using a full range of techniques to debug the system. That's not the case in a production environment, where it is essential to keep the system functioning while performing debugging. A production environment requires the following low-risk actions:

- Using non-intrusive debugging techniques, such as capturing performance information, and saving hang or crash dumps.
- Restricting the use of code changes, except in the case of critical problems. This means that you can't normally experiment using code modification to isolate root causes in production environments.
- Adjusting the testing level to the user's comfort level. This may mean passing a system through a full battery of tests in a development environment, or simply asking the question, "Is the system still running?"

Debugging Tools

Debugging a problem typically involves two phases. The initial discovery phase is used to gather information about the problem. This leads to a debugging phase, when you attempt to determine the cause of the problem. These two phases usually require the use of different tools.

Discovery Phase

Most developers are familiar with the debugging phase, yet the discovery phase is equally important to production-level debugging. The goal is usually to capture the state of the server and then allow the server to continue functioning.

Two particularly useful tools that are included with Windows operating systems are:

- **Task Manager**, which enables a system administrator to obtain values for system metrics such as CPU or memory usage and virtual memory size.
- **System Monitor** (known as Performance Monitor in Windows 2000), which enables you to log the values of a number of performance counters in order to gather information for trend analysis.

Other tools, which are not included with Windows operating systems but are available as downloads, are also useful in the discovery phase:

- **Autodump+** (ADPlus) is included in Debugging Tools for Windows version 6.0 and later. It is the primary data gathering tool for post-mortem analysis. It takes the form of a Microsoft Visual Basic® Scripting Edition (VBScript) file, which instructs CDB how to gather data, such as user dumps and logs for post mortem analysis. For more information about ADPlus, see article Q286350, "HOWTO: Use AutoDump+ to Troubleshoot 'Hangs' and 'Crashes,'" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q286350.*
- **ADPlus_AspNet.vbs** is a modified version of ADPlus, and its use is explained in Chapter 3, "Debugging Contention Problems." ADPlus_AspNet.vbs can create a full dump when a deadlock occurs. It attaches to Microsoft Internet Information Services (IIS) and waits for the health monitoring thread to shut down the ASP.NET process. When it does, ADPlus_AspNet.vbs breaks in and creates a user dump of the ASP.NET worker processes and the IIS processes running them.
- **ADPlus_KernelExit.vbs** is a modified version of ADPlus, and its use is explained in Chapter 4, "Debugging Unexpected Process Termination." ADPlus_KernelExit.vbs creates a full dump when a process terminates unexpectedly.
- **Allocation Profiler** provides a way to view the allocation of managed memory in graphical form.

Debugging Phase

The debugging phase may involve live or post-mortem debugging. Live debugging involves stepping through the code as it executes and looking for bugs or defects. Post-mortem debugging involves the analysis of data or program dumps produced by tools such as ADPlus. One advantage of post-mortem debugging is that it uses data gathered from a real problem occurrence, instead of relying on a dedicated programmer to interactively step through the code and try to reproduce the problem.

There are a variety of tools available to help you gather and analyze data during the debugging process:

- **WinDbg** is a native debugger with a graphical user interface. It attaches to a live local or remote process or opens and helps you to analyze dump files during post-mortem analysis. WinDbg is part of the Debugging Tools for Windows toolkit, along with several other console-based debuggers, notably CDB (the Console Debugger) and KD (the Kernel Debugger).
- **CorDbg** is a console-based debugger that ships with the .NET Framework SDK and is used to debug managed code. It requires the installation of three DLLs: MSDis130.dll, MSvcP70.dll, and MSvcr70.dll. This tool is useful for production debugging and data gathering because of its small number of dependencies and its size.
- **SOS.dll** is a debugger extension that is capable of displaying managed call stacks and object data. It currently works only with the debuggers provided as part of the Driver Development Kit (DDK), such as WinDbg, and it requires a .bin file that matches the type and version of the .NET runtime.

> **Note:** At present, SOS only works with the DDK debuggers. It is intended, however, that the functionality provided by SOS will be supported in the upcoming version of Visual Studio .NET. Also, the .bin file will not be required to match the type and version of the .NET runtime.

- **SieExtPub.dll** is a debugger extension used to output COM threading information.
- The **Visual Studio .NET** debugger is both a managed and native debugger that you can use to debug code written in the Visual Basic .NET and Visual C++® .NET development systems, and with Visual C# .NET. It also supports debugging of mixed language solutions and scripts. The debugger supports remote debugging, although this needs several files to be deployed on the target system.

Choosing the right debugger depends on the environment (for example if Visual Studio.NET is installed on the machine) and whether you're dealing with native or managed code. Table 1.1 summarizes the capabilities of the various debuggers. Variations include allowing you to examine managed and/or native data through live debugging or through dumps.

Table 1.1: Debugger capabilities

Situation	Visual Studio .NET	CorDbg	WinDbg	WinDbg and SOS	Upcoming version of Visual Studio .NET and SOS
Native call stacks (live attach)	Yes	No	Yes	Yes	Yes
Native call stacks (post mortem)	Yes	No	Yes	Yes	Yes
Managed call stacks (live attach)	Yes	Yes	No	Yes	Yes
Managed call stacks (post mortem)	No	No	No	Yes	Yes
Native and managed call stacks (live attach)	Yes	No	No	Yes	Yes
Native and managed call stacks (post mortem)	No	No	No	Yes	Yes
Examine .NET memory	No	No	No	Yes	Yes
Trap process exit	No (need breakpoint)	No	Yes	Yes	No (need breakpoint)

Before starting the walkthroughs described in this guide, download the Debugging Tools for Windows toolkit (which includes WinDbg and ADPlus) from the Microsoft Web site at *http://www.microsoft.com/ddk/debugging/*.

ASP.NET Process Model and Health Monitoring

The scenarios described in this guide involve ASP.NET. Therefore, before discussing them in detail, you need to understand something about how ASP.NET works, and in particular, how different versions of IIS monitor the health of an ASP.NET process. This section starts with a discussion of IIS versions 5.*x*, and concludes by discussing how health monitoring works in IIS version 6.0.

What Is the ASP.NET Process Model?

The ASP.NET process model refers to the path an HTTP request takes through IIS, along with the response that is generated and returned to the client. The process model is configured by editing the machine.config .NET machine-level configuration file.

The machine.config file is located in the *Windows Directory*\Microsoft.NET \Framework*Framework Version*\Config folder.

This XML configuration file contains a <processModel> element, whose attributes specify the parameters used by the process model. For more information on the <processModel> element and its attributes, see the "ASP.NET Process Model" entry in the .NET Framework SDK documentation, which is available from the MSDN Web site at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us /cpgenref/html/gngrfprocessmodelsection.asp.*

IIS 5.x Process Model

The IIS 5.x process model controls how the ASP.NET request travels through IIS and is finally served by the Aspnet_wp.exe process. Figure 1.1 illustrates how this works.

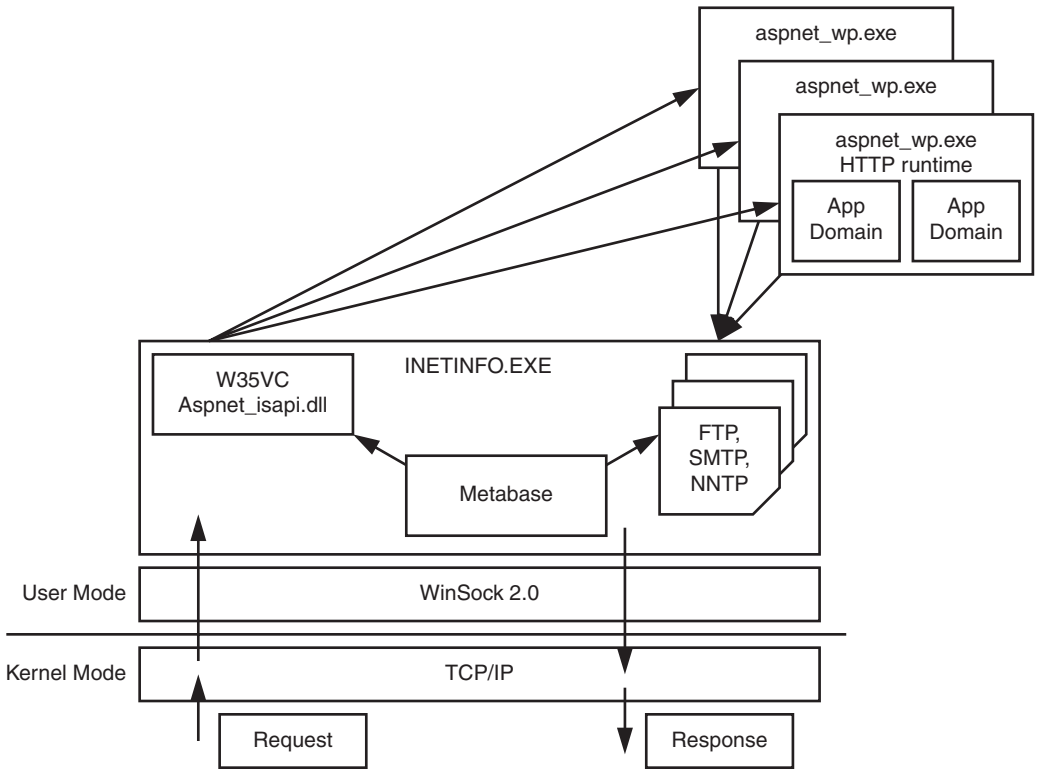

Figure 1.1
IIS 5.x process model

Both the InetInfo and Aspnet_wp executables load the Aspnet_isapi.dll. The steps below provide a more detailed look at the logic used to process an ASP.NET request.

1. IIS receives the incoming ASP.NET request and passes it to the ASP.NET ISAPI code. The request is added to the request table and is then passed to the Aspnet_wp.exe worker process by named pipes.
2. After the worker process receives the request, it sends an acknowledgement back and updates the request table to "executing." A request in the "executing" state cannot be reassigned to a different worker process if the original worker process does not execute the request to completion.
3. The worker process is now responsible for executing the request, but may call back into IIS to retrieve items, such as server variables.
4. The page response is sent asynchronously through IIS to the client. Then the request table is updated to reflect that the request is complete.

Process Model Configuration in IIS 5.*x*

When using ASP.NET with IIS 5.*x*, the aspnet_isapi.dll unmanaged DLL reads the process model settings in the machine.config file. If changes are made to the <processModel> element, IIS must be restarted for the changes to take effect. (When using IIS 6.0, the <processModel> section, by default, is not used. Instead, the Internet Services Manager UI allows you to configure applicable settings for the IIS worker process.)

Runtime Configuration

The ASP.NET HTTP runtime settings can be configured using the <httpRuntime> element, which can be defined in configuration files at the machine, site, application, and subdirectory levels. Several attributes can be used to control the runtime behavior of ASP.NET.

The following table describes the available attributes.

Table 1.2: Runtime attributes for ASP.NET

Attribute	Description
appRequestQueueLimit	The maximum number of requests that ASP.NET will queue for the application
executionTimeout	The maximum number of seconds that a request is allowed for execution
maxRequestLength	Maximum upload file size
minFreeThreads	Minimum number of threads set aside for request execution
minLocalRequestFreeThreads	Minimum number of threads set aside for execution of local requests

Health Monitoring in IIS 5.*x*

The health monitoring provided by the process model aims to provide maximum application uptime by proactively responding to possible problems before they become critical. It offers the ability to control the way in which applications behave by specifying values for parameters, such as memory usage limits and idle process timeouts.

Health monitoring occurs when the ASP.NET process model is enabled; this is controlled by the "enable" attribute of the <processModel> element in machine.config.

Every two seconds InetInfo checks the memory size, number of requests completed, and the time since the last response was received from the ASP.NET worker process. When the size of any one process reaches a certain percentage of total system memory (by default 60 percent of physical RAM, configurable through a <processModel> attribute), InetInfo suspends all requests to that Aspnet_wp.exe process, routes requests to a new worker process, and then recycles the old one. A clean restart occurs, and entries are written to the application event log (enabled by the logLevel attribute of the <processModel> element).

If process model health monitoring is enabled while debugging, the process may be recycled before you have a chance to capture an adequate picture of the problem. You can use a registry DWORD value, HKLM\Software\Microsoft\ASP.NET \UnderDebugger, to control whether the process recycles when it detects that a debugger is attached. The walkthroughs later in this guide discuss how to use this value.

IIS 6.0 Process Model in Windows .NET Server Release Candidate 1

The IIS 6.0 process model controls how the ASP.NET request is processed by IIS and then served by the worker process. Figure 1.2 on the next page illustrates how this works.

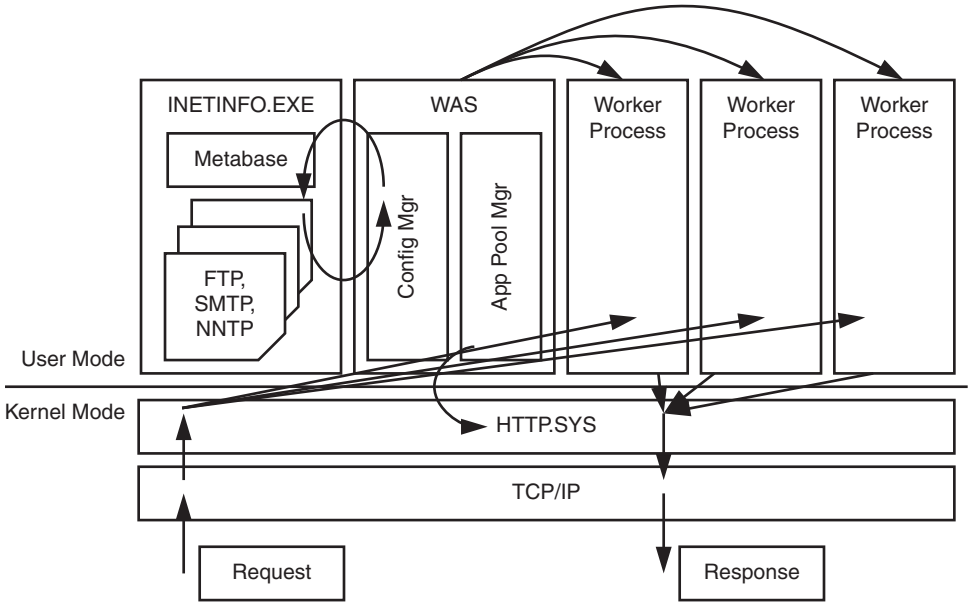

Figure 1.2
IIS 6.0 process model

With "Worker Process Isolation Mode" enabled in IIS 6.0, all configured Web applications are grouped into application pools and each pool runs in a single w3wp.exe process. In this new model, the Aspnet_Isapi.dll is loaded into the worker process serving the ASP.NET content, rather than into InetInfo.exe itself. This provides greater stability, since a failure in Aspnet_isapi.dll only impacts one worker process, rather than bringing down the entire Web server.

Requests from clients for .aspx or .asmx files are received by HTTP.sys, which is the kernel mode listener that handles all HTTP traffic for IIS 6.0. HTTP.sys then forwards requests directly to the worker process for the specific ASP.NET application. After ASP.NET has finished processing the request, the response is sent back through HTTP.sys and on to the client.

IIS 6.0 Application Pools

Proper configuration of the application pool is essential to ensure the health and reliability of an ASP.NET application. All the relevant settings are found on the application pool property sheet.

Similar to IIS 5.*x*, IIS 6.0 distinguishes between proactive and reactive recycling. IIS performs proactive recycling for known conditions and reactive recycling for unknown or dynamic conditions. For proactive recycling, IIS 6.0 checks the elapsed time, the number of requests completed, the scheduled time, and the amount of

memory used. By default, events are only logged in the application event log if the recycling occurs based on the memory limit trigger. If you want IIS to log all proactive recycling events to the application event log, run the following command from the \Inetpub\Adminscripts directory:

```
cscript adsutil.vbs set w3svc/apppools/<defaultapppool>/LogEventOnRecycle
0xffffffff
```

Note: Make sure that you change <defaultapppool> in the preceding command to the name of the application pool that you want to log proactive recycling events.

The following table describes the available IIS 6.0 application pool settings by means of the IIS MMC snap-in.

Table 1.3: Application pool settings for IIS 6.0

Setting	Description
Idle Timeout	Controls whether IIS shuts down idle worker processes. Any worker process serving the application pool is shut down after being idle for the specified amount of time.
Request Queue Limit	Controls the size of the request queue. This setting prevents large numbers of requests from queuing up and overloading the Web server.
CPU Monitoring	Specifies what action is to be taken if a CPU usage threshold is reached.
Web Garden	Controls the number of worker processes for the application pool.
Enable Pinging	Specifies how often the Web Administration Service (WAS) pings each worker process in the application pool to detect its status.
Enable Rapid Fail Protection	Configures IIS to remove the application pool from service if a specified number of crashes occurs within the specified time period. If a worker process is removed from service, HTTP.sys responds to any incoming request with a "503 Service Unavailable" message.
Startup Time Limit	Specifies the amount of time a process is allowed for startup, before IIS assumes that it has not started correctly and terminates it. Terminated processes are logged in the system event log.
Shutdown Time Limit	Tells the WAS how to cope with worker processes that hang during shutdown. If a worker process has not shut down during the specified time limit, it is terminated by the WAS.

Orphaning Failed ASP.NET Worker Processes

Web Administration Service (WAS) shuts down ASP.NET worker processes if they fail to meet established health criteria or if they fail to respond to pings. This can complicate debugging, since the failed worker process shuts down before you can attach a debugger to it.

You can reconfigure WAS to prevent the failed process from serving any more requests, while ensuring that the process continues to run; this is known as orphaning the failed process. An orphaned worker process is removed from the application pool, but left in its failed state for later debugging. In this case, the WAS starts another worker process so that the application can continue to serve requests. To configure WAS to orphan worker processes upon failure, run the following command from the \Inetpub\Adminscripts directory:

```
cscript adsutil.vbs set w3svc/apppools/orphanworkerprocess 1
```

This command applies this setting at the master level for all application pools. To apply this setting to a specific application pool only, run the command as follows:

```
cscript adsutil.vbs set w3svc/apppools/<nameofapppool>/orphanworkerprocess 1
```

Note: Replace <nameofapppool> in the preceding command with the name of the application pool to which this setting should be applied.

After WAS is configured in this way, the failing instance of W3wp.exe will not be terminated by WAS, and you will be able to hook up a debugger and investigate the cause of the problem.

System Requirements

The stack traces in this guide were created on a computer running Windows XP Professional. If you are using Windows 2000, you may find some differences in the names of function calls listed in your output.

The walkthroughs in this guide were also tested on a computer with the following configuration:

- Single processor
- 512 megabytes (MB) RAM
- Windows 2000 Advanced Server
- IIS version 5.0

Using a machine with a different configuration may lead to slightly different results, but the process to debug and analyze remains the same.

Additional Software

To perform the walkthroughs in this guide, you need to download and install the following software:

- Sample applications
- Debugging Tools for Windows
- Allocation Profiler

► To install the sample applications

1. Download DebuggingWalkthroughs.msi from *http://www.microsoft.com/downloads/release.asp?ReleaseID=44273*
2. Double-click DebuggingWalkthroughs.msi to start the .NET Debugging Walkthroughs Setup Wizard and then follow the instructions to complete the wizard.

► To build the sample applications

1. Start Visual Studio .NET, and then open **Debugging.sln** from C:\Inetpub\wwwroot\Debugging. This opens both the DebuggingCOM and DebuggingWeb applications.
2. On the **Build** menu, click **Configuration Manager**.
3. In the Configuration Manager dialog box, change the **Active Solution Configuration** to **Release**, and then click **Close**.
4. On the **Build** menu, click **Build Solution**.

► To create folders for the debuggers

- In Windows Explorer, create the following folders:
  - C:\Debuggers
  - C:\Debuggers\ZipFiles
  - C:\Debuggers\SOS
  - C:\Debuggers\AllocProf
  - C:\Symbols\DebuggingLabs

► To install the Debugging Tools for Windows toolkit

1. Download the Debugging Tools for Windows toolkit from the Microsoft Web site at *http://www.microsoft.com/ddk/debugging/*.
2. Double-click Dbg_x86_6.0.17.0.exe.
3. On the Welcome to the Debugging Tools for Windows Setup Wizard page, click **Next**.
4. On the End-User License Agreement page, if you agree with the terms and conditions, click **I agree**, and then click **Next**.

5. On the User Information page, enter your details, and then click **Next**.
6. On the Select an Installation Type page, click **Custom**, and then click **Next**.
7. On the Select an Installation Location page, change the path to **C:\Debuggers**, and then click **Next**.
8. On the Custom Installation page, click **Next**.
9. On the Begin Update page, click **Next**.
10. On the Completing the Debugging Tools for Windows Setup Wizard page, click **Finish**.

► **To install additional tools**

1. Download SOS.zip, SieExtPub.zip, and ADPlus_Scripts.zip from *http://www.microsoft.com/downloads/release.asp?ReleaseID=44274* to C:\Debuggers\ZipFiles.
2. Extract the contents of SOS.zip to C:\Debuggers\SOS.
3. Extract the contents of SieExtPub.zip to C:\Debuggers\SOS.
4. Extract the contents of ADPlus_Scripts.zip to C:\Debuggers.
5. Download the current version of the Allocation Profiler from the Microsoft .NET Framework Community Web site at *http://www.gotdotnet.com/userarea/keywordsrch.aspx?keyword=allocation%20profiler.*
6. Extract the contents of AllocationProfiler.zip to C:\Debuggers\AllocProf.

Documentation Conventions

This guide uses the following style conventions and terminology.

Table 1.3: Style conventions

Element	Meaning
Bold font	Characters that you type exactly as shown, including commands and switches. User interface elements are also bold.
Italic font	Placeholder for variables for which you supply a specific value. For example, *Filename.ext* could refer to any valid file name for the case in question. New terminology also appears in italic on first use.
`Monospace font`	Code samples.
Note	Alerts you to supplementary information.

Summary

This chapter introduced the difference between debugging in production and development environments and introduced a number of debugging tools. The chapter also explained and compared ASP.NET process models used by IIS 5.*x* and 6.0, as well as how ASP.NET uses health monitoring to maximize the reliability and robustness of applications.

The next few chapters focus on specific debugging problems and how to solve them. The chapters present three specific debugging scenarios that illustrate typical debugging problems and the use of the tools just described.

2

Debugging Memory Problems

This chapter describes how to approach debugging memory consumption problems that users might experience when using ASP.NET applications. First, the chapter discusses how memory is managed in .NET, and in particular how the garbage collector (GC) reclaims unused memory. A walkthrough then shows how to debug a memory consumption scenario.

Although reproducing this problem on your machine might not give you the exact same results because of differing operating systems and runtime versions, the output should be similar, and the debugging concepts are the same. Also, because all .NET Framework applications use the same memory architecture, you can apply what you learn about memory management in an ASP.NET context to other .NET environments, such as console applications, Microsoft® Windows® operating system applications, and Windows services.

.NET Memory Management and Garbage Collection

C and C++ programs have traditionally been prone to memory leaks because developers had to manually allocate and free memory. In Microsoft® .NET, it is not necessary to do this because .NET uses *garbage collection* to automatically reclaim unused memory. This makes memory usage safer and more efficient.

The garbage collector (GC) uses references to keep track of objects that occupy blocks of memory. When an object is set to null or is no longer in scope, the GC marks the object as reclaimable. The GC can return the blocks of memory referenced by these reclaimable objects to the operating system.

The performance benefit of a GC arises from deferring the collection of objects, as well as from performing a large number of object collections at once. GCs tend to use more memory than typical memory management routines, such as those used by the Windows-based operating systems.

The GC in .NET uses the Microsoft Win32® **VirtualAlloc()** application programming interface (API) to reserve a block of memory for its heap. A .NET managed heap is a large, contiguous region of virtual memory. The GC first reserves virtual memory, and then commits the memory as the managed heap grows. The GC keeps track of the next available address at the end of the managed heap and places the next allocation request at this location. Thus, all .NET managed memory allocations are placed in the managed heap one after another. This vastly improves allocation time because it isn't necessary for the GC to search through a free list or a linked list of memory blocks for an appropriately sized free block, as normal heap managers do. Over time, holes begin to form in the managed heap as objects are deleted. When garbage collection occurs, the GC compacts the heap and fills the holes by moving allocations using a straight memory copy. Figure 2.1 shows how this works.

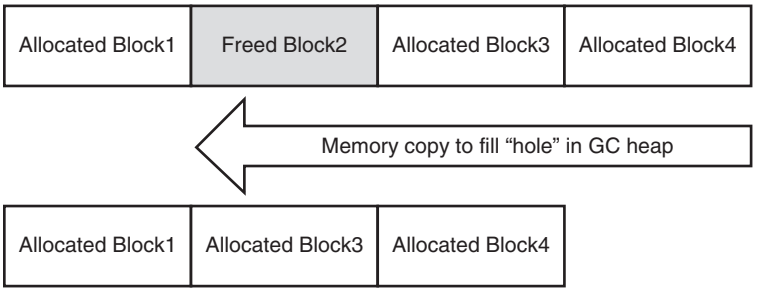

Figure 2.1
How the garbage collector compacts the heap

For more details on the .NET garbage collection mechanism, see the following references:

- "Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework", by Jeffrey Richter, *MSDN Magazine*, November 2000. (*http://msdn.microsoft.com/library/en-us/dnmag00/html/GCI.asp*).
- "Garbage Collection — Part 2: Automatic Memory Management in the Microsoft .NET Framework", by Jeffrey Richter, *MSDN Magazine*, December 2000. (*http://msdn.microsoft.com/msdnmag/issues/1200/GCI2/GCI2.asp*).
- Chapter 19, "Automatic Memory Management (Garbage Collection)" in *Applied Microsoft .NET Framework Programming* by Jeffrey Richter (Microsoft Press, 2002).

Generations

The GC improves memory management performance by dividing objects into generations based on age. When a collection occurs, objects in the youngest generation are collected. If this does not free enough memory, successively older generations can also be collected. The use of generations means that the GC only has to work with a subset of the allocated objects at any one time.

The GC currently uses three generations, numbered 0, 1, and 2. Allocated objects start out belonging to generation 0. Collections can have a depth of 0, 1, or 2. All objects that exist after a collection with a depth of 0 are promoted to generation 1. Objects that exist after a collection with a depth of 1, which will collect both generation 0 and 1, move into generation 2. Figure 2.2 shows how the migration between generations occurs.

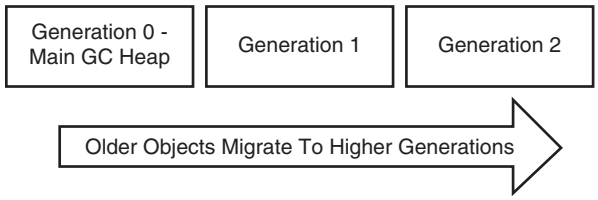

Figure 2.2
Migration between generations during multiple collections

Over time, the higher generations are filled with the oldest objects. These higher generations should be more stable and require fewer collections; therefore, fewer memory copies occur in the higher generations.

Collection for a specific generation occurs when the memory threshold for that generation is hit. In the implementation of .NET Version 1.0, the initial thresholds for generations 0, 1, and 2 are 256 kilobytes (KB), 2 megabytes (MB), and 10 MB, respectively. Note that the GC can adjust these thresholds dynamically based on an application's patterns of allocation. Objects larger than 85 KB are automatically placed in the large object heap, which is discussed later in this chapter.

Roots

The GC uses object references to determine whether or not a specific block of memory in the managed heap can be collected. Unlike other GC implementations, there is not a heap flag on each allocated block indicating whether or not the block can be collected. For each application, the GC maintains a tree of references that tracks the objects referenced by the application. Figure 2.3 on the next page shows this tree.

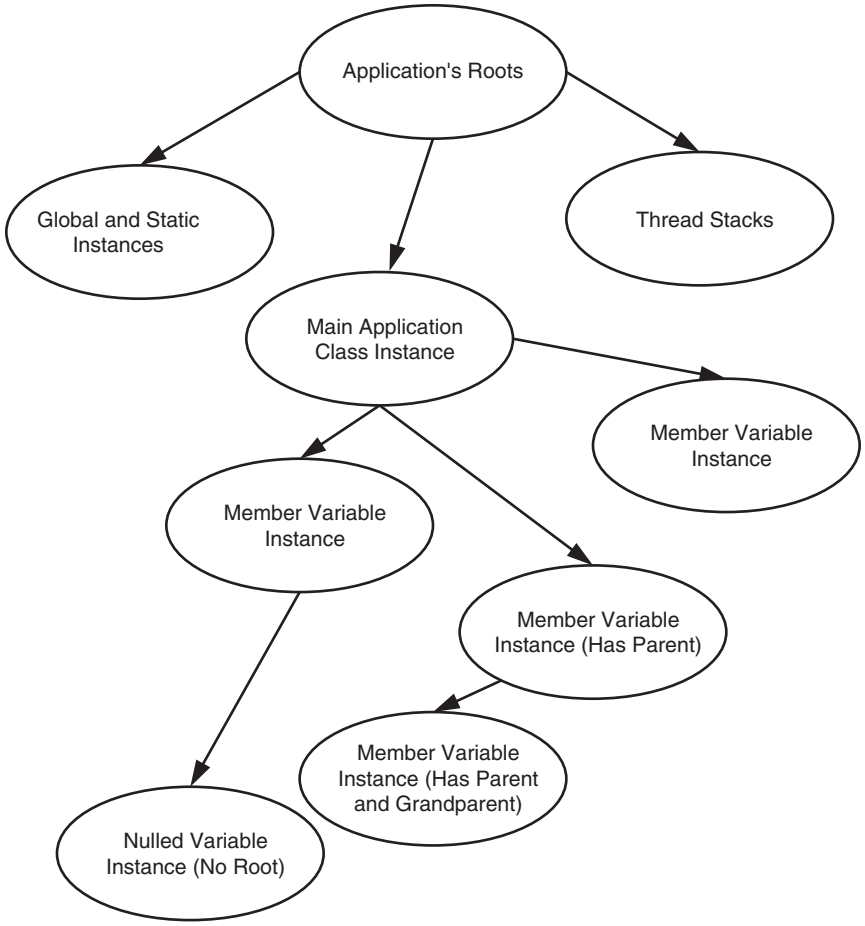

Figure 2.3
Root reference tree

The GC considers an object to be *rooted* if the object has at least one parent object that holds a reference to it. Every application in .NET has a set of *roots*, which includes global and static objects, as well as associated thread stacks and dynamically instantiated objects. Before performing a garbage collection, the GC starts from the roots and works downward to build a tree of all variable references. The GC builds a master list of all live objects, and then walks the managed heap looking for objects that are not in this live object list.

This would appear to be an expensive way of determining whether or not an object is alive, compared with using a simple flag in a memory block header or a reference counter, but it does ensure complete accuracy. For example, an object reference counter could be mistakenly over-referenced or under-referenced, and a heap flag could be mistakenly set as deleted when there are live references to the memory block. The managed heap avoids these issues by enumerating all live objects and building a list of all referenced objects before collection. As a bonus, this method also handles circular memory reference issues.

If there is a live reference to an object, that object is said to be *strongly rooted*. .NET also introduces the notion of *weakly rooted* references. A weak reference provides a way for programmers to indicate to the GC that they want to be able to access an object, but they don't want to prevent the object from being collected. Such an object is available until it is collected by the GC. For example, you could allocate a large object, and rather than fully deleting and collecting the object, you could hold onto it for possible reuse, as long as there is no memory pressure to clean up the managed heap. Thus, weak references behave somewhat like a cache.

Large Object Heap

The .NET memory manager places all allocations of 85,000 bytes or larger into a separate heap called the *large object heap*. This heap consists of a series of virtual memory blocks that are independent from the main managed heap. Using a separate heap for larger objects makes garbage collection of the main managed heap more efficient because collection requires moving memory, and moving large blocks of memory is expensive. However, the large object heap is never compacted; this is something you must consider when you make large memory allocations in .NET.

For example, if you allocate 1 MB of memory to a single block, the large object heap expands to 1 MB in size. When you free this object, the large object heap does not decommit the virtual memory, so the heap stays at 1 MB in size. If you allocate another 500-KB block later, the new block is allocated within the 1 MB block of memory belonging to the large object heap. During the process lifetime, the large object heap always grows to hold all the large block allocations currently referenced, but never shrinks when objects are released, even if a garbage collection occurs. Figure 2.4 on the next page shows an example of a large object heap.

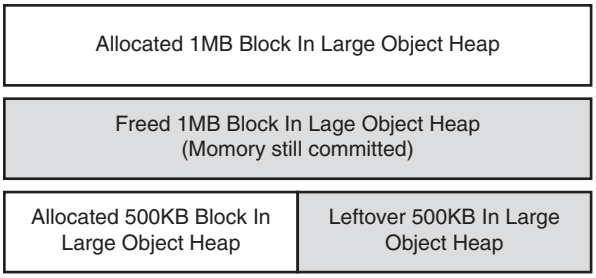

Figure 2.4
Large object heap

Scenario: Memory Consumption

Now that you have been introduced to .NET memory management and garbage collection mechanisms, let's see how these are used in ASP.NET applications. This scenario investigates how to debug memory consumption problems. As you might already know, memory leaks are caused by not deallocating memory that was dynamically allocated. A small memory leak might not be noticed and might cause only minimal damage, but large memory leaks could severely impact performance by draining available memory. In addition, there might be other memory-related problems that aren't "true" memory leaks, but which exhibit memory-leak symptoms. This scenario focuses on the latter category of memory-related problems.

Here are some typical customer scenarios that could indicate memory problems:

- **Scenario 1:** An e-commerce Web site sells products. Browser clients complain of losing data and seeing Server Application Unavailable errors. They have to log in again and don't understand why. In addition, server performance slows when memory usage increases.
- **Scenario 2:** A Web site provides a file upload facility for video files. When a browser client uploads a large file, the process recycles and the file upload fails.

Breaking Down the Thought Processes

There are many things to check when trying to debug memory-related problems. Figure 2.5 shows a flowchart you can follow when troubleshooting consumption problems.

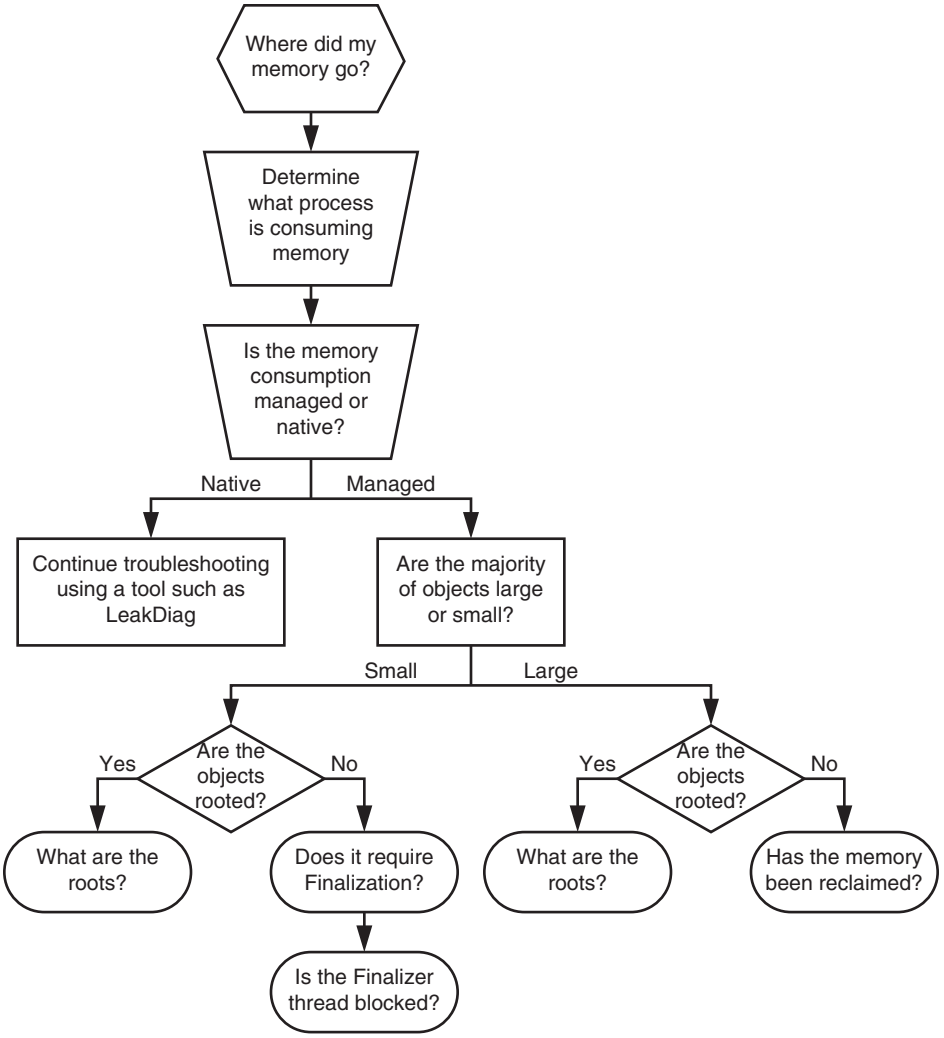

Figure 2.5
Flowchart for troubleshooting memory consumption issues

Although the thought processes are described in the walkthrough that follows, let's look at the flowchart in more detail.

Do the Errors Point to Memory Leak Symptoms?

If you're using Microsoft Internet Information Services (IIS) 5.x, check to see if there is an application event log error that indicates that an ASP process was recycled. If this has occurred, you'll see an error message similar to the following: "aspnet_wp.exe (PID: 1234) was recycled because memory consumption exceeded *n* MB (*n* percent of available RAM)."

Where Did My Memory Go?

The next step is to determine which process is using too much memory. Tools such as System Monitor (known as Performance Monitor in Windows 2000) or Task Manager can isolate those processes. Once you know which process is responsible, you can run AutoDump+ (ADPlus) to create a full memory dump of the Aspnet_wp.exe process and then, using the WinDbg debugger and the SOS.dll debugger extension, examine the differences between managed and native memory. This is discussed later in this chapter.

If the project is being run in a controlled environment (for example, in development, quality assurance (QA), or test), reproduce the problem and run ADPlus in **-hang** mode to produce a full memory dump of the Aspnet_wp.exe process. To allow more time before the process becomes recycled, you can use the **memorylimit** attribute on the <processModel> element in machine.config to increase the limit from the default of 60 percent to a higher percentage.

You can also use .NET APIs or System Monitor to gather more information. For example, with System Monitor, you can look for patterns that indicate either a .NET managed memory leak or a native memory leak.

Native Memory Consumption

If the **Private Bytes** counter in System Monitor increases while the **.NET # of Bytes in all Heaps** counter stays flat, this is evidence of native memory consumption. These counters are discussed in more detail in "Memory Consumption Walkthrough" later in this chapter.

Managed Memory Consumption

When using System Monitor, if both the **Private Bytes** counter and the .**NET # of Bytes in all Heaps** counter increase at the same rate, this is evidence of managed memory consumption. Look at the size of allocated managed memory and consider what the GC is doing. When looking at the dump file, use_SOS commands to list the managed heap size and compare it to the total size of the dump file. Consider what details you can learn about the large object heap and the generations. Look to see if large objects (85 KB or more) or smaller objects consume most of the memory. If it's the former, look at the details of the large object heap. If it's the latter, consider what generations 0, 1, and 2 contain. For large objects, determine if they are rooted, and whether or not they should be rooted. If they aren't rooted, they are candidates for collections. Determine if they are eventually collected properly.

With WinDbg and SOS.dll, it can be difficult to look at all of the small objects' details if there are many small objects. In such cases, it might be easier to use the Allocation Profiler to look at details for both large and small objects. "Memory Consumption Walkthrough" uses all of these tools to diagnose a memory consumption issue.

Memory Consumption Walkthrough

The following walkthrough presents a simplified, realistic scenario: An ASP.NET application allocates too much memory and ends up recycling because it breaches the memory limit allowed in the process model. The purpose of the walkthrough is to illustrate techniques that can be used to help you troubleshoot such a problem. Production environment scenarios might not be as clear cut as this simplified example, but you can apply similar techniques to help identify the causes of memory consumption.

This walkthrough deals with a large memory leak that could severely impact performance by draining available memory. An example ASP.NET page allocates large objects (20 MB each), and then caches them in the ASP.NET cache. By default, when memory consumption exceeds 60 percent of available RAM, the ASP.NET process is recycled. The walkthrough aims to determine which process is using too much memory, and why.

In this scenario, you will perform the following steps:

1. Browse to the ASP.NET page at *http://localhost/debugging/memory.aspx* and consider the values displayed in the process model table.
2. Follow the thought processes in the flowchart and find the errors presented both in the browser and the application event log.
3. Create a dump file and examine the dump data using WinDbg and SOS.dll.
4. Inspect the managed memory for the ASP.NET worker process and determine which objects are consuming the most memory.
5. After you've found evidence in the dumps for possible problem areas, look at the source code.

Baseline View

First, gather some baseline data for comparisons with the native and managed memory consumptions throughout the walkthrough. For instructions on how to download the sample pages, see Chapter 1, "Introduction to Production Debugging for .NET Framework Applications."

► **To view Memory.aspx**

- Open a browser window, and then browse to *http://localhost/Debugging /memory.aspx*.

  Note the information displayed on the page, especially the fields and values in the table.

The following browser view provides a baseline from which to compare data after you create objects that consume memory.

Figure 2.6
Baseline browser data

The following table describes what these fields display:

Table 2.1: Description of fields in Memory.aspx

Field	Description
StartTime	The time at which this Aspnet_wp.exe process started
Age	The length of time the process has been running
ProcessID	The ID assigned to the process
RequestCount	Number of completed requests, initially zero
Status	The current status of the process: if **Alive**, the process is running; if **Shutting Down**, the process has begun to shut down; if **ShutDown**, the process has shut down normally after receiving a shut down message from the IIS process; if **Terminated**, the process was forced to terminate by the IIS process
ShutdownReason	The reason why a process shut down: if **Unexpected**, the process shut down unexpectedly; if **Requests Limit**, requests executed by the process exceeded the allowable limit; if **Request Queue Limit**, requests assigned to the process exceeded the allowable number in the queue; if **Timeout**, the process restarted because it was alive longer than allowed; if **Idle Timeout**, the process exceeded the allowable idle time; if **Memory Limit Exceeded**, the process exceeded the per-process memory limit
PeakMemoryUsed	The maximum amount of memory the process has used. This value maps onto the **Private Bytes** (maximum amount) count in System Monitor

You can explore the code and read through the comments by opening another instance of Memory.aspx.cs in either Microsoft Visual Studio® .NET or Notepad.

Each time the **Allocate 20 MB Objects** button is clicked, five unique cache keys and five 20-MB objects are created and stored in the System.Web cache. The **Allocate 200 K Objects** button works in a similar way, but it creates 200-KB objects that are stored in the System.Web cache. The code behind this button simulates a non-fatal leaking condition for experimenting with alternative testing scenarios.

Free Memory obtains a list of cache keys from Session scope and clears the cache. It also calls the **System.GC.Collect()** method to force a garbage collection.

Note: System.GC.Collect() is used for demonstration purposes, not as a recommended procedure. Explicitly calling **GC.Collect()** changes the GC's autotuning capabilities. Repeatedly calling **GC.Collect()** suspends all threads until the collection completes. This could greatly impede performance.

Finally, **Refresh Stats** refreshes the memory statistics.

