Chapter 2—Worker and Threads

Worker and Threads

In the first chapter I looked into the handling of a page request, and, in particular, the application life cycle and page life cycle. This is the fundamental process that runs when anyone requests a page resource. However, Chapter 1 was a simplification of the situation—I assumed that only one request comes in at a time. Reality is quite different.

Managing hundreds, or even thousands, of simultaneous requests requires advanced knowledge and skills. In this and subsequent chapters you will learn those skills.

Objectives in this chapter:

\* Learn how ASP.NET handles multiple requests

\* Understand how to manage the worker processes and optimize the workload

\* Comprehend the thread model, thread pool behavior and how to optimize its usage

\* Learn how to build asynchronous handlers, pages, and tasks

# Managing the Worker Process

In Chapter 1 you learned about the worker process, w3wp.exe, which executes a request and initiates the ASP.NET engine processing. Managing the worker process is the key to managing high server demand, and keeping your applications stable and reliable.

## Managing Worker Processes and AppDomains in IIS7

One of the leading Windows application management tools is Windows Management Instrumentation (WMI). You have probably used WMI to administer basic ASP.NET-related parts of IIS. It’s a well-known way to access worker processes while your application is running.

Good Old Days: Windows Management Instrumentation

WMI stands for Windows Management Instrumentation. If you’re unfamiliar with it, there’s a good introduction in Wikipedia at:

\* http://en.wikipedia.org/wiki/Windows\_Management\_Instrumentation

It suggests that WMI is a good tool for automating the management of worker processes and application domains in IIS7. It can also help us find a better way to customize the ASP.NET environment. IIS7 worker processes are spawned by the Windows Process Activation Service (WAS), using w3wp.exe. A worker process can contain AppDomains that are typically created to handle a request.

To accommodate today’s managed-code world, Microsoft exposed the WMI and COM method calls via .NET. Instead of invoking WMI directly, you can manage IIS using .NET typed classes, rather than using untyped strings. These classes expose a direct interface to the IIS7 management level and appear to be easy to use and robust.

### Prerequisites

There are several prerequisites to writing sophisticated management applications using Visual Studio. First of all, IIS7 itself is a prerequisite. No other web server currently supports these interfaces. To test your application it must run with elevated privileges. If you run the development environment on Windows Vista, you must launch Visual Studio with Administrator rights. An application built on such a system runs well on Windows Server 2008, as long as it runs with Administrator privileges. Once Visual Studio is running with the right account, you need to reference the administration assemblies in your project. There are several managed assemblies available in the IIS7 installation folder:

<%WinDir%>\system32\inetsrv

You will find several assemblies starting with the name “Microsoft.Web”. Depending on what you’re planning to do, you’ll need one or another. For the moment, just reference all of them to ensure that all the examples in this chapter run properly. Here’s the list of what you’ll need:

\* Microsoft.Web.Administration

\* Microsoft.Web.Management.Aspnet

\* Microsoft.Web.Management.AspnetClient

\* Microsoft.Web.Management

\* Microsoft.Web.Management.Iis

\* Microsoft.Web.Management.IisClient

### Tasks to Manage

There are several tasks you can manage using the techniques described in this chapter. They will allow you to extend the available toolkit and customize the management of the ASP.NET environment. I will focus here on some of the more common tasks, but there is almost unlimited scope for extending the new management classes:

\* View the currently executing requests for a worker process

\* Get the state of all worker processes

\* Unload a specific AppDomain, or all AppDomains

\* Display all AppDomains and their properties

### Backup your Configuration

It is possible that the following samples will destroy or damage your IIS7 configuration. As it’s cumbersome to re-install your development environment after any undesirable side effects, you should test in a virtual machine or backup the IIS configuration of your production machine before you start.

To make a backup of the configuration:

\* Open a command window with Administrator rights

\* Navigate to %WinDir%\system32\inetsrv

\* Enter following command:

AppCmd add backup IIS7backup

\* The backup is created in the folder %WinDir%\system32\inetsrv\IIS7backup

You can use any name you like for the backup.

Figure 2-1. Backup command and created Backup folder with IIS7 settings

The AppCmd tool has several options. The basic structure of the command is:

AppCmd Command ObjectType ID Parameter

The *command* parameter depends on the *ObjectType* parameter. For backup purposes the *ObjectType* is Backup. You can use the following commands:

\* list – shows all available backups

\* add – creates a new backup for the current configuration

\* delete – delete a backup from disk

\* restore – restore a backup and overwrite current configuration

The usage is straightforward. Again, remember to run this tool with Administrator privileges to access the IIS settings.

### Basic tasks

The code in Listing 2-1 shows the typical structure of an IIS7 management code snippet. You can use whatever client environment you prefer. You could use a simple WPF (Windows Presentation Foundation) client to display the data in a hierarchical format (such as a TreeView), or perhaps well-formatted in a RichTextBox control.

For the purpose of demonstrating sample code, a console application is a simple, satisfactory alternative. I will use the following generic test structure for the examples:

Listing 2-1. Basic code structure for IIS7 administration code snippets

using System;

using System.Text;

using Microsoft.Web.Administration;

namespace Apress.AspNetExtensiblity.IIS7Console

{

static class IIS7Management

{

internal static string Method()

{

StringBuilder sb = new StringBuilder();

// Code goes here

return sb.ToString();

}

}

}

These static methods are invoked from the console’s entry point. The basis for most operations is the ServerManager class. From the ServerManager class there are several collections available:

\* Applications

\* Sites

\* WorkerProcesses

\* Bindings

\* VirtualDirectories

Through these collections you can reach the desired objects and properties for monitoring and managing IIS7. You always have read and write access. To save your changes, you need to explicitly call the CommitChanges method.

### Get Information about the Worker Processes

Once I’ve shown you how to loop through each worker process on a Web server, you’ll see how to display the currently executing requests, process ID, and state of each worker process, as well as the application pool to which it belongs.

#### Get the State of a Worker Process

The WorkerProcess object in the IIS7 administration has a GetState method that indicates whether a worker process is starting, running, or stopping. WorkerProcess also has two properties that particularly interest us: ApplicationPool and ProcessId. The ApplicationPool property represents the application pool to which the worker process belongs. The ProcessId property contains the process ID that uniquely identifies the worker process.

Listing 2-2. Getting information about worker processes

internal static string ShowWorkerProcesses()

{

StringBuilder sb = new StringBuilder();

try

{

ServerManager manager = new ServerManager();

foreach (WorkerProcess proc in manager.WorkerProcesses)

{

sb.AppendFormat("WorkerProcess found: {0}\n", proc.ProcessId);

sb.AppendFormat("\t|--AppPool : {0}\n", proc.AppPoolName);

sb.AppendFormat("\t|--ProcGuid: {0}\n", proc.ProcessGuid);

sb.AppendFormat("\t|--State : {0}\n", proc.State.ToString());

foreach (ApplicationDomain appDom in proc.ApplicationDomains)

{

sb.AppendFormat(

"\t+--ApplicationDomain Found: {0}\n", appDom.Id);

sb.AppendFormat(

"\t\t|--AppDomPhysPath: {0}\n", appDom.PhysicalPath);

sb.AppendFormat(

"\t\t+--AppDomVirtPath: {0}\n", appDom.VirtualPath);

}

}

return sb.ToString();

}

catch (Exception ex)

{

return ex.Message;

}

}

The worker processes are exposed by the property WorkerProcesses. The value should match the number of w3wp.exe instances running in the Windows Task Manager.

Tip: If there are no worker processes but IIS7 is running, open a website to force the process to launch.

The properties used depend on the object you wish to investigate. Refer to the documentation on MSDN to get full descriptions of all available values.

Insert ASPEXTf0203.tif

Figure 2-2. Watching worker processes

To monitor your worker processes, check the State property. The state of a process can be one of:

\* Starting

\* Running

\* Stopping

\* Unknown

Why can’t the worker process have a state of “Stopped”? After the process has been shut down, the executable is disposed of and removed from memory. The worker process then disappears from the list of processes. Therefore, the worker is first starting, then running, and, if there is nothing to do, stopping. There isn’t a specific situation where the state Unknown appears, as the name implies. I assume that it occurs when the process dies unexpectedly and hangs.

#### Get Executing Requests

One exciting new feature of IIS7 is the ability to see the requests which are currently executing in a worker process. You do this with the WorkerProcess.GetRequests method in a manner very similar to the one shown above. Retrieve the worker process and invoke the GetRequests method to get the current requests.

The GetRequests method requires an int parameter to filter the results for requests which have run for at least the number of milliseconds specified. This is very useful for displaying only long running requests. It’s a good idea to set the value to zero initially in order to get all requests, as shown in Listing 2-4.

Tip: Capturing a request to test an application within the development environment can be tricky. Use the Thread.Sleep method to make the request last long enough to be caught.

Use Windows Notepad to save the following text into a file called Sleep.aspx. Listing 2-3 shows the complete page for a request that will take a minute to execute.

Listing 2-3. Define a page running one minute to get a long-lasting request

<% System.Threading.Thread.Sleep(60000)

Response.Write ("I'm finally finished...") %>

Running the management code, as in Listing 2-4, shows that the request is running and displays various properties for the request.

Listing 2-4. Looping through some properties of a running worker process

internal static string ShowRequest()

{

StringBuilder sb = new StringBuilder();

try

{

ServerManager manager = new ServerManager();

foreach (WorkerProcess proc in manager.WorkerProcesses)

{

foreach (Request r in proc.GetRequests(0))

{

sb.AppendFormat("Request:\n");

sb.AppendFormat(" Hostname = {0}\n", r.HostName);

sb.AppendFormat(" Url = {0}\n", r.Url);

sb.AppendFormat(" Verb = {0}\n", r.Verb);

sb.AppendFormat(" IP = {0}\n", r.ClientIPAddr);

}

}

return sb.ToString();

}

catch (Exception ex)

{

return ex.Message;

}

}

The console (see Figure 2-3) shows all the requests currently running on available worker processes. Before running this on a production system, it would be advisable to add some filter conditions to avoid returning an excessive number of results.