You'll use Task Manager and System Monitor to confirm or discover more information about the ASP.NET process:

► To use Task Manager

1. Open Task Manager.
2. On the **Processes** tab, click the **View** menu, click **Select Columns**, and then select the **Virtual Memory Size** check box.

   This adds virtual memory size to the default columns.

   Note: If you are using Windows 2000 Server through a Terminal Server Client, you need to select the **Show processes from all users** check box**.**

What is the Aspnet_wp.exe process's virtual memory size? In this test, the virtual memory size of Aspnet_wp.exe is 9,820 KB. This value is a reasonable baseline.

Table 2.2: Baseline Task Manager data

Process	Virtual memory size
Aspnet_wp.exe	9,820 KB

Start the System Monitor.

► To start the System Monitor

1. In Control Panel, click **Administrative Tools**, and then double-click **Performance**.
2. On the System Monitor toolbar, click '**+**', and then in the **Add Counters** dialog box, select **.NET CLR Memory** from the **Performance object** drop-down list.

3. In the **Select instances from list** box, click **aspnet_wp**.
4. Use CTRL+Click to select the entries shown in Figure 2.7 from the Counters list box, and then click **Add**.

   Repeat the process for the **ASP.NET Applications** and **Process** performance objects, and click **Add** after selecting the appropriate instances and counters.
5. Click **Close**, and then click the **View Report** button on the toolbar to switch to text view. You should see a screen similar to the one in Figure 2.7 below.

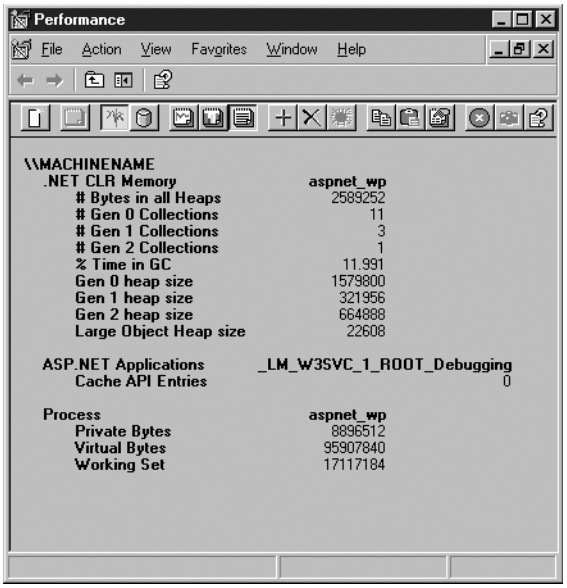

Figure 2.7
Baseline System Monitor data

Using the System Monitor report, pay particular attention to **# Bytes in all Heaps**, **Large Object Heap Size**, and **Cache API Entries**.

- **# Bytes in all Heaps** displays the sum of the **Gen 0 Heap Size**, **Gen 1 Heap Size**, **Gen 2 Heap Size**, and **Large Object Heap Size** counters. The **# Bytes in all Heaps** counter indicates the current memory allocated in bytes on the garbage collection heaps.
- **Large Object Heap Size** displays the current size, in bytes, of the large object heap. Note that this counter is updated at the end of a garbage collection, not at each allocation.
- **Cache API Entries** is the total number of entries in the application cache.

The baseline values before allocating any memory are:

- **Large Object Heap Size:** 22,608 bytes
- **# Bytes in all Heaps:** 2,589,252
- **Cache API Entries:** 0

Note: To learn more about the System Monitor .NET Performance Counters, see:

- ".NET Framework General Reference: Memory Performance Counters" on the MSDN Web site at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/gngrfmemoryperformancecounters.asp*.
- ".NET Framework Developer's Guide: Performance Counters for ASP.NET" on the MSDN Web site at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconperformancecountersforaspnet.asp*.

First Allocation Pass

Now that you've established the baseline statistics for the application and System Monitor counters with the ASP.NET process running, consider the amount of native and managed memory consumed. On the Memory.aspx page, click **Allocate 20 MB Objects** once. The data in your browser should resemble the following tables:

Table 2.3: Baseline and one allocation browser data

Field	Baseline value	New value
StartTime	07/08/2002 12:14:06 PM	07/08/2002 12:40:59 PM
Age	00:00:04.2961776	00:00:33:9585520
ProcessID	3,212	3,212
RequestCount	0	1
Status	Alive	Alive
ShutdownReason	N/A	N/A
PeakMemory Used	5,948	8,904

Updated Memory Stats	Baseline value	New value
GC.TotalMemory	780 KB	100,912 KB
Private Bytes	8,896 KB	117,616 KB

A new table appears on the page that contains two entries:

- **GC.TotalMemory** is the total amount of managed memory that the GC heap has allocated for managed objects. (This amount maps onto the System Monitor **# Bytes in all Heaps** counter for the **.NET CLR Memory Performance** object.)
- **Private Bytes** is the total size of native memory that a process has allocated that cannot be shared with other processes.

After clicking **Allocate 20 MB Objects**, the **RequestCount** field increments by one to count the completed request. You can also see that **GC.TotalMemory** and **Private Bytes** equal approximately 100 MB.

Look at the code for the ASP.NET page to see the source of the 100 MB:

```
for (int i = 0; i< 5; i++)
{
  long objSize = 20000000;
  string stime = DateTime.Now.Millisecond.ToString();
  string cachekey = "LOCache-" + i.ToString() + ":" + stime;
  Cache[cachekey] = CreateLargeObject(objSize);
  StoreCacheListInSession(cachekey);
}
```

The Private Bytes figure contains both the managed and native allocations for the process.

The example scenario displays the **Private Bytes** and **GC.TotalMemory** memory statistics on the ASP.NET page itself. When production debugging, you might need to use other tools, such as Task Manager and System Monitor, to obtain this information. You can use Task Manager to display native data, while System Monitor shows both managed and native memory data.

On the **Processes** tab in Task Manager, locate the Aspnet_wp.exe process, which has the same process ID (PID). You can see that the virtual memory size has increased to almost 115,000 KB.

Table 2.4: Baseline and one allocation Task Manager data

Process	Baseline value	New value
Aspnet_wp.exe	9,820 KB	114,840 KB

The virtual memory size has increased by about 100,000 KB. The ASP.NET process is consuming most of the memory on the machine.

Note: The **VM Size** column in Task Manager roughly maps to the System Monitor **Process: Private Bytes** counter.

In the chart view of System Monitor, you'll see that the **Private Bytes** and **# Bytes in all Heaps** counter values are both increasing at the same rate. This implies an increase in the use of managed memory, rather than native memory.

Figure 2.8 shows the increase in memory usage when the button is clicked. To change the maximum value of the y-coordinate on a graph, right-click the graph, and then select **Properties**. Click the **Graph** tab, and then change the value in the **Vertical scale Maximum** box.

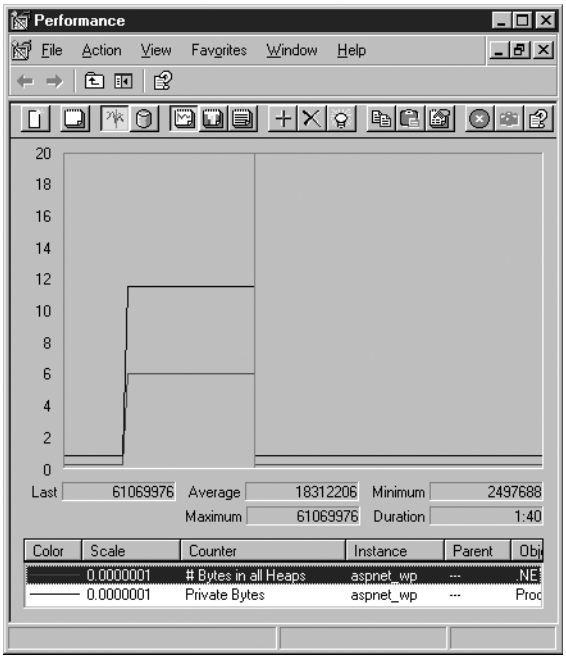

Figure 2.8
System Monitor graph

Second Allocation Pass

The next step consumes more memory. Click **Allocate 20 MB Objects** once more, making two total clicks. Note how the values reported on the ASP.NET page change, especially **RequestCount**, **PeakMemoryUsed**, and **GC.TotalMemory**.

Table 2.5: Baseline, one allocation, and two allocations browser data

Field	Baseline value	One allocation	Two allocations
StartTime	07/08/2002 12:14:06 PM	07/08/2002 12:40:59 PM	07/08/2002 13:25:51 PM
Age	00:00:04.2961776	00:00:23:9585520	00:47:33.1343328
ProcessID	3,212	3,212	3,212
RequestCount	0	1	2
Status	Alive	Alive	Alive
ShutdownReason	N/A	N/A	N/A
PeakMemory Used	5,948	8,904	113,752

Memory Stats	Baseline value	One allocation	Two allocations
GC.TotalMemory	780 KB	100,912 KB	200,916 KB
Private Bytes	8,896 KB	117,616 KB	221,450 KB

To see the results of this second button-click event on system memory, look at the details for the process in Task Manager.

Table 2.6: Baseline, one allocation, and two allocations Task Manager data

Process	Baseline value	One allocation	Two allocations
Aspnet_wp.exe	9,820 KB	114,840 KB	216,240 KB

The **VM Size** value has gone up another 100,000 KB, and the Aspnet_wp.exe process is still consuming most of the memory.

The System Monitor report and the chart views confirm these values. Now look at the **Cache API Entries** counter. As you saw in the code, each click event runs a loop that creates five cache items. The loop has been run twice, so there have been ten cache API entries.

The difference between the **#Bytes in all Heaps** and the **Large Object Heap Size** counters indicates that most of the managed memory is being used by the large object heap. This is shown in figure 2.9.

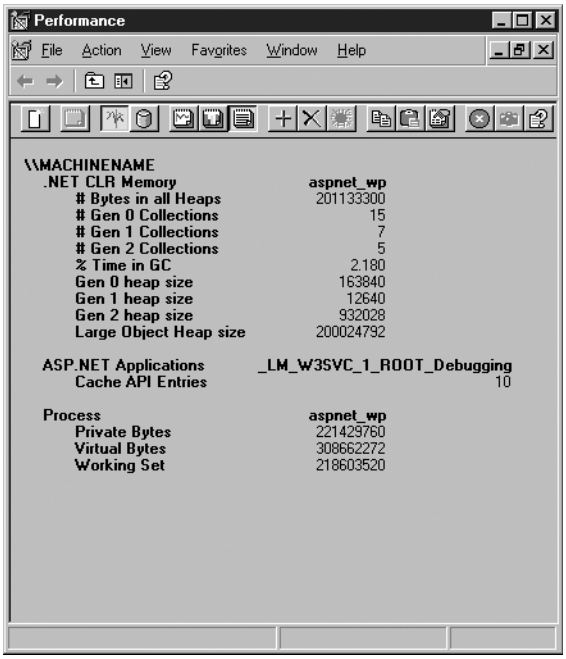

Figure 2.9
System Monitor data

Third Allocation Pass

The final step consumes even more memory. Click **Allocate 20 MB Objects** a third time, quickly switch to the **Performance** tab of the Task Manager, and examine the **Page File Usage History**.

Note: In Windows XP, the Task Manager displays **Page File Usage** on the **Performance** tab, whereas in Windows 2000, the Task Manager displays **Memory Usage** on the **Performance** tab.

Each time the **Allocate 20 MB Objects** button is clicked, the graph shows a steep incline, which levels off until the next time the button is clicked. If the process is recycled, you will see a sudden drop in **Page File Usage**. Figure 2.10 on the next page shows this happening in Task Manager.

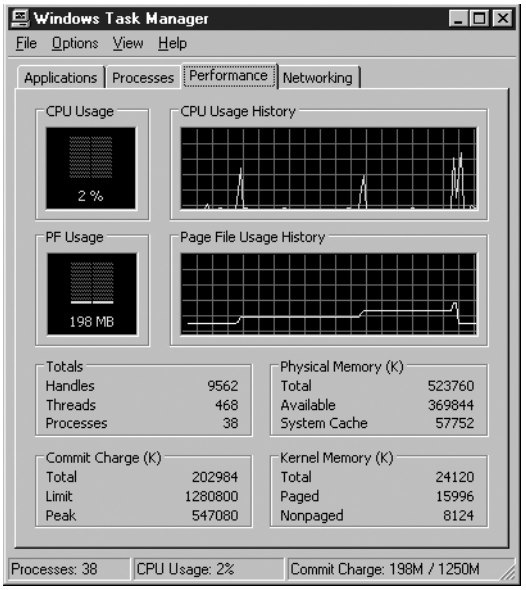

Figure 2.10
Task Manager data

Task Manager can be used for a "guesstimate" of overall system performance. Committed memory is allocated to programs and the operating system, and the Page File Usage History graphs these values. For each click event, you can see the commit charge value increase proportionately to the graph.

You can also see the results of the Aspnet_wp.exe process being recycled. As noted, memory usage increases with each button click and is not released until the process is recycled. You can check that the process has been recycled by looking in the application event log.

```
Event Type:     Error
Event Source:   ASP.NET 1.0.3705.0
Event Category: None
Event ID:       1001
Date:           6/7/2002
Time:           1:04:14 AM
User:           N/A
Computer:       machinename
Description:     aspnet_wp.exe  (PID: 2740) was recycled because
                memory consumption exceeded the 306 MB
                (60 percent of available RAM).
```

Switch back to the ASP.NET page in the browser, and click the **Refresh Stats** button. You can now see a second table added to the page, which contains data from a new process. The Process Model health monitoring shut down the original process because the memory limit was reached, and then started a new process. Compare the two tables and look particularly at the **Status** and **ShutdownReason** fields, and the **Updated Memory Stats** table.

For the first process, the value of the **Status** field has changed from **Alive** to **Terminated**, while the second process has become **Alive**. The first process has been terminated because the memory limit was exceeded. Note that ASP.NET terminates the process *after* you allocate memory and exceed the maximum memory-peak memory limit.

Because the process was recycled, the **Updated Memory Stats** table shows the baseline values again.

Exploring Further

Refresh Stats only updates the statistics. To vary the results, you can experiment with the controls on the browser interface:

- Click **Allocate 20 MB Objects** again instead of clicking **Refresh Stats**. The **RequestCount** value in the first table increments, and the second table shows a Request Count of 0 (a new worker process was created). The Process Model uses intelligent routing when the memory limit is reached. It acknowledges the request, kills the old process, and creates a new process to accommodate requests.
- Click **Allocate 20 MB Objects** one more time. This increments the second table's **RequestCount** value because we're dealing with the new worker process and the memory limit has not been met for that process.
- Click **Allocate 200 K Objects** repeatedly. This creates multiple 200-KB objects that eventually accumulate in the large object heap. You might want to try this option, as it uses a similar code path with less memory usage.

Let's review the conclusions that can be made so far:

- You're dealing with managed memory usage. The System Monitor chart view shows how the **Private Bytes** and **# Bytes in all Heaps** counters both increase at the same slope.
- The Aspnet_wp.exe process consumes most of the memory. Once the Aspnet_wp.exe process recycles, the committed memory count normalizes.
- Most of the managed heap memory is in the large object heap. The size of the large object heap is almost the same as the value of the **Total # Bytes in all Heaps** counter.

Debugging User-Mode Dump Files with WinDbg

To confirm these findings and learn more about what is happening in the ASP.NET process, use ADPlus to create a dump file, and then examine the dump file with WinDbg and SOS.dll. For more information on ADPlus, see article Q286350, "HOWTO: Use Autodump+ to Troubleshoot 'Hangs' and 'Crashes'" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q286350*. Also, if you want to run ADPlus through Terminal Server, see article Q323478, "PRB: You Cannot Debug Through a Terminal Server Session" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q323478*.

► **To start from the beginning**

- Open a command prompt window, and then type **iisreset** at the prompt to restart IIS.

By default, when memory consumption exceeds 60 percent of available RAM, an Aspnet_wp.exe process is recycled. The time before the memory limit is reached could be short, and might allow only minimal time for debugging. To prevent the process from terminating before you can debug it, modify the registry to alter the default behavior of ASP.NET when a process should be recycled.

Adding Registry Keys

You need to add two registry values: **DebugOnHighMem** and **UnderDebugger**. These registry settings tell ASP.NET to call **DebugBreak** instead of terminating the process.

Note: The following exercise requires that you make changes to the registry. It is always recommended that you back up the registry before making changes. In addition, when you have finished debugging, it is recommended that you delete the DWORDs created below. These registry DWORDs are undocumented, and should only be used during specific debugging sessions.

► **To create the registry values**

1. Start Regedit, and then locate the following key: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ASP.NET.
2. Right-click ASP.NET, point to **New**, and then select **DWORD Value**.
3. Change the name of the new entry from **New Value #1** to **DebugOnHighMem**.
4. Double-click the new name and change the HEX data value from 0 to 1.
5. Create another **New DWORD Value** and name it **UnderDebugger**. Leave the data value as HEX 0.

You should end up with the settings shown in figure 2.11.

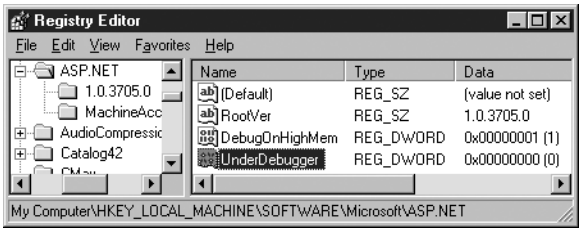

Figure 2.11
Registry Editor

The memory limit monitoring mechanism is automatically disabled whenever a debugger is attached to an Aspnet_wp.exe process, unless the value of the **UnderDebugger** DWORD value is set to zero, in which case the mechanism is not disabled.

Configuring ADPlus and Creating a Dump File

ADPlus can be configured to create a full dump if CTRL+C is pressed while it is running in **-crash** mode. Open the ADPlus.vbs script and change the value of the **Full_Dump_on_CONTRL_C** from FALSE to TRUE. Here's the relevant code from ADPlus.vbs:

```
' Set Full_Dump_on_CONTRL_C to TRUE if you wish to have AD+ produce a full
' memory dump (as opposed to a mini-memory dump) whenever CTRL-C is pressed to
stop
' debugging, when running in 'crash' mode.
' Note:  Depending upon the number of processes being monitored, this could
' generate a significant amount of memory dumps.

Const Full_Dump_on_CONTRL_C = TRUE
```

Even though the process is not crashing, you can use ADPlus in **-crash** mode to gather process data. To capture the information, run the ASP.NET pages again.

► To Run ADPlus in -crash mode

1. Browse to *http://localhost/debugging/memory.aspx*.
2. Open a command prompt and start **ADPlus** from the command line.
3. Specify the **-crash** switch and the process name, which in this case is **Aspnet_wp.exe**.

You can also use the **-quiet** option to suppress extra program information, as in this example:

```
c:\debuggers>adplus.vbs -pn aspnet_wp.exe  -crash -quiet
```

In this case, the ADPlus output will be found in a uniquely named subdirectory of C:\debuggers. To specify an output directory, use the **-o** switch to specify a path, as shown in the following example:

```
c:\debuggers>adplus.vbs -pn aspnet_wp.exe -crash -quiet -o
c:\labs\highMemory
```

Note: The **-quiet** switch ensures that you can still create a dump file if the symbol path for ADPlus is not set.

Output from the debugger will be similar to the following:

```
C:\debuggers>adplus.vbs -pn aspnet_wp.exe -crash -quiet
The '-crash' switch was used, Autodump+ is running in 'crash' mode.
The '-quiet' switch was used, Autodump+ will not display any modal dialog boxes.

Monitoring the processes that were specified on the command line
for unhandled exceptions.
------------------------------------
Attaching the CDB debugger to: Process - ASPNET_WP.EXE with PID - 344
```

You should also see a minimized window running CDB.exe.

At this point, you create a dump file so that you can examine what is consuming memory.

Note: The dump files generated by ADPlus can be extremely large. Every time the **Allocate 20 MB Objects** button is clicked, 100 MB is allocated, so consider the amount of free disk space you have on your machine. If you have 512 MB of RAM and you click the button three times, you need 300+ MB of available disk space.

To create a dump file, use the ASP.NET page to create objects that consume memory.

► **To allocate memory**

1. Switch to the Memory.aspx page in the browser.
2. Click **Allocate 20 MB Objects** three times. The memory limit is reached after the third button-click, at which point CDB.exe creates a dump.
3. Click **Refresh Stats**.

The debugger terminates the process, generates a log, and exits. Locate the files generated by ADPlus. If you didn't specify a path on the ADPlus command line, these files will be placed in a directory whose name is made up from the date and time at which the crash occurred, as shown in the following example:

```
c:\debuggers\Crash_Mode__Date_05-19-2002__Time_19-41-21PM
```

The following describes the files typically generated by ADPlus:

- **Autodump+-report.txt** is a report that ADPlus.vbs creates. It displays the list of constants that ADPlus knows about, and the constants' current settings. This report is used for diagnostic purposes if the ADPlus.vbs script doesn't function correctly after it has been modified from its original form.
- **PID-1234__process.exe.cfg** is passed to CDB.exe to script the commands that need to be run. This file can be useful for diagnosing problems when running ADPlus.vbs.
- **PID-1234__PROCESS.EXE__Reason__full/mini_timestamp.dmp** is the full dump file for the process. The number after the PID is the process ID for the dumped process. Process.exe will be replaced by the name of the process; mini_timestamp will be replaced by a timestamp; and Reason will be replaced by a string identifying the reason for the dump, for example "CTRL+C" for a crash dump. Therefore, a typical dump file name might be **PID-344__ASPNET_WP.EXE__CTRL-C__full_2002-05-19_19-43-59-798_0158.dmp**. This sample file was 358 MB in size.
- **PID-1234__PROCESS.EXE__Reason__full/mini_timestamp.log** is an output log of the debugger. This file can be useful if you are looking for some history of exceptions or events prior to the failure. Once again, parts of the file name will be replaced with the PID, the process name, and the reason.
- **Process_List.txt** is the output of Tlist.exe -v. This file is used by ADPlus.vbs to determine the PIDs and names of processes running on the machine.

Examining the Dump File

Examine the dump file that was created when Aspnet_wp.exe process was recycled.

► To examine the dump file using WinDbg

1. On the **Start** menu, point to **Debugging Tools for Windows**, and then click **WinDbg**.
2. On the **File** menu, click **Open Crash Dump**.
3. Select the appropriate dump file, and then click **Open**.
4. If you are prompted to "Save Base Workspace Information," click **No**.
5. If a Disassembly window pops up, close the window, and then on the **Window** menu, click **Automatically Open Disassembly**.

Symbol paths must be entered for the modules that will be used by the debugger to analyze the dump file. The symbol file versions that are needed on the debugging computer should match the versions on the system that produced the dump file. In this scenario, you need to include symbols for the .NET Framework SDK, Visual C Runtime, and Windows.

To enter the symbol paths, do one of the following:

- From the command line, type:

```
.sympath
SRV*c:\symbols\debugginglabs*http://msdl.microsoft.com/download/symbols;C:\
symbols\debugginglabs;C:\Program Files\Microsoft Visual Studio
.NET\FrameworkSDK\symbols;C:\windows\system32

?   Enter the symbol path for WinDbg through the _NT_SYMBOL_PATH environment
variable.
```

- On the **File** menu in WinDbg, select **Symbol File Path**, and then type:

```
SRV*c:\symbols\debugginglabs*http://msdl.microsoft.com/download/symbols;C:\
symbols\debugginglabs;C:\Program Files\Microsoft Visual Studio
.NET\FrameworkSDK\symbols;C:\windows\system32
```

The "SRV" in the path tells WinDbg to go to an external symbol server and copy symbols into the local symbol cache.

► **To use these same symbol paths for other dumps**

- On the **File** menu, select **Save Workspace As**, and then type a name for the saved paths, such as **.NET Debugging Symbols**.

WinDbg debugs native code. To examine managed code, you need to load the SOS.dll debugger extension.

Note: If you haven't already done so, install the SOS.dll debugger extension. For download instructions, see Chapter 1, "Introduction to Production Debugging for .NET Framework Applications."

► **To examine managed code**

1. Press ALT+1 to open the WinDbg command window.
2. Type **.load <SOS>\SOS.dll** to load the extension, and then replace **<SOS>** with the SOS.dll location.
3. Use the **!findtable** command to initialize SOS with the table information for the runtime version you will be debugging.

The output should be similar to the following:

```
0:001> .load sos\sos.dll
0:001> !findtable
```

```
Args: ''
Attempting data table load from SOS.WKS.4.3215.11.BIN
Failed timestamp verification.
       Binary time: 05:30:57, 2002/1/5.
       Table time: 03:40:50, 2001/9/5
Attempting data table load from SOS.WKS.4.2916.16.BIN
Failed timestamp verification.
       Binary time: 05:30:57, 2002/1/5.
       Table time: 03:38:40, 2001/6/7
Attempting data table load from SOS.WKS.4.3705.BIN
Loaded Son of Strike data table version 4 from "SOS.WKS.4.3705.BIN"
```

Note: SOS.dll requires a .bin file that matches the .NET runtime version. In the example previous, two .bin files were examined before the correct version was found. Future versions of SOS will not require a .bin file.

Because you configured ADPlus to create a full dump, and this dump was generated with a reference to CTRL+C in the filename, you know that **DebugBreak** was called and excessive memory consumption was detected by the Process Model health monitoring.

ADPlus created the dump when the Aspnet_wp.exe process was recycled. As you saw in the application event log entry, the memory consumption exceeded 60 percent of the available RAM. To troubleshoot this memory consumption issue, you need to examine the dump file, and this involves following some steps that are not always intuitive. These steps are explained in the following sections.

Analyzing the Dump: Summary

Before diving into the details, here's a summary of the steps you'll follow to analyze the dump file.

When dealing with memory consumption issues, consider what the GC is doing. You can use the **!eeheap** WinDbg command to list the managed heap size, and then compare this size to the total size of the dump file.

Next discover what .NET managed objects are taking up space on the managed heap. In this example, use the **!dumpheap** command to discover how many **System.Byte**[] objects are in the large object heap.

You need to know how long these objects will exist and how large they are. The **!gcroot** command shows that the **System.Byte**[] objects have a strong reference, which means they are rooted. The **System.Web.Caching.Cache** class references the **System.Byte**[] objects, which is why the objects are rooted.

You have now found objects referenced by the ASP.NET cache, so attempt to discover how much memory is referenced by the cache.

To find the memory address of the **System.Web.Caching.Cache**, use the **!name2ee** command qualified with the assembly and class name. This returns a MethodTable address, and you can use the **!dumpheap** command to dump the managed heap contents for the objects with that MethodTable.

The **!dumpheap** command gives the actual **System.Web.Caching.Cache** object address, which can then be used with the **!objsize** command to find the number of bytes that the root keeps alive on the GC heap. Because this size is comparable to the size of the dump file, you can infer that most of the managed memory is being referenced by the **System.Web.Caching.Cache** object.

Analyzing the Dump: Details

First consider the size of the GC heap, and then compare it to the total size of the dump file.

Use the **!eeheap** command to list the GC Heap Size and starting addresses for generation 0, 1, 2, and the large object heap. You can then compare the GC Heap Size to the total size of the dump file.

Examine the GC Heap Size with **!eeheap -gc**:

```
0:001> !eeheap -gc
generation 0 starts at 0x0110be64
generation 1 starts at 0x01109cd8
generation 2 starts at 0x01021028
 segment    begin allocated     size
01020000 01021028  0110de70 000ece48(970312)
Total Size   0xece48(970312)
------------------------------
large block  0x11e1fc04(300022788)
large_np_objects start at 17b90008
large_p_objects start  at 02020008
------------------------------
GC Heap Size  0x11f0ca4c(300993100)
```

In this case, the GC Heap Size is over 300 MB, and the size of the entire dump was 358 MB. Use the **!dumpheap -stat** command to discover which .NET managed objects are taking up space on the GC Heap. This command produces output under four columns labeled **MT**, **Count**, **TotalSize**, and **Class Name**. The **MT** output represents the Method Table, which identifies the type of the object, and the MTs are listed in descending order of total size. **TotalSize** represents the object size times the number of objects.

Note: Because of the amount of data that it has to collect and display, the **!dumpheap –stat** command can take a considerable amount of time to execute.

The following listing shows the first few of the 14,459 objects listed by the **!dumpheap –stat** command.

```
0:001> !dumpheap -stat
Bad MethodTable for Obj at 0110d2a4
Last good object: 0110d280
total 14459 objects
Statistics:
      MT    Count TotalSize Class Name
3c6185c        1        12 System.Web.UI.ValidatorCollection
3c2e110        1        12 System.Web.Configuration.MachineKeyConfigHandler
3c29778        1        12 System.Web.Configuration.HttpCapabilitiesSectionHandler
3c23240        1        12 System.Web.SafeStringResource
3c22484        1        12 System.Web.Configuration.HandlerFactoryCache
3c22028        1        12 System.Web.UI.PageHandlerFactory
3c21a48        1        12 System.Web.Configuration.HttpHandlersSectionHandler
3b5f730        1        12 System.Web.Configuration.AuthorizationConfigHandler
3b5f54c        1        12 System.Web.Configuration.AuthorizationConfig
3b5e458        1        12 System.Web.Configuration.AuthenticationConfigHandler
3b5ccb8        1        12 System.Web.SessionState.InProcStateClientManager
3b5c528        1        12 System.Web.SessionState.SessionStateSectionHandler
3b5c1c4        1        12 System.Web.SessionState.SessionOnEndTarget
3b5c108        1        12 System.Web.Caching.OutputCacheItemRemoved
3b5c078        1        12 System.Web.Security.FileAuthorizationModule
3b5bfa8        1        12 System.Web.Security.UrlAuthorizationModule
```

The end of the **dumpheap -stat** list is shown in the second listing below. The objects with the largest **TotalSize** entries are at the bottom of the MT list and include **System.String**, **System.Byte**[], and **System.Object**[]:

```
D12f28      1133     46052    System.Object[]
153cb0        88     76216    Free
321b278       85    178972 System.Byte[]
d141b0      6612    416720 System.String
Total 14459 objects
```

System.String is the last object in the list because its **TotalSize** is largest. The **Free** entry represents objects that are not referenced and have been garbage collected, but whose memory hasn't been compacted and released to the operating system yet.

This data is immediately followed by a list of the objects in the large object heap, which includes all objects over 85 KB.

```
large objects
 Address       MT     Size
17b90018  321b278 20000012 System.Byte[]
16220018  321b278 20000012 System.Byte[]
14f00018  321b278 20000012 System.Byte[]
13650018  321b278 20000012 System.Byte[]
12330018  321b278 20000012 System.Byte[]
```

```
11010018  321b278 20000012 System.Byte[]
 ec50018  321b278 20000012 System.Byte[]
 d560018  321b278 20000012 System.Byte[]
 bed0018  321b278 20000012 System.Byte[]
 a8a0018  321b278 20000012 System.Byte[]
 92d0018  321b278 20000012 System.Byte[]
 7d60018  321b278 20000012 System.Byte[]
 6850018  321b278 20000012 System.Byte[]
 53a0018  321b278 20000012 System.Byte[]
 3f50018  321b278 20000012 System.Byte[]
 2020018   d12f28     2064 System.Object[]
 2020840   d12f28     4096 System.Object[]
```

You can learn more about objects in the large object heap by using the **!gcroot** command, which finds the roots for objects. You specify an object by using its address from the **dumpheap -stat** output. For example, **!gcroot 17b90018** provides information for the first **System.Byte**[] object in the previous listing. Here's the output from **!gcroot**:

```
0:001> !gcroot 17b90018
Scan Thread 1 (4e8)
Scan Thread 5 (bb0)
Scan Thread 6 (d0)
Scan Thread 10 (43c)
Scan Thread 11 (308)
Scan Thread 12 (6e4)
Scan HandleTable 14e340
Scan HandleTable 150e40
Scan HandleTable 1a6fa8
HANDLE(Strong):37411d8:Root:020784d8(System.Object[])-
>0108b504(System.Web.HttpRuntime)->0108b9d0(System.Web.Caching.CacheSingle)-
>0108ca68(System.Web.Caching.CacheUsage)->0108ca78(System.Object[])-
>0108cb3c(System.Web.Caching.UsageBucket)-
>010f95fc(System.Web.Caching.UsageEntry[])-
>01109c8c(System.Web.Caching.CacheEntry)->00000000()
```

The Scan Thread lines show that the **!gcroot** command is scanning threads for root references to objects. The Scan HandleTable lines show that the command is also scanning the HandleTables for roots. Each application domain (or AppDomain), which is an isolated environment where applications execute, has a HandleTable, and the HandleTable is simply another source of root references. References found for a thread might be in registers, arguments, or local variables.

If the output contains **HANDLE(Strong)**, a strong reference was found. This means that the object is rooted and cannot be garbage collected. Other reference types can be found in the Appendix.

This dump shows similar data for each **System.Byte**[] object scanned by **!gcroot**. Try passing another address from the output of **dumpheap -stat** to **!gcroot**. The output

is the same as for the previous **System.Byte**[] object, except for the highlighted address for **System.Web.Caching.CacheEntry**. The other classes denote collections.

```
0:001> !gcroot 16220018
Scan Thread 1 (4e8)
Scan Thread 5 (bb0)
Scan Thread 6 (d0)
Scan Thread 10 (43c)
Scan Thread 11 (308)
Scan Thread 12 (6e4)
Scan HandleTable 14e340
Scan HandleTable 150e40
Scan HandleTable 1a6fa8
HANDLE(Strong):37411d8:Root:020784d8(System.Object[])-
>0108b504(System.Web.HttpRuntime)->0108b9d0(System.Web.Caching.CacheSingle)-
>0108ca68(System.Web.Caching.CacheUsage)->0108ca78(System.Object[])-
>0108cb3c(System.Web.Caching.UsageBucket)-
>010f95fc(System.Web.Caching.UsageEntry[])-
>01109be8(System.Web.Caching.CacheEntry)->00000000()
```

It would be useful to find out how much memory is referenced in the cache. To find the memory address of **System.Web.Caching.Cache**, use the **!name2ee** command. This command takes two parameters—the assembly name and the fully qualified class name, **System.Web.Caching.Cache**. If you don't know the assembly name, you can find it in the .NET Framework SDK documentation. Note that you can obtain similar data by using **System.Web.Caching.CacheSingle**.

```
0:001> !name2ee System.Web.dll System.Web.Caching.Cache
------------------------------
MethodTable: 03887998
EEClass: 03768814
Name: System.Web.Caching.Cache
------------------------------
```

Note: EEClass is an internal structure used to represent a .NET class. To learn more about the output from SOS commands, consult the SOS Help.

To dump the managed heap contents for objects of this type, use the **!dumpheap** command, passing the MethodTable address. For example:

```
!dumpheap -mt 03887998
```

Here's the output you'd expect from **!dumpheap**:

```
0:001> !dumpheap -mt 03887998
 Address       MT     Size
0108b8ac 03887998       12
Bad MethodTable for Obj at 0110d2a4
```

```
Last good object: 0110d280
total 1 objects
Statistics:
      MT    Count TotalSize Class Name
 3887998        1        12 System.Web.Caching.Cache
Total 1 objects
large objects
 Address       MT     Size
total 0 large objects
```

Using the **-mt** switch might seem confusing because you don't need specific information about the MethodTable; however, when the address of an object is passed to **!dumpheap -mt**, all running objects of this type in this AppDomain will be listed. If there were multiple AppDomains, each instance of the object would be listed here.

You can use this information to get the size of each object by passing the address to the **!objsize** command. In this case, there is only one address, so to dump the size of **System.Web.Caching.Cache**, use the command **!objsize 0108b8ac**. Here's the output you'd expect to see from **!objsize**:

```
0:001> !objsize 0108b8ac
sizeof(0108b8ac) = 300126128 (0x11e38fb0) bytes (System.Web.Caching.Cache)
```

Note: Execute the **!objsize** command without any parameters to list all the roots in the process.

For more information about caching in ASP.NET, see "ASP.NET Performance Tips and Best Practices" on the GotDotNet Web site at *http://www.gotdotnet.com/team/asp/ASP.NET Performance Tips and Tricks.aspx*.

Freeing Managed Memory

Free the managed memory and consider the effect on virtual memory.

► **To make sure that everything is being restarted from scratch**

1. Open a command prompt window, and then type **iisreset** at the prompt to restart IIS.
2. Browse to *http://localhost/debugging/memory.aspx*.
3. Run System Monitor and make sure that the following counters are set: **Private Bytes** and **# Bytes in all Heaps** for Aspnet_wp.exe.
4. Click **Allocate 20 MB Objects** once. You'll see output similar to Figure 2.8.

   Note the changes that System Monitor displays. Both Aspnet_wp.exe **Private Bytes** and **# Bytes in all Heaps** counters increase proportionately.

5. Click **Allocate 20 MB Objects** again and note the changes in the System Monitor chart.

   Each time the button is clicked, the **Private Bytes** and **#Bytes in All Heaps** counters increase proportionately.

Figure 2.12 shows the results in System Monitor after **Allocate 20 MB Objects** has been clicked twice. If you switch back to the browser and click **Free Memory**, you can see that the System Monitor log shows **Maximum private bytes** at 222 MB even after the button has been clicked.

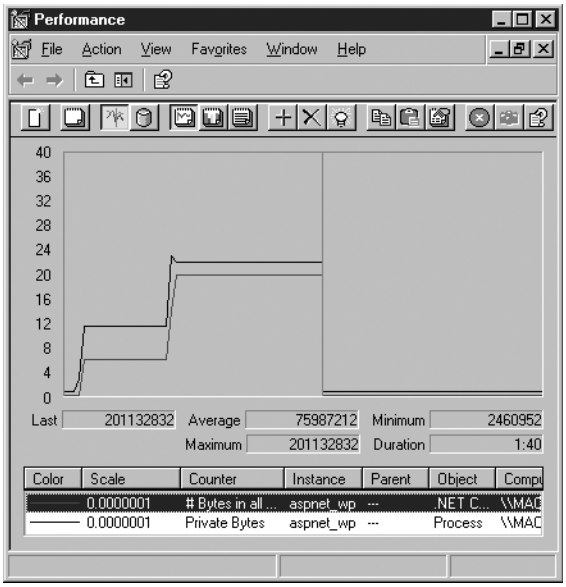

Figure 2.12
System Monitor data after allocating memory

Here's an extract from the code behind the **Free Memory** button:

```
ArrayList arr = (ArrayList)Session["CacheList"];
foreach (string s in arr)
{
   if (Cache[s] != null)
   {
      Cache.Remove(s);
      Label1.Text += "<TABLE BORDER>";
      Label1.Text += "<TR><TD>";
      Label1.Text += "Emptying out cachekey " + s;
      Label1.Text += "</TD></TR></TABLE>";
   }
}
arr.Clear();
Session["CacheList"] = arr;
```

The code also contains a call to **GC.Collect()**. This code clears the cache objects and causes a garbage collection, but it does not result in the virtual memory allocated by large object heap being freed. Figure 2.13 shows the cache being emptied and the reduction of the **GC.TotalMemory** usage, but that the **Private Bytes** remains high.

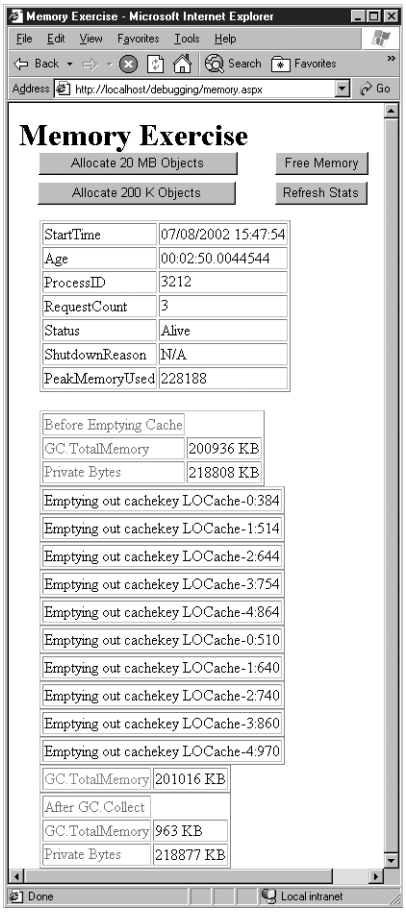

Figure 2.13
Browser data after clicking Free Memory

When the **Free Memory** button was clicked, all but 1 MB of memory was freed. The **# Bytes in all Heaps** counter decreases significantly, but the **Private Bytes** counter does not change because of the large object heap. This is shown in figure 2.14.

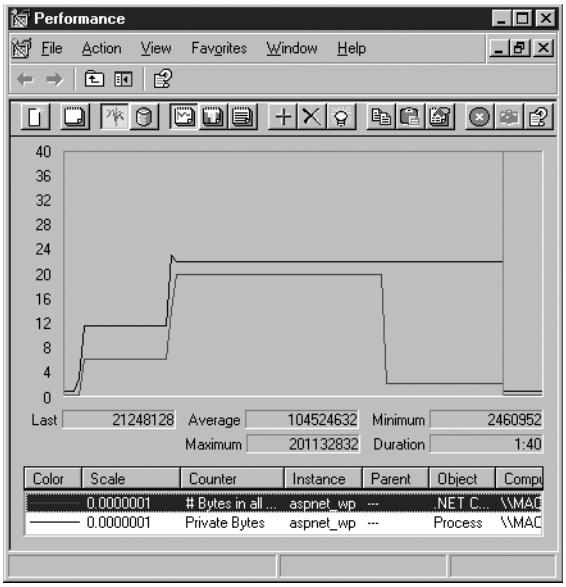

Figure 2.14
System Monitor graph after clicking Free Memory

Capturing a Dump

To learn more about the state of the process, run ADPlus in **-hang** mode to capture a dump of process state:

```
adplus.vbs -hang -pn aspnet_wp.exe -quiet
```

Start WinDbg and load the dump file, set the symbol path, and load and initialize the SOS debugger extension. Now use the **!eeheap -gc** command to find the size of the managed heap.

```
0:000> !eeheap -gc
generation 0 starts at 0x012cc0e4
generation 1 starts at 0x012afde8
generation 2 starts at 0x011c1028
 segment    begin allocated     size
011c0000 011c1028  012d6000 00114fd8(1134552)
Total Size  0x114fd8(1134552)
------------------------------
large block  0x8060(32864)
large_np_objects start at 00000000
large_p_objects start  at 021c0008
------------------------------
GC Heap Size  0x11d038(1167416)
```

The dump shows that the GC Heap Size is 1 MB, not 222 MB, which indicates that all but 1 MB was collected. Because the objects that were allocated were 20 MB, you can conclude that they have all been removed from the heap. The Private Bytes counter for Aspnet_wp is still 222 MB, so you need to investigate this further.