Insert ASPEXTf0202.tif

Figure 2-3. Output shows detailed information about certain requests

Monitoring requests is not particularly suited to a console application. Imagine you have some requests that don’t function as expected, but the errors are infrequent and not reproducible. In this situation, you could write an application to monitor a specific request and log all relevant data. When the error reappears, your log file should contain information enabling you to track down the error source.

### Get Information about the Application Domains

The first time a request for an ASP.NET page is received, the IIS7 managed engine module creates an application domain (AppDomain) in memory. I explained this in Chapter 1. The AppDomain processes requests for aspx pages, or any page that uses managed code. Unloading and enumerating AppDomains is straightforward. This section shows you how to do both.

#### Unloading a Specific AppDomain

To unload a specific AppDomain, you must be able to uniquely identify it. AppDomains have three key properties: Id, VirtualPath and PhysicalPath. One of these properties should be sufficient.

The code snippet in Listing 2-5 shows you how to obtain the desired AppDomain using a LINQ query. The nested from statements form a join to retrieve the ApplicationDomains property for each running WorkerProcess object. The where clause restricts the result to the single AppDomain defined by the given Id. For a real-life application, this value could be a variable. We saw the ShowWorkerProcesses method earlier in Listing 2-2. It’s not required, but I’ve included it here for demonstration purposes.

Listing 2-5. Retrieve information about the application domains

internal static string UnloadAppDomain(string name)

{

StringBuilder sb = new StringBuilder();

try

{

sb.Append(ShowWorkerProcesses());

ServerManager manager = new ServerManager();

var appDomains = from proc in manager.WorkerProcesses

from adc in proc.ApplicationDomains

where adc.Id == "/LM/W3SVC/4/ROOT"

select adc;

ApplicationDomain ad = appDomains.FirstOrDefault<ApplicationDomain>();

if (ad != null)

{

ad.Unload();

return name + " unloaded";

}

else

{

return "can't find " + name;

}

}

catch (Exception ex)

{

return ex.Message;

}

}

Incidentally, the AppDomain’s Id property is a path that looks like this:

/LM/W3SVC/1/ROOT

The “1” in the path listed is the site’s ID. “1” usually corresponds to the default Web site. You could find any number here, depending on how many webs you have on the machine you’re currently investigating.

If you generate a list of your server’s AppDomains and their properties first, you’ll find the right one easily (see Listing 2-2 where we extracted the AppDomain instances from the currently running worker processes).

However, the purpose of the script was to unload a particular AppDomain. To understand what happens when you unload an AppDomain, run this scenario:

\* Request a page from your server

\* Watch running processes and AppDomains and retrieve the properties

\* Launch the code and unload the domain

\* Request the page again

\* Retrieve the very same information from properties

#### Unloading all AppDomains

The next example unloads all AppDomains. Again, I use LINQ and generic features. You don’t have to do it this way, but it results in compact and highly readable code. The LINQ statement again uses a nested query.

Listing 2-6. Unload all AppDomains

internal static string UnloadAppDomains()

{

try

{

ServerManager manager = new ServerManager();

Func<ApplicationDomain, bool> unloadFunc = ⮰  
 new Func<ApplicationDomain, bool>(Unload);

var appDomains = from proc in manager.WorkerProcesses

from adc in proc.ApplicationDomains

where unloadFunc(adc) == true

select adc;

return "Unloaded " + appDomains.Count() + " domain(s)";

}

catch (Exception ex)

{

return ex.Message;

}

}

private static bool Unload(ApplicationDomain appDomain)

{

try

{

appDomain.Unload();

return true;

}

catch

{

return false;

}

}

The unload method is encapsulated in the callback function Unload. It takes the AppDomain as a parameter of type ApplicationDomain. As before, the query starts by retrieving the worker processes and the AppDomains joined to them. In the where clause, the inline method unloadFunc() is used to unload the domain. The Func type is a predefined typed delegate used to pass a parameter and retrieve a Boolean result flag. The clause forces the method call to Unload and the return value decides whether the AppDomain becomes part of the result set. This leads to a direct usage of the Count method that returns the number of successfully unloaded AppDomains.

Insert ASPEXTf0204.tif

Figure 2-4. Number of successfully unloaded AppDomains

#### Enumerating AppDomains

Dealing with AppDomains in a management environment often starts with showing all available properties. For learning purposes it makes sense to investigate the AppDomain’s properties. Listing 2-7 shows how to get this information.

Listing 2-7. Show properties of AppDomains

internal static string ShowAppDomains()

{

StringBuilder sb = new StringBuilder();

try

{

ServerManager manager = new ServerManager();

var appDomains = from proc in manager.WorkerProcesses

from adc in proc.ApplicationDomains

select

String.Format(@"Physical Path = {0}

{4}Virtual Path = {1}

{4}Process ID = {2}

{4}Is Idle = {3}",

adc.PhysicalPath,

adc.VirtualPath,

adc.Id,

adc.Idle,

Environment.NewLine);

if (appDomains.Count() == 0)

{

sb.Append("can't find AppDomains");

}

else

{

foreach (string ad in appDomains)

{

sb.AppendLine(ad);

}

}

return sb.ToString();

}

catch (Exception ex)

{

return ex.Message;

}

}

The LINQ query retrieves the worker processes and the attached AppDomains. In the select clause, the string is constructed to simplify output. You can alter this clause to retrieve the specific values you desire. With this, you can monitor your ASP.NET environment with your custom application, without requiring Internet Information Services Manager Console.

Figure 2-5. Information about a running AppDomain

You can compare this information with that available in the IIS Manager Console. Open the IIS Manager Console, navigate to the site and click on *Extended Settings* to see the settings and check against the values retrieved using the code shown above.

When working with the ApplicationDomain object, you might see many more properties than were shown in the last example. This is because all settings derive from the ConfigurationSettings type, which contains several common properties that we don’t need. The helpful properties for an AppDomain are:

\* Id

\* VirtualPath

\* PhysicalPath

\* Idle

Id returns the current internal Id of the AppDomain. The VirtualPath is as defined in the IIS settings; for the root path it is usually “/”. The PhysicalPath is the full path to the application on local disk. The Idle property is a runtime value. It’s defined as int; it returns either 0 or 1, where the latter is the value for an idle application domain. Additional values about the worker process are helpful, too. You can add these easily by extending the select clause in this manner:

adc.WorkerProcess.ProcessId

The object adc is of the current ApplicationDomain type. By using a back reference to the WorkerProcess object, all property values available there can be retrieved directly.

### Configuring the Worker Process

Unlike classic ASP, which runs in the same memory space as IIS, ASP.NET runs as a process of its own. This gives it more flexibility, stability, and power. You can use the configuration file *machine.config* to make the Webmaster’s job a lot easier.

I’ll take a closer look here at the ASP.NET process and the attributes that we can adjust. The *machine.config* file is plain XML and easy to read. As the name implies, it’s a definition at the machine level. Several portions define the default values for the application and the folder specific *web.config*. However, some parts can’t be overwritten, and there’s no point in attempting to alter others.

#### The machine.config file

As mentioned earlier, version 3.5 of the .NET framework augments, rather than replaces, .NET framework 2.0. This explains why the *machine.config* file is stored under the 2.0 hive. The path on your machine should look similar to:

%WinDir%\Microsoft.NET\Framework\v2.0.50727\CONFIG

There are several other configuration files, but we’re only examining *machine.config*. Because it’s XML, you can use Visual Studio to edit it. The worker process settings are stored in the <processModel> tag. It’s located under <system.web> and usually looks like this:

<processModel autoConfig="true" />

Microsoft sets all common values using the auto configuration option. This is good for most installations, but not all. To optimize the settings, you can set the value of this attribute to false and change whatever you like:

<processModel autoConfig="false" />

However, changing basic settings is not simple and may lead to servers not responding or performing poorly. Before going on, let me explain the various settings.

Note: Changes made to the <processModel> tag do not have immediate effect. You must restart a worker process to force it to re-read the settings. To restart a worker process, you can open Task Manager, right click the w3wp.exe found in the task list, and kill the process. The worker process will start again automatically.

#### Why Customize the machine.config Settings?

You may wonder why and when you’re supposed to change the *machine.config* settings and the <processModel> settings. Let’s consider the following scenario:

\* Your network supports only one application domain (AppDomain)

\* Each page causes one request (no subsequent AJAX calls)

\* All requests go to the same IP address

In this case, the default settings are very good and there is no reason to change anything. However, other specific scenarios could be problematic:

\* Requests to many IP addresses

\* Frequent redirects (HTTP status code 302)

\* Using authentication

\* Using more than one AppDomain

The following section describes all the processModel parameters. (A later section, *Configuring the Thread Pool*, explains this in even greater depth.)

Keep in mind that, behind the scenes, IIS must handle the available resources to manage all incoming requests. If the rate of incoming page requests is such that the fraction of memory in use or CPU power exceeds certain levels, the default values for the processModel parameters may not be optimal. However, setting higher values is not always better as this can lead to higher resource consumption and slow down even a lightly-loaded server handling simple requests. Finding the most appropriate options in the parameter jungle is a challenge.

#### A First Look Into the Attributes

Table 2-1 lists the processModel attributes available. The description column indicates the purpose of each attribute and where to find more information.