In the debugger, use the **!dumpheap** command with the "-stat" flag. The output is similar to the following:

```
  ...
  d19dd4       79     23608 System.Char[]
 32169e8      382     26304 System.Int32[]
 32174a0      179     39048 System.Collections.Hashtable/bucket[]
  d12f28     1136     46352 System.Object[]
  153bb0       91    146208      Free
 321b278       90    179136 System.Byte[]
  d141b0     6626    472128 System.String
Total 14525 objects
large objects
 Address       MT     Size
 21c0018   e82f28     2064 System.Object[]
 21c0840   e82f28     4096 System.Object[]
 21c1858   e82f28     4096 System.Object[]
 2217cb0   e82f28     2064 System.Object[]
 22184d8   e82f28     4096 System.Object[]
 22194f0   e82f28     4096 System.Object[]
 221a508   e82f28     2064 System.Object[]
 221ad30   e82f28     4096 System.Object[]
 221bd48   e82f28     2064 System.Object[]
 226e930   e82f28     2064 System.Object[]
 226f158   e82f28     2064 System.Object[]
total 11 large objects
```

In summary, the ASP.NET page allocates large objects that are 20 MB, and then caches them in the ASP.NET cache. Because the objects that are allocated are greater than 85 KB, they are allocated on the large object heap. When objects are removed from the cache in code, the managed memory (shown by the System Monitor **# Bytes in all Heaps** counter) is reduced from 200 MB to 1 MB. The private bytes for the process do not decrease, however, and stay at about 222 MB. Although the managed memory was freed, the amount of memory that was allocated by the **VirtualAlloc** API remains allocated to the process.

This is a known limitation of the version 1.0 large object heap. The large object heap grows to accommodate the amount of memory of all objects that are alive at one time. If the objects are freed, the memory is reused for new large allocations; however, the process private bytes does shrink when the objects are freed. This will be addressed in .NET Framework version 1.1.

So far, this chapter has illustrated:

- How managed code can consume excessive memory.
- How and when ASP.NET process recycling might occur.
- Techniques used to debug memory consumption problems.
- How to use tools during the debugging process, such as System Monitor, Task Manager, WinDbg, and SOS.dll.

Diagnosing Memory Leaks with the Allocation Profiler

The Allocation Profiler is another tool that can provide more insight into the causes of memory consumption. This tool examines the managed heap when an application runs and presents a graphical display of memory allocation on the heap. In this section, you use this tool to show what happens when objects greater than 85 KB are allocated on the large object heap. The Allocation Profiler shows where the largest objects are rooted and the corresponding amount of allocated memory.

Before using Allocation Profiler with ASP.NET code, you need to modify the machine.config file and change the username attribute in the <processModel> element from **machine** to **system** to enable privileges that allow you to run the tool. When you have finished using the tool, change the username attribute back to **machine** to avoid the security risk associated with running under the system account.

► To run Allocation Profiler

1. Run Allocationprofiler.exe from *C:\Debuggers\AllocProf*.
2. Select **Profile ASP.NET** from the **File** menu.

   Multiple informational command windows pop up. One of these indicates that the tool is stopping the IIS Admin service, while another indicates that it is starting the World Wide Web server. The Allocation Profiler window title changes to "Stopping IIS." IIS then restarts and the window title changes to "Waiting for ASP.NET worker process to start up." This means that the tool is waiting for the first ASP.NET page to invoke Aspnet_wp.exe.
3. Browse to *http://localhost/debugging/memory.aspx* to start the ASP.NET worker process. The Allocation Profiler window title changes to "Profiling: ASP.NET."
4. Clear the **Profiling active** check box, close the **Allocation Graph for: ASP.NET** window, and then select the **Profiling active** check box.

   The Allocation Profiler now starts gathering allocation statistics about the process.
5. Switch to the browser window and click **Allocate 20 MB Objects** twice.
6. Switch to the Allocation Profiler, and then clear the **Profiling active** check box.
7. Slide the horizontal scroll bar to the right to scroll through the allocations.

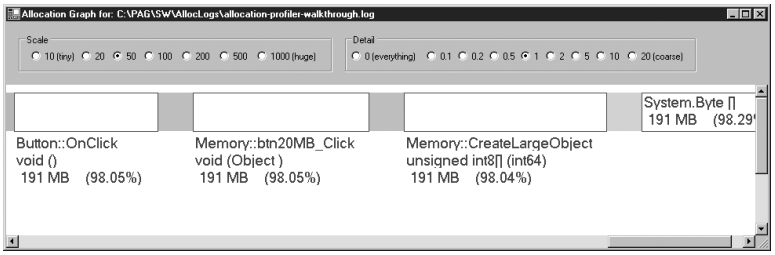

Figure 2.15
Allocation graph

The screen shows that the **Button::OnClick** handler calls the **Memory::btn20MB_Click** method, which creates 191 MB of **System.Byte** array objects.

► **To view the heap graph**

1. Close the **Allocation Graph for: ASP.NET** window. This is the Allocation Graph for ASP.NET.
2. Switch to the Allocation Profiler window, and then click **Show Heap Now**. A new Allocation Profiler window called "Heap Graph for ASP.NET" appears.
3. Scroll to the far right of this window, where you will see the allocation for the **System.Byte**[] object.

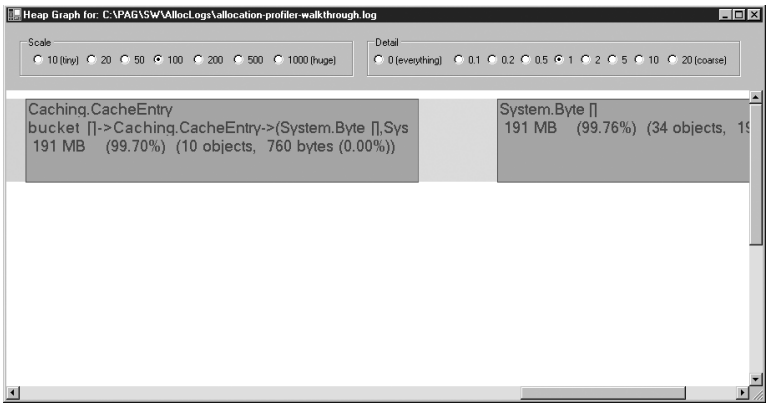

Figure 2.16
Heap graph

The second to last rectangle shows that the **System.Byte**[] object is rooted in the System.Web cache. This is shown by the references in the reference tree to Caching.CacheSingle and Caching.CacheEntry. Note that Caching.CacheEntry was called and contains 191 MB. This indicates that most of the memory is held by the cache.

► To view the histogram

1. Close the graph window.
2. In the Allocation Profiler window, click **Histogram by Age** on the **View** menu.

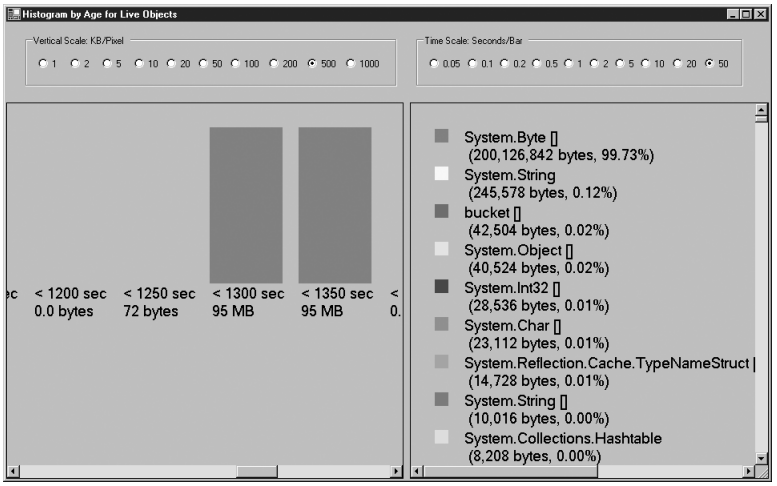

Figure 2.17
Histogram by age

3. In the left pane of the Histogram by Age for Live Objects window, scroll to the right until you see a big red bar.

   The **System.Byte**[] objects are at the top of the list of total memory taken up by Live Objects.
4. Position the mouse arrow over the red bar to obtain more information.
5. Right-click the red bar that represents **System.Byte**[], and then click **Show Who Allocated** on the **Context** menu.

   The original **Allocation graph for ASP.NET** is displayed.

Allocation Profiler can also trace the root tree, showing callers, callees, and which call allocated which memory block.

► To display root tree information

1. In the Allocation Graph for Object window, right-click the **Memory: btn20MB_Click** bar, and then select **callers & callees.** This highlights the entire call tree.
2. Right-click the highlighted area, click **Copy as text to clipboard**, and then past the text in Notepad.

The following information is displayed:

```
<root>: 191 MB    (100.00%)
 System.Web.Hosting.ISAPIRuntime::ProcessRequest:    191 MB    (100.00%)
  System.Web.HttpRuntime::ProcessRequest:    191 MB    (100.00%)
   System.Web.HttpRuntime::ProcessRequestInternal:   191 MB    (100.00%)
    System.Web.HttpApplication::System.Web.IHttpAsyncHandler.BeginProcessRequest:
191 MB    (99.96%)
     System.Web.HttpApplication::ResumeSteps:        191 MB    (99.96%)
      System.Web.HttpApplication::ExecuteStep:       191 MB    (99.96%)
       CallHandlerExecutionStep::Execute:    191 MB    (99.96%)
        System.Web.UI.Page::ProcessRequest:   191 MB    (99.96%)
         System.Web.UI.Page::ProcessRequest:  191 MB    (99.96%)
          System.Web.UI.Page::ProcessRequestMain:    191 MB    (99.96%)
           System.Web.UI.Page::RaisePostBackEvent:   191 MB    (99.94%)
            System.Web.UI.Page::RaisePostBackEvent:  191 MB    (99.94%)
         System.Web.UI.WebControls.Button::System.Web.UI.IPostBackEventHandler.RaisePostBac
kEvent:  191 MB    (99.94%)
       System.Web.UI.WebControls.Button::OnClick:    191 MB    (99.94%)
              Debugging.Memory::btn20MB_Click:       191 MB    (99.94%)
                                  ...
                           System.Byte []:           191 MB    (100.00%)
                             <bottom>:    191 MB    (100.00%)
```

This tree shows how the call from **btn20MB_Click** created the large object.

When you have finished running the Allocation Profiler, click **Kill ASP.NET**. Change the username attribute on the <processModel> element in machine.config from **system** back to **machine** to restrict privileges again, avoiding the security risk that running under **system** presents.

Allocation Profiler examined the managed heap and showed that the 20-MB large object was created by the **btn20MB_Click** event. The actual code calls from **btn20MB_Click** to **CreateLargeObject**, which creates **System.Byte**[]. With the Release build, **CreateLargeObject** does not appear due to optimization or inlining. This is discussed in Chapter 3, "Debugging Contention Problems." A Debug build would show the call from **btn20MB_Click** to **CreateLargeObject**. With both builds, you can see that the large object was rooted (and not collectable by the GC) and that a considerable amount of memory was allocated for the large objects' existence.

Conclusion

In this chapter, you have:

- Seen how to use Microsoft tools to diagnose and debug memory allocation problems.
- Learned how objects are allocated on managed heaps.
- Learned how .NET garbage collection works.

Task Manager and the System Monitor let you gather evidence that helps in the diagnosis of memory leaks. WinDbg, in conjunction with the SOS extension, allows you to pinpoint where managed memory is allocated in an application. The Allocation Profiler can be used to show how managed memory has been allocated, and also can be used to find object roots.

Finally, you have seen that you need to think carefully before using objects that are larger than 85 KB in ASP.NET applications because the memory in .NET Framework version 1.0 the large object heap is not reclaimed by the GC.

3

Debugging Contention Problems

This chapter describes how to approach debugging when ASP.NET fails to respond to browser requests. Causes may include a long-running request, or threads that deadlock or contend for a resource. Focusing on the latter, a walkthrough shows how to debug in a scenario with some controls that wait on a shared event. To set the scene, "ASP.NET Thread Pools" discusses .NET threading, the common language runtime thread pool, and how ASP.NET uses the thread pool. After the WinDbg walkthrough, there is an example that uses the Microsoft® Visual Studio® .NET debugger. For information about using CorDbg, see the Appendix.

Although reproducing this problem on your machine may not give you the exact same results because of differing operating systems and runtime versions, the output should be similar, and the debugging concepts are the same. Also, because all .NET Framework applications use the same memory architecture, you can apply what you learn about memory management in an ASP.NET context to other .NET architectures, such as console applications, Microsoft Windows® operating system applications, and Windows services.

ASP.NET Thread Pools

ASP.NET takes advantage of the .NET thread pool, thus efficiently using a pool of threads managed by the common language runtime. This section briefly describes the building blocks that ASP.NET needs to use that thread pool. First, managed threads and application domains (or AppDomains) are defined. Then the .NET thread pool, the ThreadPool manager, and various thread types involved are discussed. Understanding the thread types helps you to interpret the dump files shown in the subsequent walkthrough. As the information unfolds, you see how ASP.NET uses the .NET thread pool and handles synchronous and asynchronous calls.

Managed Threads and AppDomains

.NET threading builds upon the existing implementation of native threads and processes by using managed threads and AppDomains. *Managed threads* are .NET common-language runtime classes that are bound to native threads. Not all native threads in a process can run managed code; consequently, managed threads are a subset of the native threads in a process. *AppDomains* can be thought of as lightweight processes that isolate the managed code running in them. A native process can contain multiple AppDomains, and each AppDomain contains assemblies and managed threads. For more information, read the .NET Framework SDK Help entries for System.AppDomain and System.Theading.Thread. Figure 3.1 shows how multiple threads can exist in an AppDomain, and how a process consists of one or more AppDomains.

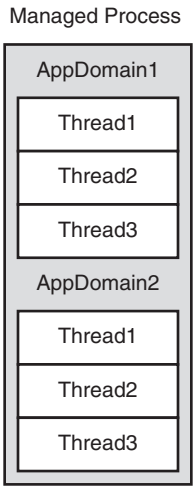

Figure 3.1
A managed process

The .NET ThreadPool

A *thread pool* is a mechanism that allows the queuing and processing of work items using a dynamically load-balanced group of threads. Using a thread pool might be more efficient than creating a thread for each individual work item because the time needed to create, start, and tear down the thread when it exits is relatively expensive. Note that there is one thread pool per AppDomain.

In addition, managing the thread's lifetime and deciding when to add new threads can be a complex task. .NET provides built-in support for using a thread pool from managed code. Developers can write code to implement callback functions that process one work item at a time using the thread pool. For more information,

see "Thread Pooling" in the .NET Framework Developer's Guide on the MSDN Web site at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthreadpooling.asp?frame=true*.

.NET implements the entire pooling and load-balancing mechanism using the native **ThreadPoolMgr** class. The **ThreadPoolMgr** class has five main thread types, and only the first two of these can run managed code. The **ThreadPoolMgr** types are:

- Completion port thread
- Worker thread
- Gate thread
- Wait thread
- Timer thread

A completion port thread executes the callback function after an item is queued by **QueueUserWorkItem()** (for more details, see the .NET Framework SDK). If the thread pool manager determines that there are not enough completion port threads available, the work item is sent to the work request queue. A worker thread services the work request queue, dequeues items, and executes the work item. To ensure that there are enough free worker threads available, a gate thread monitors the health of the thread pool. The gate thread is responsible for injecting and retiring worker and input/output (I/O) threads based on factors such as CPU utilization, garbage collection frequency, and work queue size.

The thread pool manager uses a wait thread to wait on a synchronization object. To use a wait thread, a developer can call **ThreadPool.RegisterWaitForSingleObject()**. This function attempts to find a wait thread and creates one if it does not exist. A common example of .NET Framework base classes that uses wait threads to monitor synchronization objects is the **System.Threading.WaitHandle** class.

.NET uses timer threads internally to implement the timer callback functionality of the **System.Threading.Timer** class.

ASP.NET Thread Pool

The **wmain()** function marks the beginning and end of the Aspnet_wp.exe process execution. As the entry point into Aspnet_wp.exe, the **wmain()** function first initializes the .NET thread pool. To do this, **wmain()** calls functions that choose the common language runtime flavor (which can be either workstation or server), load the common language runtime, and obtain an interface to the thread pool interface. Once this function obtains the interface for the thread pool, it can configure the number of completion port threads and the number of worker threads by querying the <processModel> and setting the appropriate values for **maxIoThreads** and **maxWorkerThreads**.

Once the process initializes the common language runtime and the thread pool, the **wmain()** function sets up the named pipes that it uses to communicate with Microsoft Internet Information Services (IIS). ASP.NET uses two named pipes to communicate with IIS: a synchronous pipe and an asynchronous pipe. The synchronous pipe makes blocking calls back to IIS. It is primarily used as a channel to obtain Internet Server Application Programming Interface (ISAPI) server variables from IIS.

The primary function of the asynchronous pipe is to transport requests from IIS to ASP.NET. In order to accomplish this, the initialization routine links the asynchronous pipe handle to a completion port callback within Aspnet_isapi.dll. When work arrives for this pipe, the .NET common language runtime calls the function called **aspnet_isapi!CorThreadPoolCompletionCallback()**. This function can be seen when viewing the native stack trace of multiple threads in the debugger.

CorThreadPoolCompletionCallback() invokes the callback function using the parameters passed. In the asynchronous pipe manager class, a completion callback function calls a function to process the message sent from IIS. The message processing function examines the request and locates the AppDomain that the request should be routed to. Once located, the AppDomain is queried for a pointer to the managed **HttpRuntime** object that corresponds to the current AppDomain. Once a pointer is obtained, the **HttpRuntime.ProcessRequest()** function is called and the request enters managed execution.

The **HttpRuntime** object processes the worker request. It decides whether it will queue the request for a worker thread or execute the request on the current thread that it is running. It does this based on conditions such as "Is this request a local request?", "Are there free completion port threads available?", and "Is this the first request to this AppDomain?"

If this is the first request, the ASP.NET request queue needs to be initialized based on the values for **minFreeThreads**, **minLocalRequestFreeThreads**, and **appRequestQueueLimit**. These values are specified in the machine.config or web.config files for the Web application.

Scenario: Contention or Deadlock Symptoms

Now that you understand some details of .NET threading and the thread pool, and how ASP.NET uses those services provided by the common language runtime, let's look at how to debug cases where ASP.NET fails to respond to browser requests. At first, debugging "hang" symptoms appears to be nebulous. You usually don't

know the immediate cause of the hang, so it's more difficult to determine what led to the results. You look at the big picture first and then narrow down the possibilities. This scenario considers what aspects of the managed code could be causing this behavior, looks at both the native and managed threads, and then focuses on the latter. Tools such as System Monitor (known as Performance Monitor in Windows 2000) may help diagnose the problem, but to debug the problem, analyzing the dump file and the source code proves most efficient.

Sometimes you may observe behavior that displays deadlock symptoms (for example, when requests do not complete because they are waiting on external resources). However, a true deadlock occurs when one thread obtains a lock and requests another lock that has already been obtained by a second thread. The second thread is waiting for the lock that the first thread owns. What a dilemma! Neither thread can execute. Both threads stop responding and appear to hang. This walk-through focuses on problems that may not be "true" deadlock scenarios.

Note: For more information on deadlocks, see article Q317723, "What Are Race Conditions and Deadlocks?" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q317723*.

Here are some real customer scenarios that could indicate contention problems:

- **Scenario 1:** A customer hosts ASP.NET sites. Their client says that pages are not returning, and/or they are monitoring the application event logs and notice that they are continually getting errors because the Aspnet_wp.exe process is recycled.
- **Scenario 2:** There is one ASP.NET application on one server. The application is facing deadlock symptoms when customers run a particular function on a page (for example, clicking the logon button on the logon page). Some internal logic causes the page to be unavailable.

Breaking Down the Thought Processes

The flowchart on the next page presents common thought processes that you can follow when troubleshooting resource contention problems.

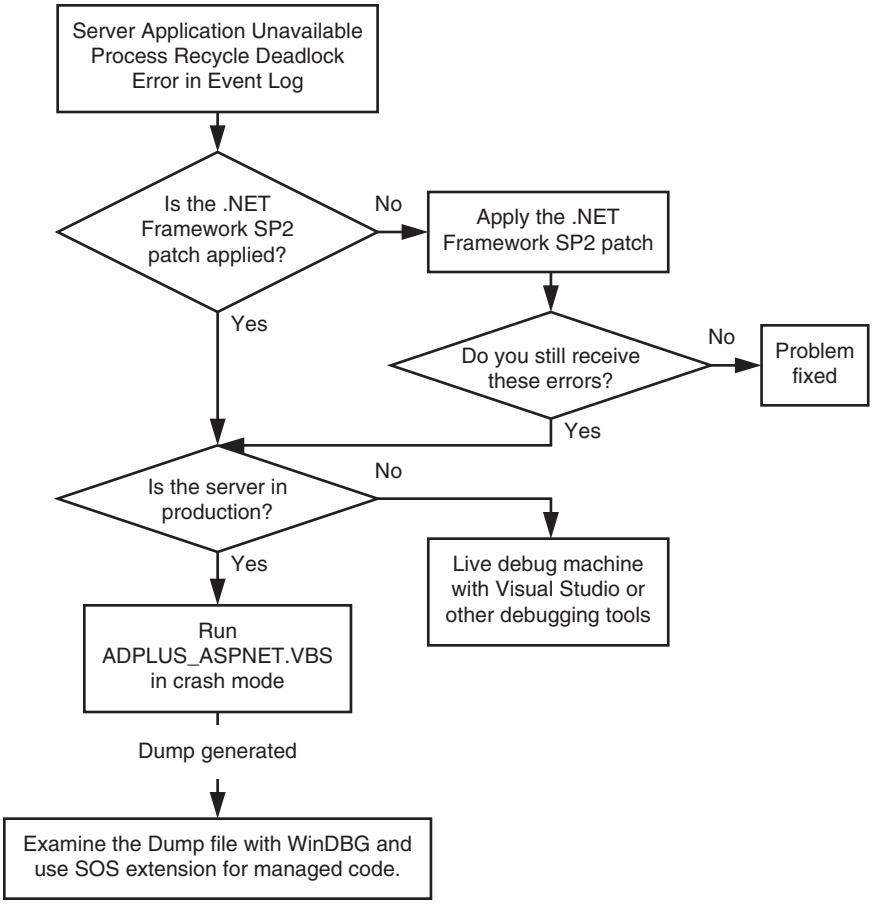

Figure 3.2
Thought flowchart

Many questions can arise. Are you dealing with hanging requests or processes? Will you be notified? With ASP.NET, you should see an error in the application event log. With other technologies, you may or may not see an error posted to the log.

Although the thought processes are illustrated in the walkthrough that follows, let's look at the flowchart in more detail.

Error in the Browser and/or Application Event Log

Errors may surface in both the application event logs and the browser. The **responseDeadlockInterval** attribute is a configuration setting in the machine.config file set to the default interval of three minutes. If the Aspnet_wp.exe process has deadlocked and the three-minute time limit has passed, the deadlock-detection mechanism in IIS recycles the process and posts the following error in the application event log: "aspnet_wp.exe was recycled because it was suspected to be in a deadlocked state. It did not send any responses for pending requests in the last 180

seconds." A Server Application Unavailable error is often received by the browser because Aspnet_wp.exe is not responding.

Deadlock Error Occurs: Apply .NET Framework Service Pack 2

When a deadlock error surfaces, make sure that the .NET Framework Service Pack 2 (SP2) has been applied. SP2 addresses a problem that causes the ASP.NET health monitor to think a process is executing when it's not. This service pack fixes the request processing logic so there isn't a phantom request.

For more information, see article Q321792, "ASP.NET Worker Process (Aspnet_wp.exe) Is Recycled Unexpectedly" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q321792*.

Deadlock Error Still Occurs

If the .NET Framework SP2 has been applied and the error still occurs, this means that the ASP.NET process is hung and you need to get a dump file. If the project is in a controlled environment (development or stress mode) and you can reproduce the problem, debugging choices include performing a live debug with CorDbg, Content Debugger (CDB), and SOS.dll, or remotely using Visual Studio .NET.

If the project is in a production environment and you can reproduce the problem, you can run Autodump+ (ADPlus) in **–hang** mode to produce a full memory dump of the Aspnet_wp.exe process. The problem with this approach is that you may not know how much time will pass before the error happens. Also, once the error does happen, you only have three minutes to run ADPlus. To allow more time to break in on the problem, you can extend the **responseDeadlockInterval** attribute of the <processModel> element in machine.config to a longer time period (maybe nine minutes instead of three). Keep in mind that the longer the time period for **responseDeadlockInterval**, the longer a deadlocked server can appear to be unavailable. Choose a time period that makes sense for the application. Also, if you want to disable the process model deadlock detection mechanism, set **responseDeadlockInterval** and **responseRestartDeadlockInterval** to infinite.

.NET Framework version 1.1 includes a **DebugOnDeadlock** registry setting that breaks into Aspnet_wp.exe and/or InetInfo.exe. This makes it easier to run ADPlus in **–crash** mode and create a dump file (the **DebugOnDeadlock** setting would be 1). This feature is also available through a hotfix. For more information, see article Q325947, "Event ID 1003 with ASP.NET Deadlock" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q325947*.

If you can't apply this hotfix, an alternative is to use ADPlus_AspNet.vbs to create a full dump when a deadlock occurs. ADPlus_AspNet.vbs is a customized version of ADPlus that sets a breakpoint and creates a dump file before the process is killed. In this example, AdPlus_AspNet.vbs runs in **-crash** mode when the process is told to recycle by the process model health monitoring.

If you use ADPlus through Terminal Server, consider the following: A native debugger cannot attach to a process running in another WinStation. Consequently, ADPlus cannot run in **-crash** mode because it scripts the CDB.exe debugger. Instead, you can use Task Scheduler to create a Remote.exe command window that has access to the system processes. For more information, see article Q323478, "You Cannot Debug Through a Terminal Server Session" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q323478*.

Process Appears to Hang, but There Is No Error

In many scenarios, you may have deadlock symptoms, but no error. For example, the server may hang and not respond. If this is the case, follow normal troubleshooting procedures first. For example, ask what else is not working. Is the request blocked in IIS or is it blocked because of third-party code running in InetInfo? If it is one of these problems, create a dump file and either examine it, or send it to Microsoft Product Support to examine.

If the problem is that the ASP.NET pages are not rendering, consider the reasons why an error message does not show. Here are some possibilities:

- ASP.NET may not know that the deadlock symptoms occurred because the worker process has not sent back a response. The process may hang before the time-out occurs.
- The debugger attached to the Aspnet_wp.exe process and disabled the deadlock detection.
- The **responseDeadlockInterval** was altered and set too high.

Examining the Dump Files

To examine dump files, you need to start WinDbg, set up your symbols as appropriate (for more information, see "Debugging" later in this chapter), and then analyze the dump file. While debugging, examine managed threads, native threads, and call stacks. If threads are processing requests at the time of the dump, examine those threads to discover why the request is not responding.

Using the SOS.dll extension, consider the thread pool information, including CPU utilization and the number of worker and completion port threads available. Identify the calling functions in the stack traces and determine if they are blocked.

Contention Walkthrough

The following walkthrough describes a simplified, realistic scenario. The purpose is to illustrate techniques to help you troubleshoot a contention problem. A production environment scenario may not be as clear cut as the contention walkthrough scenario, but you can apply similar techniques to help identify the causes of contention.

The example that is shown in this walkthrough demonstrates threads waiting on a shared resource. More specifically, this code blocks a thread by calling

WaitHandle.WaitOne() with an infinite time-out. The object contains static members, but it could have been shared in session or application scope. For simplicity, all shown requests come from one browser. A more realistic scenario would include requests from multiple browsers and multiple clients.

In this scenario, you will perform the following steps:

1. Browse to the ASP.NET page at *http://localhost/debugging/contention.aspx* and consider the values displayed in the process model table.
2. Follow the thought processes in the flowchart and find the errors presented both in the browser and the application event log.
3. Create a dump file and examine the dump data using WinDbg and SOS.dll.
4. Examine the native threads, managed threads, and call stacks.
5. Consider which threads are bound to which AppDomain, and the different stages of execution.
6. After you've found evidence in the dumps for possible problem areas in the code, look at the source code.

Baseline View

The following browser view provides a baseline from which to compare data after a deadlock occurs.

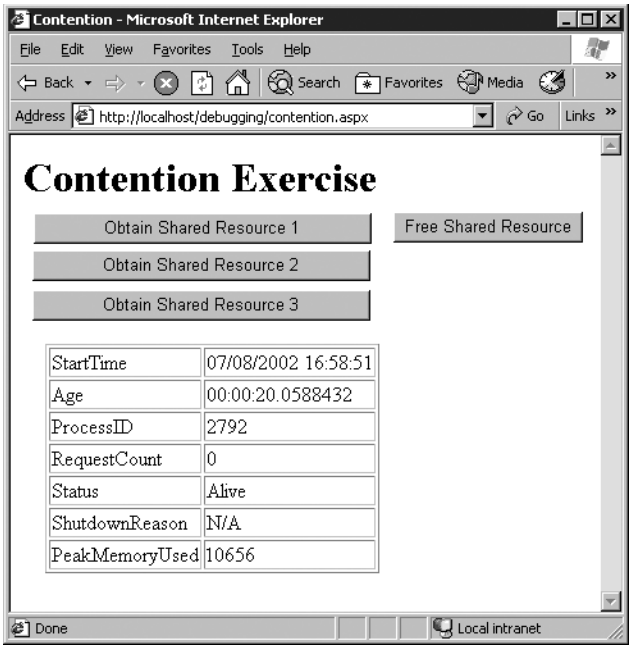

Figure 3.3
Baseline browser data

Note the information displayed, especially the data in the table. Some of the fields are self-explanatory, but descriptions are provided in the following table.

Table 3.1: Description of fields in contention.aspx

Field	Description
StartTime	The time at which this Aspnet_wp.exe process started
Age	The length of time the process has been running
ProcessID	The ID assigned to the process
RequestCount	Number of completed requests, which is initially zero
Status	The current status of the process: if **Alive**, the process is running; if **Shutting Down**, the process has begun to shut down; if **ShutDown**, the process has shut down normally after receiving a shut down message from the IIS process; if **Terminated**, the process was forced to terminate by the IIS process
ShutdownReason	The reason why a process shut down: if **Unexpected**, the process shut down unexpectedly; if **Requests Limit**, requests executed by the process exceeded the allowable limit; if **Request Queue Limit**, requests assigned to the process exceeded the allowable number in the queue; if **Timeout**, the process restarted because it was alive longer than allowed; if **Idle Timeout**, the process exceeded the allowable idle time; if **Memory Limit Exceeded**, the process exceeded the per-process memory limit
PeakMemoryUsed	The maximum amount of memory the process has used. This value maps onto the **Private Bytes** (maximum amount) count in System Monitor

You can explore the code and read through the comments by opening another instance of Contention.aspx.cs in Visual Studio .NET or Notepad.

The code behind each **Obtain Shared Resource** button on the user interface calls a function that blocks a thread by a call to **WaitHandle.WaitOne()** with an infinite time-out. More specifically, the code first creates a **ManualResetEvent** object in a nonsignaled state. Each time you click an **Obtain Shared Resource** button, code for **SharedResource.Wait()** is invoked, which calls **ManualResetEvent.WaitOne()**. The call to **WaitOne()** blocks the current thread until the **WaitHandle()** function receives a signal.

Clicking the **Free Shared Resource** button invokes **SharedResource.Free()**, which calls **ManualResetEvent.Set()**, which in turn sets the state of the specified event to **signaled**. Because **Free()** leaves **ManualResetEvent** in a signaled state, you won't be able to reproduce the blocking scenario. To be able to restart the test, the **Free Shared Resource** button also calls **SharedResource.Reset()**, which sets the state of the **ManualResetEvent** to **nonsignaled**.

► To create and view the error

1. Open a browser window, and then browse to *http://localhost/Debugging /contention.aspx*.
2. Click each **Obtain Shared Resource** button in order.
3. After three minutes, you should see an error appear in the browser window and in the application event log.

   The server returns the following error message:

   Server Application Unavailable

   The web application you are attempting to access on this web server is currently unavailable. Please hit the "Refresh" button in your web browser to retry your request.

   Administrator Note: An error message detailing the cause of this specific request failure can be found in the application event log of the Web server. Please review this log entry to discover what caused this error to occur.

4. Close the browser window. The application event log contains the following error, which you can display using the event viewer:

```
Event Type:      Error
Event Source:    ASP.NET 1.0.3705.0
Event Category:  None
Event ID:        1003
Date:            6/5/2002
Time:            9:32:49 PM
User:            N/A
Computer:        machineName
Description:
aspnet_wp.exe  (PID: 2248) was recycled because it was suspected to be in a
deadlocked state. It did not send any responses for pending requests in the
last 180 seconds.
```

   More specifically, Aspnet_isapi.dll (running in InetInfo.exe) reports the error that the browser and the application event log display. Requests that wait for a response cause the error.

5. Open a new browser window, and then browse to *http://localhost/debugging /contention.aspx*.
6. Click each **Obtain Shared Resource** button again.

The data in your browser should resemble the following table:

Table 3.2: Contention.aspx results

Field	Baseline value	New value
StartTime	07/08/2002 16:58:51	07/08/2002 17:05:32
Age	00:00:02.0588432	00:06.43.0472853
ProcessID	2,792	2,792
RequestCount	0	4
Status	Alive	Terminated
ShutdownReason	N/A	Deadlock Suspected
PeakMemoryUsed	10,656	11,248

The **RequestCount** increments to four—one to browse the page for viewing and then three for the button clicks. The buttons are ASP.NET server controls that execute code on the server. Every button click is a round trip to the server, and you can see this happening when you examine the call stacks later in this walkthrough.

The **ShutdownReason** field shows that a deadlock is suspected. As described in the "IIS 5.*x* Process Model" section of Chapter 1, "Introduction to Production Debugging for .NET Framework Applications," the process model health monitoring launches a new Aspnet_wp.exe process and terminates the old process.

Before troubleshooting further, restart IIS (by running IISReset.exe) in the command window to reinitialize the PMInfo counters.

Debugging a Dump File with WinDbg

The goal is to diagnose why the process was restarted. There may have been queued requests and/or a lack of response during the default three-minute interval set by **responseDeadlockInterval**. A dump file provides a snapshot of what occurs when the process is recycled. Because the interval may allow minimal time for the creation of the dump file, you need to modify the registry to allow debugging before the process recycles.

The ASP.NET deadlock detection mechanism is automatically disabled when a native debugger is attached to Aspnet_wp.exe, but it would be beneficial to obtain a dump before the process recycles. To circumvent this problem, you must first modify the registry.

Note: The following exercise requires that you make changes to the registry. Use caution — it is always recommended that you back up the registry before making changes. In addition, when you have finished debugging, it is recommended that you delete the DWORDs created below. These registry DWORDs are undocumented, and should only be used during specific debugging sessions.

► To modify the registry

1. Run Regedit, and then locate the key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ASP.NET.
2. Right-click **ASP.NET**, and then select **New DWORD Value** from the shortcut menu.
3. Under **Name**, change the new **New Value #1** entry to **UnderDebugger**. Keep this data value as zero.

   When **UnderDebugger** is zero, the deadlock detection mechanism works normally.

Figure 3.4 shows how the new value should appear.

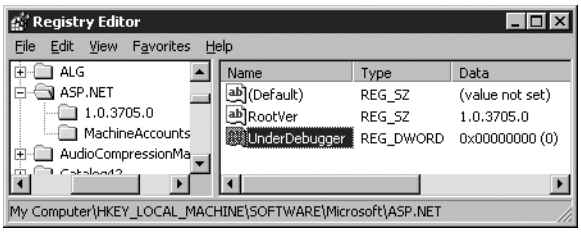

Figure 3.4
New registry key

To create a dump for this exercise, you use a customized version of ADPlus.vbs called ADPlus_AspNet.vbs. The changes to ADPlus are located in the **CreateCDBScript()** function. This is the function that is run to create the .cfg file that in turn configures the CDB debugger when it attaches to the process.

ADPlus_AspNet.vbs includes the following changes to the ADPlus.vbs script:

```
objTextFile.Writeline "* -- Setup Symbol Path to use external symbol server ---"
objTextFile.Writeline ".sympath+ SRV*c:\
symbols\debugginglabs*http://msdl.microsoft.com/download/symbols;C:\symbols\
debugginglabs"
objTextFile.Writeline "* ------------------------------------------------------"
objTextFile.Writeline "*"
objTextFile.Writeline "*"
objTextFile.Writeline "*"
objTextFile.Writeline "* -- Make sure the symbols are loaded for aspnet_wp.exe ---"
objTextFile.Writeline "x aspnet_wp!CAsyncPipeManager::Close"
objTextFile.Writeline "* ------------------------------------------------------
---"
objTextFile.Writeline "*"
objTextFile.Writeline "*"
objTextFile.Writeline "*"

'Since this is a breakpoint, any \ need to be escaped with \\
Dim EscapedCrashDir
EscapedCrashDir = Replace(ShortCrashDir, "\", "\\")
```

```
objTextFile.Writeline "* -- Add a breakpoint for CAsyncPipeManager::Close ---"
objTextFile.Writeline "bp aspnet_wp!CAsyncPipeManager::Close " & Chr(34) & _
".echo Encountered aspnet_wp!CAsyncPipeManager::Close breakpoint. Dumping process
...;.dump /mfh "  _
& EscapedCrashDir & "\\" & "PID-" & pid & "__" & packagename &
"__Recycle__full.dmp;q" & Chr(34)
```

The changes to ADPlus.vbs show how easy it is to adapt this file to your needs. If you do make these changes, a good practice is to rename the new script file to differentiate it from ADPlus.vbs.

Make sure that ADPlus_AspNet.vbs is located in the folder where you installed the debugging toolkit; by default, this is C:\Debuggers. Symbols are required, so make sure that you are connected to the Internet or that you have downloaded the symbols for ASP.NET locally, especially Aspnet_wp.pdb. Internet access is adequate because this tool sets up the path to the Microsoft external symbol server for you.

As noted in the changes, ADPlus_AspNet.vbs sets the symbol path to use the Microsoft public symbol server. The tool verifies that the symbols can be loaded for **aspnet_wp!CAsyncPipeManager::Close**.

This function is called when the process is shut down because of one of the conditions specified by the process model (for deadlocks). The tool sets a breakpoint on **aspnet_wp! CAsyncPipeManager::Close()** to create a full dump file when the breakpoint is hit.

Note: The WinDbg and CDB debuggers can set breakpoints on addresses, function names, or source file lines. The syntax to set a breakpoint on a function is **bp module_or_exe_name!function_name**.

Using this automated version of ADPlus is easier in development mode or in a controlled environment, such as stress testing. If you did not use this automated version, you would have to monitor the Aspnet_wp.exe process continually until the recycle occurs. In that case, you would run ADPlus in **-hang** mode, and timing would be tricky.

The use of the customized ADPlus script is also easier to use in production mode. You don't have to continually monitor the process; it does this for you, and it creates a dump just before the process terminates. However, if you run in **-crash** mode, which is invasive, the process is terminated when the debugger exits.

Along with a user dump, the ADPlus_AspNet.vbs script also generates a text file of the exception history, dumping the native stack traces for those exceptions (like traditional ADPlus.vbs). The default configuration of ADPlus means that for every exception, a minidump file is generated and a stack trace is logged. Obviously, when more exceptions are thrown, a larger amount of text is generated. Users have been known to run out of disk space when running this tool.

If running in production mode, throwing as few exceptions as possible is recommended. You can even disable the creation of the log file if it is too obtrusive. For more information, see "ASP.NET Performance Tips and Best Practices" on the GotDotNet Web site at *http://www.gotdotnet.com/team/asp/ASP.NET Performance Tips and Tricks.aspx*.

► To start the exercise again

1. Open a command prompt window, and then type **iisreset** at the prompt to restart IIS.
2. Open a new browser window, and then browse to *http://localhost/debugging/contention.aspx*.
3. After Aspnet_wp.exe loads, run ADPlus_AspNet.vbs against Aspnet_wp.exe in **-quiet** mode at the command line using the following command:

```
adplus_aspnet.vbs –pn aspnet_wp.exe –crash –quiet
```

> **Note:** The **-quiet** switch ensures that you can still create a dump file if the symbol path for ADPlus is not set.

You should see the following output:

```
C:\debuggers>adplus_aspnet.vbs -pn aspnet_wp.exe -crash -quiet
The '-crash' switch was used, Autodump+ is running in 'crash' mode.
The '-quiet' switch was used, Autodump+ will not display any modal dialog boxes.

Monitoring the processes that were specified on the command line
for unhandled exceptions.
------------------------------------------------------------------------
Attaching the CDB debugger to: Process - ASPNET_WP.EXE with PID - 360
```

You should also see a minimized window running Cdb.exe. The tool waits until the deadlock is detected and then creates the dump and text files. As noted in the "IIS 5.x Process Model" discussion on ASP.NET architecture in Chapter 1, ASP.NET has a standard ISAPI extension (Aspnet_isapi.dll) that runs in-proc within InetInfo.exe and uses named pipes to connect to an ASP.NET worker process (Aspnet_wp.exe). When a deadlock is detected, InetInfo.exe closes down the named pipes to Aspnet_wp.exe. When this pipe is closed, the custom breakpoint on **CAsychPipeManager::Close** is hit. The breakpoint command is then executed and the log entry and a full dump file is generated.

► To create the dump file

1. Switch to the browser that displays Contention.aspx and click each **Obtain Shared Resource** button in order.
2. Wait three minutes for the browser errors and event-log errors to be generated. When the debugger window disappears, the full dump is created.

You can check the debugging toolkit installation folder for the most recent \Crash_Mode folder. The file's name will be similar to **PID-360__ASPNET_WP.EXE__Recycle__full.dmp**.

► To examine the dump file using WinDbg

1. On the **Start** menu, point to **Debugging Tools for Windows**, and then click **WinDbg**.
2. On the **File** menu, click **Open Crash Dump**.
3. Select the appropriate dump file, and then click **Open**.
4. If you are prompted to "Save Base Workspace Information," click **No**.
5. If a Disassembly window pops up, close the window, and then on the **Window** menu, click **Automatically Open Disassembly**.

Symbol paths must be entered for the files that are used by the debugger to analyze the dump file. The symbol file versions that are needed on the debug computer should match the version on the system that produced the dump file. Include symbols from the .NET Framework SDK, Visual Studio .NET, and symbols shipped with the following System32 folder: \%WINDIR%\system32.

To enter the symbol paths, do one of the following:

- From the WinDbg command line, type:

```
.sympath SRV*C:\symbols\debugginglabs*http://msdl.microsoft.com/download/
symbols;C:\
symbols\debugginglabs;C:\Program Files\Microsoft Visual Studio
.NET\FrameworkSDK\symbols;C:\windows\system32
```

Note: If you have only installed the .NET Framework SDK (without Visual Studio .NET), then you need to replace C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK \symbols with C:\Program Files\Microsoft.NET\FrameworkSDK\symbols.

- Create a new environment variable called_NT_SYMBOL_PATH with a value of:

```
C:\symbols\debugginglabs*http://msdl.microsoft.com/download/symbols;
C:\symbols\debugginglabs;C:\Program Files\Microsoft Visual Studio
.NET\FrameworkSDK\symbols;C:\windows\system32
```

- On the **File** menu in WinDbg, click **Symbol File Path**, and then type:

```
SRV*C:\symbols\debugginglabs*http://msdl.microsoft.com/download/symbols;C:\
symbols\debugginglabs;C:\Program Files\Microsoft Visual Studio
.NET\FrameworkSDK\symbols;C:\windows\system32
```

The "SRV" in the path tells WinDbg to go to an external symbol server and copy symbols into the local symbol cache.

► **To use these same symbol paths for other dumps**

- On the **File** menu in WinDbg, click **Save Workspace As**, and then type a name for the saved paths, such as.**NET Debugging Symbols**.

Examining Native Data

First, consider how many threads you are dealing with. In WinDbg, type tilde (~) in the command line; this displays all threads in the current process. Here you have twelve threads, but you don't know yet how many are native and how many are managed.