Table 2-1. Settings available for the <processModel> tag (excerpt)

|  |  |  |
| --- | --- | --- |
| Attribute | Default | Description |
| clientConnectedCheck | 00:00:05 (five seconds) | Once a request has been waiting in the queue for this long, ASP.NET checks whether the client is still connected. |
| cpuMask | 0xffffffff | Specifies which CPU of a multiprocessor system runs ASP.NET processes. The value is a bitmask where each bit represents a CPU. Assume you set the value to 0xa, which is 1010 in binary form. CPUs are numbered beginning with 0, and read from right to left. For this value, CPUs 1 and 3 are qualified to run ASP.NET threads and CPUs 0 and 2 are not. |
| enable | true | Enables or disables the process model. |
| idleTime | Infinite | The period of inactivity (in time format hh:mm:ss), after which the worker process ends. (For example, a value of “00:20:00” will cause the worker process to shut down twenty minutes after the last request is concluded.) “Infinite” prevents the worker process from stopping at all. |
| logLevel | Errors | The quantity of errors written to the event log. Choose from “All”, “Errors”, or “None”. |
| maxAppDomains | 2000 | The number of application domains in one process. The default value, 2000, is the maximum allowed. Lower values are appropriate for hosting providers, for instance. |
| maxIoThreads | 20 | The maximum number of threads for I/O operations, counted on a per-CPU basis. Must be greater than or equal to minFreeThreads in <httpRuntime> settings. Allowed range is between 5 and 100. See the section “Understanding and using threads” in this chapter for more information. |
| maxWorkerThreads | 20 | The maximum number of worker threads, counted on a per-CPU basis. Must be greater than or equal to minFreeThreads in <httpRuntime> settings. Allowed range is between 5 and 100. See section “Understanding and using threads” in this chapter for more information. |
| memoryLimit | 60 | The maximum allowed memory size as a percentage of total system memory that one worker process is allowed to consume. Beyond this limit, a new worker process is launched and subsequent requests are redirected to this one. |
| minIoThreads | 1 | The minimum number of I/O threads. See section “Understanding and using threads” in this chapter for more information. |
| minWorkerThreads | 1 | The minimum number of worker threads. See section “Understanding and using threads” in this chapter for more information. |
| username, password | AutoGenerate | The account used to run the worker process. |
| pingFrequency | Infinite | The time interval in hh:mm:ss format used to ping the worker process to get its state. If the worker process is not running it is restarted. |
| pingTimeout | Infinite | The time, also in hh:mm:ss format, waiting for a response to a ping request. After a ping timeout is detected, the process is restarted. |
| requestLimit | Infinite | The number of requests a single worker process can handle. Beyond this value a new worker process starts. |
| requestQueueLimit | 5000 | The number of requests allowed in the queue. Any request beyond this limit receives the HTTP error 503 “Server Too Busy” response. |
| responseDeadlockInterval | 00:03:00 | The time period (again in hh:mm:ss format) allowed for the worker process to respond to a queued request. After the time period expires, the worker process is restarted. The default is three minutes. |
| restartQueueLimit | 10 | After a nonstandard (unexpected) termination of the worker process, incoming requests are queued, waiting for the worker process to become available. The value specifies the number of requests queued. |
| serverErrorMessageFile | - | A file path (either absolute or relative to the *machine.config* file path) for a file containing the error message to send to the client when a fatal error occurs. If no file is present or the attribute is not set, the string “Server unavailable” is sent. This is the default setting. |
| shutdownTimeout | 00:00:05 | The time the worker process is allowed to take when shutting the process down. If this time interval is exceeded, the worker process is forcibly terminated. |
| timeout | Infinite | If a worker process is not responding, ASP.NET launches a new one after the specific period is elapsed. |
| webGarden | False | A flag which indicates that the attribute cpuMask is being used. If False (default) this means that all CPU cores are available for worker processes and the operating system decides which CPU will handle the next request. |

#### Additional Settings Outside the ProcessModel Tag

The <processModel> tag attributes control most of the relevant settings. However, there are more tags where we can refine the configuration. One important setting is shown below:

<system.net>

<connectionManagement>

<add address="\*" maxconnection="100" />

</connectionManagement>

</system.net>

This does not impact your client connections but controls the connections you make from your application to other servers, such as fetching RSS feeds or using web services on another server. The default for the attribute maxconnection is 2, which is obviously too low. It means you cannot make more than two simultaneous connections to an IP address, from your web application. In the sample above the value is set to 100.

### Specific Tasks

Using these settings, you can perform specific tasks. The following sections explain how to use the attributes to:

\* Recycle the worker process

\* Shut down the worker process

\* Check if the client is still connected

#### Recycling the Worker Process

A common task configured in this manner is the recycling of the process. This generally improves the stability and reliability of your web application. There are five ways to recycle a process using the attributes timeout, requestLimit, memoryLimit, responseDeadlockInterval, and the pair pingFrequency/pingTimeout. They are explained one by one below:

\* timeout="48:00:00"

This involves the timeout attribute, which simply creates a new process after the specified time interval elapses. For example, the above setting will start a new process after 48 hours, or two days. The first request causes the timer to start.

\* requestLimit="1000"

Another way is to use the requestLimit attribute and an Integer value. A value of 1,000 will start a new process after 1,000 requests have been made. This can be useful if your web server’s performance degrades after a set number of requests.

\* memoryLimit="30"

A third way is to let your system watch how much memory the process is consuming. In the above example, the attribute memoryLimit is set to 30% of the total system memory. Once the limit is reached, the process is killed, a new one is created, and all existing requests are reassigned to the new process. This is helpful when you have a memory leak.

\* responseDeadlockInterval="00:02:00"

A fourth approach is to use the responseDeadlockInterval attribute. The time setting of two minutes will restart the process if there are requests in the queue and if there have been no responses for the last two minutes.

\* pingFrequency="00:00:30"

\* pingTimeout="00:00:05"

The final method for recycling the process is to use the pingFrequency and pingTimeout attributes in tandem. The system pings the process at the pingFrequency interval, and restarts it if there is no response within the pingTimeout time interval.

#### Shutting Down the Process

There are two ways of shutting down the worker process.

\* idleTimeout="00:20:00"

The first way uses the idleTimeout attribute. If the server has not served any requests for 20 minutes (in this example), then it automatically shuts down the process. If a new request comes in, a new process is started automatically. This might be useful if you experience long periods of inactivity. The server can then handle other tasks better because there is more free memory.

\* shutDownTimeout="00:00:10"

The second way is by using the shutDownTimeout attribute. This is used as a last resort after the worker process tries to shut down gracefully and fails. In this case, after the time set here has passed, a low level kill command is performed on the process to ensure its termination. This is useful in situations where the process has crashed and is no longer responding. The setting shown will force a kill after ten seconds.

#### Checking if the client is still connected

This is useful for eliminating unwanted requests from the queue.

\* clientConnectedCheck="00:00:10"

Users can become impatient. If your web server is slow to respond to their requests, they might click on the same link many times. Even if only the last request is returned to them, the server will process all the previous ones. Furthermore, if the user abandons their session with your server, the queue from that user will remain. In the example above, the server will check each request at ten seconds intervals after it has entered the queue, to ensure that the user who made it is still connected. If the user is not, the server discards that request.

# Understanding and Using Threads

In ASP.NET, threads are like magic.

Threads are well-known and easy to program using .NET techniques. However, in ASP.NET, threading is handled behind the scenes and consequently regarded by some developers as unimportant. This common misunderstanding of threading can cause poor performance, unpredictable behavior, and, in the end, unreliable applications.

## ASP.NET Thread Usage on IIS

Threading in ASP.NET is complicated by the various changes Microsoft applied over the years with each new version of IIS. As we saw in Chapter 1, IIS plays a major part in ASP.NET’s request processing. Because of the co-existence of IIS6 and IIS7, I will explain firstly how threads are handled in IIS6, and then the changes in threading introduced with IIS7.

You may also think of other hosting capabilities ASP.NET supports. Because threading has a direct and explicit relation to multiple requests and high workload, other environments, such as Visual Studio’s integrated web server, do not handle threads in any special way worth discussing.

### Thread Usage with IIS6

Each request from the outside world to the web server is handed over to an I/O thread on IIS. This thread comes from the CLR (Common Language Runtime) ThreadPool and returns the status “Pending” to IIS. Having passed the work on to the new thread, IIS is now free to service other requests, such as requests for static resources.

The CLR ThreadPool works like a queue. It can adjust itself according to the actual workload. This means that the situation depends on the frequency of incoming requests and the ability of the processor workload to respond to each request. There are two extreme situations you need to take care of.

Firstly, you might receive many simultaneous requests that are processed quickly. In this case, the ThreadPool will attempt to run only 1 or 2 threads per CPU to ensure a very low latency (waiting time).

Secondly, you might receive a few requests where one is processor intensive and somewhat long-running. Additional incoming requests will cause more threads to be spawned per CPU. The processing time will always be longer in this scenario.

A queue is a clever way to avoid allocating a lot of memory for each request, before processing starts and memory expensive objects such as HttpRequest are created. Keep in mind that the thread queue is in native memory and has no overhead caused by managed code components. Once a thread is ready for processing, we leave the unmanaged world and start working completely within the managed code realm.

The ThreadPool queue is not the only way to handle a lot of incoming requests. Within each AppDomain there are ways to handle requests that exceed the number of available threads. If there is lot of latency, the ThreadPool starts growing and launches more active threads. However, there are physical limitations; either the system runs out of threads, or the available memory restricts the number of threads.

Note: By default the ThreadPool has a limitation of 25 worker threads per CPU and 1000 I/O completion threads.

ASP.NET sets its own limits to ThreadPool usage. If this limit is exceeded, incoming requests are still handled but another queue is now built on the application level and performance becomes significantly worse. You can control the settings with the following parameters of the <httpRuntime> tag:

\* minFreeThreads

\* minLocalRequestFreeThreads

Performance Counters are widely used in Windows to monitor a system running with high workload. To observe the internal behavior of the ThreadPool you can use the following counter: “ASP.NET Applications\Requests in Application Queue”.

Any value but zero shows that there is a performance problem on that system, as it indicates that the ThreadPool has at some time run out of threads.

#### Managing the ThreadPool

Usually the ThreadPool does not require managing, because few sites exceed the default limitations. Additionally, the autoConfig settings allow the ASP.NET engine to optimize behavior as much as possible. However, maintaining an ASP.NET system that is running near its limits is the goal of this book. If you experience performance issues, it’s time to explore manual intervention. The auto settings assume that the number of concurrently executing requests per CPU is twelve. An application with high latency might require higher values.

### Thread Usage with IIS7

In chapter 1, I discussed IIS7 and integrated mode several times. As mentioned earlier, the differences between IIS7 and IIS6 are significant for basic page request processing.

First of all, the queues built on the application level are gone, due to poor performance. The biggest difference is that IIS6 restricts the number of threads, while IIS7 restricts the number of requests. Each thread in IIS6 handles a request and this—indirectly—limits the number of requests handled in parallel. IIS7 is able, due to the tight integration of ASP.NET within IIS7, to directly restrict the number of requests. It is when you employ asynchronous processing that this makes a difference. Although the processing pipeline is usually synchronous (see Chapter 1), both, Handlers and Modules, can be set up to run asynchronously. I’ll explain this, with examples, later in this chapter.

For requests that are processed synchronously, the number of threads equals the number of requests, as each request runs in a single thread. If the processing is asynchronous, the number of threads may differ from the number of requests. Imagine that an incoming request starts processing and is running a long lasting action. The thread is handed over to ASP.NET and IIS accepts the next incoming request. This leads to more concurrently running requests than threads. ASP.NET gets the request as incoming IIS I/O thread. The CLR ThreadPool is immediately asked to create and start a new thread and this thread becomes responsible for this very request. As quickly as it starts, it returns with status of “Pending”. After this, IIS checks the number of requests currently executing. If this value is too high, the next request is put in a process-wide global queue. This queue is in native code within IIS7.

#### Configuring the ThreadPool

In the <processModel> configuration settings are the following parameters:

\* autoConfig

\* maxWorkerThreads

\* maxIoThreads

\* minWorkerThreads

\* minIoThreads

The settings for IIS6 minFreeThreads and minFreeThreads in the httpRuntime tag are still there, but they do nothing. They are there merely for backwards compatibility. But where can you find the settings for IIS7? With .NET 3.5 SP1 the settings become available[[1]](#footnote-1) in web.config, as shown below:

<system.web>

<applicationPool maxConcurrentRequestsPerCPU="12"

maxConcurrentThreadsPerCPU="0"

requestQueueLimit="5000"/>

</system.web>

The *web.config* settings override the settings in the Registry mentioned in the footnote.

They are not values that you would change frequently. You might experiment with different quantities, but the default is usually appropriate. The current settings are a compromise between requests for static resources such as images, and dynamic resources such as aspx pages.

In chapter 6, I’ll explain the extensibility concepts of resource management. This includes the creation of design time expressions to retrieve resources. It’s possible to suppress code compilation when your aspx pages consist solely of declarative markup, resources, and expressions. This is a way to optimize overall system performance under particular circumstances.

Now, you could try setting the number of threads per request to 0. If you have only static requests and no (or very few) dynamic ones, and if these dynamic requests are fast due to such techniques, this might improve performance. Imagine that the requests are now being executed within the IIS I/O thread and that there is no handover procedure to the CLR ThreadPool. Less overhead leads to faster resource processing.

The opposite scenario appears when you have many heavily asynchronous operations. In this case, the limit of twelve threads might be too low. Imagine a scenario you’ll probably work with: Ajax enabled applications. These applications have fewer page requests but many background tasks created by JavaScript/web service pairs. This is heavily asynchronous and a single webpage can send large numbers of such requests. If you look into browser-based Outlook clients or Word-like text processing interfaces, you’ll see that a lot of background work happens even when the user merely moves the mouse pointer. Settings of up to several thousand are possible in order to handle many simple requests. 5000 is the limit, without making changes elsewhere in the system.

#### Server Too Busy

You might be wondering when and why the “Server Too Busy” status is sent to the client. This is HTTP status error 503 and appears when the limit of concurrent requests exceeds the default value of 5000. The actual value is exposed by the “ASP.NET/Requests Current” performance counter.

See the section “Install a Performance Counter” for details on how to monitor and log such values and discover what is happening to your server at any given moment.

## Tune the threading

The previous explanations might discourage you from changing the settings. However, they’re worth exploring when performance issues arise or strange errors occur. This section shows some solutions for the following error messages:

\* A process serving application pool ‘name’ exceeds time limits during shut down

\* System.InvalidOperationException: There are not enough free threads in the ThreadPool object to complete the operation.

\* HttpException (0x80004005): Request timed out.

To solve the problems above, try the following:

\* Limit the number of requests that can execute at the same time to approximately twelve per CPU. This limit works well for most applications.

\* Permit web service callbacks to freely use threads in the ThreadPool.

\* Select an appropriate value for the maxconnections attribute. Base your selection on the number of IP addresses and AppDomains that are used (more details follow).

### Tuning Task by Task

This section gives some examples of how and when to change the default settings of the following values:

\* maxWorkerThreads

\* minWorkerThreads

\* maxIoThreads

\* minFreeThreads

\* minLocalRequestFreeThreads

\* maxconnection

\* executionTimeout

#### Set maxWorkerThreads and maxIoThreads

ASP.NET uses the following two configuration settings to limit the number of worker threads and completion threads used:

<processModel maxWorkerThreads="20" maxIoThreads="20">

The maxWorkerThreads attribute and the maxIoThreads attribute are implicitly multiplied by the number of CPUs. In the example, the maximum number of worker threads is 40 if you have a dual-core processor.

#### Set minFreeThreads and minLocalRequestFreeThreads

ASP.NET also contains the following configuration settings which determine how many worker threads and completion port threads must be available to start a remote request or a local request:

<httpRuntime minFreeThreads="8" minLocalRequestFreeThreads="8">

If there are insufficient threads available, the request is queued until sufficient threads are free to handle the request. Therefore, ASP.NET will not execute more than the following number of requests at the same time:

(maxWorkerThreads \* numCPUs ) - minFreeThreads

The minFreeThreads parameter and the minLocalRequestFreeThreads attributes are not implicitly multiplied by the number of CPUs. Assuming you have four CPUs, the formula given equals 24 parallel requests ((8 x 4) – 8).

#### Set minWorkerThreads

ASP.NET also contains a configuration setting for the number of worker threads to be made available immediately to service a remote request.

<processModel minWorkerThreads="1">

Threads that are controlled by this setting can be created at a much faster rate than worker threads that are created from the CLR’s default capabilities.

Requests may suddenly fill the request queue due to a slow-down on a back end server, a sudden burst of requests from the client end, or something similar that would cause a sudden rise in the number of requests in the queue.

The default value for the minWorkerThreads parameter is 1. Microsoft recommends that you set the value for the minWorkerThreads parameter to half of the maxWorkerThreads value. By default, the minWorkerThreads parameter is implicitly multiplied by the number of CPUs.

#### Set maxconnection

The maxconnection attribute determines how many connections can be made to a specific IP address. The setting was mentioned before, but here is a more complex scenario:

<connectionManagement>

<add address="\*" maxconnection="2">

<add address="10.6.205.84" maxconnection="20">

</connectionManagement>

The maxconnection setting applies at the AppDomain level. Consequently, only two connections (by default) can be made to a specific IP address from each AppDomain in your process.

In this example, all IP addresses accept two connections, except for the specified one at 10.6.205.84, which accepts 20.

#### Set executionTimeout

ASP.NET uses the following configuration setting to limit the request execution time (in seconds):

<httpRuntime executionTimeout="90"/>

This refers to the Server.ScriptTimeout property exposed by the HttpServer object. If you increase the value of the executionTimeout attribute, you may have to also modify the responseDeadlockInterval attribute of the <processModel> tag.

### Install a Performance Counter

To monitor the current workload, you can use a Performance Counter. Open the *Reliability and Performance Monitor* tool (PerfMon) by entering *PerfMon* in the *Run* window. Install the counter using these steps:

1. Click on “Performance Monitor” in the tree view.

2. Right click and choose “New” and then “Data Collector Set”.

Figure 2-6. Install new data collection using the Performance Monitor tool

3. Give the collection an appropriate name (such as “MonitorIIS”) and click “Next”.

4. Choose a directory in which the data is to be saved. The default path is %systemdrive%\PerfLogs\Admin\MonitorIIS, if you have named the collection “MonitorIIS”. Click “Next”.

5. In the last step, you can choose an account with which to run the counter. This is, by default, the current account. If you’re logged on as Administrator, it’s fine. Otherwise, select an appropriate account by clicking on “Change”.

6. Finish the wizard with the “Finish” button.

You can now open your collection within the object’s tree view at the left and navigate the path to Reliability and Performance > Data Collector Sets > User Defined > MonitorIIS. There is a default entry here named “System Monitor Log”. You can modify this or add a new data collector. Let’s add a new one to demonstrate the process. Right click on the leaf entry in the tree named MonitorIIS and choose New > Data Collector. A wizard launches and asks for a name and collection type. Name it “IISRequest” and choose Performance counter data collector as the type.

Figure 2-7. Add a new Data Collector

Click on “Next” to add the counter. On the wizard’s next screen, choose “Add…” and search for the counter. In the left hand area you’ll find the Available counters group. Search for the section “ASP.NET Application” and expand it. The list is sorted alphabetically. Scroll down to “Requests In Application Queue”. In the “Instances of selected object” group, you’ll find all applications. Accept the default selection \_\_Total\_\_ to view all requests. Click “Add>>” and “OK”. The monitor begins to watch the counter and saves the results in the file chosen.

To view the results, open the path Reliability and Performance > Reports > User Defined > MonitorIIS > IISRequest. It might take some time to load the results, depending on workload and settings. There are several ways to modify the counter to bind it to specific events. For example, you could add a scheduler to run the counter at a specific time. For instance, if you experience problems between 1 and 2 a.m. on your server, it doesn’t make sense to run the tool the whole day and collect an unnecessarily large amount of data.