Each thread is in Suspend state, and is unfrozen. The WinDbg online Help explains that "Each thread has a suspend count associated with it. If it is one or higher, the system will not run the thread. If it is zero or lower, the system will run the thread when appropriate." The following are the twelve threads:

```
0:000> ~
.  0  id: 168.248   Suspend: 1 Teb 7ffde000 Unfrozen
   1  id: 168.420   Suspend: 1 Teb 7ffdd000 Unfrozen
   2  id: 168.ab0   Suspend: 1 Teb 7ffdc000 Unfrozen
   3  id: 168.a18   Suspend: 1 Teb 7ffdb000 Unfrozen
   4  id: 168.9c4   Suspend: 1 Teb 7ffda000 Unfrozen
   5  id: 168.d84   Suspend: 1 Teb 7ffd9000 Unfrozen
   6  id: 168.2ec   Suspend: 1 Teb 7ffd8000 Unfrozen
   7  id: 168.7b0   Suspend: 1 Teb 7ffd7000 Unfrozen
   8  id: 168.3c8   Suspend: 1 Teb 7ffd6000 Unfrozen
   9  id: 168.a4    Suspend: 1 Teb 7ffd5000 Unfrozen
  10  id: 168.eb8   Suspend: 1 Teb 7ffd4000 Unfrozen
  11  id: 168.11c   Suspend: 1 Teb 7ffaf000 Unfrozen
  12  id: 168.894   Suspend: 1 Teb 7ffae000 Unfrozen
  13  id: 168.e94   Suspend: 1 Teb 7ffad000 Unfrozen
```

The WinDbg online Help also says "A thread can also be frozen by the debugger. This is similar to suspending the thread in some ways. However, it is solely a debugger setting; nothing in Windows itself will recognize anything different about this thread. All threads are unfrozen by default. When the debugger causes a process to execute, threads that are frozen will not execute. However, if the debugger detaches from the process, all threads will unfreeze."

Look at the native call stacks for all threads that were running when the dump occurred. The call stack is a data structure that tracks function calls and the parameters passed into these functions. The **~*k** command shows 20 stack frames by default, the **~*** command displays all threads, and the **k** command displays the stack frame of the given thread.

Type **~*k 200** to display the entire call stacks for all twelve threads (given that each has under 200 stack frames).

```
0:000> ~*k 200

.  0  id: 168.248   Suspend: 1 Teb 7ffde000 Unfrozen
ChildEBP RetAddr
0012f874 77f7e76f SharedUserData!SystemCallStub+0x4
0012f878 77e775b7 ntdll!NtDelayExecution+0xc
0012f8d0 77e61bf1 kernel32!SleepEx+0x61
0012f8dc 00422c17 kernel32!Sleep+0xb
0012ff44 004237db aspnet_wp!wmain+0x30b [e:\dna\src\xsp\wp\main.cxx @
222]
0012ffc0 77e7eb69 aspnet_wp!wmainCRTStartup+0x131
[f:\vs70builds\9111\vc\crtbld\crt\src\crtexe.c @ 379]
0012fff0 00000000 kernel32!BaseProcessStart+0x23

   1  id: 168.420   Suspend: 1 Teb 7ffdd000 Unfrozen
ChildEBP RetAddr
0086f7ac 77f7f49f SharedUserData!SystemCallStub+0x4
0086f7b0 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
0086f84c 79281971 kernel32!WaitForMultipleObjectsEx+0x12c
0086f87c 79282444 mscorwks!Thread::DoAppropriateWaitWorker+0xc1
0086f8d0 79317c0a mscorwks!Thread::DoAppropriateWait+0x46
0086f918 036545f4 mscorwks!WaitHandleNative::CorWaitOneNative+0x6f
0086f9ec 03c6c90f 0x36545f4
0086fa24 03c6c373 0x3c6c90f
0086fa80 03daf66b 0x3c6c373
0086fac8 03daf543 0x3daf66b
0086fb20 0377f42d 0x3daf543
0086fba4 792e0069 0x377f42d
0086fbe4 792830c1 mscorwks!ComCallMLStubCache::CompileMLStub+0x1af
0086fc2c 79260b19 mscorwks!Thread::DoADCallBack+0x5c
0086fc94 00cea179 mscorwks!ComCallMLStubCache::CompileMLStub+0x2c2
0086ff24 0042218c 0xcea179
0086ff40 7a13ecc2 aspnet_wp!CAsyncPipeManager::ProcessCompletion+0x1d4
[e:\dna\src\xsp\wp\asyncpipemanager.cxx @ 516]
0086ff84 792cf905 aspnet_isapi!CorThreadPoolCompletionCallback+0x41
[e:\dna\src\xsp\isapi\threadpool.cxx @ 773]
0086ffb4 77e802ed mscorwks!ThreadpoolMgr::CompletionPortThreadStart+0x93
0086ffec 00000000 kernel32!BaseThreadStart+0x37

   2  id: 168.ab0   Suspend: 1 Teb 7ffdc000 Unfrozen
ChildEBP RetAddr
0096ff04 77f7e76f SharedUserData!SystemCallStub+0x4
0096ff08 77e775b7 ntdll!NtDelayExecution+0xc
0096ff60 77e61bf1 kernel32!SleepEx+0x61
0096ff6c 792d00bc kernel32!Sleep+0xb
0096ffb4 77e802ed mscorwks!ThreadpoolMgr::GateThreadStart+0x4c
0096ffec 00000000 kernel32!BaseThreadStart+0x37

   3  id: 168.a18   Suspend: 1 Teb 7ffdb000 Unfrozen
ChildEBP RetAddr
00c6fefc 77f7f4af SharedUserData!SystemCallStub+0x4
00c6ff00 77e7788b ntdll!NtWaitForSingleObject+0xc
00c6ff64 77e79d6a kernel32!WaitForSingleObjectEx+0xa8
```

```
00c6ff74 0042289c kernel32!WaitForSingleObject+0xf
00c6ff80 7c00fbab aspnet_wp!DoPingThread+0x10 [e:\dna\src\xsp\wp\main.cxx
@ 412]
00c6ffb4 77e802ed MSVCR70!_threadstart+0x6c
[f:\vs70builds\9466\vc\crtbld\crt\src\thread.c @ 196]
00c6ffec 00000000 kernel32!BaseThreadStart+0x37

   4  id: 168.9c4   Suspend: 1 Teb 7ffda000 Unfrozen
ChildEBP RetAddr
0101fe80 77f7f49f SharedUserData!SystemCallStub+0x4
0101fe84 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
0101ff20 77e74c70 kernel32!WaitForMultipleObjectsEx+0x12c
0101ff38 791cccf6 kernel32!WaitForMultipleObjects+0x17
0101ffa0 791ccc6b mscorwks!DebuggerRCThread::MainLoop+0x90
0101ffac 791ccc1e mscorwks!DebuggerRCThread::ThreadProc+0x55
0101ffb4 77e802ed mscorwks!DebuggerRCThread::ThreadProcStatic+0xb
0101ffec 00000000 kernel32!BaseThreadStart+0x37

   5  id: 168.d84   Suspend: 1 Teb 7ffd9000 Unfrozen
ChildEBP RetAddr
031bfe98 77f7f49f SharedUserData!SystemCallStub+0x4
031bfe9c 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
031bff38 77e74c70 kernel32!WaitForMultipleObjectsEx+0x12c
031bff50 791d3f35 kernel32!WaitForMultipleObjects+0x17
031bff70 791cdd05 mscorwks!WaitForFinalizerEvent+0x5a
031bffb4 77e802ed mscorwks!GCHeap::FinalizerThreadStart+0x96
031bffec 00000000 kernel32!BaseThreadStart+0x37

   6  id: 168.2ec   Suspend: 1 Teb 7ffd8000 Unfrozen
ChildEBP RetAddr
0353ff1c 77f7f4af SharedUserData!SystemCallStub+0x4
0353ff20 77e7788b ntdll!NtWaitForSingleObject+0xc
0353ff84 77e79d6a kernel32!WaitForSingleObjectEx+0xa8
0353ff94 792cf5dc kernel32!WaitForSingleObject+0xf
0353ffb4 77e802ed mscorwks!ThreadpoolMgr::WorkerThreadStart+0x2e
0353ffec 00000000 kernel32!BaseThreadStart+0x37

   7  id: 168.7b0   Suspend: 1 Teb 7ffd7000 Unfrozen
ChildEBP RetAddr
0363f7b0 77f7f49f SharedUserData!SystemCallStub+0x4
0363f7b4 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
0363f850 79281971 kernel32!WaitForMultipleObjectsEx+0x12c
0363f880 79282444 mscorwks!Thread::DoAppropriateWaitWorker+0xc1
0363f8d4 79317c0a mscorwks!Thread::DoAppropriateWait+0x46
0363f91c 036545f4 mscorwks!WaitHandleNative::CorWaitOneNative+0x6f
0363f9ec 03c6c90f 0x36545f4
0363fa24 03c6c373 0x3c6c90f
0363fa80 03daf66b 0x3c6c373
0363fac8 03daf543 0x3daf66b
0363fb20 0377f42d 0x3daf543
0363fba4 792e0069 0x377f42d
0363fbe4 792830c1 mscorwks!ComCallMLStubCache::CompileMLStub+0x1af
0363fc2c 79260b19 mscorwks!Thread::DoADCallBack+0x5c
```

```
0363fc94 00cea179 mscorwks!ComCallMLStubCache::CompileMLStub+0x2c2
0363ff24 0042218c 0xcea179
0363ff40 7a13ecc2 aspnet_wp!CAsyncPipeManager::ProcessCompletion+0x1d4
[e:\dna\src\xsp\wp\asyncpipemanager.cxx @ 516]
0363ff84 792cf905 aspnet_isapi!CorThreadPoolCompletionCallback+0x41
[e:\dna\src\xsp\isapi\threadpool.cxx @ 773]
0363ffb4 77e802ed mscorwks!ThreadpoolMgr::CompletionPortThreadStart+0x93
0363ffec 00000000 kernel32!BaseThreadStart+0x37

   8  id: 168.3c8   Suspend: 1 Teb 7ffd6000 Unfrozen
ChildEBP RetAddr
0387ff44 77f7e76f SharedUserData!SystemCallStub+0x4
0387ff48 77e775b7 ntdll!NtDelayExecution+0xc
0387ffa0 792d05b2 kernel32!SleepEx+0x61
0387ffb4 77e802ed mscorwks!ThreadpoolMgr::TimerThreadStart+0x30
03880038 792067d5 kernel32!BaseThreadStart+0x37
00d1203b 000000ff mscorwks!CreateTypedHandle+0x16

   9  id: 168.a4   Suspend: 1 Teb 7ffd5000 Unfrozen
ChildEBP RetAddr
03eafe24 77f7efff SharedUserData!SystemCallStub+0x4
03eafe28 77cc1ac9 ntdll!NtReplyWaitReceivePortEx+0xc
03eaff90 77cc167e RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0xf6
03eaff94 77cc1505 RPCRT4!RecvLotsaCallsWrapper+0x9
03eaffac 77cc1670 RPCRT4!BaseCachedThreadRoutine+0x64
03eaffb4 77e802ed RPCRT4!ThreadStartRoutine+0x16
03eaffec 00000000 kernel32!BaseThreadStart+0x37

  10  id: 168.eb8   Suspend: 1 Teb 7ffd4000 Unfrozen
ChildEBP RetAddr
040af7b0 77f7f49f SharedUserData!SystemCallStub+0x4
040af7b4 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
040af850 79281971 kernel32!WaitForMultipleObjectsEx+0x12c
040af880 79282444 mscorwks!Thread::DoAppropriateWaitWorker+0xc1
040af8d4 79317c0a mscorwks!Thread::DoAppropriateWait+0x46
040af91c 036545f4 mscorwks!WaitHandleNative::CorWaitOneNative+0x6f
040af9ec 03c6c90f 0x36545f4
040afa24 03c6c373 0x3c6c90f
040afa80 03daf66b 0x3c6c373
040afac8 03daf543 0x3daf66b
040afb20 0377f42d 0x3daf543
040afba4 792e0069 0x377f42d
040afbe4 79283dc1 mscorwks!ComCallMLStubCache::CompileMLStub+0x1af
040afc2c 79260b19 mscorwks!Thread::DoADCallBack+0x5c
040afc94 00cea179 mscorwks!ComCallMLStubCache::CompileMLStub+0x2c2
040aff24 0042218c 0xcea179
040aff40 7a13ecc2 aspnet_wp!CAsyncPipeManager::ProcessCompletion+0x1d4
[e:\dna\src\xsp\wp\asyncpipemanager.cxx @ 516]
040aff84 792cf905 aspnet_isapi!CorThreadPoolCompletionCallback+0x41
[e:\dna\src\xsp\isapi\threadpool.cxx @ 773]
040affb4 77e802ed mscorwks!ThreadpoolMgr::CompletionPortThreadStart+0x93
040affec 00000000 kernel32!BaseThreadStart+0x37
```

```
  11  id: 168.11c   Suspend: 1 Teb 7ffaf000 Unfrozen
ChildEBP RetAddr
041aff28 004221ab aspnet_wp!CAsyncPipeManager::Close
[e:\dna\src\xsp\wp\asyncpipemanager.cxx @ 122]
041aff40 7a13ecc2 aspnet_wp!CAsyncPipeManager::ProcessCompletion+0x1f3
[e:\dna\src\xsp\wp\asyncpipemanager.cxx @ 535]
041aff84 792cf905 aspnet_isapi!CorThreadPoolCompletionCallback+0x41
[e:\dna\src\xsp\isapi\threadpool.cxx @ 773]
041affb4 77e802ed mscorwks!ThreadpoolMgr::CompletionPortThreadStart+0x93
041affec 00000000 kernel32!BaseThreadStart+0x37

  12  id: 168.894   Suspend: 1 Teb 7ffae000 Unfrozen
ChildEBP RetAddr
042afe24 77f7efff SharedUserData!SystemCallStub+0x4
042afe28 77cc1ac9 ntdll!NtReplyWaitReceivePortEx+0xc
042aff90 77cc167e RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0xf6
042aff94 77cc1505 RPCRT4!RecvLotsaCallsWrapper+0x9
042affac 77cc1670 RPCRT4!BaseCachedThreadRoutine+0x64
042affb4 77e802ed RPCRT4!ThreadStartRoutine+0x16
042affec 00000000 kernel32!BaseThreadStart+0x37

  13  id: 168.e94   Suspend: 1 Teb 7ffad000 Unfrozen
ChildEBP RetAddr
00000000 00000000 kernel32!BaseThreadStartThunk
```

WinDbg doesn't display the managed call stack. It displays native call stacks, but some of the managed calls are wrappers for native calls. The middle section of this stack (without correlated modules and functions) represents managed threads. Threads 1, 7, and 10 have managed code on them. You can identify them by the unmapped code regions between the native resolved function names. This is JIT-compiled code, and it doesn't map to an address range for a loaded module. You will extract the managed thread data shortly.

Note: Experienced debuggers scan through the threads and examine the threads that are questionable. However, many people wonder how one knows which threads are questionable. The answer is experience. You could examine these threads, conclude that three threads are each waiting and have similar call stacks, and decide to check out only those threads. In this example, however, each thread is examined, and the differences and similarities of the dumps are considered. Then, next time you look at a similar dump, you too might go directly to those irregular threads.

Thread 0 is the ASP.NET main thread as shown by the call to **aspnet_wp!wmain()**. This thread has initialized the common language runtime, connected the named pipes back to InetInfo.exe, and created the ping thread. Once it has completed the initialization, it loops, waiting for the process to exit.

The listings for threads 7 and 10 look similar to thread 1, which is shown in the following code sample. Each of these managed threads shows a native call to **mscorwks!WaitHandleNative::CorWaitOneNative()** and appears to be waiting for a response. As noted, the middle section is managed thread information that WinDbg cannot display on its own. You can enlist the help of SOS.dll to interpret this section. Also note that these three threads are completion port threads (see **mscorwks!ThreadpoolMgr::CompletionPortThreadStart()** below). As noted in the discussion on .NET thread pools at the beginning of this chapter, ASP.NET first runs its pages on I/O completion port threads, and then if they get used up, they run on worker threads.

```
   1  id: 168.420   Suspend: 1 Teb 7ffdd000 Unfrozen
ChildEBP RetAddr
0086f7ac 77f7f49f SharedUserData!SystemCallStub+0x4
0086f7b0 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
0086f84c 79281971 kernel32!WaitForMultipleObjectsEx+0x12c
0086f87c 79282444 mscorwks!Thread::DoAppropriateWaitWorker+0xc1
0086f8d0 79317c0a mscorwks!Thread::DoAppropriateWait+0x46
0086f918 036545f4 mscorwks!WaitHandleNative::CorWaitOneNative+0x6f
0086f9ec 03c6c90f 0x36545f4
0086fa24 03c6c373 0x3c6c90f
0086fa80 03daf66b 0x3c6c373
0086fac8 03daf543 0x3daf66b
0086fb20 0377f42d 0x3daf543
0086fba4 792e0069 0x377f42d
0086fbe4 792830c1 mscorwks!ComCallMLStubCache::CompileMLStub+0x1af
0086fc2c 79260b19 mscorwks!Thread::DoADCallBack+0x5c
0086fc94 00cea179 mscorwks!ComCallMLStubCache::CompileMLStub+0x2c2
0086ff24 0042218c 0xcea179
0086ff40 7a13ecc2 aspnet_wp!CAsyncPipeManager::ProcessCompletion+0x1d4
[e:\dna\src\xsp\wp\asyncpipemanager.cxx @ 516]
0086ff84 792cf905 aspnet_isapi!CorThreadPoolCompletionCallback+0x41
[e:\dna\src\xsp\isapi\threadpool.cxx @ 773]
0086ffb4 77e802ed mscorwks!ThreadpoolMgr::CompletionPortThreadStart+0x93
0086ffec 00000000 kernel32!BaseThreadStart+0x37
```

Threads 2, 6, and 8 are ThreadpoolMgr threads. Note the calls to **mscorwks!ThreadpoolMgr::GateThreadStart**, **mscorwks!ThreadpoolMgr::WorkerThreadStart**, and **mscorwks!ThreadpoolMgr::TimerThreadStart**, respectively. These threads are present in ASP.NET (but not always in console applications) because of the ASP.NET use of the thread pool.

- Thread 2 is a GateThread, which monitors the health of the thread pool. It is responsible for injecting and retiring worker and I/O threads based on indicators, such as CPU utilization, garbage-collection frequency, and thread starvation. There is only one thread of this type. It is created when the first I/O or work request is executed.

- Thread 6 is a worker thread as shown by the call to **mscorwks!ThreadpoolMgr::WorkerThreadStart()** in the listing. This thread is waiting for work to be assigned to it.
- Thread 8 is a **TimerThread**, which is used to manage timed callbacks specified using the **System.Threading.Timer** class. There is only one thread of this type, which is created when the first **System.Threading.Timer** object is created.

The threads discussed above are managed threads and thread-pool threads. Let's examine some of the other threads:

- Thread 3 is the ASP.NET ping thread as shown by the call to **aspnet_wp!DoPingThread()**. This thread responds to the pings it receives from the health monitoring thread in InetInfo.exe. The **UnderDebugger** registry change you made has disabled InetInfo from pinging this worker process.
- Thread 4 is a debugger thread as shown by the call to **mscorwks!DebuggerRCThread()**. All managed processes have a debugger thread, even if there is no debugger attached. There is only one debugger thread in a given process. This thread exists so that a managed debugger can attach and control the managed code. Threads react differently based on whether or not a managed or native debugger attaches invasively.

  If you break into the process with a managed debugger (CorDbg or Visual Studio .NET in common language runtime mode), it attaches noninvasively. The entire process is not frozen; instead, only the managed threads are frozen. The native threads that don't contain common language runtime references will continue to run. Options that use a managed debugger to attach invasively are available, but they are beyond the scope of this guide.

  If you use a native debugger, it can attach invasively or noninvasively. When you break into the process invasively and quit, the process terminates. Consequently all threads — managed and native — are shut down. If you attach noninvasively, the process (and threads) stop temporarily. The debugger pauses the native and managed code and provides a snapshot. It then can detach and the process resumes.

  In IIS 5.*x*, there is only one Aspnet_wp.exe worker process and one debugger thread. Consequently, only one debugger can be attached to the Aspnet_wp.exe process at a time. This can pose a problem if you're dealing with multiple Web applications on the same computer. In IIS 6.0, you can coerce an AppDomain to run in a separate application pool. For more information, see "IIS 5.*x* Process Model" and "IIS 6.0 Process Model" in Chapter 1. Separate application pools provide multiple W3wp.exe processes. Multiple debugger threads are created in these processes (one in each), allowing you to debug more efficiently.
- Thread 5 is a finalizer thread as shown by the call to **mscorwks!GCHeap::FinalizerThreadStart**. Just like the debugger thread, there is always one finalizer thread per process in version 1.0 of the .NET Framework. The finalizer thread calls the **Finalize()** method of applicable objects when the GC determines that the object is unreachable (not rooted). In Visual Basic .NET, you can implement

a finalize method. The same function in the Visual C#™ development tool and managed extensions for the Visual C++® development system is performed by a destructor, which maps onto a finalize method. For more information, see the following articles in the .NET Framework SDK:

- .NET Framework General Reference
- Implementing Finalize
- Dispose to Clean Up Unmanaged Resources.

Other sources include:

- Applied Microsoft .NET Framework Programming by Jeffrey Richter (Microsoft Press, 2002).
- "Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework", by Jeffrey Richter, *MSDN Magazine*, November 2000. (*http://msdn.microsoft.com/library/en-us/dnmag00/html/GCI.asp*).
- "Garbage Collection — Part 2: Automatic Memory Management in the Microsoft .NET Framework", by Jeffrey Richter, *MSDN Magazine*, December 2000. (*http://msdn.microsoft.com/msdnmag/issues/1200/GCI2/GCI2.asp*).
- Chapter 19, "Automatic Memory Management (Garbage Collection)" in *Applied Microsoft .NET Framework Programming* by Jeffrey Richter (Microsoft Press, 2002).

- Thread 9 is a lightweight remote procedure call (RPC) thread used for local remote procedure call (LRPC) communication. Here it handles incoming RPC calls to this process.
- Thread 11 is the thread that AdPlus_AspNet.vbs broke in on when Aspnet_isapi.dll (running in InetInfo.exe) was about to shut down the process. Remember that the tool set a breakpoint on **aspnet_wp!CAsyncPipeManager::Close** and dumped the process when this code was triggered. This thread is also a completion port thread.

You'll find several features common to all ASP.NET dumps: the main thread, the ping thread, the debugger thread, the ThreadpoolMgr thread(s), and the finalizer thread.

Examining Managed Data

The native output has set the stage, but you want to see where the action is. To examine the managed code in WinDbg, you must load the SOS.dll debugger extension.

► **To load the debugger extension**

1. In WinDbg, type **.load SOS\SOS.dll** to load the extension (where "SOS" is the location of SOS.dll).
2. Type **!findtable** to initialize SOS with the table information for the runtime version you are debugging.

WinDbg displays the following:

```
0:000> .load sos\sos.dll
0:000> !findtable
Args: ''
Attempting data table load from SOS.WKS.4.3705.209.BIN
Failed timestamp verification.
       Binary time: 05:30:57, 2002/1/5.
       Table time: 03:04:42, 2002/2/15
Attempting data table load from SOS.WKS.4.3705.BIN
Loaded Son of Strike data table version 4 from "SOS.WKS.4.3705.BIN"
```

To list and examine the managed threads and their functions, type **!threads**.

There are five managed threads, which include the finalizer thread and the thread pool worker threads. Examine threads 1, 7, and 10. These are also the same threads whose call stacks made their last managed native call to **CorWaitOneNative()**. Note the lock count values; for this scenario, the number of threads with lock counts greater than 0 will most likely equal the number of times you clicked the **Obtain Shared Resource** buttons. The following shows three threads with a lock count of 1:

```
ThreadCount: 5
UnstartedThread: 0
BackgroundThread: 5
PendingThread: 0
DeadThread: 0
```

	1	5	6	7	10
ID	420	d84	2ec	7b0	eb8
ThreadOBJ	00154008	00155d80	00174a30	0017d060	001f1618
State	200a220	b220	1800220	2000220	2000220
PreEmptive GC	Enabled	Enabled	Enabled	Enabled	Enabled
GC Alloc Context	00000000: 00000000	00000000: 00000000	00000000: 00000000	00000000: 00000000	00000000: 00000000
Domain	001a58b8	00165058	00165058	001a58b8	001a58b8
Lock Count	1	0	0	1	1
Apt	MTA	MTA	MTA	MTA	MTA
Exception		(Finalizer)	(Threadpool Worker)		

The SOS **!threads** command provides other information that may be useful, such as the threading model, AppDomain, and thread state. In this case, all threads are initialized to multithreaded apartment (MTA) threads, and there are two active domains. Every ASP.NET application always has at least two AppDomains—the default domain and the virtual folder's application domain. The default domain houses threads that aren't running managed code in a specific AppDomain. Threads 1, 7, and 10 are each in the same domain (001a58b8). The thread states are discussed later in this section.

According to the .NET Framework SDK, an AppDomain is "an isolated environment where applications execute. It provides isolation, unloading, and security boundaries for executing managed code."

The AppDomain information can provide answers for questions like these:

- There are three blocked threads. Are they all in the same AppDomain? If so, what else is loaded in that AppDomain?
- What threads are in the default domain? These threads aren't running managed code. What are they doing? The default domain threads include the finalizer thread and the worker thread, which hasn't been given work to do yet. Once the worker thread has some work, it moves out of the default domain.
- What application is using up all the threads? If you see 25 threads in one AppDomain, that could help answer why there might not be other threads available to process requests.

For specific information on the AppDomain, type **!dumpdomain <addr>**, where **<addr>** is the domain address listed by the **!threads** command. In this scenario, typing **!dumpdomain 1a58b8** lists the assemblies and modules in the domain specified.

```
0:000> !dumpdomain 001a58b8
Domain: 001a58b8
LowFrequencyHeap: 001a591c
HighFrequencyHeap: 001a5970
StubHeap: 001a59c4
Name: /LM/W3SVC/1/ROOT/Debugging-1-126678048540984363
Assembly: 0017d240 [System, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]
ClassLoader: 00177e90
  Module Name
00178a68 c:\windows\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll

Assembly: 0017f830 [System.Drawing, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a]
ClassLoader: 00177f08
  Module Name
0017f8f8 c:\windows\assembly\gac\system.drawing\1.0.3300.0__b03f5f7f11d50a3a\syste
m.drawing.dll
```

Note the AppDomain name for these three threads. In this case, it is:

```
/LM/W3SVC/1/ROOT/Debugging-1-126678048540984363.
```

Now check the name of the AppDomain that threads 5 and 6 are in (165058). Remember that thread 5 is a finalizer thread and thread 6 is a ThreadpoolWorker wait thread. Use the command **!dumpdomain 165058**. Again, 165058 is the domain address listed by the **!threads** command.

```
0:000> !dumpdomain 165058
Domain: 00165058
LowFrequencyHeap: 001650bc
HighFrequencyHeap: 00165110
StubHeap: 00165164
Name: DefaultDomain
Assembly: 0017d240 [System, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]
ClassLoader: 00177e90
  Module Name
00178a68 c:\windows\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll

Assembly: 0017f830 [System.Drawing, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a]
ClassLoader: 00177f08
  Module Name
0017f8f8 c:\windows\assembly\gac\system.drawing\1.0.3300.0__b03f5f7f11d50a3a\syste
m.drawing.dll
```

The name of this AppDomain is **DefaultDomain**. AppDomains own resources held in memory, and in order to unload a module or assembly, the AppDomain must be unloaded. The default AppDomain cannot be unloaded until the application is no longer running.

This AppDomain information did not reveal anything significant. That happens sometimes — you check out different paths, and some reveal more than others.

The SOS **!threads** command also provided thread state information. The state column indicates what the thread is doing, for example, waiting or running. The values correspond to byte representations of bit flags set for a particular state. You'll look at some threads' states here. To explore more, match the values to the state table in the Appendix.

Looking at the **!threads** output, you can see that thread 1has the state value of 0x0200a220. The following interpretation applies:

Table 3.3: Thread state for thread 1 is 0x0200a220

Mnemonic	Value	Description
TS_Interruptible	0x02000000	Sitting in either a Sleep(), Wait(), or Join()
TS_InMTA	0x00008000	Thread is part of the MTA
TS_CoInitialized	0x00002000	CoInitialize has been called for this thread
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Is it now legal to attempt a Join()?

Threads 7 and 10 are similar, with the state value of 0x02000220, except that they exclude TS_InMTA and TS_CoInitialized. A thread will not live in an apartment until after **CoInitialize** is called. **CoInitialize** has not been called yet for these two threads because they're waiting for the resource that thread 1 accessed first.

Again, these threads are created by clicking the **Obtain Shared Resource** buttons. They each have a lock count of 1 and made their last managed native call to **CorWaitOneNative().**

Let's look at the thread state values for some the other managed threads. Again, this output provides more information on the threads.

Table 3.4: Thread state for thread 5 is 0x0000b220

Mnemonic	Value	Description
TS_InMTA	0x00008000	Thread is part of the MTA
TS_CoInitialized	0x00002000	CoInitialize has been called for this thread
TS_WeOwn	0x00001000	Exposed object initiated this thread
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicates if it now legal to attempt a Join()

Table 3.5: Thread state for thread 6 is 0x1800220

Mnemonic	Value	Description
TS_TPWorkerThread	0x01000000	Is this a thread pool worker thread? (If not, it is a thread pool CompletionPort thread.)
TS_ThreadPoolThread	0x00800000	Is this a thread pool thread?
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicates if it now legal to attempt a Join()

ASP.NET uses thread pooling to provide the application with a pool of worker threads that are managed by the system. There is only one **ThreadPool** object per process. The **!threadpool** command confirms that CPU utilization is minimal. There is one worker thread and five completion port threads. In this example, the maximum worker thread is 25, which implies that there is one CPU.

```
0:000> !threadpool
CPU utilization 2%
Worker Thread: Total: 1 Running: 1 Idle: 1 MaxLimit: 25 MinLimit: 1
Work Request in Queue: 0
----------------------------------
Number of Timers: 0
----------------------------------
Completion Port Thread: Total: 5 Free: 1 MaxFree: 2 CurrentLimit: 5 MaxLimit: 25
MinLimit: 1
```

The worker thread is managed thread 6 waiting for work, and the completion port threads are runtime threads 1, 7, 10, 11, and 12. Threads 1, 7, and 10 are managed; they have work to do, but are waiting for a shared resource. Threads 11 and 12 are native. Thread 11 is the thread on which we set a breakpoint with ADPlus_AspNet.vbs. It is a completion port thread whose work item is to cleanly shut down the worker process. Thread 12 is just waiting for work as shown by the "Free:1" output from the command.

When using the SOS.dll extension, the output for some commands varies depending on whether you are working with a debug build or a release build:

- **Debug build.** The debug build aids in testing and debugging. This build contains error-checking, argument-verification, and debugging information that is not available in the release build. Consequently, the debug build is larger, slower, and uses more memory. Performance is also slower because the executables contain symbolic debugging information. Additional code paths are executed because of parameter checking and output of diagnostic messages for debugging purposes.
- **Release build.** The release build is smaller and faster, and it uses less memory than the debug build. It is fully optimized, debugging asserts are disabled, and debugging information is stripped from the binaries. Because the compiler repositions and reorganizes instructions, the debugger can't always identify the source code that corresponds to the instructions.
- **Release build with .ini files.** If the release build is coupled with applicable .ini files, the results simulate the debug build. The .ini files must have the same name as the assembly and be in the same folder from where the assembly was loaded. The .ini file includes the **AllowOptimize** and **GenerateTrackingInfo** settings, which are set to 0 and 1, respectively. For more information, see the .NET Framework SDK article "Making An Image Easier To Debug" on the MSDN Web site at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconmakingimageeasiertodebug.asp*.

For more information on compiling options, see the .NET Framework SDK. Unless otherwise specified, release build examples from the dump file are used here.

You can examine the managed trace by typing **~*e !clrstack**, as shown in the following command sample. This command is similar to the **~*k** WinDbg command noted earlier, but it applies to managed calls. WinDbg now displays the current enter stack pointer (ESP), the current enter instruction pointer (EIP), and the method signature for each frame with a managed thread. The **!clrstack** command would show only the managed threads.

```
0:000> ~*e !clrstack
Thread 0
Not a managed thread.
Thread 1
ESP       EIP
0086f948  7ffe0304 [FRAME: ECallMethodFrame] [DEFAULT] Boolean
```

```
System.Threading.WaitHandle.WaitOneNative(I,UI4,Boolean)
0086f95c  03ce73f6 [DEFAULT] [hasThis] Boolean
System.Threading.WaitHandle.WaitOne(I4,Boolean)
0086f970  03c410e2 [DEFAULT] Boolean Debugging.SharedResource.Wait()
0086f974  03c41059 [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared1_Click(Object,Class System.EventArgs)
0086f980  03ce7375 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)
0086f994  03ce7152 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.System.Web.UI.IPostBackEventHandl
er.RaisePostBackEvent(String)
0086f9a4  03ce7103 [DEFAULT] [hasThis] Void
System.Web.UI.Page.RaisePostBackEvent(Class
System.Web.UI.IPostBackEventHandler,String)
0086f9ac  03ce7062 [DEFAULT] [hasThis] Void
System.Web.UI.Page.RaisePostBackEvent(Class
System.Collections.Specialized.NameValueCollection)
0086f9bc  03c6ebd8 [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequestMain()
0086f9f4  03c6c90f [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequest()
0086fa2c  03c6c373 [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequest(Class System.Web.HttpContext)
0086fa34  03c6c34c [DEFAULT] [hasThis] Void
System.Web.HttpApplication/CallHandlerExecutionStep.Execute()
0086fa44  03c600b0 [DEFAULT] [hasThis] Class System.Exception
System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef
Boolean)
0086fa8c  03daf66b [DEFAULT] [hasThis] Void
System.Web.HttpApplication.ResumeSteps(Class System.Exception)
0086fad0  03daf543 [DEFAULT] [hasThis] Class System.IAsyncResult
System.Web.HttpApplication.System.Web.IHttpAsyncHandler.BeginProce
ssRequest(Class System.Web.HttpContext,Class
System.AsyncCallback,Object)
0086faec  0377f648 [DEFAULT] [hasThis] Void
System.Web.HttpRuntime.ProcessRequestInternal(Class
System.Web.HttpWorkerRequest)
0086fb28  0377f42d [DEFAULT] Void
System.Web.HttpRuntime.ProcessRequest(Class
System.Web.HttpWorkerRequest)
0086fb34  0377b127 [DEFAULT] [hasThis] I4
System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
0086fbf0  792Oeab0 [FRAME: ContextTransitionFrame]
0086fcd0  792Oeab0 [FRAME: ComMethodFrame]
Thread 2
Not a managed thread.
Thread 3
Not a managed thread.
Thread 4
Not a managed thread.
Thread 5
ESP       EIP
Thread 6
```

```
ESP       EIP
Thread 7
ESP       EIP
0363f94c  7ffe0304 [FRAME: ECallMethodFrame] [DEFAULT] Boolean
System.Threading.WaitHandle.WaitOneNative(I,UI4,Boolean)
0363f960  03ce73f6 [DEFAULT] [hasThis] Boolean
System.Threading.WaitHandle.WaitOne(I4,Boolean)
0363f974  03c41115 [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared2_Click(Object,Class
System.EventArgs)
0363f980  03ce7375 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)
0363f994  03ce7152 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.System.Web.UI.IPostBackEventHandl
er.RaisePostBackEvent(String)
0363f9a4  03ce7103 [DEFAULT] [hasThis] Void
System.Web.UI.Page.RaisePostBackEvent(Class
System.Web.UI.IPostBackEventHandler,String)
0363f9ac  03ce7062 [DEFAULT] [hasThis] Void
System.Web.UI.Page.RaisePostBackEvent(Class
System.Collections.Specialized.NameValueCollection)
0363f9bc  03c6ebd8 [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequestMain()
0363f9f4  03c6c90f [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequest()
0363fa2c  03c6c373 [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequest(Class System.Web.HttpContext)
0363fa34  03c6c34c [DEFAULT] [hasThis] Void
System.Web.HttpApplication/CallHandlerExecutionStep.Execute()
0363fa44  03c600b0 [DEFAULT] [hasThis] Class System.Exception
System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef
Boolean)
0363fa8c  03daf66b [DEFAULT] [hasThis] Void
System.Web.HttpApplication.ResumeSteps(Class System.Exception)
0363fad0  03daf543 [DEFAULT] [hasThis] Class System.IAsyncResult
System.Web.HttpApplication.System.Web.IHttpAsyncHandler.BeginProce
ssRequest(Class System.Web.HttpContext,Class
System.AsyncCallback,Object)
0363faec  0377f648 [DEFAULT] [hasThis] Void
System.Web.HttpRuntime.ProcessRequestInternal(Class
System.Web.HttpWorkerRequest)
0363fb28  0377f42d [DEFAULT] Void
System.Web.HttpRuntime.ProcessRequest(Class
System.Web.HttpWorkerRequest)
0363fb34  0377b127 [DEFAULT] [hasThis] I4
System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
0363fbf0  7920eab0 [FRAME: ContextTransitionFrame]
0363fcd0  7920eab0 [FRAME: ComMethodFrame]
Thread 8
Not a managed thread.
Thread 9
Not a managed thread.
Thread 10
```

```
ESP       EIP
040af94c  7ffe0304 [FRAME: ECallMethodFrame] [DEFAULT] Boolean
System.Threading.WaitHandle.WaitOneNative(I,UI4,Boolean)
040af960  03ce73f6 [DEFAULT] [hasThis] Boolean
System.Threading.WaitHandle.WaitOne(I4,Boolean)
040af974  03c4116d [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared3_Click(Object,Class
System.EventArgs)
040af980  03ce7375 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)
040af994  03ce7152 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.System.Web.UI.IPostBackEventHandl
er.RaisePostBackEvent(String)
040af9a4  03ce7103 [DEFAULT] [hasThis] Void
System.Web.UI.Page.RaisePostBackEvent(Class
System.Web.UI.IPostBackEventHandler,String)
040af9ac  03ce7062 [DEFAULT] [hasThis] Void
System.Web.UI.Page.RaisePostBackEvent(Class
System.Collections.Specialized.NameValueCollection)
040af9bc  03c6ebd8 [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequestMain()
040af9f4  03c6c90f [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequest()
040afa2c  03c6c373 [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequest(Class System.Web.HttpContext)
040afa34  03c6c34c [DEFAULT] [hasThis] Void
System.Web.HttpApplication/CallHandlerExecutionStep.Execute()
040afa44  03c600b0 [DEFAULT] [hasThis] Class System.Exception
System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef
Boolean)
040afa8c  03daf66b [DEFAULT] [hasThis] Void
System.Web.HttpApplication.ResumeSteps(Class System.Exception)
040afad0  03daf543 [DEFAULT] [hasThis] Class System.IAsyncResult
System.Web.HttpApplication.System.Web.IHttpAsyncHandler.BeginProce
ssRequest(Class System.Web.HttpContext,Class
System.AsyncCallback,Object)
040afaec  0377f648 [DEFAULT] [hasThis] Void
System.Web.HttpRuntime.ProcessRequestInternal(Class
System.Web.HttpWorkerRequest)
040afb28  0377f42d [DEFAULT] Void
System.Web.HttpRuntime.ProcessRequest(Class
System.Web.HttpWorkerRequest)
040afb34  0377b127 [DEFAULT] [hasThis] I4
System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
040afbf0  7920eab0 [FRAME: ContextTransitionFrame]
040afcd0  7920eab0 [FRAME: ComMethodFrame]
Thread 11
Not a managed thread.
Thread 12
Not a managed thread.
Thread 13
Not a managed thread.
```

Because this is a release build, the call stack may not report as many method calls as expected. Code transformations, such as inlining, occur during optimization. The following shows how the code has been optimized by the JIT compiler. Managed threads 1, 7, and 10 each call the same function. Thread 1 ran first and included **Debugging.SharedResource.Wait().** The other two frames don't include this call.

```
Thread 1
ESP       EIP
0086f948  7ffe0304 [FRAME: ECallMethodFrame] [DEFAULT] Boolean
System.Threading.WaitHandle.WaitOneNative(I,UI4,Boolean)
0086f95c  03cd77be [DEFAULT] [hasThis] Boolean
System.Threading.WaitHandle.WaitOne(I4,Boolean)
0086f970  03c310e2 [DEFAULT] Boolean Debugging.SharedResource.Wait()
0086f974  03c31059 [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared1_Click(Object,Class
System.EventArgs)
0086f980  03cd7735 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)

Thread 7
ESP       EIP
0363f94c  7ffe0304 [FRAME: ECallMethodFrame] [DEFAULT] Boolean
System.Threading.WaitHandle.WaitOneNative(I,UI4,Boolean)
0363f960  03cd77be [DEFAULT] [hasThis] Boolean
System.Threading.WaitHandle.WaitOne(I4,Boolean)
0363f974  03c31115 [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared2_Click(Object,Class
System.EventArgs)
0363f980  03cd7735 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)

Thread 10
ESP       EIP
040af94c  7ffe0304 [FRAME: ECallMethodFrame] [DEFAULT] Boolean
System.Threading.WaitHandle.WaitOneNative(I,UI4,Boolean)
040af960  03cd77be [DEFAULT] [hasThis] Boolean
System.Threading.WaitHandle.WaitOne(I4,Boolean)
040af974  03c3116d [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared3_Click(Object,Class
System.EventArgs)
040af980  03cd7735 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)
```

If you use ILDasm with the release build and examine the Microsoft intermediate language (MSIL) for each button-click event, you will find that each function's IL code is practically identical. But if you disassemble the code, you will see differences between the JIT-compiled (optimized) and non-JIT compiled code. Keep in mind that JIT-compiled managed code is converted to x86 assembly (native code).

Use the WinDbg **!u** command to disassemble the code for the thread 1 button-click event. You pass in the EIP, and the disassembly begins at the current address.

From the SOS **!clrstack** command output for thread 1, use the EIP value as the address that is passed to the **!u** command:

```
0086f974  03c31059 [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared1_Click(Object,Class
System.EventArgs)

0:010> !u 03c31059
Normal JIT generated code
[DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared1_Click(Object,Class
System.EventArgs)
Begin 03c31050, size 3b
03c31050 56              push    esi
03c31051 8bf1            mov     esi,ecx
03c31053 ff1554c07503    call    dword ptr [0375c054]
(Debugging.SharedResource.Wait)  //call Wait() function
03c31059 25ff000000      and     eax,0xff
03c3105e 8b8ec8000000    mov     ecx,[esi+0xc8]
03c31064 8bf1            mov     esi,ecx
03c31066 8b01            mov     eax,[ecx]
03c31068 ff9090010000    call    dword ptr [eax+0x190]
03c3106e 8bc8            mov     ecx,eax
03c31070 8b15d49c0702    mov     edx,[02079cd4] ("<BR>Obtained Shared
Resource 1")
03c31076 e8e5325dff      call    03204360 (System.String.Concat)
03c3107b 8bd0            mov     edx,eax
03c3107d 8bce            mov     ecx,esi
03c3107f 8b01            mov     eax,[ecx]
03c31081 ff9094010000    call    dword ptr [eax+0x194]
03c31087 5e              pop     esi
03c31088 c20400          ret     0x4
```

The first button-click event calls **SharedResource.Wait()**. This code path has not been JIT-compiled before. The JIT compiler compiles the method call the first time it is called, and it includes a call to **Debugging.SharedResource.Wait()**. You can use the **!u** command to determine where this call goes as shown in the following example:

```
0:010> !u  03c310e2
```

In this case, 03c310e2 is the EIP address for the **Debugging.SharedResource.Wait()** function, as you can see from the **!clrstack** output for thread 1.