You can define stop conditions based on time frame or the size of the log file. You can also launch a task when the scheduler stops. Depending on the resources of the server you could run an application that automatically analyses the content, sending the log file to an email address, or just copying it to another location.

This short description does not cover the Reliability and Performance Monitor tool in all its glory. It’s just a “teaser” so that you know there are powerful tools available to monitor a server and find bottlenecks, failures and performance loss easily. You can also gather hard data to confirm a problem, find a solution, and validate that your solution fixes the problem.

## Threading and Asynchronous Operations

For most scenarios, the internal thread handling is well-designed and functional. However, there are situations where you reach the limits of the default settings. In particular, if a site comes under pressure from too many requests, the internal thread pool can run out of threads and the server will no longer respond as expected. I’ll discuss several techniques for overcoming this. I’ll consider how the common language runtime thread pool is used by ASP.NET to service requests, as well as looking into the pooling mechanisms used for handlers, modules, and applications.

I’ll also show the threading usage independent of other techniques and the asynchronous processing of requests using internal features to solve common issues.

### Threading in ASP.NET

To efficiently service multiple client requests, web servers make extensive use of concurrency by launching multiple processes and spawning multiple threads to service requests. Considering the construction and behavior of the ASP.NET engine, it seems that developers need not concern themselves with threading at all, as the challenging aspects are handled internally. This is correct for most scenarios; page requests are serviced on the same thread and a separate instance is created to service each new request. However, there are scenarios where you reach the limits of this model—as every model has limits—and need to extend the behavior.

First of all, a clear understanding of the internal behavior is required. Some parts have been explained in the previous sections but I will repeat it from the perspective of threading. The process-wide CLR thread pool services requests. The thread pool size is set to 25 worker threads and 25 I/O threads by default. Recall the <processModel> settings explained already:

<processModel enable="true"

maxWorkerThreads="25"

maxIoThreads="25" />

As explained in chapter 1, for each incoming request an instance of the type HttpApplication is created. To avoid reallocating applications and modules, each AppDomain holds a pool of applications and modules. The size of this pool is also 25—that means that 25 requests can be handled per worker process.

#### The Need for Asynchrony

Imagine that you need to exclusively request a resource. As long as one person is using an application this is not a problem. However, web applications are typically employed by many, if not thousands, of concurrent users. The thread pool and thread handling design allows several requests to be executed in parallel to improve the user’s experience when using a web application. Limiting the number of threads is important in order to allow more than one request per CPU or core. Creating a vast number of threads can cause a system to a grind to a halt.

However, a request can launch different kinds of tasks. Processing a page and sending resources to the client is the most common action. Requesting data from a database, RSS feed, or web service is another. Requesting data can be time consuming, rather than processor intensive. What happens then? The number of threads in the thread pool quickly reaches its limit and subsequent requests are not handled as expected. Despite this, the CPU is only idling. As you learned in the previous sections, the settings allow you to change the number of concurrent threads. But this is not a solution either, as your system has to handle both kinds of requests: short-lived requests for resources and long running queries against other servers.

To make things clearer, look at Listing 2-8. It delays a request while demanding little from the CPU. It displays the thread ID to show whether or not a new thread is created to handle the request.

Listing 2-8. A simple page which slows down a request (Threading/SlowThread.aspx)

<%@ Page Language="C#" %>

<%@ Import Namespace="System.Reflection" %>

<%@ Import Namespace="System.Threading" %>

<script runat="server">

protected void Page\_Load(object src, EventArgs e)

{

System.Threading.Thread.Sleep(3000);

Response.Output.Write("Slow Response, Thread ID={0}",

AppDomain.GetCurrentThreadId());

}

</script>

To test it, open the page in a browser. Open the page in more browser windows and refresh all the pages within the three second (3000 millisecond) period. You’ll see that the thread ID exposed by the script changes for each request. Once the period is over the next request receives a recycled thread from a previous request. Several long running requests can fill the thread pool and easily reach the limit of 25 concurrent threads.

Removing the Sleep call (Listing 2-9) will result in a faster running page. You won’t be able to request the page again while the server is handling the previous one. The thread IDs demonstrate that all requests run on the same thread, and the thread pool is never filled up to its limit.

Listing 2-9. A simple page which performs well (Threading/FastThread.aspx)

<%@ Page Language="C#" %>

<%@ Import Namespace="System.Reflection" %>

<script runat="server">

protected void Page\_Load(object src, EventArgs e)

{

Response.Output.Write("Fast Response, Thread ID={0}",

AppDomain.GetCurrentThreadId());

}

</script>

Microsoft offers a stress test tool for simulating multiple concurrent requests even in the development environment. See the sidebar about the Web Stress Tool for more information.

Web Stress TOOL

The web application stress tool can be downloaded from:

\* http://www.microsoft.com/downloads/details.aspx?familyid=e2c0585a-062a-439e-a67d-75a89aa36495&displaylang=en

Unfortunately the tool is quite old and Microsoft has not refreshed it to support newer environments. So if you run it on Vista, a DLL named msvcp50.dll is missing. You may find the file on the web or you can safely copy it from the support files section provided with this book. Perform these steps to install the stress test tool:

\* Download the setup.exe from the address shown above

\* Download msvcp50.dll from the Apress support web site

\* Copy the unzipped file to folder %WinDir%\system (you must run with higher privileges to do this). Note that this is really system, not system32!

\* Install the stress test tool

\* Run it once to check that it’s working. Start it as an Administrator.

Now you can use the tool as outlined in the following sections.

#### Working with the Stress Test Tool

A stress test is what the name implies—your application is forced to handle as many requests as necessary to feel stressed. To walk through this stress test, you’ll need to set up the pages shown above (Listings 2-8 and 2-9) within your IIS environment, which you’re able to run from a browser manually. Follow the steps below to set up the script in the stress test tool. I have set IIS to use this path:

http://localhost/Threading/SlowThread.aspx

http://localhost/Threading/FastThread.aspx

Firstly, set up the script by adding the content tree—the pages requested from the tool.

Figure 2-8. Set the content for a stress test

The content tree allows you to set the actions the tool performs several times. Alternatively, you can record a session to save the page load action. However, in this script, only one page is called using a GET command, so recording is not necessary.

Figure 2- 9. Settings used to force the test

The settings define how the tool operates on the server. In Figure 2-9 I set thirty threads, which create thirty concurrent requests. This runs for twenty seconds as quickly as possible. There is no throttling and no other options are used. This is the configuration that should cause the server to create twenty five threads and bring the thread pool to its limit.

Table 2-2 shows the results of some stress tests. Remember that there is a wide range of possible values for response time depending on your machine, its configuration, and what other applications and services are currently running. It’s the relative values that are significant here.

Table 2-2. Test results using a web stress test tool

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | File | Hits | Average response time |
| 100 | FastThread.aspx | 21093 | 45 ms |
| 20 | FastThread.aspx | 21034 | 8.3 ms |
| 100 | SlowThread.aspx | 33 | 6,108 ms |
| 20 | SlowThread.aspx | 32 | 4,654 ms |

These results are not surprising. The server is less able to handle the requests as the number of parallel threads increases. Because we’re forcing IIS to queue the requests, this causes additional overhead. Running the client against both FastThread.aspx and SlowThread.aspx, however, reduces the average response time for FastThread.aspx requests to 2.05 seconds and only 98 hits being handled. For SlowThread.aspx, the number of threads makes no significant difference. This is the worst case—because of long running requests, fast ones are not served quickly. Increasing the CPU power will not help. The slow page does not consume any CPU power because there is nothing to calculate. The delay you see is due to the saturation of the thread pool. Even the fast requests are queued until a thread is released.

This demonstrates that some pages can influence the performance of other requests even if they have nothing to do with each other. What is the solution?

I discussed previously the various settings available for improving the system’s behavior. In the stress test I used 100 threads. Increasing the thread pool limits might help. However, the real world does not have a defined number of requests regularly arriving. Finding the right value is anything but easy. It could change depending on user behavior, server settings, network connection, and application code—which means that there is no right answer.

The solution we’re looking for should free the threads in the thread pool to hold the utilization low and have enough threads available at any time. In other words, each page should behave like a fast running page. This brings us to asynchronous handlers.

Note: The next chapter is dedicated the world of handlers and modules. Here we discuss only the portion directly related to threading issues. For more information about handlers, refer to chapter 3.

### Custom Thread Pool and Asynchronous Handlers

While most ASP.NET pages and handlers are serviced synchronously on threads drawn from the thread pool, it is possible to create handlers and pages that service requests asynchronously.

Asynchronous handlers implement the IHttpAsyncHandler interface, which derives from IHttpHandler.

Listing 2-10. Definition of the IHttpAsynchHandler interface

public interface IHttpAsyncHandler : IHttpHandler

{

IAsyncResult BeginProcessRequest(HttpContext ctx,

AsyncCallback cb,

object obj);

void EndProcessRequest(IAsyncResult ar);

}

The interface follows the typical pattern for asynchronous actions—it has a method for indicating the beginning and one for indicating the end of a process. Usually handlers have a method named ProcessRequest. Instead of calling this method, the asynchronous handler calls the BeginProcessRequest method. In this method, you launch a new thread and manage things, which can take some time. The method returns immediately, providing a reference to an IAsyncResult instance. This frees the thread from the thread pool. A new thread is then used to perform the long running action. When the internal (or private) thread returns, the EndProcessRequest method is called. The IAsyncResult instance is handed over in order to parameterize the call. Cleanup actions, such as closing a database connection, are best placed here.

#### Asynchronous Handler with a Delegate

Creating an asynchronous handler with a delegate is the most common approach. Delegates called with BeginInvoke implicitly create a new thread.

Listing 2-11. A handler using IHttpAsyncHandler

<%@ WebHandler Language="C#" Class="Apress.Threading.AsyncHandlers.AsyncHandler" %>

using System;

using System.Web;

using System.Threading;

using System.Diagnostics;

using System.Reflection;

namespace Apress.Threading.AsyncHandlers

{

public delegate void ProcessRequestDelegate(HttpContext ctx);

public class AsyncHandler : IHttpAsyncHandler

{

public void ProcessRequest(HttpContext ctx)

{

System.Threading.Thread.Sleep(2000);

ctx.Response.Output.Write(

"Async Delegate, Thread ID={0}",

AppDomain.GetCurrentThreadId());

}

public bool IsReusable

{

get { return true; }

}

public IAsyncResult BeginProcessRequest(HttpContext ctx, ⮰

AsyncCallback cb, ⮰

object obj)

{

ProcessRequestDelegate prg = new ProcessRequestDelegate(ProcessRequest);

return prg.BeginInvoke(ctx, cb, obj);

}

public void EndProcessRequest(IAsyncResult ar)

{

}

}

}

Call this handler by using a browser, as before, to test that it’s running properly:

http://localhost/Threading/AsyncThreadDelegate.ashx

The process starts with the method BeginProcessRequest. Using the delegate and its BeginInvoke method, the method ProcessRequest is called. Then the Sleep method simulates something long running without performing anything on the CPU. Because there is nothing to clean up, no code is required in EndProcessRequest.

Now it’s time to see what the stress test tool is reporting.

If you run the test as before, you can see that the results are nearly identical. What is going on here? Why write asynchronous delegates if there is no speed improvement at all? The reason is in the way ASP.NET internally handles the threading.

Recall that when I first introduced the term “thread pool”, it was called a “process wide thread pool”. The asynchronous handler still runs in the very same process, which leads to a lack of improvement. The original thread is freed, but the new one is taken from the same thread pool. The same thing would happen if you used ThreadPool.QueueUserWorkItem from the System.Threading namespace. We need to find where we can get a new thread from another thread source.

#### Asynchronous Handler with Custom Threads

Before we can implement the whole solution, another interface is required—IAsynchResult—as the return result is now essential. The definition is short—just four properties (Listing 2-12).

Listing 2-12. The definition of IAsyncResult

public interface IAsyncResult

{

public object AsyncState { get; }

public bool CompletedSynchronously { get; }

public bool IsCompleted { get; }

public WaitHandle AsyncWaitHandle { get; }

}

Our actual implementation of the IAsynchResult interface (Listing 2-13) has two additional properties: a reference to the HttpContext object, and a reference to the callback object. The callback method is invoked later, when processing is complete. The AsyncState object is optional; you can use it to store private data. AsyncWaitHandle returns a WaitHandle object, which is used to signal when the request is complete. Using the CompleteRequest method implemented additionally to the requirements of the interface, the calling class can execute the EndProcessRequest method. In this routine, the WaitHandle object is triggered. Either way, the IsCompleted property shows that the object has reached the completed state.

Listing 2-13. The implementation of IAsyncResult (from AsyncThreadCallback.ashx)

class AsyncRequestState : IAsyncResult

{

internal HttpContext \_ctx;

internal AsyncCallback \_cb;

internal object \_data;

private bool \_isCompleted;

private ManualResetEvent \_ completeEvent;

public AsyncRequestState(HttpContext ctx,

AsyncCallback cb,

object data)

{

\_ctx = ctx;

\_cb = cb;

\_data = data;

}

internal HttpContext CurrentContext

{

get

{

return \_ctx;

}

}

internal void CompleteRequest()

{

\_isCompleted = true;

lock (this)

{

if (\_completeEvent!= null)

\_ completeEvent.Set();

}

// invoke registered callback, if any

if (\_cb != null)

\_cb(this);

}

public object AsyncState

{

get

{

return (\_data);

}

}

public bool CompletedSynchronously

{

get

{

return (false);

}

}

public bool IsCompleted

{

get

{

return (\_isCompleted);

}

}

public WaitHandle AsyncWaitHandle

{

get

{

lock (this)

{

if (\_completeEvent == null)

\_ completeEvent = new ManualResetEvent(false);

return \_completeEvent;

}

}

}

}

The next step is to spawn a new thread to process the request. The method called on this new thread will need access to the state cached in the AsyncRequestState class shown above. To pass necessary data to this object, a parameterized thread is used, based on ParameterizedThreadStart class.

The handler itself is similar to the one already introduced. The definition for the ashx page looks like this:

<%@ WebHandler Language="C#"   
 Class="Apress.Threading.AsyncHandlers.CustomAsyncHandler" %>

The ProcessRequest method must be present, as it is required by the interface IHttpAsyncHandler, but we don’t use it. The whole work is split between the BeginProcessRequest and EndProcessRequest methods. In the BeginProcessRequest method the AsyncRequestState object is created, along with the reference to the context and callback.

Listing 2-14. The implementation of AsyncResult (AsyncThreadCallback.ashx)

public class CustomAsyncHandler : IHttpAsyncHandler

{

public void ProcessRequest(HttpContext ctx)

{

// not used

}

public bool IsReusable

{

get { return false; }

}

public IAsyncResult BeginProcessRequest(HttpContext ctx,

AsyncCallback cb,

object obj)

{

AsyncRequestState reqState = ⮰

new AsyncRequestState(ctx, cb, obj);

ParameterizedThreadStart ts = new ParameterizedThreadStart(ProcessThread);

Thread t = new Thread(ts);

t.Start();

return reqState;

}

public void EndProcessRequest(IAsyncResult ar)

{

AsyncRequestState ars = ar as AsyncRequestState;

if (ars != null)

{

ars.CurrentContext.Response.Write("End Request reached");

}

}

private void ProcessThread(object obj)

{

Thread.Sleep(2000);

AsyncRequestState asr = obj as AsyncRequestState;

asr.CurrentContext.Response.Output.Write( ⮰

"Async Thread, Thread ID = {0}", ⮰

AppDomain.GetCurrentThreadId());

// signal end of processing

asr.EndRequest();

}

}

Let’s look at the code to understand how it’s working. Figure 2-10 shows the thread usage during internal processing. Initiated by the request, the process starts by calling the BeginProcessRequest method. The AsyncHandler creates the helper objects required and launches the new custom thread independent of the thread pool. The custom thread runs and the thread pool thread is released. Once the process is completed, the end is signaled via the callback method and the request is finished.

Figure 2-10. First step of the processing using an asynchronous handler

The response held in memory is waiting for the thread to complete. This might seem disappointing, because it doesn’t make anything faster. The page is still waiting for the slow operation to finish, and so is the user. Remember the thread pool issue, however; once the pool runs out of threads, the performance of the whole server is reduced and no more requests are handled properly. The advantage of custom threads is not a faster response for one user but for all users.

In step two, as shown in Figure 2-11, the call to EndRequest indicates the end of the process and terminates the request. Once called, the response is complete and sent to the browser. This means that you can add content to the response at any time before the EndProcessRequest method is called. In the example code (Listing 2-14) the Response.Output.Write method demonstrates this.

Figure 2-11. Second step of the processing using an asynchronous handler

Run the stress test again to see whether you find an improvement. Under pressure it performs better, and the thread pool no longer runs out of threads. This allows the site to accept incoming requests at any time, and the whole server appears to be more responsive. Again, the processing time for a single page is the same.

#### Asynchronous Handler and a Custom Thread Pool

This technique is useful for many extensibility projects. In Chapter 3, I’ll show more examples of practical implementations that solve common issues. All handlers explained there use simple synchronous operations. You can implement them all as asynchronous counterparts using the techniques described here. Using asynchronous handlers you’ll see how to improve the overall performance of an ASP.NET site drastically. The thread issues and handlers that I discuss here are just one solution.

There is another issue that can be found easily using the stress test tool. In one of the previous paragraphs I wrote “accept incoming requests at any time”. This is quite a claim, as it assumes that the server will respond no matter how many requests arrive, just by creating new threads. However, the operating system, CPU power, available memory, and architectural restrictions of the hardware may stop the code from creating new threads. It is possible for too many threads to be created, in which case they will block each other and make the system slow and unresponsive.

You might ask “Why not use a thread pool?” Unfortunately, .NET has only one thread pool internally and it is used by ASP.NET. That’s why using ThreadPool.QueueUserWorkItem does not work. Our task now is the creation of our own thread pool. Several people have done this successfully. Mike Woodring wrote a version which is useful and fits our needs, using the code in an asynchronous handler. The original version can be found on his page:

http://www.bearcanyon.com/

I have adapted his code to suit our needs for a handler.

There are several clever aspects to this code. The usage of HttpContext is much simpler. In the previous example, a reference to the context object was stored and the use of HttpContext.Current was forbidden. It is not good programming practice to change well known method calls.

Listing 2-15. Delegates and base interface for the thread pool

public delegate void WorkRequestDelegate(object state, ⮰  
 DateTime requestEnqueueTime );

public delegate void ThreadPoolDelegate();

#region IWorkRequest interface

public interface IWorkRequest

{

bool Cancel();

}

#endregion

The following code shows the implementation in several steps. The core functionality is explained inline.

Listing 2-16. The basic thread pool implementation

public sealed class ThreadPool : WaitHandle

{

#region ThreadPool constructors

public ThreadPool( int initialThreadCount, int maxThreadCount, string poolName )

: this( initialThreadCount, maxThreadCount, poolName,

DEFAULT\_NEW\_THREAD\_TRIGGER\_TIME,

DEFAULT\_DYNAMIC\_THREAD\_DECAY\_TIME,

DEFAULT\_THREAD\_PRIORITY,

DEFAULT\_REQUEST\_QUEUE\_LIMIT )

{

}

public ThreadPool( int initialThreadCount, int maxThreadCount, string poolName,

int newThreadTrigger, int dynamicThreadDecayTime,

ThreadPriority threadPriority, int requestQueueLimit )

{

Handle = stopCompleteEvent.Handle;

if( maxThreadCount < initialThreadCount )

{

throw new ArgumentException("Maximum thread count must be >= initial ⮰

thread count.", "maxThreadCount");

}

if( dynamicThreadDecayTime <= 0 )

{

throw new ArgumentException("Dynamic thread decay time cannot be <= 0."