As shown in the code below, the **Wait()** method calls **ManualResetEvent()**. For the second or third button click, take the EIP value as the address passed to the **!u** command. In this example, the Thread 10s value is used.

```
040af974  03c3116d [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared3_Click(Object,Class
System.EventArgs)

0:010> !u 03c3116d
Normal JIT generated code
[DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared3_Click(Object,Class
System.EventArgs)
Begin 03c31158, size 4c
03c31158 56              push    esi
03c31159 8bf1            mov     esi,ecx
//code runs inline, after JITting
03c3115b 6a00            push    0x0
03c3115d 8b0d98910702    mov     ecx,[02079198] (Object:
System.Threading.ManualResetEvent)   //moves the address of the
object into ecx
03c31163 baffffffff      mov     edx,0xffffffff    //pass in -1 to
ManualResetEvent (infinite wait)
03c31168 8b01            mov     eax,[ecx]  //deference ecx (function
pointer)
03c3116a ff5050          call    dword ptr [eax+0x50]     //call eax
plus 50 (offset in the ManualResetEvent to WaitOne())
03c3116d b801000000      mov     eax,0x1
03c31172 25ff000000      and     eax,0xff
03c31177 8b8ec8000000    mov     ecx,[esi+0xc8]
03c3117d 8bf1            mov     esi,ecx
03c3117f 8b01            mov     eax,[ecx]
03c31181 ff9090010000    call    dword ptr [eax+0x190]
03c31187 8bc8            mov     ecx,eax
03c31189 8b15dc9c0702    mov     edx,[02079cdc] ("<BR>Obtained Shared
Resource 3")
03c3118f e8cc315dff      call    03204360 (System.String.Concat)
03c31194 8bd0            mov     edx,eax
03c31196 8bce            mov     ecx,esi
03c31198 8b01            mov     eax,[ecx]
03c3119a ff9094010000    call    dword ptr [eax+0x194]
03c311a0 5e              pop     esi
03c311a1 c20400          ret     0x4
```

When the other click events call the method again, the optimized code does not perform an "external" call to **SharedResource.Wait()**; rather, it runs the code inline.

Look again at threads 5 and 6 and note that the **!clrstack** output does not include any stack data for these managed threads. Managed code can run on these threads, but isn't currently active. The example below shows the relevant lines extracted from the **!clrstack** and **!threads** outputs.

```
Output from !clrstack
Thread 5
ESP       EIP
Thread 6
ESP       EIP
```

```
Output from !threads
  5   454 00156c10      b220 Enabled  00000000:00000000 00166030     0 MTA
(Finalizer)
  6   5e8 00174a30   1800220 Enabled  00000000:00000000 00166030     0 MTA
(Threadpool Worker)
```

Again, you can see threads 1, 7, and 10 performing similar actions. Earlier you looked at the native output of one of these managed threads (thread 7). SOS.dll output fills in the missing information for the managed data. To build both managed and native portions of the stack, use Notepad or a similar editor to cut and paste the output from **!clrstack** into the middle portion where there were previously no mapped modules.

```
   1  id: 580.ffc   Suspend: 1 Teb 7ffdd000 Unfrozen
NATIVE
ChildEBP RetAddr
0086f7ac 77f7f49f SharedUserData!SystemCallStub+0x4
0086f7b0 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
0086f84c 79281971 kernel32!WaitForMultipleObjectsEx+0x12c
0086f87c 79282444 mscorwks!Thread::DoAppropriateWaitWorker+0xc1
0086f8d0 79317c0a mscorwks!Thread::DoAppropriateWait+0x46
0086f918 036545f4 mscorwks!WaitHandleNative::CorWaitOneNative+0x6f

MANAGED
ESP       EIP
0086f948  7ffe0304 [FRAME: ECallMethodFrame] [DEFAULT] Boolean
System.Threading.WaitHandle.WaitOneNative(I,UI4,Boolean)
0086f95c  03cd77be [DEFAULT] [hasThis] Boolean
System.Threading.WaitHandle.WaitOne(I4,Boolean)
0086f970  03c310e2 [DEFAULT] Boolean Debugging.SharedResource.Wait()
0086f974  03c31059 [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared1_Click(Object,Class
System.EventArgs)
0086f980  03cd7735 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)
0086f994  03cd7512 [DEFAULT] [hasThis] Void
System.Web.UI.WebControls.Button.System.Web.UI.IPostBackEventHandl
er.RaisePostBackEvent(String)
0086f9a4  03cd74c3 [DEFAULT] [hasThis] Void
System.Web.UI.Page.RaisePostBackEvent(Class
System.Web.UI.IPostBackEventHandler,String)
0086f9ac  03cd7422 [DEFAULT] [hasThis] Void
System.Web.UI.Page.RaisePostBackEvent(Class
System.Collections.Specialized.NameValueCollection)
0086f9bc  03c5f050 [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequestMain()
0086f9f4  03c5cd7f [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequest()
0086fa2c  03c5c7eb [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequest(Class System.Web.HttpContext)
0086fa34  03c5c7c4 [DEFAULT] [hasThis] Void
```

```
System.Web.HttpApplication/CallHandlerExecutionStep.Execute()
0086fa44  03c50528 [DEFAULT] [hasThis] Class System.Exception
System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef
Boolean)
0086fa8c  03dafb2b [DEFAULT] [hasThis] Void
System.Web.HttpApplication.ResumeSteps(Class System.Exception)
0086fad0  03daf9fb [DEFAULT] [hasThis] Class System.IAsyncResult
System.Web.HttpApplication.System.Web.IHttpAsyncHandler.BeginProce
ssRequest(Class System.Web.HttpContext,Class
System.AsyncCallback,Object)
0086faec  0377f5b8 [DEFAULT] [hasThis] Void
System.Web.HttpRuntime.ProcessRequestInternal(Class
System.Web.HttpWorkerRequest)
0086fb28  0377f39d [DEFAULT] Void
System.Web.HttpRuntime.ProcessRequest(Class
System.Web.HttpWorkerRequest)
0086fb34  0377b097 [DEFAULT] [hasThis] I4
System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
0086fbf0  7920eab0 [FRAME: ContextTransitionFrame]
0086fcd0  7920eab0 [FRAME: ComMethodFrame]

NATIVE
0086fbe4 792830c1 mscorwks!ComCallMLStubCache::CompileMLStub+0x1af
0086fc2c 79260b19 mscorwks!Thread::DoADCallBack+0x5c
0086fc94 00cea179 mscorwks!ComCallMLStubCache::CompileMLStub+0x2c2
0086ff24 0042218c 0xcea179
0086ff40 7a13ecc2 aspnet_wp!CAsyncPipeManager::ProcessCompletion+0x1d4
[e:\dna\src\xsp\wp\asyncpipemanager.cxx @ 516]
0086ff84 792cf905 aspnet_isapi!CorThreadPoolCompletionCallback+0x41
[e:\dna\src\xsp\isapi\threadpool.cxx @ 773]
0086ffb4 77e802ed mscorwks!ThreadpoolMgr::CompletionPortThreadStart+0x93
0086ffec 00000000 kernel32!BaseThreadStart+0x37
```

Read through the various events and calls. Note that the **ButtonObtainShared1_Click()** event calls **SharedResource.Wait()**, which calls **System.Threading.WaitHandle.WaitOne()**. This thread is waiting —it makes a runtime call to **CorWaitOneNative()**, which in turn makes an operating system call to **WaitForMultipleObjectsEx()**.This is seen in the three of the managed threads (except the thread1 call to **Wait()** as mentioned before). This gives a great picture of what is causing the deadlock!

Using the Debug Build

The debug build output for the managed threads also includes the source file and line number at the time of the dump. You can use the **!clrstack** command to view this information. For example, in the following debug build frame, the **!clrstack** output tells you to go to the SharedResource.cs file and check out line 55.

```
0373f940  03df1303 [DEFAULT] Boolean Debugging.SharedResource.Wait()
  at [+0x5b] [+0x2d] C:\Inetpub\wwwroot\Debugging\SharedResource.cs:55
```

```
0373f960  03df136b [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared1_Click(Object,Class System.EventArgs)
  at [+0x13] [+0x5] c:\inetpub\wwwroot\debugging\contention.aspx.cs:120
```

Provided that the debug build is used, you can also check out other **!clrstack** options, such as **-l** and **-p**. For example, type **~1s** (to go to that thread), and then type **!clrstack -p** to get the following results:

```
0373f960  03df136b [DEFAULT] [hasThis] Void
Debugging.Contention.btnObtainShared1_Click(Object,Class System.EventArgs)
  at [+0x13] [+0x5] c:\inetpub\wwwroot\debugging\contention.aspx.cs:120
    PARAM: this: 0x011ce548 (ASP.Contention_aspx)
    PARAM: class System.EventArgs sender: 0x011d2f60
    PARAM: unsigned int8 e: 0x12186b0
```

What jumps out is the PARAM information in the ASP.Contention_aspx **btnObtainShared1_Click** frame. You can use the **!dumpobj** command to discover more about this parameter. The **!DumpObj <addr>** command dumps an object on the managed heap and displays information about the object's fields. Type **!dumpobj 0x011ce548**. You can then dump other objects using the same syntax.

For the same thread, you can examine the locals for the frames that contain debugging information. Type **!clrstack -l** and see if the LOCAL information is helpful.

```
0373f940  03df1303 [DEFAULT] Boolean Debugging.SharedResource.Wait()
  at [+0x5b] [+0x2d] C:\Inetpub\wwwroot\Debugging\SharedResource.cs:55
    LOCAL: bool CS$00000003$00000000: false
    LOCAL: int32 CS$00000002$00000001: 142
    LOCAL: int32 CS$00000002$00000002: 0
```

Reviewing What You've Learned

In this scenario, the dumps have indicated that it's time to look at the source code. The **~*k 200** WinDbg command showed three managed threads with the last managed native call to **mscorwks!WaitHandleNative::CorWaitOneNative()**.

You loaded SOS.dll and used the **!threads** command to list and examine the managed threads and their functions. The same three threads show up in the same AppDomain, with lock counts of one, and with state of **Wait**. The **!clrstack** command output shows a call to the **ButtonObtainShared1_Click()** event, which calls **SharedResource.Wait()**, which in turn calls **System.Threading.WaitHandle.WaitOne()**. If you were using the debug build, the output even points to the page and line of code in question: C:\Inetpub\wwwroot\Debugging\SharedResource.cs:55.

Looking at the Source Code

Now let's discover what the code is waiting on. The **btnObtainShared1_Click()** event calls **Debugging.SharedResource.Wait()**. Look at the source code for **Debugging.SharedResource.Wait()**.

```
//From SharedResource.cs
public static bool Wait()
{
   System.Diagnostics.Debug.WriteLine("About to wait : " +
         Thread.CurrentThread.GetHashCode().ToString());
   _sharedEvent.WaitOne(-1, false);
   System.Diagnostics.Debug.WriteLine("Finished Waiting: " +
         Thread.CurrentThread.GetHashCode().ToString());
   return true;
}
```

The SDK documentation for the **WaitHandle.WaitOne()** method says that when this method is "overridden in a derived class, [it] blocks the current thread until the current WaitHandle receives a signal." The documentation also says that because -1 was passed as the first parameter, the thread waits indefinitely until a signal is received.

Look at the **_sharedEvent** object to determine its scope.

```
//From SharedResource.cs
public class SharedResource
{
   static SharedResource()
   {
      _sharedEvent = new ManualResetEvent(false);

   }
   private static ManualResetEvent _sharedEvent;
```

Note that there is a static constructor and a static field called **_sharedEvent**. This means that there is only one copy of this class per AppDomain.

To reiterate, the code first creates a **ManualResetEvent** in a nonsignaled state. Each time you click an **Obtain Shared Resource** button, code for **SharedResource.Wait()** is invoked, which calls **ManualResetEvent.WaitOne()**. The threads are waiting for a shared resource, not their own copy. **WaitOne()** blocks the current thread until the WaitHandle receives a signal. You don't provide the signal within the three-minute interval, so the Aspnet_wp.exe process recycles.

If you don't have the source, you can use ILDasm and examine the MSIL code.

To change which thread is being examined, use the ~ command, passing it the thread number you want to use. For example, to view the call stack for thread 1, use **~1s**.

```
0:000> ~1s
```

Now look back at the native call stack using the **kb** command. In the examples shown below, you are only interested in the first six frames so you can pass **6** to the **kb** command. You want to verify that each thread is waiting on the same event.

```
0:001> kb 6
ChildEBP RetAddr  Args to Child
0086f7ac 77f7f49f 77e74bd8 00000001 0086f7f8 SharedUserData!SystemCallStub+0x4
0086f7b0 77e74bd8 00000001 0086f7f8 00000000 ntdll!ZwWaitForMultipleObjects+0xc
0086f84c 79281971 00000001 0086f92c 00000001
kernel32!WaitForMultipleObjectsEx+0x12c
0086f87c 79282444 00000001 0086f92c 00000001
mscorwks!Thread::DoAppropriateWaitWorker+0xc1
0086f8d0 79317c0a 00000001 0086f92c 00000001
mscorwks!Thread::DoAppropriateWait+0x46
0086f918 036545f4 0086f924 00000000 ffffffff
mscorwks!WaitHandleNative::CorWaitOneNative+0x6f
```

Examine the **kernel32!WaitForMultipleObjectsEx()** function. You can look up the function definition on the MSDN Web site at *http://msdn.microsoft.com/library/default.asp* by searching for **WaitForMultipleObjectsEx** . Here's the definition:

```
DWORD WaitForMultipleObjectsEx(
  DWORD nCount,             // number of handles in array
  CONST HANDLE *lpHandles,  // object-handle array
  BOOL bWaitAll,            // wait option
  DWORD dwMilliseconds,     // time-out interval
  BOOL bAlertable           // alertable option
);
```

In the above listing, the first argument passed has the value 1, while the second is the address of the object handle array. Because you know there is one item in the handle array, you can dump the second parameter with the **dd <address> l<number of DWORDs to dump>** command; for example, use **dd 0086f92c l1**.

If you check the other threads, you can see that they are all waiting on the same handle, which has the value 250.

```
0:001> dd 0086f92c l1
0086f92c  00000250
0:007> ~7s
eax=000000c0 ebx=0363f7fc ecx=000003ff edx=00000000 esi=00000000 edi=7ffdf000
eip=7ffe0304 esp=0363f7b4 ebp=0363f850 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
SharedUserData!SystemCallStub+4:
```

```
7ffe0304 c3              ret
0:007> kb 6
ChildEBP RetAddr  Args to Child
0363f7b0 77f7f49f 77e74bd8 00000001 0363f7fc SharedUserData!SystemCallStub+0x4
0363f7b4 77e74bd8 00000001 0363f7fc 00000000 ntdll!ZwWaitForMultipleObjects+0xc
0363f850 79281971 00000001 0363f930 00000001
kernel32!WaitForMultipleObjectsEx+0x12c
0363f880 79282444 00000001 0363f930 00000001
mscorwks!Thread::DoAppropriateWaitWorker+0xc1
0363f8d4 79317c0a 00000001 0363f930 00000001
mscorwks!Thread::DoAppropriateWait+0x46
0363f91c 036545f4 0363f928 00000000 ffffffff
mscorwks!WaitHandleNative::CorWaitOneNative+0x6f
0:007> dd 0363f7fc l1
0363f7fc  00000250
0:007> ~10s
eax=000000c0 ebx=040af7fc ecx=000003ff edx=00000000 esi=00000000 edi=7ffdf000
eip=7ffe0304 esp=040af7b4 ebp=040af850 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
SharedUserData!SystemCallStub+4:
7ffe0304 c3              ret
0:010> kb 6
ChildEBP RetAddr  Args to Child
040af7b0 77f7f49f 77e74bd8 00000001 040af7fc SharedUserData!SystemCallStub+0x4
040af7b4 77e74bd8 00000001 040af7fc 00000000 ntdll!ZwWaitForMultipleObjects+0xc
040af850 79281971 00000001 040af930 00000001
kernel32!WaitForMultipleObjectsEx+0x12c
040af880 79282444 00000001 040af930 00000001
mscorwks!Thread::DoAppropriateWaitWorker+0xc1
040af8d4 79317c0a 00000001 040af930 00000001
mscorwks!Thread::DoAppropriateWait+0x46
040af91c 036545f4 040af928 00000000 ffffffff
mscorwks!WaitHandleNative::CorWaitOneNative+0x6f
0:010> dd 040af930  l1
040af930  00000250
```

Exploring Further

Try these ideas to learn more:

- Although each **Obtain Shared Resource** button invokes the functions with the same code, the output for the call stacks shows multiple code paths. You may be dealing with the same code, but not the same function.

  Click the same **Obtain Shared Resource** button three times and compare these results to those that occur when you click three different **Obtain Shared Resource** buttons. The output should show the same call stack three times. Each thread would be obtaining the same resource.

- Click an **Obtain Shared Resource** button at least 25 times and consider any thread pool information gathered from either System Monitor or the dumps and SOS.dll. What happens when the thread pool is exhausted?
- Compare the results when you use **AutoResetEvent** instead of **ManualResetEvent**. From the .NET Framework SDK: "You use the **AutoResetEvent** class to make a thread wait until some event puts it in the signaled state by calling **AutoResetEvent.Set**. Unlike the **ManualResetEvent**, the **AutoResetEvent** is automatically reset to nonsignaled by the system after a single waiting thread has been released. If no threads are waiting, the event object's state remains signaled."

Debugging with Visual Studio .NET

Visual Studio .NET contains an excellent debugger that can be used to display detailed information about both managed and native code. This section demonstrates how you can use Visual Studio .NET to obtain much of the same information you can obtain using WinDbg, SOS.dll, and CorDbg.

Dumping Process Information

Before you create a dump, you must ensure that the registry and the application are correctly configured.

► **To configure the registry and the application**

1. Run Regedit, and then locate the key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ASP.NET.
2. Look at the **UnderDebugger** key, and make sure that its data value is set to 1.
3. Start the Visual Studio .NET IDE, and then open the DebuggingWeb project. Make sure that the active configuration is the Debug configuration. To change the configuration:
   a. Right-click the solution in Solution Explorer, and then click **Configuration Manager**.
   b. Set the **Active Solution Configuration** to **Debug**.
4. In Solution Explorer, right-click the project, and then click **Set as Startup Project**.
5. Right-click Contention.aspx, and then click **Set as Start Page**.
6. Press F5 to start the application.
7. When the form displays in the browser window, click each **Obtain Shared Resource** button once.
8. Close all dockable windows.

9. On the **Debug** menu, click **Break All**, or press CTRL+ALT+BREAK. On the **Debug** menu, point to **Windows**, and then click **Disassembly**.

   You should see disassembly and source windows displayed. Figure 3.5 shows the source window:

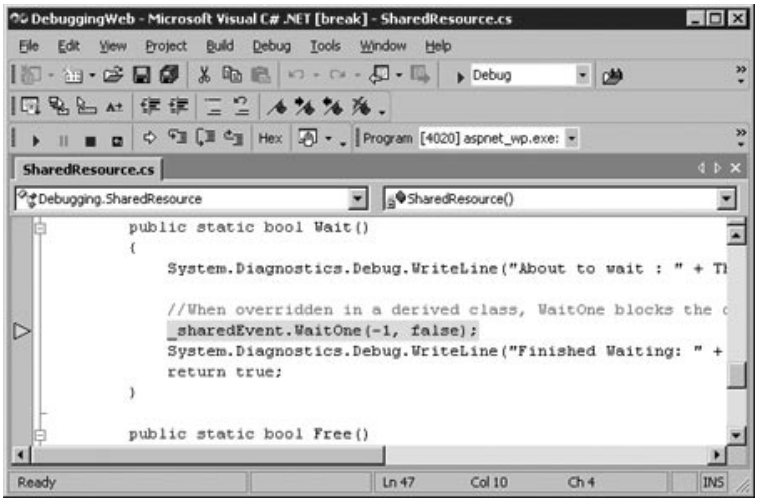

Figure 3.5
Source code window

10. To open the thread window, on the **Debug** menu, point to **Windows,** and then click **Threads**.

In Visual Studio .NET, you can also issue debugger commands from the command line.

► **To issue debugger commands from the command line**

1. On the **View** menu, point to **Other Windows**, and then select **Command Window**.

   You can use the tilde (~) command to list the current threads:

```
>~
 Index Id     Name                                Location
-------------------------------------------------------------------------
*1     3584   <No Name>        System.Threading.WaitHandle::WaitOne
 2     2740   <No Name>        System.Threading.WaitHandle::WaitOne
 3     3360   <No Name>        System.Threading.WaitHandle::WaitOne
```

2. To open a window that displays the call stack, on the **Debug** menu, point to **Windows**, and then select **Call Stack**.

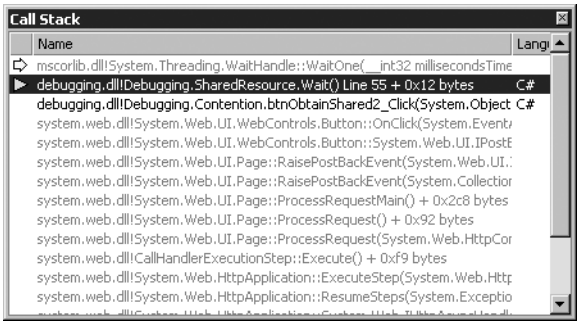

Figure 3.6
Call Stack window

3. To open the Macro Explorer window, on the **Tools** menu, point to **Macros**, and then click **Macro Explorer**.
4. Expand **Samples**, and then expand **VSDebugger**.
5. Right-click **DumpStacks**, and the click **Run**.

The output from the macro appears in the command window and is similar to the following example:

```
>kb
 Index  Function
-------------------------------------------------------------------------------
 1      mscorlib.dll!System.Threading.WaitHandle::WaitOne(__int32
millisecondsTimeout = -1, bool exitContext = false)
*2      debugging.dll!Debugging.SharedResource.Wait()
 3
debugging.dll!Debugging.Contention.btnObtainShared1_Click(System.O
bject sender = {System.Web.UI.WebControls.Button},
System.EventArgs e = {System.EventArgs})
 4
system.web.dll!System.Web.UI.WebControls.Button::OnClick(System.Ev
entArgs e = {System.EventArgs})
 5
system.web.dll!System.Web.UI.WebControls.Button::System.Web.UI.IPo
stBackEventHandler.RaisePostBackEvent(String* eventArgument =
null)
 6
system.web.dll!System.Web.UI.Page::RaisePostBackEvent(System.Web.U
I.IPostBackEventHandler sourceControl =
{System.Web.UI.WebControls.Button}, String* eventArgument = null)
 7
system.web.dll!System.Web.UI.Page::RaisePostBackEvent(System.Colle
ctions.Specialized.NameValueCollection postData =
{System.Web.HttpValueCollection})
 8      system.web.dll!System.Web.UI.Page::ProcessRequestMain()
 9      system.web.dll!System.Web.UI.Page::ProcessRequest()
 10
system.web.dll!System.Web.UI.Page::ProcessRequest(System.Web.HttpC
ontext context = {System.Web.HttpContext})
```

```
 11      system.web.dll!CallHandlerExecutionStep::Execute()
 12
system.web.dll!System.Web.HttpApplication::ExecuteStep(System.Web.
HttpApplication.IExecutionStep step =
{System.Web.HttpApplication.CallHandlerExecutionStep}, bool
completedSynchronously = true)
 13
system.web.dll!System.Web.HttpApplication::ResumeSteps(System.Exce
ption error = null)
 14
system.web.dll!System.Web.HttpApplication::System.Web.IHttpAsyncHa
ndler.BeginProcessRequest(System.Web.HttpContext context =
{System.Web.HttpContext}, System.AsyncCallback cb =
{System.AsyncCallback}, System.Object extraData =
{System.Web.HttpContext})
 15
system.web.dll!System.Web.HttpRuntime::ProcessRequestInternal(Syst
em.Web.HttpWorkerRequest wr)
 16
system.web.dll!System.Web.HttpRuntime::ProcessRequest(System.Web.H
ttpWorkerRequest wr)
 17
system.web.dll!System.Web.Hosting.ISAPIRuntime::ProcessRequest(__i
nt32 ecb, __int32 iWRType)
```

Switch the thread to 2740 to display similar information for the next thread. Only the third frame differs from the first thread's call stack.

```
>kb
 Index  Function
-------------------------------------------------------------------------------
 1       mscorlib.dll!System.Threading.WaitHandle::WaitOne(__int32
millisecondsTimeout = -1, bool exitContext = false)
*2       debugging.dll!Debugging.SharedResource.Wait()
 3
debugging.dll!Debugging.Contention.btnObtainShared2_Click(System.O
bject sender = {System.Web.UI.WebControls.Button},
System.EventArgs e = {System.EventArgs})
```

Switch the thread to 3360 to display similar information for the next thread. Only the third frame differs from the first thread's call stack.

```
>kb
 Index  Function
-------------------------------------------------------------------------------
 1       mscorlib.dll!System.Threading.WaitHandle::WaitOne(__int32
millisecondsTimeout = -1, bool exitContext = false)
*2       debugging.dll!Debugging.SharedResource.Wait()
 3
debugging.dll!Debugging.Contention.btnObtainShared3_Click(System.O
bject sender = {System.Web.UI.WebControls.Button},
System.EventArgs e = {System.EventArgs})
```

You can see that this information is very similar to that obtained using WinDbg and the SOS extension.

Debugging a Hung Process with Visual Studio .NET

Complete the following steps to attach to the process.

► **To debug a hung process**

1. Open a command prompt window, and then type **iisreset** at the prompt to restart IIS.
2. Right-click Contention.aspx in Visual Studio .NET, and then click **View in Browser**.
3. On the **Debug** menu, click **Processes**. A window that displays process information for local or remote machines appears.
4. In the **Processes** dialog box, select the **Show system processes** and **Show processes in all sessions** check boxes.
5. Select **aspnet_wp.exe**, and then click **Attach.** The **Attach to Process** dialog box appears.
6. Make sure that the **Common Language Runtime** item is selected, and then click **OK**.
7. In the **Processes** dialog box, make sure that the Aspnet_wp.exe process is listed in the **Debugged Processes** list, and then click **Close**.

You are now attached to the process and can use the debugger windows to display information about the process.

Conclusion

In this chapter, you have been introduced to some of the problems that can arise when executing multithreaded code. You have also discovered how managed and unmanaged threads are used within ASP.NET.

Using either WinDbg with the SOS.dll extension or Visual Studio .NET, you have been able to trace the execution of the threads within an ASP.NET worker process, and use a dump file to pinpoint the position in the code where contention occurred.

4

Debugging Unexpected Process Termination

This chapter describes how to approach debugging when the ASP.NET process terminates unexpectedly and the error and what caused it are unknown. A native exception might have been thrown, or the process might have been instructed to terminate. Troubleshooting steps for these scenarios are similar, regardless of what caused the problem. This chapter focuses on the latter scenario, and the walk-through uses a COM Interop example to demonstrate unexpected terminations. However, keep in mind that these crashes can be caused by a variety of problems and are not a specific COM Interop issue. In the sample application, ASP.NET calls a COM component that deliberately causes Aspnet_wp.exe process to terminate. "Handling COM Threading in ASP.NET" gives some background information about how ASP.NET uses COM Interop and how the **ASPCompat** page directive works.

Although reproducing this problem on your machine may not give you the exact same results because of differing operating systems and runtime versions, the output should be similar, and the debugging concepts are the same. Also, because all Microsoft® .NET Framework applications use the same memory architecture, you can apply what you learn about COM Interop and debugging fatal exceptions in an ASP.NET context to other .NET environments, such as console applications, Microsoft Windows® operating system applications, and Windows services.

Handling COM Threading in ASP.NET

Any new technology has to be able to work with existing technologies because rewriting existing code can be cost prohibitive, and there is always the possibility of introducing new bugs when rewriting. Before you integrate older COM components with newer ASP.NET code, you should familiarize yourself with COM+ applications. The old COM rules still apply, and the ASP.NET architecture has adapted to accommodate these rules.

Note: For an in-depth discussion of COM threading models, see "COM Threading and Application Architecture in COM+ Applications" available on the MSDN Web site at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomser/html/comthread.asp*.

ASP.NET and COM Interop

Typically, ASP.NET pages call managed .NET objects, so for the most part, programming with ASP.NET does not require intimate knowledge of COM, COM threading, or all of the COM rules. However, when migrating existing Web applications to the .NET Framework, it is sometimes necessary to call native COM components to perform some business functionality.

There are two ways to call native COM components from .NET: *early binding* and *late binding*. When using early binding, the information about the native COM component is known at design time. Early binding provides faster access to COM objects than late binding, and it allows you to include a reference to an imported COM component when you compile your assembly. When you use early binding and import a native COM component for use with the .NET Framework, you need to create an Interop assembly. A .NET assembly must contain type metadata, so you must create an Interop assembly that causes required metadata to be generated from the COM type library. In Microsoft Visual Studio® .NET, you would use Add References or the Tlbimp.exe utility in the .NET Framework SDK. Once you have an Interop assembly, you can create an instance of the native COM component using the "new" keyword in the same way you would create any managed object.

With late binding, information about the native COM component is not known until runtime. To create a late-bound server instance of a COM component, you can use the **HttpServerUtility.CreateObject()** method by passing the component's programmatic identifier (ProgID) to **Server.CreateObject()**.

COM Components and ASP.NET

It is important to consider how COM and ASP.NET instantiate the components that you request. In Windows 2000 and XP, where the COM component is instantiated depends largely on how the threading model of the native COM component is marked in the registry, as either **Free**, **Apartment**, **Neutral**, or **Both**. This chapter uses an example that focuses on **Free** and **Apartment**.

Components Marked Free

When your ASP.NET code calls a COM component that is marked **Free**, the component is instantiated on the same thread pool thread that the ASP.NET page started running on. The thread pool threads that process ASP.NET pages, such as input/output (I/O) completion port threads and worker threads, are initialized as

multithreaded apartment (MTA) threads. Because the component is marked **Free** and the ASP.NET thread is initialized as MTA, no thread switch is necessary and the performance penalty is marginal.

Components Marked Apartment

Traditionally, business COM components that are called from either ASP or MTS/COM+ have been marked **Apartment**. The single-threaded apartment (STA) threading model is not compatible with the default threading model for the ASP.NET thread pool, which is MTA. As a result, calling a native COM component marked **Apartment** from an ASP.NET page results in a thread switch and COM cross-apartment marshalling.

If the COM component marked **Apartment** is not configured in COM+, the component will be instantiated on the Host STA thread of the Aspnet_wp.exe process. If the COM component marked **Apartment** is configured in COM+, a thread switch occurs to a COM+ STA worker thread from the MTA thread that the ASP.NET started running on. In the unconfigured case, if the component marked **Apartment** makes a long running blocking call, no new unconfigured components marked **Apartment** can be instantiated or perform work on that same thread. Under stress, this presents a severe bottleneck. To work around this issue, a new directive called **ASPCompat** was introduced to the **System.Web.UI.Page** object.

How ASPCompat Works

The **ASPCompat** attribute minimizes thread switching due to incompatible COM threading models. More specifically, if a COM component is marked **Apartment**, the **ASPCompat = "true"** directive on an ASP.NET page runs the component marked **Apartment** on one of the COM+ STA worker threads.

Assume that you are requesting a page called UnexpectedCompat.aspx that contains the directive **ASPCompat ="true"**. When the page is compiled, the page compiler checks to see if the page requires **ASPCompat** mode. Because this value is present, the page compiler modifies the generated page class to implement the IHttpAsyncHandler interface, adds methods to implement this interface, and modifies the page class constructor to reflect that **ASPCompatMode** will be used.

The two methods that are required to implement the IHttpAsyncHandler interface are **BeginProcessRequest** and **EndProcessRequest**. The implementation of these methods contains calls to **this.ASPCompatBeginProcessRequest** and **this.ASPCompatEndProcessRequest**, respectively.

You can view the code that the page compiler creates by setting **Debug="true"** in the <compilation> section of the web.config or machine.config files. In our example the file location of the code is C:\Windows\Microsoft.NET\Framework\v1.0.3705\Temporary ASP.NET Files\debugging\34374a37\79f52287\0n9-hgqi.cs.

The following is the source code for the class definition, the class constructor, and the implemented methods:

```
public class UnexpectedCompat_aspx : Debugging.Unexpected,
System.Web.SessionState.IRequiresSessionState, System.Web.IHttpAsyncHandler {
...
public UnexpectedCompat_aspx() {
          ...
          this.AspCompatMode = true;
     }
...
public virtual System.IAsyncResult BeginProcessRequest(System.Web.HttpContext
context, System.AsyncCallback cb, object data)
{
     return this.AspCompatBeginProcessRequest(context, cb, data);
}

public virtual void EndProcessRequest(System.IAsyncResult ar)
{
     this.AspCompatEndProcessRequest(ar);
}
```

The **Page.ASPCompatBeginProcessRequest()** method determines if the page is already running on a COM+ STA worker thread. If it is, the call can continue to execute synchronously.

A more common scenario is a page running on a .NET MTA thread-pool thread. **ASPCompatBeginProcessRequest()** makes an asynchronous call to the native function **ASPCompatProcessRequest()** within Aspnet_isapi.dll. The following describes what happens when invoking COM+ in the latter scenario:

1. The native **ASPCompatProcessRequest()** function constructs an **ASPCompatAsyncCall** class that contains the callback to the ASP.NET page and a context object created by ASP.NET. The native **ASPCompatProcessRequest()** function then calls a method that creates a COM+ activity and posts the activity to COM+.
2. COM+ receives the request and binds the activity to an STA worker thread.

3. Once the thread is bound to an activity, it calls the **ASPCompatAsyncCall::OnCall()** method, which initializes the intrinsics so they can be called from the ASP.NET (this is similar to classic ASP code). This function calls back into managed code so that the **ProcessRequest()** method of the page can continue executing.
4. The page callback is invoked and the **ProcessRequest()** function on that page continues to run on that STA thread. This process reduces the number of thread switches required to run native COM components marked **Apartment**.
5. Once the ASP.NET page finishes executing, it calls **Page.ASPCompat EndProcessRequest()** to complete the request.

Scenario: Unexpected Process Termination

Now that you know how COM threading models relate to ASP.NET, let's see how problems can occur when mixing native and managed code, and how to fix them.

In this scenario, a Server Application Unavailable error message is displayed in the browser, and there is an "aspnet_wp.exe stopped unexpectedly" entry in the application event log. Other processes continue to run, but the Aspnet_wp.exe process recycles, thus causing any related state or cache to be purged. These errors are not diagnostic — they don't point to the cause of the problem. It is known, however, that because the process terminated, something caused **kernel32!ExitProcess()** to be called. A snapshot of the process taken before it recycles may provide clues.

This scenario may present itself in many ways, including the following:

- **Scenario 1:** The big picture could be that a production Web site crashes and the connection to the browser client is lost.
- **Scenario 2:** A more specific example might be that a client browses to a page on a financial Web site and then tries to check stock quotes. The client makes an entry on the page, clicks Enter, and the Server Application Unavailable error message returns in the browser. The following entry appears in the Application log: "aspnet_wp.exe(PID:1234) stopped unexpectedly". Yet these messages are vague, and it's unclear what's wrong.

Breaking Down the Thought Processes

The flowchart on the next page shows a walkthrough that helps discover more information about the problem so you can troubleshoot this scenario. The goal is to teach a core set of techniques that can be applied to similar production scenarios and architectures.

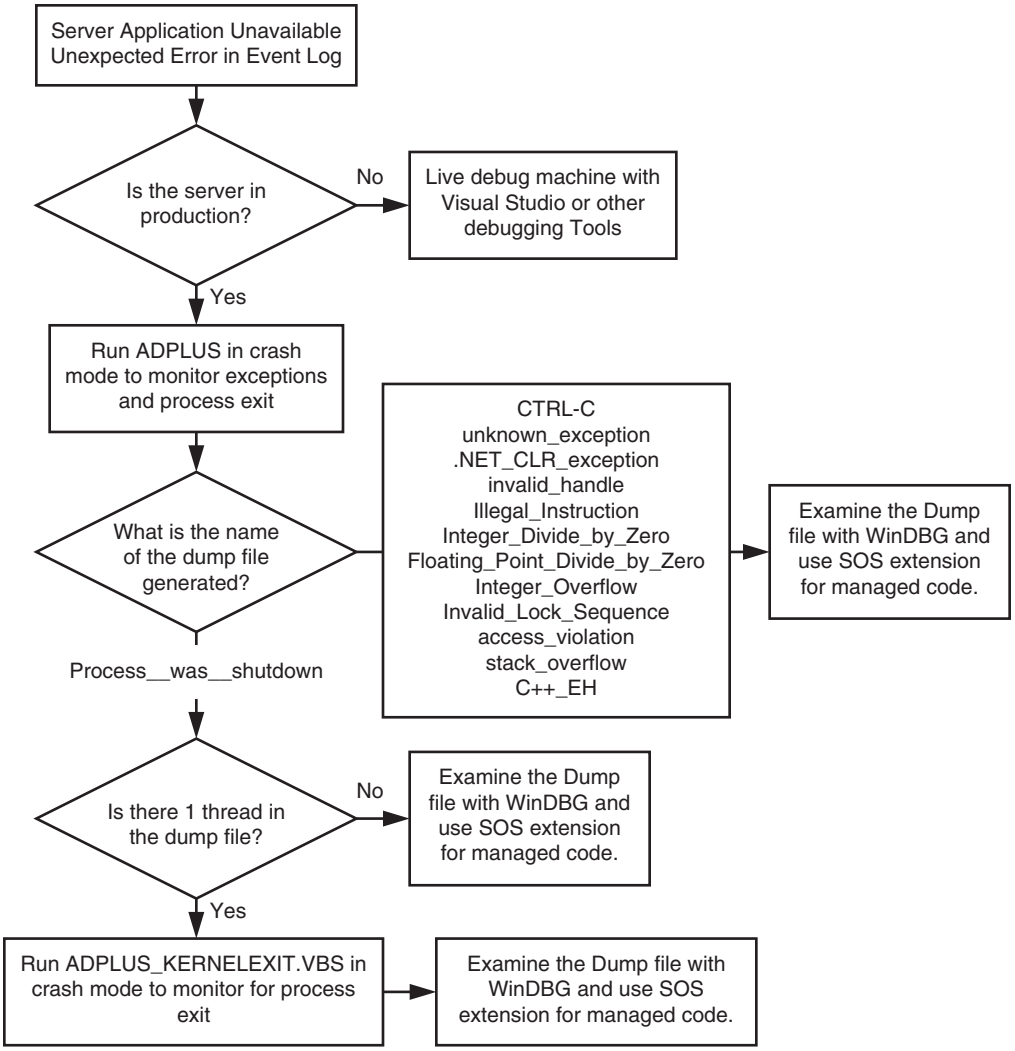

Figure 4.1
Thought Process Flowchart

Although the thought processes are described in the walkthrough that follows, let's look at the flowchart in more detail.

Is the Server in Production?

If the server is in production and the error is reproducible, you can create a memory dump to examine thread activity at the time the problem occurs. The amount of time before the problem reproduces dictates how long you have to wait for the dump to be created after you attach the debugger. If the server is not in production, debug with your favorite tool.

Run ADPlus to Monitor the Process for Exceptions

Run ADPlus in **-crash** mode against the Aspnet_wp.exe process, and then reproduce the problem to create a dump file.

ADPlus in **-crash** mode handles both first-chance and second-chance exceptions. The difference between the two is the stage at which they are handled.

- A first-chance exception may mean that something has gone wrong, but the application has handled the problem. ADPlus in **-crash** mode handles first-chance exceptions by outputting a stack trace to a log, creating a mini-dump, and continuing to execute.
- A second-chance exception is an unhandled first-chance exception. ADPlus handles second-chance exceptions by logging a stack trace, generating a full memory dump of the process, and terminating the debugger and process.

If ADPlus breaks in when the process is in the final stages of termination, the dump will only show one thread (why there is one thread is explained later in this chapter). You want to catch the thread activity before it terminates. A customized ADPlus.vbs script may accomplish this.

For more information on ADPlus, see article Q286350 "HOWTO: Use Autodump+ to Troubleshoot 'Hangs' and 'Crashes'" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q286350*. Also, if you want torun ADPlus through Terminal Server, see article Q323478, "You Cannot Debug Through a Terminal Server Session" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q323478*.

Examining the Dump File

If the name of the dump file does not contain the text Process_was_shutdown, examine the dump file with WinDbg and SOS.dll. If the name does contain the text Process_was_shutdown, check how many threads are in the dump file. If there is only one thread, run a modified version of ADPlus (described later in this chapter) to break in and create a dump before the process exits. Examine the dump file with WinDbg and SOS.dll. Consider what both the native and managed stack traces show. Determine why an exception or shutdown occurred and if the issue can be addressed with a workaround or fix.

Unexpected Process Termination Walkthrough

Now that you have seen how Server Application Unavailable errors can occur and how you can approach debugging them, here is a walkthrough that describes a simplified scenario of a realistic occurrence. The purpose is to illustrate trouble-shooting techniques that help you solve the problem.

Here's the scenario: The Aspnet_wp.exe process has terminated, but the cause is not obvious. Error messages show, but they are not descriptive enough to solve the problem. You must trace the call of execution even if it hops across multiple threads. Ultimately, you discover that the process terminated because of the code. The managed code makes a native call that aborts the process.

The code also incorporates the concept of COM Interop and includes multiple ATL classes that use different threading models. The first button-click event calls an unconfigured COM component marked **Apartment** and the second button-click event calls an unconfigured COM component marked **Free**. The walkthrough concentrates on the former scenario and shows how the .NET framework handles the calls.

In this scenario, you will perform the following steps:

1. Browse to the ASP.NET page and consider the values displayed in the processModel table.
2. Follow the thought processes in the previous flowchart and find the errors presented both in the browser and event log.
3. Run ADPlus to monitor the process, trap the error, and create a dump file.
4. Look at the dump file name and determine if the process terminated or the process raised a second-chance exception.
5. Use WinDbg and SOS.dll to peruse the data within the dump file, specifically the native and managed threads. By merging the managed and native call stacks, you see the managed code calling to the native code and point to the portion of the native code at fault.

Baseline View

First, gather some baseline data for comparisons with subsequent output.

► **To view Unexpected.aspx**

1. Open a command prompt window, and then type **iisreset** at the prompt to restart IIS.
2. Open a browser window, and then browse to *http://localhost/debugging /unexpected.aspx*.

Figure 4.2 presents the baseline values with which to compare subsequent output. Note the information displayed, especially the data in the table. Some of the labels are self-explanatory. (For more information about these counters, see Chapter 2, "Debugging Memory Problems.") Here, the **ProcessID**, **RequestCount**, **Status**, and **ShutdownReason** fields change as the exercise continues. The **RequestCount** is zero because the value was queried while the first request was executing, so the request hasn't finished yet. **RequestCount** really means the number of requests completed. The **Status** is **Alive**, which indicates that the process is running.