, "dynamicThreadDecayTime");

}

if( newThreadTrigger <= 0 )

{

throw new ArgumentException("New thread trigger time cannot be <= 0."

, "newThreadTrigger");

}

this.initialThreadCount = initialThreadCount;

this.maxThreadCount = maxThreadCount;

this.requestQueueLimit = (requestQueueLimit < 0 ? ⮰

DEFAULT\_REQUEST\_QUEUE\_LIMIT : requestQueueLimit);

this.decayTime = dynamicThreadDecayTime;

this.newThreadTrigger = new TimeSpan(TimeSpan.TicksPerMillisecond \* ⮰

newThreadTrigger);

this.threadPriority = threadPriority;

this.requestQueue = new Queue(requestQueueLimit < 0 ? ⮰

4096 : requestQueueLimit);

if( poolName == null )

{

throw new ArgumentNullException("poolName", ⮰

"Thread pool name cannot be null");

}

else

{

this.threadPoolName = poolName;

}

}

The next region sets several ThreadPool properties. The Priority and the DynamicThreadDecay properties are not thread safe. They can only be set before Start is called.

public ThreadPriority Priority

{

get { return(threadPriority); }

set

{

if( hasBeenStarted )

{

throw new InvalidOperationException("Cannot adjust thread ⮰

priority after pool has been started.");

}

threadPriority = value;

}

}

public int DynamicThreadDecay

{

get { return(decayTime); }

set

{

if( hasBeenStarted )

{

throw new InvalidOperationException("Cannot adjust dynamic thread ⮰

decay time after pool has been started.");

}

if( value <= 0 )

{

throw new ArgumentException("Dynamic thread decay time cannot ⮰

be <= 0.", "value");

}

decayTime = value;

}

}

public int NewThreadTrigger

{

get { return((int)newThreadTrigger.TotalMilliseconds); }

set

{

if( value <= 0 )

{

throw new ArgumentException("New thread trigger time cannot ⮰

be <= 0.", "value");

}

lock( this )

{

newThreadTrigger = new TimeSpan(TimeSpan.TicksPerMillisecond \* ⮰

value);

}

}

}

public int RequestQueueLimit

{

get { return(requestQueueLimit); }

set { requestQueueLimit = (value < 0 ? DEFAULT\_REQUEST\_QUEUE\_LIMIT : value);

}

public int AvailableThreads

{

get { return(maxThreadCount - currentThreadCount); }

}

public int MaxThreads

{

get { return(maxThreadCount); }

set

{

if( value < initialThreadCount )

{

throw new ArgumentException("Maximum thread count must be ⮰

>= initial thread count.", ⮰

"MaxThreads");

}

maxThreadCount = value;

}

}

public bool IsStarted

{

get { return(hasBeenStarted); }

}

public bool PropogateThreadPrincipal

{

get { return(propogateThreadPrincipal); }

set { propogateThreadPrincipal = value; }

}

public bool PropogateCallContext

{

get { return(propogateCallContext); }

set { propogateCallContext = value; }

}

public bool PropogateHttpContext

{

get { return(propogateHttpContext); }

set { propogateHttpContext = value; }

}

public bool PropogateCASMarkers

{

get { return(propogateCASMarkers); }

// When CompressedStack get/set is opened up,

// add the following setter back in.

//

// set { propogateCASMarkers = value; }

}

public bool IsBackground

{

get { return(useBackgroundThreads); }

set

{

if( hasBeenStarted )

{

throw new InvalidOperationException("Cannot adjust background ⮰

status after pool has been started.");

}

useBackgroundThreads = value;

}

}

#endregion

#region ThreadPool events

public event ThreadPoolDelegate Started;

public event ThreadPoolDelegate Stopped;

#endregion

public void Start()

{

lock( this )

{

if( hasBeenStarted )

{

throw new InvalidOperationException("Pool has already ⮰  
 been started.");

}

hasBeenStarted = true;

Next, we have to check to see if there were already items posted to the queue before Start was called. If this is the case, we’ll reset their timestamps to the current time.

if( requestQueue.Count > 0 )

{

ResetWorkRequestTimes();

}

for( int n = 0; n < initialThreadCount; n++ )

{

ThreadWrapper thread = ⮰

new ThreadWrapper( this, true, threadPriority, ⮰

string.Format("{0} (static)", threadPoolName) );

thread.Start();

}

if( Started != null )

{

Started();

}

}

}

The Stop method can be used to stop the whole pool immediately. The StopAndWait method lets all threads run until they end normally and then stops the pool. The overloaded version defines a timeout to prevent the method blocking due to threads held in a deadlock.

public void Stop()

{

InternalStop(false, Timeout.Infinite);

}

public void StopAndWait()

{

InternalStop(true, Timeout.Infinite);

}

public bool StopAndWait( int timeout )

{

return InternalStop(true, timeout);

}

InternalStop is the method that completes the order. After checking that there is something to stop we set the property that exposes the stopping process. Monitor.PulseAll informs all threads that the state of the object has changed.

If the wait option is set, the thread is waiting using WaitOne. This method is defined in the base class WaitHandle.

private bool InternalStop( bool wait, int timeout )

{

if( !hasBeenStarted )

{

throw new InvalidOperationException("Cannot stop a thread pool ⮰  
 that has not been started yet.");

}

lock(this)

{

stopInProgress = true;

Monitor.PulseAll(this);

}

if( wait )

{

bool stopComplete = WaitOne(timeout, true);

if( stopComplete )

{

hasBeenStarted = false;

stopInProgress = false;

requestQueue.Clear();

stopCompleteEvent.Reset();

}

return(stopComplete);

}

return(true);

}

#endregion

#region ThreadPool.PostRequest(early bound)

// Overloads for the early bound WorkRequestDelegate-based targets.

//

public bool PostRequest( WorkRequestDelegate cb )

{

return PostRequest(cb, (object)null);

}

public bool PostRequest( WorkRequestDelegate cb, object state )

{

IWorkRequest notUsed;

return PostRequest(cb, state, out notUsed);

}

public bool PostRequest( WorkRequestDelegate cb, object state, ⮰

out IWorkRequest reqStatus )

{

WorkRequest request =

new WorkRequest( cb, state,

propogateThreadPrincipal, propogateCallContext,

propogateHttpContext, propogateCASMarkers );

reqStatus = request;

return PostRequest(request);

}

#endregion

#region ThreadPool.PostRequest(late bound)

// Overloads for the late bound Delegate.DynamicInvoke-based targets.

//

public bool PostRequest( Delegate cb, object[] args )

{

IWorkRequest notUsed;

return PostRequest(cb, args, out notUsed);

}

public bool PostRequest( Delegate cb, object[] args, out IWorkRequest reqStatus )

{

WorkRequest request =

new WorkRequest( cb, args,

propogateThreadPrincipal, propogateCallContext,

propogateHttpContext, propogateCASMarkers );

reqStatus = request;

return PostRequest(request);

}

#endregion

// The actual implementation of PostRequest.

//

bool PostRequest( WorkRequest request )

{

lock(this)

{

A requestQueueLimit of -1 means the queue is “unbounded” and that there is no explicit limit on the maximum number of requests allowed in the queue.

if( (requestQueueLimit == -1) || ⮰

(requestQueue.Count < requestQueueLimit) )

{

try

{

requestQueue.Enqueue(request);

Monitor.Pulse(this);

return(true);

}

catch

{

}

}

}

return(false);

}

void ResetWorkRequestTimes()

{

lock( this )

{

DateTime newTime = DateTime.Now;

foreach( WorkRequest wr in requestQueue )

{

wr.workingTime = newTime;

}

}

}

The class comes with few default parameters. The time values are provided in milliseconds. The constant DEFAULT\_DYNAMIC\_THREAD\_DECAY\_TIME is set to five minutes.

const int DEFAULT\_DYNAMIC\_THREAD\_DECAY\_TIME = 5 \* 60 \* 1000;

const int DEFAULT\_NEW\_THREAD\_TRIGGER\_TIME = 500;

const ThreadPriority DEFAULT\_THREAD\_PRIORITY = ThreadPriority.Normal;

const int DEFAULT\_REQUEST\_QUEUE\_LIMIT = -1; // unbounded

#endregion

#region Private ThreadPool member variables

private bool hasBeenStarted = false;

private bool stopInProgress = false;

private readonly string threadPoolName;

private readonly int initialThreadCount; // Initial # of threads to create (called "static threads" in this class).

private int maxThreadCount; // Cap for thread count. Threads added above initialThreadCount are called "dynamic" threads.

private int currentThreadCount = 0; // Current # of threads in the pool (static + dynamic).

private int decayTime; // If a dynamic thread is idle for this period of time w/o processing work requests, it will exit.

private TimeSpan newThreadTrigger; // If a work request sits in the queue this long before being processed, a new thread will be added to queue up to the max.

private ThreadPriority threadPriority;

private ManualResetEvent stopCompleteEvent = new ManualResetEvent(false); // Signaled after Stop called and last thread exits.

private Queue requestQueue;

private int requestQueueLimit; // Throttle for maximum # of work requests that can be added.

private bool useBackgroundThreads = true;

private bool propogateThreadPrincipal = false;

private bool propogateCallContext = false;

private bool propogateHttpContext = false;

private bool propogateCASMarkers = false;

Listing 2-17. Private embedded class that holds thread information

class ThreadInfo

{

public static ThreadInfo Capture( bool propogateThreadPrincipal, bool propogateCallContext,

bool propogateHttpContext, bool propogateCASMarkers )

{

return new ThreadInfo( propogateThreadPrincipal, propogateCallContext,

propogateHttpContext, propogateCASMarkers );

}

public static ThreadInfo Impersonate( ThreadInfo ti )

{

if( ti == null ) throw new ArgumentNullException("ti");

ThreadInfo prevInfo = Capture(true, true, true, true);

Restore(ti);

return(prevInfo);

}

public static void Restore( ThreadInfo ti )

{

if( ti == null ) throw new ArgumentNullException("ti");

if( miSetLogicalCallContext != null )

{

miSetLogicalCallContext.Invoke(Thread.CurrentThread, ⮰  
 new object[]{ti.callContext});

}

Restore HttpContext assigns the stored context as current one.