Figure 4.2
Baseline data for Unexpected.aspx

You can explore the code and read through the comments by opening another instance of Unexpected.aspx.cs in Visual Studio .NET or Notepad. Notably, the STA button-click event handler is managed code that makes a call to native code and passes parameters that ultimately cause the application to abort or crash.

To start the exercise, click the **Call STA COM Object** button. The browser returns the following error message:

> Server Application Unavailable
>
> The web application you are attempting to access on this web server is currently unavailable. Please hit the "Refresh" button in your web browser to retry your request.
>
> **Administrator Note:** An error message detailing the cause of this specific request failure can be found in the system event log of the Web server. Please review this log entry to discover what caused this error to occur.

The application event log shows the following error:

```
Event Type:       Error
Event Source:     ASP.NET 1.0.3705.0
Event Category:   None
Event ID:         1000
Date:             6/7/2002
Time:             12:59:54 PM
```

```
User:            N/A
Computer:        machineName
Description:
aspnet_wp.exe  (PID: 2920) stopped unexpectedly.
```

Close and reopen the browser window, and then browse to *http://localhost/debugging/unexpected.aspx*. Figure 4.3 shows the tables that should be displayed in the browser.

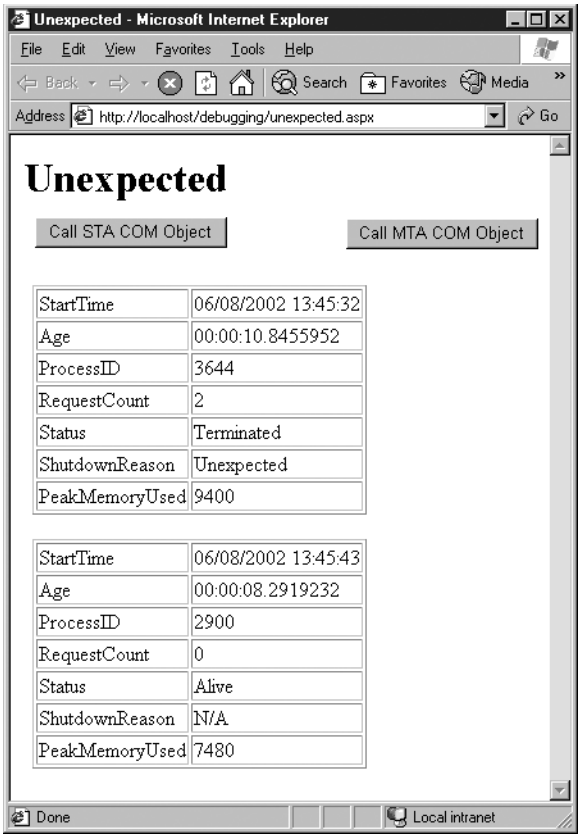

Figure 4.3
Data from Unexpected.aspx following the crash

The **RequestCount** increments to two—one to browse the page for viewing and then one for the first button click. The browser shows the **Status** as **Terminated** and the **ShutdownReason** as Unexpected. As described in "IIS 5.*x* Process Model" in Chapter1, the processModel health monitoring launches a new Aspnet_wp.exe process after the first process terminates. A new process is accompanied by a new ProcessID.

Debugging a Dump File with WinDbg

Without looking at the code, you can speculate about what caused the crash — a stack overflow, code that called **ProcessExit()** or **TerminateProcess()**, or an external monitoring process that shut down the Aspnet_wp.exe process. To obtain a snapshot of what the threads are doing, you can run ADPlus in **-crash** mode against Aspnet_wp.exe using the following command line.

```
adplus.vbs -crash -pn aspnet_wp.exe -quiet
```

Note: The **-quiet** switch ensures that you can still create a dump file if the symbol path for ADPlus is not set.

To cause the failure, click the **Call STA COM Object** button again. A minimized window runs CDB.exe; the tool waits for a call to **ProcessExit()** and then creates the dump and the text files.

When the debugger window disappears, the full dump is created. You can check the C:\Debuggers directory for the most recent \Crash_Mode folder. The file's name will be similar to **PID-1692__ASPNET_WP.EXE__Process_was_shutdown__full_2002-06-07_01-04-02-746_069C.dmp**.

Exploring Further

Try these ideas to learn more:

- **Use the debug build.** In the examples in this document, a release build is used. Server messages and dump output differ with a debug build.
- **Examine the log that ADPlus generates.** Consider the history of what the system loads at the time of the crash.

Examining the Dump File

Examine the dump file that was created when Aspnet_wp.exe process was recycled.

► **To examine the dump file using WinDbg**

1. On the **Start** menu, point to **Debugging Tools for Windows**, and then click **WinDbg**.
2. On the **File** menu, click **Open Crash Dump**.
3. Select the appropriate dump file, and then click **Open**.

4. If you are prompted to "Save Base Workspace Information," click **No**.
5. If a Disassembly window pops up, close the window, and then on the **Window** menu, click **Automatically Open Disassembly**.

Symbol paths must be entered for the files that will be used by the debugger to analyze the dump file. The versions of symbol files needed on the debug computer are those that match the version on the system that produced the dump file. Include symbols from the .NET Framework.SDK, Visual Studio .NET, and symbols shipped with the following System32 folder: \%WINDIR%\system32.

To enter the symbol paths, you may do one of the following:

- From the command line, type:

```
.sympath SRV*C:\symbols\debugginglabs*http://msdl.microsoft.com/download/
symbols;C:\
symbols\debugginglabs;C:\Program Files\Microsoft Visual Studio
.NET\FrameworkSDK\symbols;C:\windows\system32
```

> **Note:** If you have only installed the .NET Framework SDK (without Visual Studio .NET) then you need to replace the C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK \symbols with C:\Program Files\Microsoft.NET\FrameworkSDK\symbols.

- Enter the symbol path for WinDbg with the _NT_SYMBOL_PATH environment variable.
- On the **File** menu in WinDbg, click **Symbol File Path**, and then type:

```
SRV*C:\symbols\debugginglabs*http://msdl.microsoft.com/download/symbols;C:\
symbols\debugginglabs;C:\Program Files\Microsoft Visual Studio
.NET\FrameworkSDK\symbols;C:\windows\system32
```

The "SRV" in the path tells WinDbg to go to an external symbol server and copy symbols into the local symbol cache.

► **To use these same symbol paths for other dumps**

- On the **File** menu in WinDbg, click **Save Workspace As**, and then type a name for the saved paths, such as **.NET Debugging Symbols**.

As you look at the dump output, examine the call stacks by typing **~*kb** (for example), and then verify that all symbols have been loaded successfully. For example, if you see the following message, you need to modify your symbol path to include the correct symbols.

```
040af6d4 10001017 MSVCR70!abort+0xe
*** WARNING: Unable to verify checksum for DebuggingCOM.dll
*** ERROR: Symbol file could not be found. Defaulted to export symbols for
DebuggingCOM.dll -
WARNING: Stack unwind information not available. Following frames may be wrong.
040af6f4 77d281a5 DebuggingCOM+0x1017
```

To address this problem, modify the symbol path to include the symbols for the DLL in question. For example, use the following command:

```
.sympath+ c:\inetpub\wwwroot\Debugging\DebuggingCOM\Release
```

Then type **.reload**.

Examining the Native and Managed Output

Consider how many threads you are dealing with. Type tilde (~) in the command window to display all threads in the current process. In this example, there is one thread:

```
0:000> ~
  0  Id: 7aa.208 Suspend: -1 Teb: 7ffd6000 Unfrozen
```

This thread is in Suspend state, unfrozen, which means that the system will run the thread when appropriate.

Look at the native call stack for the thread that was running when the dump occurred. The call stack tracks function calls and the parameters passed into these functions. The **k** command displays the stack frame of a given thread. If there were more than one thread, you would run the command **~*k 200** and list the call stacks for all threads loaded. Because there is only one thread, you use **k 200** to display the entire call stack for this thread (given that this thread is under 200 stack frames in size).

```
0:000> k 200
ChildEBP RetAddr
0387ff44 77f7e76f SharedUserData!SystemCallStub+0x4
0387ff48 77e775b7 ntdll!NtDelayExecution+0xc
0387ffa0 792d05b2 kernel32!SleepEx+0x61
0387ffb4 77e802ed mscorwks!ThreadpoolMgr::TimerThreadStart+0x30
03880038 792067d5 kernel32!BaseThreadStart+0x37
00d1203b 000000ff mscorwks!CreateTypedHandle+0x16
```

Note: Your output may be different to the sample shown above because of timing and environment factors.

There is only one thread running because the process is in the final stages of termination. All threads except the last one have been destroyed, and managed threads have been cleaned up. If you were to run the SOS **!threads** command, you would see "XXX" for these managed thread IDs that do not map to native threads in the process.

Given the simplicity of the scenario, the thread that is output here may actually be the thread that calls **kernel32!ExitProcess**, but this would be unusual in production mode. A production Web server most likely will serve many pages simultaneously and many threads will be running, thus complicating the troubleshooting process.

Also, because the process is in a shutdown state, managed structures in the process's memory may be inconsistent with native structures, or they may be unreliable. It is difficult to determine what thread terminated the process and whether this call was directly invoked from an ASP.NET page, so the next step is to attach a custom debugger script to the process and set a breakpoint on one of the functions in the call stack that is called before the process exits. Other threads (managed or native) may or may not provide more information.

Determining where to set a breakpoint depends on whether you have matching symbols for the modules in question. Sometimes having hotfixes installed can affect whether the symbols will resolve to the DLLs or not, and this may mean that you cannot set a breakpoint.

To create a dump for this exercise, use a customized version of ADPlus.vbs called ADPlus_KernelExit.vbs. (For download instructions, see Chapter 1, "Introduction to Production Debugging for .NET Framework Applications.") The changes to ADPlus are located within the **CreateCDBScript()** function, which creates the .cfg file that configures the CDB debugger when it attaches to the process.

Make sure that ADPlus_KernelExit.vbs is located in the C:\Debuggers folder. Symbols are required, so make sure that you are connected to the Internet or that you have downloaded the symbols for **kernel32.pdb** locally. Internet access is adequate because this tool sets up the path to the Microsoft external symbol server for you.

The following lines have been added to the ADPlus.vbs script:

```
objTextFile.Writeline "* -- Setup Symbol Path to use external symbol server --"
objTextFile.Writeline ".sympath+ SRV*c:\symbols\debugginglabs*http://
msdl.microsoft.com/download/symbols;c:\symbols
\debugginglabs"
objTextFile.Writeline "* ------------------------------------------------------------
---"
objTextFile.Writeline "*"
objTextFile.Writeline "*"
objTextFile.Writeline "*"
objTextFile.Writeline "* -- Make sure the symbols are loaded for mscoree.dll --"
objTextFile.Writeline "x kernel32!ExitProcess"
objTextFile.Writeline "* ------------------------------------------------------------
---"
objTextFile.Writeline "*"
objTextFile.Writeline "*"
objTextFile.Writeline "*"

'Since this is a breakpoint, any \ need to be escaped with \\
Dim EscapedCrashDir
EscapedCrashDir = Replace(ShortCrashDir, "\", "\\")
```

```
objTextFile.Writeline "* -- Add a breakpoint for CAsyncPipeManager::Close --"
objTextFile.Writeline "bp kernel32!ExitProcess " & Chr(34) & ".echo Encountered
kernel32!ExitProcess breakpoint. Dumping process ...;.dump /mfh " &
EscapedCrashDir & "\\" & "PID-" & pid & "__" & packagename &
"__Kernel32ExitProcess__full.dmp;q" & Chr(34)
objTextFile.Writeline "* ------------------------------------"
```

As noted, ADPlus_KernelExit.vbs sets the symbol path to use the Microsoft public symbol server. The tool verifies that the symbols can be loaded for the **kernel32!ExitProcess()** function. Because you know that the process terminates, the tool sets a breakpoint on **kernel32!ExitProcess()** to create a full dump file when the breakpoint is hit.

Along with a user dump, the ADPlus_KernelExit.vbs script also generates a text file that provides a history of exceptions, and dumps the native stack traces for those exceptions (the same as in the unmodified ADPlus.vbs). Keep in mind that throwing many exceptions over an extended period of time can produce a very large text file.

Throwing as few exceptions as possible is preferable because catching exceptions is extremely resource intensive. You can even disable the creation of the log file if it is too obtrusive. For more information, see "ASP.NET Performance Tips and Best Practices" on the GotDotNet Web site at *http://www.gotdotnet.com/team/asp/ASP.NET Performance Tips and Tricks.aspx.*

► **To prepare for debugging**

1. Open a command prompt window, and then type **iisreset** at the prompt to restart IIS.
2. Open a new browser window, and then browse to the ASP.NET sample page at *http://localhost/debugging/unexpected.aspx*.
3. Run ADPlus_KernelExit.vbs in **-crash** mode and specify which Aspnet_wp.exe process to attach to.
4. In the command prompt window, change to the debuggers folder and type:

   ```
   adplus_kernelexit.vbs -crash -pn aspnet_wp.exe –quiet
   ```

   A minimized CDB.exe window shows on the task bar.
5. On the ASP.NET page, click **Call STA COM Object** and the CDB.exe window becomes active. The browser should display the following message:

   Server Application Unavailable

   The web application you are attempting to access on this web server is currently unavailable. Please hit the "Refresh" button in your web browser to retry your request.

> **Administrator Note:** An error message detailing the cause of this specific request failure can be found in the system event log of the Web server. Please review this log entry to discover what caused this error.

6. Look in the C:\debuggers directory for the most recent \Crash_Mode folder that has a name similar to **Crash_Mode__Date_06-10-2002__Time_15-56-01PM**. The dump file has a name similar to **PID-3968__ASPNET_WP.EXE__Kernel32ExitProcess__full.dmp**.
7. Open WinDbg, and then open the crash dump and set the symbol path like you did before.
8. Select the **Reload** checkbox, or type **.reload** in the command window.

► To examine the native threads

- List the native threads using the tilde (~)command.

```
0:010> ~
   0  Id: ecc.8d8 Suspend: 1 Teb: 7ffde000 Unfrozen
   1  Id: ecc.e4 Suspend: 1 Teb: 7ffdd000 Unfrozen
   2  Id: ecc.d28 Suspend: 1 Teb: 7ffdc000 Unfrozen
   3  Id: ecc.670 Suspend: 1 Teb: 7ffdb000 Unfrozen
   4  Id: ecc.9e8 Suspend: 1 Teb: 7ffda000 Unfrozen
   5  Id: ecc.d08 Suspend: 1 Teb: 7ffd9000 Unfrozen
   6  Id: ecc.d88 Suspend: 1 Teb: 7ffd8000 Unfrozen
   7  Id: ecc.55c Suspend: 1 Teb: 7ffd7000 Unfrozen
   8  Id: ecc.c40 Suspend: 1 Teb: 7ffd6000 Unfrozen
   9  Id: ecc.f54 Suspend: 1 Teb: 7ffd5000 Unfrozen
  10  Id: ecc.59c Suspend: 1 Teb: 7ffd4000 Unfrozen
  11  Id: ecc.f40 Suspend: 1 Teb: 7ffaf000 Unfrozen
  12  Id: ecc.cfc Suspend: 1 Teb: 7ffae000 Unfrozen
  13  Id: ecc.564 Suspend: 1 Teb: 7ffad000 Unfrozen
```

- There are considerably more threads in this dump file than in previous dump. List the call stacks of the native threads using the **~*k 200** command.

```
0:010> ~*k 200

   0  Id: ecc.8d8 Suspend: 1 Teb: 7ffde000 Unfrozen
ChildEBP RetAddr
0012f874 77f7e76f SharedUserData!SystemCallStub+0x4
0012f878 77e775b7 ntdll!NtDelayExecution+0xc
0012f8d0 77e61bf1 kernel32!SleepEx+0x61
0012f8dc 00422c17 kernel32!Sleep+0xb
0012ff44 004237db aspnet_wp!wmain+0x30b [e:\dna\src\xsp\wp\main.cxx @
   222]
0012ffc0 77e7eb69 aspnet_wp!wmainCRTStartup+0x131
   [f:\vs70builds\9111\vc\crtbld\crt\src\crtexe.c @ 379]
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

```
   1  Id: ecc.e4 Suspend: 1 Teb: 7ffdd000 Unfrozen
ChildEBP RetAddr
009df2dc 77f7f4af SharedUserData!SystemCallStub+0x4
009df2e0 77e7788b ntdll!NtWaitForSingleObject+0xc
009df344 77e79d6a kernel32!WaitForSingleObjectEx+0xa8
009df354 771da2e7 kernel32!WaitForSingleObject+0xf
009df370 772afc14 ole32!GetToSTA+0x6d
009df394 772af180
    ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0xe2
009df464 771cf0af ole32!CRpcChannelBuffer::SendReceive2+0xa6
009df4ec 77cc28f7 ole32!CAptRpcChnl::SendReceive+0x95
009df514 77d28c20 RPCRT4!NdrClientInitializeNew+0x17
009df550 771c07c8 RPCRT4!NdrProxySendReceive+0x41
009df63c 771c955f ole32!CoTaskMemFree+0xf
009df654 792107dc ole32!CStdIdentity::CInternalUnk::Release+0x3b
009df684 00177226 mscorwks!SafeRelease+0x56
WARNING: Frame IP not in any known module. Following frames may be wrong.
00000000 00000000 0x177226

   2  Id: ecc.d28 Suspend: 1 Teb: 7ffdc000 Unfrozen
ChildEBP RetAddr
00adff04 77f7e76f SharedUserData!SystemCallStub+0x4
00adff08 77e775b7 ntdll!NtDelayExecution+0xc
00adff60 77e61bf1 kernel32!SleepEx+0x61
00adff6c 792d00bc kernel32!Sleep+0xb
00adffb4 77e802ed mscorwks!ThreadpoolMgr::GateThreadStart+0x4c
00adffec 00000000 kernel32!BaseThreadStart+0x37

   3  Id: ecc.670 Suspend: 1 Teb: 7ffdb000 Unfrozen
ChildEBP RetAddr
00ddfefc 77f7f4af SharedUserData!SystemCallStub+0x4
00ddff00 77e7788b ntdll!NtWaitForSingleObject+0xc
00ddff64 77e79d6a kernel32!WaitForSingleObjectEx+0xa8
00ddff74 0042289c kernel32!WaitForSingleObject+0xf
00ddff80 7c00fbab aspnet_wp!DoPingThread+0x10
    [e:\dna\src\xsp\wp\main.cxx @ 412]
00ddffb4 77e802ed MSVCR70!_endthread+0xaa
00ddffec 00000000 kernel32!BaseThreadStart+0x37

   4  Id: ecc.9e8 Suspend: 1 Teb: 7ffda000 Unfrozen
ChildEBP RetAddr
011bfe80 77f7f49f SharedUserData!SystemCallStub+0x4
011bfe84 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
011bff20 77e74c70 kernel32!WaitForMultipleObjectsEx+0x12c
011bff38 791cccf6 kernel32!WaitForMultipleObjects+0x17
011bffa0 791ccc6b mscorwks!DebuggerRCThread::MainLoop+0x90
011bffac 791ccc1e mscorwks!DebuggerRCThread::ThreadProc+0x55
011bffb4 77e802ed mscorwks!DebuggerRCThread::ThreadProcStatic+0xb
011bffec 00000000 kernel32!BaseThreadStart+0x37

   5  Id: ecc.d08 Suspend: 1 Teb: 7ffd9000 Unfrozen
ChildEBP RetAddr
0335ff08 77f7e76f SharedUserData!SystemCallStub+0x4
```

```
0335ff0c 77e775b7 ntdll!NtDelayExecution+0xc
0335ff64 77e61bf1 kernel32!SleepEx+0x61
0335ff70 791cde2c kernel32!Sleep+0xb
0335ffb4 77e802ed mscorwks!GCHeap::FinalizerThreadStart+0x1f9
0335ffec 00000000 kernel32!BaseThreadStart+0x37

   6  Id: ecc.d88 Suspend: 1 Teb: 7ffd8000 Unfrozen
ChildEBP RetAddr
0347ff4c 77f7ef9f SharedUserData!SystemCallStub+0x4
0347ff50 77e73b3f ntdll!ZwRemoveIoCompletion+0xc
0347ff7c 792cf8b8 kernel32!GetQueuedCompletionStatus+0x27
0347ffb4 77e802ed
    mscorwks!ThreadpoolMgr::CompletionPortThreadStart+0x46
0347ffec 00000000 kernel32!BaseThreadStart+0x37

   7  Id: ecc.55c Suspend: 1 Teb: 7ffd7000 Unfrozen
ChildEBP RetAddr
0379ff1c 77f7f4af SharedUserData!SystemCallStub+0x4
0379ff20 77e7788b ntdll!NtWaitForSingleObject+0xc
0379ff84 77e79d6a kernel32!WaitForSingleObjectEx+0xa8
0379ff94 792cf5dc kernel32!WaitForSingleObject+0xf
0379ffb4 77e802ed mscorwks!ThreadpoolMgr::WorkerThreadStart+0x2e
0379ffec 00000000 kernel32!BaseThreadStart+0x37

   8  Id: ecc.c40 Suspend: 1 Teb: 7ffd6000 Unfrozen
ChildEBP RetAddr
0394ff44 77f7e76f SharedUserData!SystemCallStub+0x4
0394ff48 77e775b7 ntdll!NtDelayExecution+0xc
0394ffa0 792d05b2 kernel32!SleepEx+0x61
0394ffb4 77e802ed mscorwks!ThreadpoolMgr::TimerThreadStart+0x30
0394ffec 00000000 kernel32!BaseThreadStart+0x37

   9  Id: ecc.f54 Suspend: 1 Teb: 7ffd5000 Unfrozen
ChildEBP RetAddr
03c7fe24 77f7efff SharedUserData!SystemCallStub+0x4
03c7fe28 77cc1ac9 ntdll!NtReplyWaitReceivePortEx+0xc
03c7ff90 77cc167e RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0xf6
03c7ff94 77cc1505 RPCRT4!RecvLotsaCallsWrapper+0x9
03c7ffac 77cc1670 RPCRT4!BaseCachedThreadRoutine+0x64
03c7ffb4 77e802ed RPCRT4!ThreadStartRoutine+0x16
03c7ffec 00000000 kernel32!BaseThreadStart+0x37

  10  Id: ecc.59c Suspend: 1 Teb: 7ffd4000 Unfrozen
ChildEBP RetAddr
03daf618 7923b594 kernel32!ExitProcess
03daf624 792176bb mscorwks!SafeExitProcess+0x38
03daf630 79210cdd mscorwks!ForceEEShutdown+0x3f
03daf644 7917b297 mscorwks!CorExitProcess+0x5a
03daf654 7c00bb88 mscoree!CorExitProcess+0x3d
03daf65c 7c00440a MSVCR70!__crtExitProcess+0x25
03daf668 7c00f791 MSVCR70!_cinit+0x101
```

```
03daf678 7c012ac8 MSVCR70!_exit+0xe
03daf6c0 7c012799 MSVCR70!raise+0xae
03daf6cc 10005877 MSVCR70!abort+0xe
03daf6d0 77cc2f58 DebuggingCOM!CSTA::RaiseError+0x17
    [c:\inetpub\wwwroot\debugging\debuggingcom\sta.cpp @ 50]
03daf6f0 77d281a5 RPCRT4!Invoke+0x30
03dafad8 77d28d3e RPCRT4!NdrStubCall2+0x1fb
03dafb30 7713bb3d RPCRT4!CStdStubBuffer_Invoke+0x3f
03dafb90 772aec81 OLEAUT32!CUnivStubWrapper::Invoke+0xe1
03dafecc 77d43d6a ole32!StubInvoke+0xa5
03daff2c 77d43dd0 USER32!DispatchMessageWorker+0x2e9
03daff38 771e04c8 USER32!DispatchMessageW+0xb
03daff6c 771dbdc5 ole32!CDllHost::STAWorkerLoop+0x52
03daff8c 771dbd0e ole32!CDllHost::WorkerThread+0xb8
03daff90 771debb7 ole32!DLLHostThreadEntry+0x9
03daffa8 771dec16 ole32!CRpcThread::WorkerLoop+0x1e
03daffb4 77e802ed ole32!CRpcThreadCache::RpcWorkerThreadEntry+0x1a
03daffec 00000000 kernel32!BaseThreadStart+0x37

  11  Id: ecc.f40 Suspend: 1 Teb: 7ffaf000 Unfrozen
ChildEBP RetAddr
03f0ff1c 77f7e76f SharedUserData!SystemCallStub+0x4
03f0ff20 77e775b7 ntdll!NtDelayExecution+0xc
03f0ff78 77e61bf1 kernel32!SleepEx+0x61
03f0ff84 771c987b kernel32!Sleep+0xb
03f0ff90 771debb7 ole32!CROIDTable::WorkerThreadLoop+0x12
03f0ffa8 771dec16 ole32!CRpcThread::WorkerLoop+0x1e
03f0ffb4 77e802ed ole32!CRpcThreadCache::RpcWorkerThreadEntry+0x1a
03f0ffec 00000000 kernel32!BaseThreadStart+0x37

  12  Id: ecc.cfc Suspend: 1 Teb: 7ffae000 Unfrozen
ChildEBP RetAddr
0403ff20 77f7f4af SharedUserData!SystemCallStub+0x4
0403ff24 77e7788b ntdll!NtWaitForSingleObject+0xc
0403ff88 771debdf kernel32!WaitForSingleObjectEx+0xa8
0403ffa8 771dec16 ole32!CRpcThread::WorkerLoop+0x58
0403ffb4 77e802ed ole32!CRpcThreadCache::RpcWorkerThreadEntry+0x1a
0403ffec 00000000 kernel32!BaseThreadStart+0x37

  13  Id: ecc.564 Suspend: 1 Teb: 7ffad000 Unfrozen
ChildEBP RetAddr
0413ff30 77f7e76f SharedUserData!SystemCallStub+0x4
0413ff34 77e775b7 ntdll!NtDelayExecution+0xc
0413ff8c 792d0444 kernel32!SleepEx+0x61
0413ffb4 77e802ed mscorwks!ThreadpoolMgr::WaitThreadStart+0x22
0413ffec 00000000 kernel32!BaseThreadStart+0x37
```

Thread Discussion

The following table summarizes the functions that each thread was executing at the point the dump file was generated. These functions were extracted from the previous thread output.

Table 4.1: Summary of thread state

ThreadID	Purpose
0	aspnet_wp!wmain
1	ole32!GetToSTA
2	mscorwks!ThreadpoolMgr::GateThreadStart
3	aspnet_wp!DoPingThread
4	mscorwks!DebuggerRCThread::MainLoop
5	mscorwks!GCHeap::FinalizerThreadStart
6	mscorwks!ThreadpoolMgr::CompletionPortThreadStart
7	mscorwks!ThreadpoolMgr::WorkerThreadStart
8	mscorwks!ThreadpoolMgr::TimerThreadStart
9	RPCRT4!RecvLotsaCallsWrapper
10	ole32!DLLHostThreadEntry (EXIT THREAD)
11	ole32!CRpcThread::WorkerLoop
12	ole32!CRpcThread::WorkerLoop
13	mscorwks!ThreadpoolMgr::WaitThreadStart

WinDbg without SOS.dll doesn't display the managed call stack; instead, it displays native call stacks. However, some of the managed calls are wrappers for native calls.

Note: When attaching to a process with Visual Studio .NET in mixed mode (native and managed), Visual Studio .NET displays both managed and native thread information in one window.

Let's examine more extensively what the threads are and what they're doing:

- Thread 0 is the ASP.NET main thread as shown by the call to **aspnet_wp!wmain.** This thread has initialized the runtime, connected the named pipes back to InetInfo.exe, and created the ping thread. Once it has completed the initialization, it loops, waiting for the process to exit.
- Thread 1 is a managed thread that has both managed and native code running on it, along with a call to the **ole32!GetToSTA** function. To learn more about this thread, you'll use the SOS.dll and SieExtPub.dll managed extension, which is explained later in this chapter.
- Threads 2, 6, 7, 8, and 13 are ThreadpoolMgr threads. Note the calls to **mscorwks!ThreadpoolMgr::GateThreadStart**,

mscorwks!ThreadpoolMgr::WorkerThreadStart, and **mscorwks!ThreadpoolMgr::TimerThreadStart**, respectively. These threads are present in ASP.NET (but not always in console applications) because of the ASP.NET use of the thread pool.

- Thread 2 is a GateThread that monitors the health of the thread pool. It is responsible for injecting and retiring worker and I/O threads based on indicators, such as CPU utilization, garbage-collection frequency, and thread starvation. There is only one thread of this type. It is created when the first I/O or work request is executed.
- Thread 7 is a managed Threadpool worker thread. This thread is waiting for work to be assigned to it.
- Thread 8 is a TimerThread that is used to manage timed callbacks specified using the **System.Threading.Timer** class. There is only one thread of this type, and it is created when the first **System.Threading.Timer** object is created.
- Thread 6 is an I/O completion port thread that is waiting for a managed or native work item.
- Thread 13 is a WaitThread that is used to wait on handles. When the wait is signaled or times out, a callback is executed on a worker thread. Each wait thread can wait on up to 64 distinct wait handles concurrently. New wait threads are created as necessary.

- Thread 3 is the ASP.NET ping thread as shown by the call to **aspnet_wp!DoPingThread**. This thread responds to the pings it receives from the health-monitoring thread in InetInfo.exe.
- Thread 4 is a debugger thread as shown by the call to **mscorwks!DebuggerRCThread**. All managed processes have a debugger thread, even if there is no debugger attached. There is always one debugger thread in a given process. This thread exists so that a managed debugger can attach and control the managed code. Threads react differently depending on whether a managed or native debugger attaches invasively or not.

  If you break into the process with a managed debugger (for example, CorDbg or Visual Studio .NET), it attaches noninvasively. The entire process is not frozen; instead, only the managed threads are frozen. The native threads that don't contain runtime references continue running. Options to use a managed debugger to attach invasively are available.

  If you use a native debugger, it can attach invasively or noninvasively. When you break into the process invasively and quit, the process terminates, and consequently all threads—managed and native — are shut down. If you attach noninvasively, the process (and threads) stop temporarily. The debugger pauses the native and managed code and provides a snapshot in time. It then can detach and the process resumes.

In IIS 5.x, there is only one Aspnet_wp.exe worker process and one debugger thread. Consequently, only one debugger can be attached to the Aspnet_wp.exe process at a time. This can pose a problem if you're dealing with multiple Web applications on the same machine. In IIS 6.0, you can coerce an AppDomain to run in a separate application pool. (For more information, see "IIS 5.x Process Model" and "IIS 6.0 Process Model" in Chapter 1.) Separate application pools provide multiple W3wp.exe processes. Multiple debugger threads are created in these processes (one in each), allowing you to debug more efficiently.

- Thread 5 is a finalizer thread as shown by a call to **mscorwks!GCHeap::FinalizerThreadStart**. Just like the debugger thread, there is always one finalizer thread in .NET Framework version 1.0. The finalizer thread calls the **Finalize()** method of objects when the garbage collector (GC) determines that the object is unreachable. In Visual Basic .NET, you can implement a **Finalize()** method. The same function in the Visual C#™ development tool and managed extensions for the Visual C++® development system is performed by a destructor. To learn more, see the SDK for articles such as ".NET Framework General Reference" and "Implementing Finalize and Dispose to Clean Up Unmanaged Resources."
- Thread 9 is a lightweight remote procedure call (RPC) thread (note the call to **RPCRT4!RecvLotsaCallsWrapper**) used for local remote procedure call (LRPC) communication. It handles incoming RPC calls to this process and is waiting for work.
- Thread 10 is a HostThreadEntry thread that COM has created to instantiate the COM component marked **Apartment**. You need to discover why **DebuggingCOM!CSTA::RaiseError** calls **abort()** by checking the code, which is explained later in this chapter.
- Threads 11 and 12 are worker threads that are waiting for COM to dispatch work.

Checking the Code

The following shows the relevant code from STA.cpp. Based on the input parameters, this function offers three actions when called.

```
STDMETHODIMP CSTA::RaiseError(LONG errorid, LONG errorseverity)
{
     switch(errorid)
     {
          case 1:
               abort();
               return E_FAIL;
          case 0:
               return E_FAIL;
          default:
               return S_OK;
     }
}
```

The function **CSTA::RaiseError()** calls **abort()** when 1 is passed as the first parameter. If you ran **~*kb** and looked at the call stack, you would see that 1 and 5 are indeed passed. The crash is caused by calling **abort()**, which eventually causes the process to exit. As a result, the Aspnet_wp.exe process terminates and a new process is created.

How is **abort()** being called? What called **CSTA::RaiseError()**? Can you prevent passing 1 and 5 to the function? Let's look at the managed threads.

You can use two extension DLLs to examine managed output in WinDbg: SOS.dll and SieExtPub.dll. The former provides most of the functionality you need for this walkthrough. You'll use the latter to examine MTA/STA thread switching.

To examine managed threads, you must load the SOS.dll debugger extension.

► To examine the managed threads

1. In WinDbg, type **.load SOS\SOS.dll** to load the extension. Amend the path as necessary; in this case the location of SOS.dll is in the SOS folder.
2. Type **!findtable** to initialize SOS with the table information for the runtime version you are debugging.
3. Type **!threads**, and you should see results similar to the data shown below:

```
0:010> !threads
ThreadCount: 5
UnstartedThread: 0
BackgroundThread: 5
PendingThread: 0
DeadThread: 0
```

	1	5	6	7	10
ID	f20	74c	cc0	6c4	6c0
ThreadOBJ	00155008	00156df8	00171b68	00179650	001f9b58
State	a220	b220	1800220	4220	220
PreEmptive GC	Enabled	Enabled	Enabled	Enabled	Enabled
GC Alloc Context	00000000: 00000000	00000000: 00000000	00000000: 00000000	00000000: 00000000	00000000: 00000000
Domain	001a68c0	00166030	00166030	00166030	00166030
Lock Count	1	0	0	1	1
Apt	MTA	MTA	MTA	STA	Unk
Exception		(Finalizer)	(Threadpool Worker)	(GC)	

Three out of five managed threads initialized to the MTA threading model. Thread 10 is initialized to STA and shows **(GC)**. This means that the GC was run on the thread at some point. Four out of five threads are in the same AppDomain. Thread 1 is running in a different AppDomain because it's running in the ASP.NET virtual folder.

The following tables match the thread state to the values in the Appendix State chart and provide more information about the possible execution states for the thread. A thread can be in more than one state at a given time.

Table 4.2: Breakdown of state for thread 1, state value = 0xa220

Symbol	Value	Description
TS_InMTA	0x00008000	Thread is part of the MTA
TS_CoInitialized	0x00002000	CoInitialize has been called for this thread
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicates whether it is now legal to attempt a Join()

Table 4.3: Breakdown of state for thread 6, state value = 0xb220

Symbol	Value	Description
TS_InMTA	0x00008000	Thread is part of the MTA
TS_CoInitialized	0x00002000	CoInitialize has been called for this thread
TS_WeOwn	0x00001000	Exposed object initiated this thread
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicates whether it is now legal to attempt a Join()

Table 4.4: Breakdown of state for thread 7, state value = 0x01800220

Symbol	Value	Description
TS_TPWorkerThread	0x01000000	Indicates that this a threadpool worker thread. (if not, it is a threadpool completionport thread)
TS_ThreadPoolThread	0x00800000	Indicates that this a threadpool thread
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicates whether it is now legal to attempt a Join()

Table 4.5: Breakdown of state for thread 10, state value = 0x4220

Symbol	Value	Description
TS_InSTA	0x00004000	Thread hosts an STA
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicates whether it is now legal to attempt a Join()

Table 4.6: Breakdown of state for thread 12, state value = 0x220

Symbol	Value	Description
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicates whether it is now legal to attempt a Join()

Display the managed call stacks by typing **~*e !clrstack**. This command is similar to the **~*k** WinDbg command, but it applies to managed calls. WinDbg now displays the current enter stack pointer (ESP), the current enter instruction pointer (EIP), and the method signature for each frame with a managed thread. The **!clrstack** command displays only the managed threads, not any native call-stack information.

Note: Unless otherwise specified, the dump files were created using release builds.

```
0:010> ~*e !clrstack
Thread 0
Not a managed thread.
Thread 1
ESP       EIP
0096f95c  7ffe0304 [FRAME: ComPlusMethodFrameStandalone] [DEFAULT]
          [hasThis] Void DebuggingCOMLib.STAClass.RaiseError(I4,I4)
0096f970  03a00e06 [DEFAULT] [hasThis] Void
          Debugging.Unexpected.btnSTA_Click(Object,Class System.EventArgs)
0096f980  03ca2f55 [DEFAULT] [hasThis] Void
          System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)
0096f994  03ca2d32 [DEFAULT] [hasThis] Void
          System.Web.UI.WebControls.Button.System.Web.UI.IPostBackEventHandl
          er.RaisePostBackEvent(String)
0096f9a4  03ca2ce3 [DEFAULT] [hasThis] Void
          System.Web.UI.Page.RaisePostBackEvent(Class
          System.Web.UI.IPostBackEventHandler,String)
0096f9ac  03ca2c42 [DEFAULT] [hasThis] Void
          System.Web.UI.Page.RaisePostBackEvent(Class
          System.Collections.Specialized.NameValueCollection)
0096f9bc  03a4a5e0 [DEFAULT] [hasThis] Void
          System.Web.UI.Page.ProcessRequestMain()
0096f9f4  03a48317 [DEFAULT] [hasThis] Void
          System.Web.UI.Page.ProcessRequest()
0096fa2c  03a47d7b [DEFAULT] [hasThis] Void
          System.Web.UI.Page.ProcessRequest(Class System.Web.HttpContext)
0096fa34  03a47d54 [DEFAULT] [hasThis] Void
          System.Web.HttpApplication/CallHandlerExecutionStep.Execute()
0096fa44  03a5a230 [DEFAULT] [hasThis] Class System.Exception
          System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef
          Boolean)
0096fa8c  03a5997b [DEFAULT] [hasThis] Void
          System.Web.HttpApplication.ResumeSteps(Class System.Exception)
0096fad0  03a59853 [DEFAULT] [hasThis] Class System.IAsyncResult
          System.Web.HttpApplication.System.Web.IHttpAsyncHandler.BeginProce
          ssRequest(Class System.Web.HttpContext,Class
          System.AsyncCallback,Object)
```

```
0096faec  0377fed0 [DEFAULT] [hasThis] Void
          System.Web.HttpRuntime.ProcessRequestInternal(Class
          System.Web.HttpWorkerRequest)
0096fb28  0377fcbd [DEFAULT] Void
          System.Web.HttpRuntime.ProcessRequest(Class
          System.Web.HttpWorkerRequest)
0096fb34  0377b9af [DEFAULT] [hasThis] I4
          System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
0096fbf0  7920eab0 [FRAME: ContextTransitionFrame]
0096fcd0  7920eab0 [FRAME: ComMethodFrame]
Thread 2
Not a managed thread.
Thread 3
Not a managed thread.
Thread 4
Not a managed thread.
Thread 5
Not a managed thread.
Thread 6
ESP       EIP
Thread 7
ESP       EIP
Thread 8
Not a managed thread.
Thread 9
Not a managed thread.
Thread 10
ESP       EIP
Thread 11
Not a managed thread.
Thread 12
ESP       EIP
Thread 13
Not a managed thread.
```

Earlier you looked at the native output of one of the managed threads (as shown in thread 1). SOS.dll output fills in the missing information for the managed data. To build both managed and native portions of the stack, cut and paste the output from **!clrstack** into the middle portion where there were previously no mapped modules.

```
NATIVE
   1  Id: f80.f20 Suspend: 1 Teb: 7ffdd000 Unfrozen
ChildEBP RetAddr
0096f2dc 77f7f4af SharedUserData!SystemCallStub+0x4
0096f2e0 77e7788b ntdll!NtWaitForSingleObject+0xc
0096f344 77e79d6a kernel32!WaitForSingleObjectEx+0xa8
0096f354 771da2e7 kernel32!WaitForSingleObject+0xf
0096f370 772afc14 ole32!GetToSTA+0x6d
0096f394 772af180 ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0xe2
0096f464 771cf0af ole32!CRpcChannelBuffer::SendReceive2+0xa6
0096f4ec 77cc28f7 ole32!CAptRpcChnl::SendReceive+0x95
0096f514 77d28c20 rpcrt4!NdrClientInitializeNew+0x17
0096f550 771c07c8 rpcrt4!NdrProxySendReceive+0x41
0096f63c 771c955f ole32!CoTaskMemFree+0xf
```

```
0096f654 792107dc ole32!CStdIdentity::CInternalUnk::Release+0x3b
0096f684 00174b7e mscorwks!SafeRelease+0x56
WARNING: Frame IP not in any known module. Following frames may be wrong.
00000000 00000000 0x174b7e
MANAGED
0096f95c  7ffe0304 [FRAME: ComPlusMethodFrameStandalone] [DEFAULT]
       [hasThis] Void DebuggingCOMLib.STAClass.RaiseError(I4,I4)
0096f970  03a00e06 [DEFAULT] [hasThis] Void
       Debugging.Unexpected.btnSTA_Click(Object,Class System.EventArgs)
0096f980  03ca2f55 [DEFAULT] [hasThis] Void
       System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)
0096f994  03ca2d32 [DEFAULT] [hasThis] Void
       System.Web.UI.WebControls.Button.System.Web.UI.IPostBackEventHandl
       er.RaisePostBackEvent(String)
0096f9a4  03ca2ce3 [DEFAULT] [hasThis] Void
       System.Web.UI.Page.RaisePostBackEvent(Class
       System.Web.UI.IPostBackEventHandler,String)
0096f9ac  03ca2c42 [DEFAULT] [hasThis] Void
       System.Web.UI.Page.RaisePostBackEvent(Class
System.Collections.Specialized.NameValueCollection)
       0096f9bc  03a4a5e0 [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequestMain()
       0096f9f4  03a48317 [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequest()
       0096fa2c  03a47d7b [DEFAULT] [hasThis] Void
System.Web.UI.Page.ProcessRequest(Class System.Web.HttpContext)
0096fa34  03a47d54 [DEFAULT] [hasThis] Void
       System.Web.HttpApplication/CallHandlerExecutionStep.Execute()
0096fa44  03a5a230 [DEFAULT] [hasThis] Class System.Exception
       System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef
       Boolean)
0096fa8c  03a5997b [DEFAULT] [hasThis] Void
       System.Web.HttpApplication.ResumeSteps(Class System.Exception)
0096fad0  03a59853 [DEFAULT] [hasThis] Class System.IAsyncResult
       System.Web.HttpApplication.System.Web.IHttpAsyncHandler.BeginProce
       ssRequest(Class System.Web.HttpContext,Class
       System.AsyncCallback,Object)
0096faec  0377fed0 [DEFAULT] [hasThis] Void
       System.Web.HttpRuntime.ProcessRequestInternal(Class
       System.Web.HttpWorkerRequest)
0096fb28  0377fcbd [DEFAULT] Void
       System.Web.HttpRuntime.ProcessRequest(Class
       System.Web.HttpWorkerRequest)
0096fb34  0377b9af [DEFAULT] [hasThis] I4
       System.Web.Hosting.ISAPIRuntime.ProcessRequest(I,I4)
0096fbf0  7920eab0 [FRAME: ContextTransitionFrame]
0096fcd0  7920eab0 [FRAME: ComMethodFrame]
```

The **!clrstack** output for Thread1 does not show all of the call-stack frames. When you use the **!dumpstack** command, the output shows all of the stack frames, including the origin of the thread and the reference to the type of thread you are dealing with, which is a completion port thread. The following example shows the end of the output from the **!dumpstack** command for thread 1.

Note: You must change to the appropriate thread before running the **!dumpstack** command. For example, to change to thread 1, execute ~**1s**.

```
0:01> !dumpstack
...
0096ff24 0042218c aspnet_wp!CAsyncPipeManager::ProcessCompletion+0x1d4
       [e:\dna\src\xsp\wp\asyncpipemanager.cxx:516], calling
aspnet_wp!CAsyncPipeManager::ProcessMessage
       [e:\dna\src\xsp\wp\asyncpipemanager.cxx:597]
0096ff40 7a13ecc2 aspnet_isapi!CorThreadPoolCompletionCallback+0x41
       [e:\dna\src\xsp\isapi\threadpool.cxx:773]
0096ff84 792cf905 mscorwks!ThreadpoolMgr::CompletionPortThreadStart+0x93
0096ffb4 77e802ed kernel32!BaseThreadStart+0x37
```

In the managed stack, **Debugging.Unexpected.btnSTA_Click** calls **DebuggingCOMLib.STAClass**. Is there a specific call from one to the other in code? In the native stack, the calls work their way to **ole32!GetToSTA**.

Look at the code in the **Debugging.Unexpected.btnSTA_Click** event.

```
private void btnSTA_Click(object sender, System.EventArgs e)
{
   DebuggingCOMLib.STAClass staobj =  new DebuggingCOMLib.STAClass();
   staobj.RaiseError(1,5);
   Label1.Text += "STA Call Completed sucessfully";
}
```

If the source code is not available, you can examine the assembly by supplying the instruction pointer for the call-stack frame to the **!u** command. The instruction pointer can be retrieved from the **!clrstack**: output.

```
0096f970  03a00e06 [DEFAULT] [hasThis] Void
       Debugging.Unexpected.btnSTA_Click(Object,Class System.EventArgs)
```

To disassemble this function, type **!u 03a00e06**.

```
0:010> !u 03a00e06
Normal JIT generated code
[DEFAULT] [hasThis] Void Debugging.Unexpected.btnSTA_Click(Object,Class
System.EventArgs)
Begin 03a00de0, size 54
03a00de0 57              push    edi
03a00de1 56              push    esi
03a00de2 8bf9            mov     edi,ecx
03a00de4 b988c17503      mov     ecx,0x375c188 (MT:
       DebuggingCOMLib.STAClass)
03a00de9 e879147b75      call    mscorwks!JIT_NewCrossContext (791b2267)
03a00dee 8bf0            mov     esi,eax
03a00df0 8bce            mov     ecx,esi
```

```
03a00df2 ff15e4c17503     call    dword ptr [0375c1e4]
       (DebuggingCOMLib.STAClass..ctor)
03a00df8 6a05             push    0x5
03a00dfa 8bce             mov     ecx,esi
03a00dfc ba01000000       mov     edx,0x1
03a00e01 8b01             mov     eax,[ecx]
03a00e03 ff504c           call    dword ptr [eax+0x4c]
03a00e06 8b8fbc000000     mov     ecx,[edi+0xbc]
03a00e0c 8bf1             mov     esi,ecx
03a00e0e 8b01             mov     eax,[ecx]
03a00e10 ff9090010000     call    dword ptr [eax+0x190]
03a00e16 8bc8             mov     ecx,eax
03a00e18 8b15a89c1702     mov     edx,[02179ca8] ("STA Call Completed
       sucessfully")
03a00e1e e83d3590ff       call    03304360 (System.String.Concat)
03a00e23 8bd0             mov     edx,eax
03a00e25 8bce             mov     ecx,esi
03a00e27 8b01             mov     eax,[ecx]
03a00e29 ff9094010000     call    dword ptr [eax+0x194]
03a00e2f 5e               pop     esi
03a00e30 5f               pop     edi
03a00e31 c20400           ret     0x4
```

The code in both C# and x86 assemblies shows the creation of a COM wrapper. It does this by making a call to **DebuggingCOMLib.STAClass..ctor**), which then calls a method on a **staobj.RaiseError(1,5)** COM object, which in turn calls **MSVCR70!abort**. This confirms the findings in the native code for thread 10.

You now have seen the code and have proven that it directly calls **MSVCR70!abort()**, which causes **kernel32!ExitProcess()** to be called and the Aspnet_wp.exe process to terminate. Although you have technically solved the problem, it would be helpful to learn more. Thread 10 contains a call to **MSVCR70!abort()**. If there are multiple calls to the DLL, how can you determine which call causes a problem? SOS.dll can provide you with information on the COM threading models for the managed threads using the **!comstate** command.

```
0:010> !comstate
      ID       TEB      APT    APTId    CallerTID    Context
  0   230    7ffde000    MTA       6e            0    00145008
  1   f20    7ffdd000    MTA       75            0    00145008
  2   cdc    7ffdb000    Ukn
  3   e80    7ffda000    Ukn
  4   b94    7ffdc000    MTA        0            0    00145008
  5   874    7ffd9000    Ukn
  6   74c    7ffd8000    MTA        0            0    00145008
  7   cc0    7ffd7000    MTA        0            0    00145008
  8   764    7ffd6000    MTA        0            0    00145008
  9   b1c    7ffd5000    Ukn
 10   6c4    7ffd4000    STA        0            0    00145178
 11   114    7ffaf000    Ukn
 12   6c0    7ffae000    Ukn        0            0    00000000
 13   524    7ffad000    Ukn
```

The **!comstate** output shows that the apartment type of each thread is either MTA, Ukn, or STA. The Ukn entry implies that the threads haven't been initialized to a particular threading model yet.

The other apartment type that stands out is STA, and as noted earlier, thread 10 is an STA thread. You can use two methods to determine on which thread this call continues to execute.

The first method is to guess. There is an entry for **ole32!GetToSTA()**, so you know that you need to switch to an STA thread in the process. The output of **!comstate** indicates that thread 10 is the only thread initialized to the STA threading model.

The second method is to troubleshoot using the SieExtPub.dll extension to WinDbg. Make sure that the SieExtPub.dll file is located in your C:\Debuggers or applicable folder. Type **.load sos\sieextpub.dll** to load the extension. Type **!comcalls** to display thread switching.

```
0:010> .load sos\sieextpub.dll
Thank you for using SIEExtPub.

0:010> !comcalls
       Thread 1 - MTA
Target Process ID: 00000f80 = 3968
Target Thread  ID: 000006c4 = 10  (STA)
```

This output shows that Thread 1, which is an MTA thread, calls Thread 10, which is an STA thread.

The previous example required an MTA/STA thread switch to allow the managed code to call the native code. This is because a component marked **Apartment** cannot be instantiated from an MTA thread with a thread switch due to COM marshalling. An alternative approach would be to avoid the marshalling overhead by using the **ASPCompat** page directive, which forces the page's **ProcessRequest** method to execute on a COM+ STA worker thread.

Using ASPCompat

To view the changes that **ASPCompat** can bring to the previous scenario, rerun the examples, but make one code change—add the following directive to the ASP.NET page. The Unexpectedcompat.aspx example page contains this page directive:

```
<%@ Page language="c#" ASPCompat="true" AutoEventWireup="false"
Inherits="Debugging.Unexpected" %>
```

This is the only change to the code. Because this code inherits from the same DLL as the previous ASP.NET page, recompiling the DLL is not necessary.

As noted earlier, when a managed object is created from a thread that is initialized to the MTA threading model, and it calls a COM component marked **Apartment**, it must follow COM threading rules. **ASPCompat** complies by running the page's **ProcessRequest** method on a thread initialized to STA threading model.

► To start with a clean slate

1. Open a command prompt window, and then type **iisreset** at the prompt to restart IIS.
2. Browse to *http://localhost/debugging/unexpectedcompat.aspx.*

The following figure shows a table similar to what the browser displays:

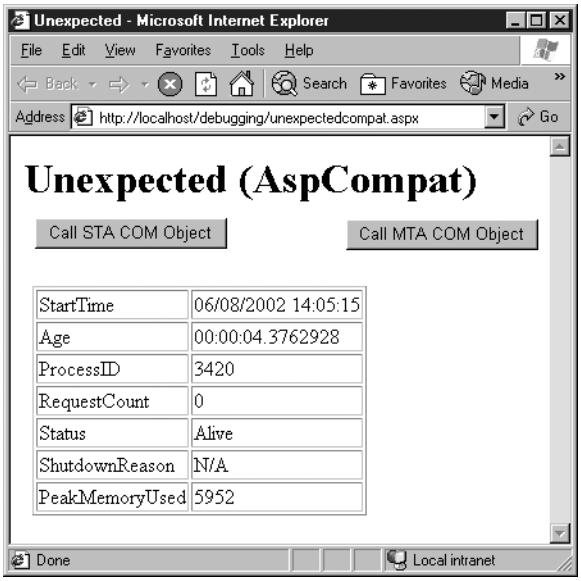

Figure 4.4
Baseline data for Unexpectedcompat.aspx

Run ADPlus_KernelExit.vbs in **-crash** mode from the command line:

```
ADPLUS_KernelExit.vbs -crash -pn aspnet_wp.exe -quiet
```

Click the **Call STA COM Object** button. The browser displays a Server Application Unavailable error and the application event log shows the following entry: "aspnet_wp.exe(PID:2992) stopped unexpectedly."

Look in the latest \Crash folder under C:\Debuggers for a file with a name similar to **PID-2916__ASPNET_WP.EXE__Kernel32ExitProcess__full.dmp**.

Open the file in WinDbg, and ensure the symbol path is set up correctly:

Enter the symbol path in the command line as it is shown below. Replace C:\Windows with your windows folder location.

```
srv*C:\symbols\debugginglabs*http://msdl.microsoft.com/download/symbols;C:\symbols
\debugginglabs;C:\program files\microsoft visual studio
.net\frameworksdk\symbols;C:\windows\system32;C:\inetpub\wwwroot\debugging\
debuggingCOM\release.
```

Dump the native threads using the tilde '~' command:

```
0:000> ~
   0  Id: b64.db0 Suspend: 1 Teb: 7ffde000 Unfrozen
   1  Id: b64.b98 Suspend: 1 Teb: 7ffdd000 Unfrozen
   2  Id: b64.60c Suspend: 1 Teb: 7ffdc000 Unfrozen
   3  Id: b64.d9c Suspend: 1 Teb: 7ffdb000 Unfrozen
   4  Id: b64.b1c Suspend: 1 Teb: 7ffda000 Unfrozen
   5  Id: b64.680 Suspend: 1 Teb: 7ffd9000 Unfrozen
   6  Id: b64.d1c Suspend: 1 Teb: 7ffd8000 Unfrozen
   7  Id: b64.3ac Suspend: 1 Teb: 7ffd7000 Unfrozen
   8  Id: b64.184 Suspend: 1 Teb: 7ffd6000 Unfrozen
   9  Id: b64.980 Suspend: 1 Teb: 7ffd5000 Unfrozen
  10  Id: b64.d8 Suspend: 1 Teb: 7ffd4000 Unfrozen
  11  Id: b64.c30 Suspend: 1 Teb: 7ffaf000 Unfrozen
  12  Id: b64.f54 Suspend: 1 Teb: 7ffae000 Unfrozen
  13  Id: b64.198 Suspend: 1 Teb: 7ffad000 Unfrozen
  14  Id: b64.8b0 Suspend: 1 Teb: 7ffac000 Unfrozen
  15  Id: b64.9c4 Suspend: 1 Teb: 7ffab000 Unfrozen
  16  Id: b64.c5c Suspend: 1 Teb: 7ffaa000 Unfrozen
  17  Id: b64.9a0 Suspend: 1 Teb: 7ffa9000 Unfrozen
  18  Id: b64.da0 Suspend: 1 Teb: 7ffa8000 Unfrozen
  19  Id: b64.934 Suspend: 1 Teb: 7ffa7000 Unfrozen
  20  Id: b64.5c0 Suspend: 1 Teb: 7ffa6000 Unfrozen
```

Now dump the native call stacks.

```
0:000> ~*k 200

   0  Id: b64.db0 Suspend: 1 Teb: 7ffde000 Unfrozen
ChildEBP RetAddr
0012f874 77f7e76f SharedUserData!SystemCallStub+0x4
0012f878 77e775b7 ntdll!NtDelayExecution+0xc
0012f8d0 77e61bf1 kernel32!SleepEx+0x61
```

```
0012f8dc 00422c17 kernel32!Sleep+0xb
0012ff44 004237db aspnet_wp!wmain+0x30b [e:\dna\src\xsp\wp\main.cxx @
       222]
       0012ffc0 77e7eb69 aspnet_wp!wmainCRTStartup+0x131
[f:\vs70builds\9111\vc\crtbld\crt\src\crtexe.c @ 379]
0012fff0 00000000 kernel32!BaseProcessStart+0x23

   1  Id: b64.b98 Suspend: 1 Teb: 7ffdd000 Unfrozen
ChildEBP RetAddr
0086ff4c 77f7ef9f SharedUserData!SystemCallStub+0x4
0086ff50 77e73b3f ntdll!ZwRemoveIoCompletion+0xc
0086ff7c 792cf8b8 kernel32!GetQueuedCompletionStatus+0x27
0086ffb4 77e802ed mscorwks!ThreadpoolMgr::CompletionPortThreadStart+0x46
0086ffec 00000000 kernel32!BaseThreadStart+0x37

   2  Id: b64.60c Suspend: 1 Teb: 7ffdc000 Unfrozen
ChildEBP RetAddr
0096ff04 77f7e76f SharedUserData!SystemCallStub+0x4
0096ff08 77e775b7 ntdll!NtDelayExecution+0xc
0096ff60 77e61bf1 kernel32!SleepEx+0x61
0096ff6c 792d00bc kernel32!Sleep+0xb
0096ffb4 77e802ed mscorwks!ThreadpoolMgr::GateThreadStart+0x4c
0096ffec 00000000 kernel32!BaseThreadStart+0x37

   3  Id: b64.d9c Suspend: 1 Teb: 7ffdb000 Unfrozen
ChildEBP RetAddr
00c6fefc 77f7f4af SharedUserData!SystemCallStub+0x4
00c6ff00 77e7788b ntdll!NtWaitForSingleObject+0xc
00c6ff64 77e79d6a kernel32!WaitForSingleObjectEx+0xa8
00c6ff74 0042289c kernel32!WaitForSingleObject+0xf
00c6ff80 7c00fbab aspnet_wp!DoPingThread+0x10 [e:\dna\src\xsp\wp\main.cxx
       @ 412]
       00c6ffb4 77e802ed msvcr70!_threadstart+0x6c
[f:\vs70builds\9466\vc\crtbld\crt\src\thread.c @ 196]
00c6ffec 00000000 kernel32!BaseThreadStart+0x37

   4  Id: b64.b1c Suspend: 1 Teb: 7ffda000 Unfrozen
ChildEBP RetAddr
0101fe80 77f7f49f SharedUserData!SystemCallStub+0x4
0101fe84 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
0101ff20 77e74c70 kernel32!WaitForMultipleObjectsEx+0x12c
0101ff38 791cccf6 kernel32!WaitForMultipleObjects+0x17
0101ffa0 791ccc6b mscorwks!DebuggerRCThread::MainLoop+0x90
0101ffac 791ccc1e mscorwks!DebuggerRCThread::ThreadProc+0x55
0101ffb4 77e802ed mscorwks!DebuggerRCThread::ThreadProcStatic+0xb
0101ffec 00000000 kernel32!BaseThreadStart+0x37

   5  Id: b64.680 Suspend: 1 Teb: 7ffd9000 Unfrozen
ChildEBP RetAddr
031bff08 77f7e76f SharedUserData!SystemCallStub+0x4
031bff0c 77e775b7 ntdll!NtDelayExecution+0xc
031bff64 77e61bf1 kernel32!SleepEx+0x61
031bff70 791cde2c kernel32!Sleep+0xb
031bffb4 77e802ed mscorwks!GCHeap::FinalizerThreadStart+0x1f9
031bffec 00000000 kernel32!BaseThreadStart+0x37
```

```
   6  Id: b64.d1c Suspend: 1 Teb: 7ffd8000 Unfrozen
ChildEBP RetAddr
0354ff1c 77f7f4af SharedUserData!SystemCallStub+0x4
0354ff20 77e7788b ntdll!NtWaitForSingleObject+0xc
0354ff84 77e79d6a kernel32!WaitForSingleObjectEx+0xa8
0354ff94 792cf5dc kernel32!WaitForSingleObject+0xf
0354ffb4 77e802ed mscorwks!ThreadpoolMgr::WorkerThreadStart+0x2e
0354ffec 00000000 kernel32!BaseThreadStart+0x37

   7  Id: b64.3ac Suspend: 1 Teb: 7ffd7000 Unfrozen
ChildEBP RetAddr
0364ff4c 77f7ef9f SharedUserData!SystemCallStub+0x4
0364ff50 77e73b3f ntdll!ZwRemoveIoCompletion+0xc
0364ff7c 792cf8b8 kernel32!GetQueuedCompletionStatus+0x27
0364ffb4 77e802ed mscorwks!ThreadpoolMgr::CompletionPortThreadStart+0x46
0364ffec 00000000 kernel32!BaseThreadStart+0x37

   8  Id: b64.184 Suspend: 1 Teb: 7ffd6000 Unfrozen
ChildEBP RetAddr
0387ff44 77f7e76f SharedUserData!SystemCallStub+0x4
0387ff48 77e775b7 ntdll!NtDelayExecution+0xc
0387ffa0 792d05b2 kernel32!SleepEx+0x61
0387ffb4 77e802ed mscorwks!ThreadpoolMgr::TimerThreadStart+0x30
03880038 792067d5 kernel32!BaseThreadStart+0x37
00d1203b 000000ff mscorwks!CreateTypedHandle+0x16

   9  Id: b64.980 Suspend: 1 Teb: 7ffd5000 Unfrozen
ChildEBP RetAddr
03bffe18 77f7f49f SharedUserData!SystemCallStub+0x4
03bffe1c 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
03bffeb8 77d46db9 kernel32!WaitForMultipleObjectsEx+0x12c
03bfff14 77d46e5b user32!RealMsgWaitForMultipleObjectsEx+0x13c
03bfff30 757832b1 user32!MsgWaitForMultipleObjects+0x1d
03bfff80 77c37fb8 comsvcs!CSTAThread::WorkerLoop+0x1df
03bfffb4 77e802ed msvcrt!_endthreadex+0xa0
03bfffec 00000000 kernel32!BaseThreadStart+0x37

  10  Id: b64.d8 Suspend: 1 Teb: 7ffd4000 Unfrozen
ChildEBP RetAddr
03cffe24 77f7efff SharedUserData!SystemCallStub+0x4
03cffe28 77cc1ac9 ntdll!NtReplyWaitReceivePortEx+0xc
03cfff90 77cc167e rpcrt4!LRPC_ADDRESS::ReceiveLotsaCalls+0xf6
03cfff94 77cc1505 rpcrt4!RecvLotsaCallsWrapper+0x9
03cfffac 77cc1670 rpcrt4!BaseCachedThreadRoutine+0x64
03cfffb4 77e802ed rpcrt4!ThreadStartRoutine+0x16
03cfffec 00000000 kernel32!BaseThreadStart+0x37
```

```
  11  Id: b64.c30 Suspend: 1 Teb: 7ffaf000 Unfrozen
ChildEBP RetAddr
03dffe18 77f7f49f SharedUserData!SystemCallStub+0x4
03dffe1c 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
03dffeb8 77d46db9 kernel32!WaitForMultipleObjectsEx+0x12c
03dfff14 77d46e5b user32!RealMsgWaitForMultipleObjectsEx+0x13c
03dfff30 757832b1 user32!MsgWaitForMultipleObjects+0x1d
03dfff80 77c37fb8 comsvcs!CSTAThread::WorkerLoop+0x1df
03dfffb4 77e802ed msvcrt!_endthreadex+0xa0
03dfffec 00000000 kernel32!BaseThreadStart+0x37

  12  Id: b64.f54 Suspend: 1 Teb: 7ffae000 Unfrozen
ChildEBP RetAddr
03effe18 77f7f49f SharedUserData!SystemCallStub+0x4
03effe1c 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
03effeb8 77d46db9 kernel32!WaitForMultipleObjectsEx+0x12c
03efff14 77d46e5b user32!RealMsgWaitForMultipleObjectsEx+0x13c
03efff30 757832b1 user32!MsgWaitForMultipleObjects+0x1d
03efff80 77c37fb8 comsvcs!CSTAThread::WorkerLoop+0x1df
03efffb4 77e802ed msvcrt!_endthreadex+0xa0
03efffec 00000000 kernel32!BaseThreadStart+0x37

  13  Id: b64.198 Suspend: 1 Teb: 7ffad000 Unfrozen
ChildEBP RetAddr
03fffe18 77f7f49f SharedUserData!SystemCallStub+0x4
03fffe1c 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
03fffeb8 77d46db9 kernel32!WaitForMultipleObjectsEx+0x12c
03ffff14 77d46e5b user32!RealMsgWaitForMultipleObjectsEx+0x13c
03ffff30 757832b1 user32!MsgWaitForMultipleObjects+0x1d
03ffff80 77c37fb8 comsvcs!CSTAThread::WorkerLoop+0x1df
03ffffb4 77e802ed msvcrt!_endthreadex+0xa0
03ffffec 00000000 kernel32!BaseThreadStart+0x37

  14  Id: b64.8b0 Suspend: 1 Teb: 7ffac000 Unfrozen
ChildEBP RetAddr
040ffe18 77f7f49f SharedUserData!SystemCallStub+0x4
040ffe1c 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
040ffeb8 77d46db9 kernel32!WaitForMultipleObjectsEx+0x12c
040fff14 77d46e5b user32!RealMsgWaitForMultipleObjectsEx+0x13c
040fff30 757832b1 user32!MsgWaitForMultipleObjects+0x1d
040fff80 77c37fb8 comsvcs!CSTAThread::WorkerLoop+0x1df
040fffb4 77e802ed msvcrt!_endthreadex+0xa0
040fffec 00000000 kernel32!BaseThreadStart+0x37

  15  Id: b64.9c4 Suspend: 1 Teb: 7ffab000 Unfrozen
ChildEBP RetAddr
041ffe18 77f7f49f SharedUserData!SystemCallStub+0x4
041ffe1c 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
041ffeb8 77d46db9 kernel32!WaitForMultipleObjectsEx+0x12c
041fff14 77d46e5b user32!RealMsgWaitForMultipleObjectsEx+0x13c
041fff30 757832b1 user32!MsgWaitForMultipleObjects+0x1d
041fff80 77c37fb8 comsvcs!CSTAThread::WorkerLoop+0x1df
041fffb4 77e802ed msvcrt!_endthreadex+0xa0
041fffec 00000000 kernel32!BaseThreadStart+0x37
```

```
  16  Id: b64.c5c Suspend: 1 Teb: 7ffaa000 Unfrozen
ChildEBP RetAddr
042ff8e0 7923b594 kernel32!ExitProcess
042ff8ec 792176bb mscorwks!SafeExitProcess+0x38
042ff8f8 7921Ocdd mscorwks!ForceEEShutdown+0x3f
042ff90c 7917b297 mscorwks!CorExitProcess+0x5a
042ff91c 7c00bb88 mscoree!CorExitProcess+0x3d
042ff924 7c00440a msvcr70!__crtExitProcess+0x25
[f:\vs70builds\9466\vc\crtbld\crt\src\crt0dat.c @ 451]
042ff930 7c00f791 msvcr70!doexit+0xa9
[f:\vs70builds\9466\vc\crtbld\crt\src\crt0dat.c @ 402]
042ff940 7c012ac8 msvcr70!_exit+0xe
[f:\vs70builds\9466\vc\crtbld\crt\src\crt0dat.c @ 302]
042ff988 7c012799 msvcr70!raise+0xae
[f:\vs70builds\9466\vc\crtbld\crt\src\winsig.c @ 509]
042ff994 10005877 msvcr70!abort+0xe
[f:\vs70builds\9466\vc\crtbld\crt\src\abort.c @ 48]

  17  Id: b64.9a0 Suspend: 1 Teb: 7ffa9000 Unfrozen
ChildEBP RetAddr
043ffe18 77f7f49f SharedUserData!SystemCallStub+0x4
043ffe1c 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
043ffeb8 77d46db9 kernel32!WaitForMultipleObjectsEx+0x12c
043fff14 77d46e5b user32!RealMsgWaitForMultipleObjectsEx+0x13c
043fff30 757832b1 user32!MsgWaitForMultipleObjects+0x1d
043fff80 77c37fb8 comsvcs!CSTAThread::WorkerLoop+0x1df
043fffb4 77e802ed msvcrt!_endthreadex+0xa0
043fffec 00000000 kernel32!BaseThreadStart+0x37

  18  Id: b64.da0 Suspend: 1 Teb: 7ffa8000 Unfrozen
ChildEBP RetAddr
044ffef0 77f7f4af SharedUserData!SystemCallStub+0x4
044ffef4 77e7788b ntdll!NtWaitForSingleObject+0xc
044fff58 77e79d6a kernel32!WaitForSingleObjectEx+0xa8
044fff68 75780195 kernel32!WaitForSingleObject+0xf
044fff80 77c37fb8
comsvcs!CSTAThreadPool::LoadBalanceThreadControlLoop+0x21
044fffb4 77e802ed msvcrt!_endthreadex+0xa0
044fffec 00000000 kernel32!BaseThreadStart+0x37

  19  Id: b64.934 Suspend: 1 Teb: 7ffa7000 Unfrozen
ChildEBP RetAddr
045ffef0 77f7f4af SharedUserData!SystemCallStub+0x4
045ffef4 77e7788b ntdll!NtWaitForSingleObject+0xc
045fff58 77e79d6a kernel32!WaitForSingleObjectEx+0xa8
045fff68 7577fd59 kernel32!WaitForSingleObject+0xf
045fff80 77c37fb8 comsvcs!CSTAThreadPool::KillThreadControlLoop+0x21
045fffb4 77e802ed msvcrt!_endthreadex+0xa0
045fffec 00000000 kernel32!BaseThreadStart+0x37

  20  Id: b64.5c0 Suspend: 1 Teb: 7ffa6000 Unfrozen
ChildEBP RetAddr
0485fee0 77f7f49f SharedUserData!SystemCallStub+0x4
```

```
0485fee4 77e74bd8 ntdll!ZwWaitForMultipleObjects+0xc
0485ff80 792d0467 kernel32!WaitForMultipleObjectsEx+0x12c
0485ffb4 77e802ed mscorwks!ThreadpoolMgr::WaitThreadStart+0x45
0485ffec 00000000 kernel32!BaseThreadStart+0x37
```

Note: Your output may be different from the sample shown above because of the differences in the names of function calls between Windows 2000 and Windows XP.

Thread Discussion

The following table summarizes the functions that were being executed by the threads at the point the dump was created.

Table 4.7: Summary of thread state information

Thread ID	Purpose
0	aspnet_wp!wmain
1	mscorwks!ThreadpoolMgr::CompletionPortThreadStart
2	mscorwks!ThreadpoolMgr::GateThreadStart
3	aspnet_wp!DoPingThread
4	mscorwks!DebuggerRCThread::MainLoop
5	mscorwks!GCHeap::FinalizerThreadStart
6	mscorwks!ThreadpoolMgr::WorkerThreadStart
7	mscorwks!ThreadpoolMgr::CompletionPortThreadStart
8	mscorwks!ThreadpoolMgr::TimerThreadStart
9	comsvcs!CSTAThread::WorkerLoop
10	rpcrt4!RecvLotsaCallsWrapper
11	comsvcs!CSTAThread::WorkerLoop
12	comsvcs!CSTAThread::WorkerLoop
13	comsvcs!CSTAThread::WorkerLoop
14	comsvcs!CSTAThread::WorkerLoop
15	comsvcs!CSTAThread::WorkerLoop
16	comsvcs!CSTAThread::WorkerLoop (EXIT)
17	comsvcs!CSTAThread::WorkerLoop
18	comsvcs!CSTAThreadPool::LoadBalanceThreadControlLoop
19	comsvcs!CSTAThreadPool::KillThreadControlLoop
20	mscorwks!ThreadpoolMgr::WaitThreadStart

There are eight **comsvcs!CSTAThread::WorkerLoop** threads— seven threads, plus one thread for each CPU on the machine. There are also two **comsvcs!CSTAThreadPool** threads, five **mscorwks!ThreadpoolMgr** threads, and one **rpcrt4!RecvLotsaCallsWrapper** thread. For more information about the COM+ thread pool, see article Q282490, "INFO: Thread Pool Differences Between COM+ and MTS" in the Microsoft Knowledge Base at *http://support.microsoft.com /default.aspx?scid=kb;en-us;Q282490.*

Examine the threading models by using the SOS.dll. Load the extension and initialize it using the **!findtable** command. Then type **!comstate** to dump information on the threading models of the COM threads:

```
0:000> !comstate
      ID       TEB      APT    APTId    CallerTID    Context
  0   db0 7ffde000      MTA       80            0    00145008
  1   b98 7ffdd000      MTA        0            0    00145008
  2   60c    7ffdc000   Ukn
  3   d9c    7ffdb000   Ukn
  4   b1c    7ffda000   Ukn
  5   680    7ffd9000   MTA        0            0    00145008
  6   d1c    7ffd8000   MTA        0            0    00145008
  7   3ac    7ffd7000   MTA        0            0    00145008
  8   184    7ffd6000   MTA        0            0    00145008
  9   980    7ffd5000   STA       81            0    00145178
 10   d8     7ffd4000   Ukn
 11   c30    7ffaf000   STA        0            0    00145230
 12   f54    7ffae000   STA        0            0    001452e8
 13   198    7ffad000   STA        0            0    001453a0
 14   8b0    7ffac000   STA        0            0    00145458
 15   9c4    7ffab000   STA        0            0    00145510
 16   c5c    7ffaa000   STA        0            0    001457f0
 17   9a0    7ffa9000   STA        0            0    00145680
 18   da0    7ffa8000   Ukn
 19   934    7ffa7000   Ukn
 20   5c0    7ffa6000   Ukn
```

In this aspnet_wp process, there are eight threads initialized to the STA threading model. These are the **comsvcs!CSTAThread::WorkerLoop** threads that were mentioned previously.

Examining the Managed Threads

Use the **!threads** command to dump information on the managed threads:

```
0:000> !threads
ThreadCount: 5
UnstartedThread: 0
BackgroundThread: 5
PendingThread: 0
DeadThread: 0
```

	1	5	6	17	16
ID	b98	680	d1c	9a0	c5c
ThreadOBJ	00153a68	00156c10	00174a30	001785c8	001ac710
State	a220	b220	1800220	4220	4220
PreEmptive GC	Enabled	Enabled	Enabled	Enabled	Enabled
GC Alloc Context	00000000: 00000000	00000000: 00000000	00000000: 00000000	00000000: 00000000	00000000: 00000000
Domain	00166030	00166030	00166030	00166030	001a6918
Lock Count	0	0	0	0	2
Apt	MTA	MTA	MTA	STA	STA
Exception		(Finalizer)	(Threadpool Worker)		(GC)

The following tables decode the state field for each managed thread:

Table 4.8: Breakdown of state for thread 1, state value = 0xa220

Symbol	Value	Description
TS_InMTA	0x00008000	Thread is part of the MTA
TS_CoInitialized	0x00002000	CoInitialize has been called for this thread
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicates whether it is now legal to attempt a Join()

Table 4.9: Breakdown of state for thread 5, state value = 0xb220

Symbol	Value	Description
TS_InMTA	0x00008000	Thread is part of the MTA
TS_CoInitialized	0x00002000	CoInitialize has been called for this thread
TS_WeOwn	0x00001000	Exposed object initiated this thread
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicated whether it is now legal to attempt a Join()

Table 4.10: Breakdown of state for thread 6, state value = 0x01800220

Symbol	Value	Description
TS_TPWorkerThread	0x01000000	Indicates that this a threadpool worker thread. (if not, it is a threadpool completionport thread)
TS_ThreadPoolThread	0x00800000	Indicates that this is a threadpool thread
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicates whether it is now legal to attempt a Join()

Table 4.11: Breakdown of state for threads 16 and 17, state value = 0x4220

Symbol	Value	Description
TS_InSTA	0x00004000	Thread hosts an STA
TS_Background	0x00000200	Thread is a background thread
TS_LegalToJoin	0x00000020	Indicates whether it is now legal to attempt a Join()

Threads 16 and 17 are STA threads created by COM+, but they are initialized by runtime thread 17, which is waiting for either managed or native work. As seen in the native thread output, Thread 16 called **msvcr70!abort**, which called **mscorwks!CorExitProcess**. Then **mscorwks!CorExitProcess** made a call to **mscorwks!ForceEEShutdown,** which finally called **kernel32!ExitProcess**. To examine this and other managed threads more closely, type **~*e !clrstack**.

```
0:000> ~*e !clrstack
Thread 0
Not a managed thread.
Thread 1
ESP       EIP
Thread 2
Not a managed thread.
Thread 3
Not a managed thread.
Thread 4
Not a managed thread.
Thread 5
ESP       EIP
Thread 6
ESP       EIP
Thread 7
Not a managed thread.
Thread 8
Not a managed thread.
Thread 9
Not a managed thread.
Thread 10
Not a managed thread.
Thread 11
Not a managed thread.
Thread 12
Not a managed thread.
Thread 13
Not a managed thread.
Thread 14
Not a managed thread.
Thread 15
Not a managed thread.
Thread 16
ESP       EIP
042ff9c8  77e75cb5 [FRAME: ComPlusMethodFrameStandalone] [DEFAULT]
```

```
       [hasThis] Void DebuggingCOMLib.STAClass.RaiseError(I4,I4)
042ff9dc  03a00e86 [DEFAULT] [hasThis] Void
       Debugging.Unexpected.btnSTA_Click(Object,Class System.EventArgs)
042ff9ec  04748b6d [DEFAULT] [hasThis] Void
       System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)
042ffa00  0474894a [DEFAULT] [hasThis] Void
       System.Web.UI.WebControls.Button.System.Web.UI.IPostBackEventHandl
       er.RaisePostBackEvent(String)
042ffa10  047488fb [DEFAULT] [hasThis] Void
       System.Web.UI.Page.RaisePostBackEvent(Class
       System.Web.UI.IPostBackEventHandler,String)
042ffa18  0474885a [DEFAULT] [hasThis] Void
       System.Web.UI.Page.RaisePostBackEvent(Class
       System.Collections.Specialized.NameValueCollection)
042ffa28  04620ac0 [DEFAULT] [hasThis] Void
       System.Web.UI.Page.ProcessRequestMain()
042ffa60  03a9f09f [DEFAULT] [hasThis] Void
       System.Web.UI.Page.ProcessRequest()
042ffa98  03a9f012 [DEFAULT] [hasThis] Void System.Web.Util.ASPCompat
       ApplicationStep.System.Web.HttpApplication+IExecutionStep.Execute()
042ffa9c  03a5a6f0 [DEFAULT] [hasThis] Class System.Exception
       System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef
       Boolean)
042ffae4  03a9ef8c [DEFAULT] [hasThis] Void System.Web.Util.ASPCompat
       ApplicationStep.ExecuteASPCompat Code()
042ffb10  03a9ee54 [DEFAULT] [hasThis] Void System.Web.Util.ASPCompat
       ApplicationStep.OnASPCompat Execution()
042ffb7c  001ec90a [FRAME: ContextTransitionFrame]
Thread 17
ESP       EIP
Thread 18
Not a managed thread.
Thread 19
Not a managed thread.
Thread 20
Not a managed thread.
```

Thread 16 is the only managed thread that is executing .NET code. You can combine Thread 16's managed and native frames together to see how one talks to the other.

```
  16  Id: b64.c5c Suspend: 1 Teb: 7ffaa000 Unfrozen
ChildEBP RetAddr
042ff8e0 7923b594 kernel32!ExitProcess
042ff8ec 792176bb mscorwks!SafeExitProcess+0x38
042ff8f8 79210cdd mscorwks!ForceEEShutdown+0x3f
042ff90c 7917b297 mscorwks!CorExitProcess+0x5a
042ff91c 7c00bb88 mscoree!CorExitProcess+0x3d
042ff924 7c00440a msvcr70!__crtExitProcess+0x25
       [f:\vs70builds\9466\vc\crtbld\crt\src\crt0dat.c @ 451]
042ff930 7c00f791 msvcr70!doexit+0xa9
       [f:\vs70builds\9466\vc\crtbld\crt\src\crt0dat.c @ 402]
042ff940 7c012ac8 msvcr70!_exit+0xe
       [f:\vs70builds\9466\vc\crtbld\crt\src\crt0dat.c @ 302]
```

```
042ff988 7c012799 msvcr70!raise+0xae
       [f:\vs70builds\9466\vc\crtbld\crt\src\winsig.c @ 509]
042ff994 10005877 msvcr70!abort+0xe
       [f:\vs70builds\9466\vc\crtbld\crt\src\abort.c @ 48]
Thread 16
ESP       EIP
042ff9c8  77e75cb5 [FRAME: ComPlusMethodFrameStandalone] [DEFAULT]
       [hasThis] Void DebuggingCOMLib.STAClass.RaiseError(I4,I4)
042ff9dc  03a00e86 [DEFAULT] [hasThis] Void
       Debugging.Unexpected.btnSTA_Click(Object,Class System.EventArgs)
042ff9ec  04748b6d [DEFAULT] [hasThis] Void
       System.Web.UI.WebControls.Button.OnClick(Class System.EventArgs)
042ffa00  0474894a [DEFAULT] [hasThis] Void
       System.Web.UI.WebControls.Button.System.Web.UI.IPostBackEventHandl
       er.RaisePostBackEvent(String)
042ffa10  047488fb [DEFAULT] [hasThis] Void
       System.Web.UI.Page.RaisePostBackEvent(Class
       System.Web.UI.IPostBackEventHandler,String)
042ffa18  0474885a [DEFAULT] [hasThis] Void
       System.Web.UI.Page.RaisePostBackEvent(Class
       System.Collections.Specialized.NameValueCollection)
042ffa28  04620ac0 [DEFAULT] [hasThis] Void
       System.Web.UI.Page.ProcessRequestMain()
042ffa60  03a9f09f [DEFAULT] [hasThis] Void
       System.Web.UI.Page.ProcessRequest()
042ffa98  03a9f012 [DEFAULT] [hasThis] Void System.Web.Util.ASPCompat
       ApplicationStep.System.Web.HttpApplication+IExecutionStep.Execute(
       )
042ffa9c  03a5a6f0 [DEFAULT] [hasThis] Class System.Exception
       System.Web.HttpApplication.ExecuteStep(Class IExecutionStep,ByRef
       Boolean)
042ffae4  03a9ef8c [DEFAULT] [hasThis] Void System.Web.Util.ASPCompat
       ApplicationStep.ExecuteASPCompat Code()
042ffb10  03a9ee54 [DEFAULT] [hasThis] Void System.Web.Util.ASPCompat
       ApplicationStep.OnASPCompat Execution()
042ffb7c  001ec90a [FRAME: ContextTransitionFrame]
...
this is an extract of the output from !dumpstack to show the starting frames of
the thread's call stack
042fff2c 75782f91 comsvcs!CSTAThread::ProcessQueueWork+0x32, calling
       comsvcs!CSTAThread::DoWork
042fff44 75783251 comsvcs!CSTAThread::WorkerLoop+0x17f, calling
       comsvcs!CSTAThread::ProcessQueueWork
042fff7c 77c2ab33 msvcrt!free+0xc8, calling msvcrt!_SEH_epilog
042fff80 77c37fb8 msvcrt!_endthreadex+0xa0
042fff8c 77f52dbb ntdll!RtlAllocateHeap+0x81f, calling
       ntdll!RtlpUpdateIndexInsertBlock
042fffb4 77e802ed kernel32!BaseThreadStart+0x37
```

Note: You must change to the appropriate thread before running the **!dumpstack** command. For example, to change to thread 16, execute **~16s.**

When you use **ASPCompat**, the process terminates on one thread instead of multiple threads. In the previous example, the call from **Debugging.Unexpected.btnSTA_Click()** to **DebuggingCOMLib.STAClass.RaiseError()** occurred on thread 1, and the call to the **msvcr70!abort()** and ultimately **kernel32!ExitProcess()** occurred on thread 10. Here, all calls take place on thread 16 because **ASPCompat** forces the **ProcessRequest** method to execute in a thread that has been initialized to the STA threading model.

For the reasons shown above, Microsoft recommends that you use the supported scenario of **ASPCompat** when calling COM components marked **Apartment** from ASP.NET pages. This scenario has proven to perform better and be more reliable in Microsoft stress labs.

Another server-based technology, XML Web services, doesn't support **ASPCompat**. See article Q303375, "INFO: XML Web Services and Apartment Objects" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q303375*. The component must be registered in COM+ as a library application, and **Synchronization** must be set to **Required** so that it runs on a COM+ STA worker thread similarly to the examples above.

Exploring Further

This walkthrough includes code to call COM components marked either **Apartment** or **Free.** Try instantiating other components marked either **Both** or **Neutral**. This can be accomplished by creating **DebuggingComLib**, **BothClass**, or **DebuggingComLib.NeutralClass** objects from the ASP.NET pages.

Try these ideas to learn more:

1. Consider how the walkthrough results differ if you click the second button-click event, which calls an MTA COM object.
2. Create an unconfigured COM component marked **Apartment** with a method that calls **Sleep()**. Then make another request to create another component instance of the same type and see if the object creation fails or succeeds.

Conclusion

This walkthrough has shown you how to use Microsoft tools, in particular WinDbg, the SOS extension DLL, and ADPlus, to investigate a crash in an ASP.NET application. In this scenario, the crash was caused by a call to **ExitProcess**, and as a result the process exited.

Using the dump files produced by ADPlus, you have discovered how to find information about the threads executing within a process, and about how to use call-stack information to determine which methods are executing on each thread.

You have also learned about the **ASPCompat** attribute, which provides better and more reliable performance when calling COM components marked **Apartment** from ASP.NET pages.

Appendix

This appendix contains the following:

- **Thread state values.** This contains a list of possible thread states and their descriptions.
- **Application code.** This lists the Visual C# code for the three ASP.NET applications used in the scenarios.
- **Debugging with CorDbg.** This shows you how to use CorDbg as an alternative to WinDbg for debugging managed code. It is an addendum to Chapter 3, "Debugging Contention Problems."
- **Exploring further.** This contains a list of articles to read for more information.

Thread State Values

The following table lists the possible thread states that are used to construct the thread state values reported by the !thread command of the SOS.dll debugger extension:

Table 5.1: Thread state values and descriptions

Mnemonic	Value	Description
TS_Unknown	0x00000000	Threads are initialized to null
TS_StopRequested	0x00000001	Process will be stopped at the next opportunity
TS_GCSuspendPending	0x00000002	Waiting to get to safe spot for garbage collector (GC)
TS_UserSuspendPending	0x00000004	User suspension at the next opportunity
TS_DebugSuspendPending	0x00000008	Indicates whether the debugger is suspending threads
TS_GCOnTransitions	0x00000010	Indicates if a GC is forced on stub transitions (GCStress only)
TS_LegalToJoin	0x00000020	Indicates if it is now legal to attempt a Join()
TS_Hijacked	0x00000080	Return address has been hijacked
TS_Background	0x00000200	Thread is a background thread
TS_Unstarted	0x00000400	Thread has never been started
TS_Dead	0x00000800	Thread is dead
TS_WeOwn	0x00001000	Exposed object initiated this thread
TS_CoInitialized	0x00002000	CoInitialize has been called for this thread
TS_InSTA	0x00004000	Thread hosts an STA
TS_InMTA	0x00008000	Thread is part of the MTA
TS_ReportDead	0x00010000	Thread is in WaitForOtherThreads()
TS_SyncSuspended	0x00080000	Thread is suspended with WaitSuspendEvent
TS_DebugWillSync	0x00100000	Debugger will wait for this thread to sync
TS_RedirectingEntryPoint	0x00200000	Redirecting entry point. Do not call managed entry point when set
TS_SuspendUnstarted	0x00400000	Indicates latching a user suspension on an unstarted thread
TS_ThreadPoolThread	0x00800000	Indicates if this is a thread pool thread

Mnemonic	Value	Description
TS_TPWorkerThread	0x01000000	Indicates if this is a thread pool worker thread (if not, it is a thread pool completion port thread)
TS_Interruptible	0x02000000	Thread is sitting in either a Sleep(), Wait(), or Join()
TS_Interrupted	0x04000000	Thread was awakened by an interrupt APC
TS_AbortRequested	0x08000000	Process will be stopped at the next opportunity to trip the thread
TS_AbortInitiated	0x10000000	Set when abort is begun
TS_UserStopRequested	0x20000000	Set when a user stop is requested. This is different from TS_StopRequested
TS_GuardPageGone	0x40000000	Stack overflow, not yet reset
TS_Detached	0x80000000	Thread was detached by DllMain

Application Code

This section of the appendix lists the Visual C#™ development tool code for the three ASP.NET applications used in the scenarios.

Contention.aspx.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace Debugging
{
   /// <summary>
   /// Summary description for Contention.
   /// </summary>
   public class Contention : System.Web.UI.Page
   {
       protected System.Web.UI.WebControls.Button btnObtainShared2;
       protected System.Web.UI.WebControls.Button btnFreeShared;
       protected System.Web.UI.WebControls.Button btnObtainShared3;
       protected System.Web.UI.WebControls.Label Label1;
       protected System.Web.UI.WebControls.Button btnObtainShared1;
```

```
        private void Page_Load(object sender, System.EventArgs e)
        {
            PMHealth.OutputProcessInfoTable(Label1);
        }



        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            //
            // CODEGEN: This call is required by the ASP.NET Web Form
            // Designer.
            //
            InitializeComponent();
            base.OnInit(e);
        }

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.btnObtainShared2.Click += new
System.EventHandler(this.btnObtainShared2_Click);
            this.btnFreeShared.Click += new
System.EventHandler(this.btnFreeShared_Click);
            this.btnObtainShared3.Click += new
System.EventHandler(this.btnObtainShared3_Click);
            this.btnObtainShared1.Click += new
System.EventHandler(this.btnObtainShared1_Click);
            this.Load += new System.EventHandler(this.Page_Load);

        }
        #endregion

        private void btnObtainShared1_Click(object sender,
System.EventArgs e)
        {
            SharedResource.Wait();
            Label1.Text += "<BR>Obtained Shared Resource 1";
        }

        private void btnObtainShared2_Click(object sender,
System.EventArgs e)
        {
            SharedResource.Wait();
            Label1.Text += "<BR>Obtained Shared Resource 2";
        }

        private void btnObtainShared3_Click(object sender,
System.EventArgs e)
        {
```

```
         SharedResource.Wait();
         Label1.Text += "<BR>Obtained Shared Resource 3";
      }

      private void btnFreeShared_Click(object sender,
System.EventArgs e)
      {
         SharedResource.Free();
         Label1.Text = "Freed Shared Resource";
         SharedResource.Reset();
         Label1.Text = Label1.Text += "<BR>Resetting Event";
      }
   }
}
```

Memory.aspx.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Management;

/*************************************************************
Title: Memory.aspx
Purpose: Running this ASP.NET page can demonstrate memory
consumption in either small or large increments.
To consider: How the Garbage Collector cleans up objects stored in
memory and how the size of those objects determines how, when, and if
the memory is every freed or reusable.

Partial Lab Instructions:
In the browser window, click Allocate 20 MB Objects once. Switch back
to Task Manager. Click the Performance tab and examine the Page File
Usage History. Click the same button again (2 total). Click the same
button once more (3 total) and immediately switch to the Performance
tab of Task Manager. Note: the sudden drop off.
Switch back to the browser window. Click Refresh Stats in the browser
window.
Note: There were two processes, PID 2740, and now a new process, ID
3128, and that 2740 was shutdown due to memory limit exceeded. (PID
numbers will vary from run to run.)
Examine the Application Event Log for Errors. Note: PID 2740 was
recycled because it exceeded 306 MB (60 of physical RAM)
Click the Refresh button on the browser page.
```

```
Using System Monitor. Note: Private Bytes and # Bytes in All Heaps
are both increasing.
Click the 20 MB Button once more.
Note: the Cache API Entries. And the # Bytes in all heaps. You can
also look at the Large Object Heap Size to determine that most of the
GC Heap memory is in the large object heap.

**********************************************************/

namespace Debugging
{
   /// <summary>
   /// Summary description for Memory.
   /// </summary>
   public class Memory : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.Label Label1;
      protected System.Web.UI.WebControls.Button btn20MB;
      protected System.Web.UI.WebControls.Button btnFree;
      protected System.Web.UI.WebControls.Button btnRefresh;
      protected System.Web.UI.WebControls.Button btn200K;

      private void Page_Load(object sender,
                             System.EventArgs e)
      {
         PMHealth.OutputProcessInfoTable(Label1);
      }


      // Web Form Designer generated code omitted
      private void btn20MB_Click(object sender,
                                 System.EventArgs e)
      {
         //Allocate 20 MB objects
         //Use a System.Web.SessionState Session object
         //Get the CacheList Session object provided by
         //ASP.NET
         //If the object is null then populate it with
         //a unique value.
         if (Session["CacheList"] == null)
         {
            Session["CacheList"] = new ArrayList();
         }
         for (int i = 0; i< 5; i++)
         {
            long objSize = 20000000;
            //Populate the variable with a unique value
            string stime = DateTime.Now.Millisecond.ToString();
            string cachekey = "LOCache-" + i.ToString() + ":" +
stime;
```

```
            //System.Web.Caching.Cache Page.Cache
            //Associate a key with a cached object
            Cache[cachekey] = CreateLargeObject(objSize);
            //Store the cache key in a Session scope
            StoreCacheListInSession(cachekey);
         }
         PMHealth.OutputUpdatedMemoryStatsTable(Label1);
      }
      private byte[] CreateLargeObject(long size)
      {
         //Create and return a byte of specified size
         byte[] largeByteArray = new byte[size];
         return largeByteArray;
      }
      private void StoreCacheListInSession(string list)
      {
         //Store the list of cache keys in a Session object
         //Specifically, dynamically populate an ArrayList with
         //cached values and add to Session variable
         ArrayList arr = (ArrayList)Session["CacheList"];
         arr.Add(list);
         Session["CacheList"] = arr;
      }

      private void btnFree_Click(object sender,
                                 System.EventArgs e)
      {
         //Display message if the Session object is null.
         if (Session["CacheList"] == null)
         {
            Label1.Text += "<TABLE BORDER>";
            Label1.Text += "<TR><TD>No Entries in Cache (CacheList is
null)</TD></TR></TABLE>";

            PMHealth.OutputGCTotalMemoryandCollect(Label1);
         }
         else
         {
            ArrayList arr = (ArrayList)Session["CacheList"];
            if (arr.Count == 0)
            {
               Label1.Text += "<TABLE BORDER>";
               Label1.Text += "<TR><TD>No Entries in Cache
(CacheList.Count is 0)</TD></TR></TABLE>";

               PMHealth.OutputGCTotalMemoryandCollect(Label1);
            }
            else
            {
               //If Session object is populated, clear its contents.
               Label1.Text += "<TABLE BORDER>";
               Label1.Text += "<TR><TD><FONT COLOR=GREEN>";
               Label1.Text += "Before Emptying Cache";
               Label1.Text += "</FONT></TD></TR>";
               Label1.Text += "<TR><TD><FONT COLOR=GREEN>";
```

```
                //Retrieve the number of bytes currently thought to be
                //allocated and convert to KB
                Label1.Text += "GC.TotalMemory </TD><TD>" +
(GC.GetTotalMemory(false) / 1000).ToString() + " KB";
                Label1.Text += "</FONT></TD></TR>";
                Label1.Text += "<TR><TD><FONT COLOR=GREEN>";
                //Get the amount of physical memory mapped to the
                //process context and convert to KB
                Label1.Text += "WorkingSet</TD><TD>" +
(Environment.WorkingSet / 1000).ToString() + " KB";
                Label1.Text += "</FONT></TD></TR></TABLE>";

                foreach (string s in arr)
                {
                   if (Cache[s] != null)
                   {
                      //Remove each item from the cache
                      //using the cache key
                      Cache.Remove(s);
                      Label1.Text += "<TABLE BORDER>";
                      Label1.Text += "<TR><TD>";
                      Label1.Text +=
                            "Emptying out cachekey " + s;
                      Label1.Text += "</TD></TR></TABLE>";
                   }
                }
                //Remove elements from the arraylist and
                //Session object
                arr.Clear();
                Session["CacheList"] = arr;
                PMHealth.OutputGCTotalMemoryandCollect(Label1);
             }
          }
       }

       private void btnRefresh_Click(object sender,
                                     System.EventArgs e)
       {
           PMHealth.OutputUpdatedMemoryStatsTable(Label1);
       }

       private void btn200K_Click(object sender,
                                  System.EventArgs e)
       {
          //Allocate 200K objects
          //Use a System.Web.SessionState.HttpSessionState
          //Page.Session
          //Get the CacheList Session object provided by
          //ASP.NET. If the object is null then populate it
          //with a unique value.
          if (Session["CacheList"] == null)
          {
             Session["CacheList"] = new ArrayList();
```

```
            }
            for (int i = 0; i< 5; i++)
            {
               long objSize = 200000;
               //Populate the variable with a unique value
               string stime =
                      DateTime.Now.Millisecond.ToString();
               string cachekey = "LOCache-" + i.ToString() +
                                 ":" + stime;
               //System.Web.Caching.Cache Page.Cache
               //Associate a key with a cached object
               Cache[cachekey] = CreateLargeObject(objSize);
               //Store the cache key in a Session scope
               StoreCacheListInSession(cachekey);
            }
         }
      }
   }
```

Unexpected.aspx.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using DebuggingCOMLib;

namespace Debugging
{
   /// <summary>
   /// Summary description for Unexpected.
   /// </summary>
   public class Unexpected : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.Label Label1;
      protected System.Web.UI.WebControls.Button btnSTA;
      protected System.Web.UI.WebControls.Button btnMTA;

      private void Page_Load(object sender, System.EventArgs e)
      {
         PMHealth.OutputProcessInfoTable(Label1);
      }
```

```
        // Web Form Designer generated code omitted

        private void btnSTA_Click(object sender,
                                  System.EventArgs e)
        {
            DebuggingCOMLib.STAClass staobj =
                        new DebuggingCOMLib.STAClass();
            staobj.CallCOM(1);
            Label1.Text += "STA Call Completed successfully";
        }

        private void btnMTA_Click(object sender,
                                  System.EventArgs e)
        {
            DebuggingCOMLib.MTAClass mtaobj =
                         new DebuggingCOMLib.MTAClass();
            mtaobj.CallCOM(1);
            Label1.Text += "MTA Call Completed sucessfully";
        }
    }
}
```

Pmhealth.cs

Pmhealth.cs contains the code that displays the Process Model statistics for the ASP.NET applications used in the scenario ASP.NET pages.

```
using System;
using System.Web;
using System.Runtime.InteropServices;

namespace Debugging
{
        /// <summary>
        /// Summary description for PMHealth.
        /// </summary>
        public class PMHealth
        {
                static PMHealth()
                {
                        //
                        // TODO: Add constructor logic here
                        //
                }
                public static void
OutputProcessInfoTable(System.Web.UI.WebControls.Label lbl)
                {
                        //ProcessModelInfo Class inherits from the System.Web
                         //Namespace.
                        //The static method GetHistory populates the
                         //ProcessInfo object.
```

```
                        //Properties of ProcessInfo provide information about
                          //the ASP.NET worker processes.
                        ProcessInfo[] history =
ProcessModelInfo.GetHistory(100);
                        for( int i=0; i<history.Length; i++ )
                        {
                                lbl.Text += "<TABLE BORDER>";
                                lbl.Text += "<TR><TD>";
                                //System.DateTime ProcessInfo.StartTime gets
                                  //the time at which the process started.
                                lbl.Text += "StartTime<TD>"      +
history[i].StartTime.ToString()       + "<BR>";
                                lbl.Text += "<TR><TD>";
                                //System.TimeSpan ProcessInfo.Age gets the
                                  //length of time the process has been running.
                                lbl.Text += "Age<TD>"            +
history[i].Age.ToString()             + "<BR>";
                                lbl.Text += "<TR><TD>";
                                //ProcessInfo.ProcessID. Get the ID assigned
                                 //to the process.
                                lbl.Text += "ProcessID<TD>"      +
history[i].ProcessID.ToString()       + "<BR>";
                                lbl.Text += "<TR><TD>";
                                //ProcessInfo.RequestCount.  Get the number of
                                 //start requests.
                                lbl.Text += "RequestCount<TD>"   +
history[i].RequestCount.ToString()    + "<BR>";
                                lbl.Text += "<TR><TD>";
                                //System.String Memory.GetProcessStatus.  Get
                                  //the current status of a process.
                                lbl.Text += "Status<TD>"         +
GetProcessStatus( history[i].Status ) + "<BR>";
                                lbl.Text += "<TR><TD>";
                                //System.String Memory.GetShutdownReason. Get
                                  //the reason why a process shut down.
                                lbl.Text += "ShutdownReason<TD>" +
GetShutdownReason( history[i].ShutdownReason) + "<BR>";
                                lbl.Text += "<TR><TD>";
                                //ProcessInfo.PeakMemoryUsed gets the maximum
                                  //amount of memory the process has used.
                                lbl.Text += "PeakMemoryUsed<TD>" +
history[i].PeakMemoryUsed.ToString()  + "<BR>";
                                lbl.Text += "</TABLE><P><P>";
                        }

                }
                private static String GetProcessStatus( ProcessStatus ps )
                {
                        //ProcessStatus indicates the current status of a
                         //process.
                        String s = "Unknown";
                        switch(ps)
                        {
```

```
                                //The process is running
                                case ProcessStatus.Alive:
                                        s = "Alive";
                                        break;


                                //The process has shut down normally after
                                //receiving a shut down message from the IIS process.
                                case ProcessStatus.ShutDown:
                                        s = "Shutdown";
                                        break;

                                //The process has begun to shut down.
                                case ProcessStatus.ShuttingDown:
                                        s = "Shutting Down";
                                        break;

                                //The process was forced to terminate by the
                                //IIS process.
                                case ProcessStatus.Terminated:
                                        s = "Terminated";
                                        break;
                        }
                        return s;
                }

                private static String GetShutdownReason(
ProcessShutdownReason psr )
                {
                        //ProcessShutdownReason indicates why a process has
                        //shut down.
                        String s = "Unknown";
                        switch(psr)
                        {

                                        //The process was restarted because a
                                        //deadlock was suspected
                                case ProcessShutdownReason.DeadlockSuspected:
                                        s = "Deadlock Suspected";
                                        break;

                                        //The process exceeded the allowable
                                        //idle time.
                                case ProcessShutdownReason.IdleTimeout:
                                        s = "Idle Timeout";
                                        break;

                                        //The process exceeded the per-process
                                        //memory limit.
                                case ProcessShutdownReason.MemoryLimitExceeded:
                                        s = "Memory Limit Exceeded";
                                        break;
```

```
                                        //No reason available
                                case ProcessShutdownReason.None:
                                        s = "N/A";
                                        break;

                                        //The process was restarted because it
                                       //didn't respond to a ping from the IIS
                                        //process
                                case ProcessShutdownReason.PingFailed:
                                        s = "Ping Failed";
                                        break;

                                        //Requests assigned to the process
                                       //exceeded the allowable number in the queue.
                                case ProcessShutdownReason.RequestQueueLimit:
                                        s = "Request Queue Limit";
                                        break;

                                        //Requests executed by the process
                                       //exceeded the allowable limit.
                                case ProcessShutdownReason.RequestsLimit:
                                        s = "Requests Limit";
                                        break;

                                        //The process restarted because it was
                                       //alive longer than allowed.
                                case ProcessShutdownReason.Timeout:
                                        s = "Timeout";
                                        break;

                                        //The process shut down unexpectedly.
                                case ProcessShutdownReason.Unexpected:
                                        s = "Unexpected";
                                        break;
                        }
                        return s;
                }
                public static void
OutputUpdatedMemoryStatsTable(System.Web.UI.WebControls.Label lbl)
                {
                        //Update the Memory Statistics
                        lbl.Text += "<TABLE BORDER>";
                        lbl.Text += "<TR><TD><FONT COLOR=BLUE>";
                        lbl.Text += "Updated Memory Stats";
                        lbl.Text += "</FONT></TD><TD></TD></TR>";
                        lbl.Text += "<TR><TD><FONT COLOR=BLUE>";
                        //Retrieve the number of bytes currently thought to
                       //be allocated and convert to KB
                        lbl.Text += "GC.TotalMemory </TD><TD>" +
(GC.GetTotalMemory(false) / 1024).ToString() + " KB";
                        lbl.Text += "</FONT></TD></TR>";
                        lbl.Text += "<TR><TD><FONT COLOR=BLUE>";
                        //Get the amount of physical memory mapped to the
                       //process context and convert to KB
```

```
                        lbl.Text += "Private Bytes</TD><TD>" +
GetMemoryInfoFromOS() + " KB";
                        lbl.Text += "</FONT></TD></TR></TABLE>";
                }

                public static void
OutputGCTotalMemoryandCollect(System.Web.UI.WebControls.Label lbl)
                {
                        lbl.Text += "<TABLE BORDER>";
                        lbl.Text += "<TR><TD><FONT COLOR=RED>";
                        //Display allocated memory before calling Garbage
                        //Collector
                        //Retrieve the number of bytes currently thought to
                        //be allocated and convert to KB
                        lbl.Text += "GC.TotalMemory </TD><TD>" +
(GC.GetTotalMemory(false) / 1024).ToString() + " KB";
                        lbl.Text += "</FONT></TD></TR></TABLE>";
                        //Force Garbage Collection of all Generations
                        //NOTE: Calling System.GC.Collect() manually can
                        //degrade performance of your application
                        //We call it in this lab so that you don't have
                        //to wait for a Generation 2 collection to
                        //occur before you can see the results in
                        //the heap.
                        GC.Collect();
                        lbl.Text += "<TABLE BORDER>";
                        lbl.Text += "<TR><TD><FONT COLOR=GREEN>";
                        //Display allocated memory after calling Garbage
                        //Collector
                        lbl.Text += "After GC.Collect";
                        lbl.Text += "</FONT></TD><TD></TD></TR>";
                        lbl.Text += "<TR><TD><FONT COLOR=GREEN>";
                        //Retrieve the number of bytes currently thought to
                        //be allocated and convert to KB
                        lbl.Text += "GC.TotalMemory </TD><TD>" +
(GC.GetTotalMemory(false) / 1024).ToString() + " KB";
                        lbl.Text += "</FONT></TD></TR>";
                        lbl.Text += "<TR><TD><FONT COLOR=GREEN>";
                        //Get the amount of physical memory mapped to the
                        //process context and convert to KB
                        lbl.Text += "Private Bytes</TD><TD>" +
GetMemoryInfoFromOS() + " KB";
                        lbl.Text += "</FONT></TD></TR></TABLE>";
                }
                [StructLayout(LayoutKind.Sequential)]
                        public class PROCESS_MEMORY_COUNTERS
                {
                        public int cb;
                        public int PageFaultCount;
                        public int PeakWorkingSetSize;
                        public int WorkingSetSize;
                        public int QuotaPeakPagedPoolUsage;
                        public int QuotaPagedPoolUsage;
```

```
                        public int QuotaPeakNonPagedPoolUsage;
                        public int QuotaNonPagedPoolUsage;
                        public int PagefileUsage;
                        public int PeakPagefileUsage;
                }
                [DllImport("psapi.dll")]
                public static extern int GetProcessMemoryInfo (
                        int hProcess,
                        [Out] PROCESS_MEMORY_COUNTERS counters,
                        int size
                        );

                private static int GetMemoryInfoFromOS()
                {
                        PROCESS_MEMORY_COUNTERS counters = new
PROCESS_MEMORY_COUNTERS();
                        GetProcessMemoryInfo (-1, counters, 40);
                        if (counters != null && counters.PagefileUsage != 0)
                                return counters.PagefileUsage/1024;
                        return 0;
                }
        }
}
```

Debugging with CorDbg

This section is an addendum to Chapter 3, "Debugging Contention Problems."

CorDbg provides an alternative to WinDbg for debugging managed code using a command line interface instead of a GUI. The difference between using this tool and using WinDbg is that you do not have to create a dump file or terminate the process.

This section shows you how to use CorDbg to attach to the Aspnet_wp.exe process, list the loaded modules in the process, and display call stack traces for specified threads. After listing specific CorDbg commands, there are examples of how to use these commands. Finally, there is an example showing you how to automate this process using a batch file.

Note: For more information on CorDbg, see the Microsoft® .NET Framework SDK documentation and the CorDbg program Help.

The following table lists the commonly used CorDbg commands.

Table5.2: CorDbg commands and descriptions

Command	Description
pro[**cessenum**]	Enumerates all managed processes and the application domains in each process.
a[**ttach**] *pid*	Attaches the debugger to the running process specified by the *pid* argument.
l[**ist**] **mod**	Lists the loaded assemblies in the process by AppDomain.
w[**here**] [*count*]	Displays a stack trace for the current thread. If you do not specify an argument, the tool displays a complete stack trace. If you specify an argument, the tool displays the specified number of stack frames.
de[**tach**]	Detaches the debugger from the current process. The process automatically continues and runs as if a debugger is not attached to it.
q[**uit**]	Stops the current process (if detach was not previously executed) and exits the debugger.
? [*command* ...]	Displays descriptions for the specified commands. If you do not specify any arguments, Cordbg.exe displays a list of debugger commands.

Using CorDbg to Display Call Stack Information

Using CorDbg, you obtain the call stacks for the three threads created by the button-click events in Contention.aspx.

► **To display call stack information**

1. To start the Aspnet_wp.exe process so that you can attach to it, browse to *http://localhost/Debugging/Contention.aspx.*
2. Click the three **Obtain Shared Resource** buttons on Contention.aspx. The page does not return data.
3. Open the Visual Studio .NET Command Prompt, and then type **cordbg** to start the debugger.

Note: You need to have the .NET Framework SDK \Bin folder in your path to be able to run CorDbg. If you are using Visual Studio® .NET, you can open a Visual Studio .NET Command Prompt window, which has the path set correctly.

You should see a startup banner and prompt like the following:

```
Microsoft (R) Common Language Runtime Test Debugger Shell Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

(cordbg)
```

4. Using the CorDbg **pro** command, find the Aspnet_wp.exe process ID among the listed running processes.

```
(cordbg) pro

PID=0xab8 (3820)
Name=C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705\aspnet_wp.exe
        ID=2  AppDomainName=/LM/w3svc/1/root/Debugging-1-
126695838789138931
        ID=1  AppDomainName=DefaultDomain
```

Note: If you have other managed processes running, you will see other entries listed here. To ensure you are using the Aspnet_wp.exe process ID related to the correct browser session, close any other browser windows and other managed applications, and then restart IIS.

---

5. Attach the debugger to the running Aspnet_wp.exe process by using the **attach** command followed by the Aspnet_wp.exe process ID returned in the previous step.

   Note that the symbols are not loaded for runtime components, so don't worry about the "cannot load symbol" warning messages. The first few lines of output from the command are shown below:

```
(cordbg) a 3820
Process 3820/0xab8 created.
...
```

6. List the loaded modules in the process using the **l mod** command.

   The command line is shown below:

```
(cordbg) l mod
```

7. Display and examine the call stack trace for all current threads using the ***w** command. Specify an argument of 200; this lists up to 200 levels of stack trace.

```
(cordbg) *w 200
```

8. Now detach from the process, and quit the debugger:

```
(cordbg) de
(cordbg) q
```

Using a Script File

An alternative to running each of the commands individually is to use a .cmd file that includes the commands and writes the output to a disk.

The code listing for managed_threads.cmd follows the output shown below. To run this file, make sure that managed_threads.cmd is located in the C:\Debuggers folder.

► To run the script

1. Open a command prompt window, and then type **iisreset** at the prompt to restart IIS.
2. Browse to *http://localhost/Debugging/contention.aspx.*
3. Click the **Obtain Shared Resource 1**, **2**,and **3** buttons.
4. At the command line, type the following:

```
C:\Debuggers>managed_threads.cmd 3820
```

   Make sure that you replace "3280" with the ID of your Aspnet_wp.exe process.

The following listing shows the contents of the output file (C:\Debuggers\managed_threads.txt), with commands shown in bold:

```
Microsoft (R) Common Language Runtime Test Debugger Shell Version
1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

(cordbg) a 3820
Process 3820/0xeec created.
Warning: couldn't load symbols for
c:\windows\microsoft.net\framework\v1.0.3705\mscorlib.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.drawing\1.0.3300.0__b03f5f7f11d50a3
a\system.drawing.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.directoryservices\1.0.3300.0__b03f5f7f11d50a3a\
system.directoryservices.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.messaging\1.0.3300.0__b03f5f7f11d50a3a\
system.messaging.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.serviceprocess\1.0.3300.0__b03f5f7f11d50a3a\
system.serviceprocess.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.data\1.0.3300.0__b77a5c561934e089\system.data.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.web\1.0.3300.0__b03f5f7f11d50a3a\system.web.dll
```

```
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.enterpriseservices\1.0.3300.0__b03f5f7f11d50a3a\
system.enterpriseservices.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.enterpriseservices\1.0.3300.0__b03f5f7f11d50a3a\
system.enterpriseservices.thunk.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.web.regularexpressions\1.0.3300.0__b03f5f7f11d50a3a
\system.web.regularexpressions.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.xml\1.0.3300.0__b77a5c561934e089\system.xml.dll
Warning: couldn't load symbols for
c:\windows\microsoft.net\framework\v1.0.3705\mscorlib.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.drawing\1.0.3300.0__b03f5f7f11d50a3a\system.drawing.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.directoryservices\1.0.3300.0__b03f5f7f11d50a3a\
system.directoryservices.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.messaging\1.0.3300.0__b03f5f7f11d50a3a\
system.messaging.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.serviceprocess\1.0.3300.0__b03f5f7f11d50a3a\
system.serviceprocess.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.data\1.0.3300.0__b77a5c561934e089\system.data.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.web\1.0.3300.0__b03f5f7f11d50a3a\system.web.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.xml\1.0.3300.0__b77a5c561934e089\system.xml.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.web.regularexpressions\1.0.3300.0__b03f5f7f11d50a3a
\system.web.regularexpressions.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.enterpriseservices\1.0.3300.0__b03f5f7f11d50a3a\
system.enterpriseservices.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.enterpriseservices\1.0.3300.0__b03f5f7f11d50a3a\
system.enterpriseservices.thunk.dll
Warning: couldn't load symbols for
c:\windows\assembly\gac\system.web.services\1.0.3300.0__b03f5f7f11d50a3a\
system.web.services.dll
Warning: couldn't load symbols for
c:\windows\microsoft.net\framework\v1.0.3705\temporary asp.net
files\debugging\34374a37\79f52287\assembly\dl\2b680d64\198fa8c2_9d17c201\
interop.debuggingcomlib.dll
[thread 0xf30] Thread created.
Unable to  determine existence of prolog, if any
[thread 0xe00] Thread created.
[thread 0xf40] Thread created.
[thread 0x6f4] Thread created.
```

```
[thread 0xd78] Thread created.
[thread 0xf30]
Thread 0xf30 R 0x7FFE0304:  <unknown>
(cordbg) l mod
Module 0x00174f0c,
c:\windows\assembly\gac\system.drawing\1.0.3300.0__b03f5f7f11d50a3a\
system.drawing.dll -- AD #2 -- Symbols not loaded
Module 0x00135824,
c:\windows\assembly\gac\system.serviceprocess\1.0.3300.0__b03f5f7f11d50a3a\
system.serviceprocess.dll -- AD #1 -- Symbols not loaded
Module 0x00179404,
c:\windows\assembly\gac\system.serviceprocess\1.0.3300.0__b03f5f7f11d50a3a\
system.serviceprocess.dll -- AD #2 -- Symbols not loaded
Module 0x0016f3cc,
c:\windows\assembly\gac\system.xml\1.0.3300.0__b77a5c561934e089\
system.xml.dll -- AD #1 -- Symbols not loaded
Module 0x000b5ff4,
c:\windows\assembly\gac\system\1.0.3300.0__b77a5c561934e089\
system.dll -- AD #1 -- Symbols not loaded
Module 0x00177514,
c:\windows\assembly\gac\system.data\1.0.3300.0__b77a5c561934e089\
system.data.dll -- AD #2 -- Symbols not loaded
Module 0x0011897c,
c:\windows\assembly\gac\system.enterpriseservices\1.0.3300.0__b03f5f7f11d50a3a\
system.enterpriseservices.thunk.dll -- AD #1 -- Symbols not loaded
Module 0x001190ec,
c:\windows\assembly\gac\system.data\1.0.3300.0__b77a5c561934e089\
system.data.dll -- AD #1 -- Symbols not loaded
Module 0x00090ac4,
c:\windows\microsoft.net\framework\v1.0.3705\mscorlib.dll -- AD #1 -
Symbols not loaded
Module 0x015dc294,
c:\windows\assembly\gac\system.web.services\1.0.3300.0__b03f5f7f11d50a3a\
system.web.services.dll -- AD #2 -- Symbols not loaded
Module 0x0017ad4c,
c:\windows\assembly\gac\system.web.regularexpressions\1.0.3300.0__b03f5f7f11d50a3a
\system.web.regularexpressions.dll -- AD #2 -- Symbols not loaded
Module 0x0017ae74,
c:\windows\assembly\gac\system.xml\1.0.3300.0__b77a5c561934e089\
system.xml.dll -- AD #2 -- Symbols not loaded
Module 0x015db804,
c:\windows\microsoft.net\framework\v1.0.3705\temporary asp.net
files\debugging\34374a37\79f52287\pvnt75wl.dll -- AD #2
Module 0x0017f99c,
c:\windows\microsoft.net\framework\v1.0.3705\temporary asp.net
files\debugging\34374a37\79f52287\assembly\dl\18040135\6ef006c4_9d17c201\
debugging.dll -- AD #2
Module 0x00178fbc,
c:\windows\assembly\gac\system.web\1.0.3300.0__b03f5f7f11d50a3a\
system.web.dll -- AD #2 -- Symbols not loaded
Module 0x001755fc,
c:\windows\assembly\gac\system.directoryservices\1.0.3300.0__b03f5f7f11d50a3a\
system.directoryservices.dll -- AD #2 -- Symbols not loaded
```

```
Module 0x00171efc,
c:\windows\microsoft.net\framework\v1.0.3705\mscorlib.dll -- AD #2 -
Symbols not loaded
Module 0x0016a894,
c:\windows\microsoft.net\framework\v1.0.3705\temporary asp.net
files\debugging\34374a37\79f52287\assembly\dl\2b680d64\198fa8c2_9d17c201\
interop.debuggingcomlib.dll -- AD #2 -- Symbols not loaded
Module 0x00142ccc,
c:\windows\assembly\gac\system.web\1.0.3300.0__b03f5f7f11d50a3a\
system.web.dll -- AD #1 -- Symbols not loaded
Module 0x0010b2ac,
c:\windows\assembly\gac\system.directoryservices\1.0.3300.0__b03f5f7f11d50a3a\
system.directoryservices.dll -- AD #1 -- Symbols not loaded
Module 0x0016ba8c,
c:\windows\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll -
AD #2 -- Symbols not loaded
Module 0x00142ba4,
c:\windows\assembly\gac\system.enterpriseservices\1.0.3300.0__b03f5f7f11d50a3a\
system.enterpriseservices.dll -- AD #1 -- Symbols not loaded
Module 0x0011880c,
c:\windows\assembly\gac\system.messaging\1.0.3300.0__b03f5f7f11d50a3a\
system.messaging.dll -- AD #1 -- Symbols not loaded
Module 0x000bd6b4,
c:\windows\assembly\gac\system.drawing\1.0.3300.0__b03f5f7f11d50a3a\
system.drawing.dll -- AD #1 -- Symbols not loaded
Module 0x015da5b4,
c:\windows\assembly\gac\system.enterpriseservices\1.0.3300.0__b03f5f7f11d50a3a\
system.enterpriseservices.thunk.dll -- AD #2 -- Symbols not loaded
Module 0x015d6dbc,
c:\windows\assembly\gac\system.enterpriseservices\1.0.3300.0__b03f5f7f11d50a3a\
system.enterpriseservices.dll -- AD #2 -- Symbols not loaded
Module 0x0017718c,
c:\windows\microsoft.net\framework\v1.0.3705\temporary asp.net
files\debugging\34374a37\79f52287\0rzdgjzl.dll -- AD #2
Module 0x001770fc,
c:\windows\assembly\gac\system.messaging\1.0.3300.0__b03f5f7f11d50a3a\
system.messaging.dll -- AD #2 -- Symbols not loaded
Module 0x00142c34,
c:\windows\assembly\gac\system.web.regularexpressions\1.0.3300.0__b03f5f7f11d50a3a
\system.web.regularexpressions.dll -- AD #1 -- Symbols not loaded
(cordbg) *w 200


Thread 0xd78 R 0x7FFE0304:  <unknown>
Thread 0xd78 Current State:Normal
*       0x7FFE0304:  <unknown>
-- Unmanaged transition --
0)  mscorlib!System.Threading.WaitHandle::WaitOne +0036 [no source
information available]
1)  debugging!Debugging.SharedResource::Wait +005b in
C:\Inetpub\wwwroot\Debugging\SharedResource.cs:55
2)  debugging!Debugging.Contention::btnObtainShared3_Click +0013 in
c:\inetpub\wwwroot\debugging\contention.aspx.cs:71
```

```
                sender=(0x012113a8) <System.Web.UI.WebControls.Button>
                e=(0x012193c4) <System.EventArgs>
3)  system.web!System.Web.UI.WebControls.Button::OnClick +006d [no
source information available]
4)  system.web!System.Web.UI.WebControls.Button::System.Web.UI.
IPostBackEventHandler.RaisePostBackEvent +003a [no source information available]
5)  system.web!System.Web.UI.Page::RaisePostBackEvent +0013 [no
source information available]
6)  system.web!System.Web.UI.Page::RaisePostBackEvent +0022 [no
source information available]
7)  system.web!System.Web.UI.Page::ProcessRequestMain +04f0 [no
source information available]
8)  system.web!System.Web.UI.Page::ProcessRequest +0077 [no source
information available]
9)  system.web!System.Web.UI.Page::ProcessRequest +0013 [no source
information available]
10)  system.web!CallHandlerExecutionStep::Execute +00b4 [no source
information available]
11)  system.web!System.Web.HttpApplication::ExecuteStep +0058 [no
source information available]
12)  system.web!System.Web.HttpApplication::ResumeSteps +00f3 [no
source information available]
13)  system.web!System.Web.HttpApplication::System.Web.
IHttpAsyncHandler.BeginProcessRequest +00e3 [no source information available]
14)  system.web!System.Web.HttpRuntime::ProcessRequestInternal +01e0
[no source information available]
15)  system.web!System.Web.HttpRuntime::ProcessRequest +003d [no
source information available]
16)  system.web!System.Web.Hosting.ISAPIRuntime::ProcessRequest +00b7
[no source information available]
--- Managed transition ---
       0x00421EAE:  <unknown>



Thread 0xf30 R 0x7FFE0304:  <unknown>
Thread 0xf30 Current State:Normal
*      0x7FFE0304:  <unknown>
--- Unmanaged transition ---
0)  mscorlib!System.Threading.WaitHandle::WaitOne +0036 [no source
information available]
1)  debugging!Debugging.SharedResource::Wait +005b in
C:\Inetpub\wwwroot\Debugging\SharedResource.cs:55
2)  debugging!Debugging.Contention::btnObtainShared1_Click +0014 in
c:\inetpub\wwwroot\debugging\contention.aspx.cs:59
                sender=(0x011e91c4) <System.Web.UI.WebControls.Button>
                e=(0x011f136c) <System.EventArgs>
3)  system.web!System.Web.UI.WebControls.Button::OnClick +006d [no
source information available]
4)  system.web!System.Web.UI.WebControls.Button::System.Web.UI.
IPostBackEventHandler.RaisePostBackEvent +003a [no source information available]
5)  system.web!System.Web.UI.Page::RaisePostBackEvent +0013 [no
source information available]
```

```
6)  system.web!System.Web.UI.Page::RaisePostBackEvent +0022 [no
source information available]
7)  system.web!System.Web.UI.Page::ProcessRequestMain +04f0 [no
source information available]
8)  system.web!System.Web.UI.Page::ProcessRequest +0077 [no source
information available]
9)  system.web!System.Web.UI.Page::ProcessRequest +0013 [no source
information available]
10)  system.web!CallHandlerExecutionStep::Execute +00b4 [no source
information available]
11)  system.web!System.Web.HttpApplication::ExecuteStep +0058 [no
source information available]
12)  system.web!System.Web.HttpApplication::ResumeSteps +00f3 [no
source information available]
13)  system.web!System.Web.HttpApplication::System.Web.
IHttpAsyncHandler.BeginProcessRequest +00e3 [no source information available]
14)  system.web!System.Web.HttpRuntime::ProcessRequestInternal +01e0
[no source information available]
15)  system.web!System.Web.HttpRuntime::ProcessRequest +003d [no
source information available]
16)  system.web!System.Web.Hosting.ISAPIRuntime::ProcessRequest +00b7
[no source information available]
--- Managed transition ---
       0x00421EAE:  <unknown>



Thread 0x6f4 R 0x7FFE0304:  <unknown>
Thread 0x6f4 Current State:Normal
*      0x7FFE0304:  <unknown>
--- Unmanaged transition ---
0)  mscorlib!System.Threading.WaitHandle::WaitOne +0036 [no source
information available]
1)  debugging!Debugging.SharedResource::Wait +005b in
C:\Inetpub\wwwroot\Debugging\SharedResource.cs:55
2)  debugging!Debugging.Contention::btnObtainShared2_Click +0013 in
c:\inetpub\wwwroot\debugging\contention.aspx.cs:65
               sender=(0x012043ec) <System.Web.UI.WebControls.Button>
               e=(0x0120c888) <System.EventArgs>
3)  system.web!System.Web.UI.WebControls.Button::OnClick +006d [no
source information available]
4)  system.web!System.Web.UI.WebControls.Button::System.Web.UI.
IPostBackEventHandler.RaisePostBackEvent +003a [no source information available]
5)  system.web!System.Web.UI.Page::RaisePostBackEvent +0013 [no
source information available]
6)  system.web!System.Web.UI.Page::RaisePostBackEvent +0022 [no
source information available]
7)  system.web!System.Web.UI.Page::ProcessRequestMain +04f0 [no
source information available]
8)  system.web!System.Web.UI.Page::ProcessRequest +0077 [no source
information available]
9)  system.web!System.Web.UI.Page::ProcessRequest +0013 [no source
information available]
```

```
10)  system.web!CallHandlerExecutionStep::Execute +00b4 [no source
information available]
11)  system.web!System.Web.HttpApplication::ExecuteStep +0058 [no
source information available]
12)  system.web!System.Web.HttpApplication::ResumeSteps +00f3 [no
source information available]
13)  system.web!System.Web.HttpApplication::System.Web.
IHttpAsyncHandler.BeginProcessRequest +00e3 [no source information available]
14)  system.web!System.Web.HttpRuntime::ProcessRequestInternal +01e0
[no source information available]
15)  system.web!System.Web.HttpRuntime::ProcessRequest +003d [no
source information available]
16)  system.web!System.Web.Hosting.ISAPIRuntime::ProcessRequest +00b7
[no source information available]
--- Managed transition ---
       0x00421EAE:  <unknown>



Thread 0xe00 R 0x7FFE0304:  <unknown>
Thread 0xe00 Current State:Background
*      0x7FFE0304:  <unknown>



Thread 0xf40 R 0x7FFE0304:  <unknown>
Thread 0xf40 Current State:Background
*      0x7FFE0304:  <unknown>

(cordbg) de
(cordbg) q
```

The managed_threads.cmd Script

This is the code listing for the script that produced the output in the previous section:

```
@echo off
if "%_echo%"=="1" echo on
setlocal

REM FILENAME: MANAGED_THREADS.CMD
REM PURPOSE:  To Dump the managed threads for an .NET Process
REM DATE:     5/9/2002
REM AUTHOR:   SIE, GSSC, Microsoft Corporation


if /i "%1" == "" (
   GOTO :HELP
) else (
       cordbg !a %1 !l mod !*w 200 !de !q >managed_threads.txt
```

```
        echo managed threads output is in managed_threads.txt
        GOTO :EOF
)



:HELP
echo.
echo USAGE: %~xn0 [process ID from TLIST.EXE]
echo.
echo %~xn0 will create a log of the managed threads in a .NET Application
echo.
echo For Example:
tlist
goto :EOF

:EOF
endlocal
```

Exploring Further

For more information, read the following articles:

- "COM Threading and Application Architecture in COM+ Applications" on the MSDN Web site at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomser/html/comthread.asp*
- "Thread.ApartmentState Property" on the MSDN Web site at *http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemThreadingThreadClassApartmentStateTopic.asp*
- "STAThreadAttribute Class" on the MSDN Web site at *http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemSTAThreadAttributeClassTopic.asp*
- "MTAThreadAttribute Class" on the MSDN Web site at *http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemMTAThreadAttributeClassTopic.asp*
- Article Q303375, "INFO: XML Web Services and Apartment Objects" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q303375*
- Article Q308095, "PRB: Creating STA Components in the Constructor in ASP.NET ASPCOMPAT Mode Negatively Affects Performance" in the Microsoft Knowledge Base at *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q308095*
- "FIX: Various Problems When You Call Transactional COM+ Components from ASP.NET (Q318000)" on the Microsoft Product Support Services Web site at *http://support.microsoft.com/default.aspx?scid=fh;EN-US;KBHOWTO*
- "HOWTO: Set the COM Apartment Type in Managed Threads (Q318402") on the Microsoft Product Support Services Web site at *http://support.microsoft.com/default.aspx?scid=fh;EN-US;KBHOWTO*

- Article Q318000, "FIX: Problems When You Call Transactional COM+ Components" in the Microsoft Knowledge Base at *http://support.microsoft.com /default.aspx?scid=fh;EN-US;KBHOWTO*
- Article Q308095, "PRB: Performance Degradation When Creating STA Components" in the Microsoft Knowledge Base at *http://support.microsoft.com /default.aspx?scid=fh;EN-US;KBHOWTO*
- Article Q309330, "BUG: InvalidCastException Calling COM Component Marked as STA" in the Microsoft Knowledge Base at *http://support.microsoft.com /default.aspx?scid=fh;EN-US;KBHOWTO*
- Article Q291837, "INFO: Do Not Make Blocking Calls from an STA Component" in the Microsoft Knowledge Base at *http://support.microsoft.com /default.aspx?scid=fh;EN-US;KBHOWTO*