CallContext.SetData(HttpContextSlotName, ti.httpContext);

The thread identity is restored. It's important that this is done after restoring call context, since restoring call context also overwrites the current thread principal setting. If propogateCallContext and propogateThreadPrincipal are both true, then the following is redundant. However, since propagating call context requires the use of reflection to capture and restore call context, the author wanted that behavior to be independently switchable so that it could be disabled. The thread principal is still allowed to be propagated. In the event that call context propagation changes it no longer propagates the thread principal.

Thread.CurrentPrincipal = ti.principal;

}

private ThreadInfo( bool propogateThreadPrincipal, ⮰

bool propogateCallContext, ⮰

bool propogateHttpContext, bool propogateCASMarkers )

{

if( propogateThreadPrincipal )

{

principal = Thread.CurrentPrincipal;

}

if( propogateHttpContext )

{

httpContext = HttpContext.Current;

}

if( propogateCallContext && (miGetLogicalCallContext != null) )

{

callContext = ⮰

(LogicalCallContext)miGetLogicalCallContext.Invoke(⮰

Thread.CurrentThread, null);

callContext = (LogicalCallContext)callContext.Clone();

}

}

IPrincipal principal;

LogicalCallContext callContext;

HttpContext httpContext;

// Cached type information.

//

const BindingFlags bfNonPublicInstance = BindingFlags.Instance | BindingFlags.NonPublic;

const BindingFlags bfNonPublicStatic = BindingFlags.Static | BindingFlags.NonPublic;

static MethodInfo miGetLogicalCallContext =

typeof(Thread).GetMethod("GetLogicalCallContext", bfNonPublicInstance);

static MethodInfo miSetLogicalCallContext =

typeof(Thread).GetMethod("SetLogicalCallContext", bfNonPublicInstance);

static string HttpContextSlotName;

static ThreadInfo()

{

Look up the value of HttpContext.CallContextSlotName—if it exists—to find the name of the call context slot where HttpContext.Current is stored. If this field isn’t present, try for the original “HttpContext” slot name.

FieldInfo fi = typeof(HttpContext).GetField("CallContextSlotName", bfNonPublicStatic);

if( fi != null )

{

HttpContextSlotName = (string)fi.GetValue(null);

}

else

{

HttpContextSlotName = "HttpContext";

}

}

}

Listing 2-18. Implementation of the base interface to handle the incoming requests

class WorkRequest : IWorkRequest

{

internal const int PENDING = 0;

internal const int PROCESSED = 1;

internal const int CANCELLED = 2;

public WorkRequest( WorkRequestDelegate cb, object arg, ⮰

bool propogateThreadPrincipal, ⮰

bool propogateCallContext, ⮰

bool propogateHttpContext, bool propogateCASMarkers )

{

targetProc = cb;

procArg = arg;

procArgs = null;

Initialize( propogateThreadPrincipal, propogateCallContext,

propogateHttpContext, propogateCASMarkers );

}

public WorkRequest( Delegate cb, object[] args, ⮰

bool propogateThreadPrincipal, ⮰

bool propogateCallContext, ⮰

bool propogateHttpContext, bool propogateCASMarkers )

{

targetProc = cb;

procArg = null;

procArgs = args;

Initialize( propogateThreadPrincipal, propogateCallContext,

propogateHttpContext, propogateCASMarkers );

}

void Initialize( bool propogateThreadPrincipal, bool propogateCallContext,

bool propogateHttpContext, bool propogateCASMarkers )

{

workingTime = timeStampStarted = DateTime.Now;

threadInfo = ThreadInfo.Capture( propogateThreadPrincipal, propogateCallContext,

propogateHttpContext, propogateCASMarkers );

}

public bool Cancel()

{

If the work request was pending, it’s marked as cancelled. Otherwise, this method was called too late. Note that this call can cancel an operation without any race conditions. But if the result of this test-and-set indicates the request is in the “processed” state, it might actually be about to be processed.

return(Interlocked.CompareExchange(ref state, CANCELLED, PENDING) == PENDING);

}

internal Delegate targetProc; // Function to call.

internal object procArg; // State to pass to function.

internal object[] procArgs; // Used with Delegate.DynamicInvoke.

internal DateTime timeStampStarted; // Time work request was originally enqueued (held constant).

internal DateTime workingTime; // Current timestamp used for triggering new threads (moving target).

internal ThreadInfo threadInfo; // Everything we know about a thread.

internal int state = PENDING; // The state of this particular request.

}

Listing 2-19. An internal wrapper class around the pool

class ThreadWrapper

{

ThreadPool pool;

bool isPermanent;

ThreadPriority priority;

string name;

public ThreadWrapper( ThreadPool pool, bool isPermanent,

ThreadPriority priority, string name )

{

this.pool = pool;

this.isPermanent = isPermanent;

this.priority = priority;

this.name = name;

lock( pool )

{

// Update the total # of threads in the pool.

//

pool.currentThreadCount++;

}

}

public void Start()

{

Thread t = new Thread(new ThreadStart(ThreadProc));

t.ApartmentState = ApartmentState.MTA;

t.Name = name;

t.Priority = priority;

t.IsBackground = pool.useBackgroundThreads;

t.Start();

}

void ThreadProc()

{

bool done = false;

while( !done )

{

WorkRequest wr = null;

ThreadWrapper newThread = null;

lock( pool )

{

As long as the request queue is empty and a shutdown hasn’t been initiated, wait for a new work request to arrive.

bool timedOut = false;

while( !pool.stopInProgress && !timedOut && (pool.requestQueue.Count == 0) )

{

if( !Monitor.Wait(pool, (isPermanent ? Timeout.Infinite : pool.decayTime)) )

{

Timed out waiting for something to do. Only dynamically created threads will reach this point, so bail out.

//

timedOut = true;

}

}

The loop above exited because one of the following conditions was met:

\* ThreadPool.Stop was called to initiate a shutdown.

\* A dynamic thread timed out waiting for a work request to arrive.

\* There were items in the work queue to process.

If the loop exited because there was work to be done, this means that a shutdown hadn’t been initiated, and the code wasn’t running in a dynamic thread that timed out. Pull the request off the queue and prepare to process it:

if( !pool.stopInProgress && ⮰

!timedOut && (pool.requestQueue.Count > 0) )

{

wr = (WorkRequest)pool.requestQueue.Dequeue();

Debug.Assert(wr != null);

Check whether this work request languished in the queue too long. If it was in the queue longer than the new thread trigger time, and if it hadn’t reached the max thread count cap, add a new thread to the pool.

If that was the case, create the new thread object and update the current number of threads in the pool, but defer starting the new thread until the lock is released.

TimeSpan requestTimeInQ = ⮰

DateTime.Now.Subtract(wr.workingTime);

if( (requestTimeInQ >= pool.newThreadTrigger) && ⮰

(pool.currentThreadCount < pool.maxThreadCount) )

{

Note that the constructor for ThreadWrapper will update pool.currentThreadCount.

newThread = ⮰

new ThreadWrapper( pool, false, priority, ⮰

string.Format("{0} (dynamic)", ⮰

pool.threadPoolName) );

Since the current request we just dequeued is stale, everything else behind it in the queue is also stale. Therefore, reset the timestamps of the remaining pending work requests so that we don’t start creating threads for every subsequent request.

pool.ResetWorkRequestTimes();

}

}

else

{

Code flow should only reach this point if this is a dynamic thread that timed out waiting for a work request, or if the pool is shutting down.

pool.currentThreadCount--;

if( pool.currentThreadCount == 0 )

{

If there is no thread running in the pool, the pool stops and this is signaled.

if( pool.Stopped != null )

{

pool.Stopped();

}

pool.stopCompleteEvent.Set();

}

done = true;

}

} // lock

After all this is completed, the lock is no longer required.

if( !done && (wr != null) )

{

Now check to see if this request has been cancelled while stuck in the work queue. If the work request was pending, mark it processed and proceed to handle. Otherwise, the request must have been cancelled before we plucked it off the request queue.

if( Interlocked.CompareExchange(ref wr.state, ⮰

WorkRequest.PROCESSED, WorkRequest.PENDING) != ⮰

WorkRequest.PENDING )

{

The request was cancelled before flow could get here and we have to bail out.

continue;

}

if( newThread != null )

{

Add a dynamic thread to the pool just by starting it.

newThread.Start();

}

Dispatch the work request.

ThreadInfo originalThreadInfo = null;

try

{

Impersonate as much as possible what we know about the thread that issued the work request.

originalThreadInfo = ThreadInfo.Impersonate(wr.threadInfo);

WorkRequestDelegate targetProc = wr.targetProc as ⮰

WorkRequestDelegate;

if( targetProc != null )

{

targetProc(wr.procArg, wr.timeStampStarted);

}

else

{

wr.targetProc.DynamicInvoke(wr.procArgs);

}

}

catch( Exception e )

{

}

finally

{

Restore our worker thread's identity.

ThreadInfo.Restore(originalThreadInfo);

}

}

}

The worker thread is now exiting the pool.

}

}

}

#### Using the Custom Thread Pool

Now that we have a custom thread pool implementation, it’s time to use it. The following example is simply an extended version of the previous one. Instead of using single threads, however, and risking having too many of them, we can use the custom thread pool and set some limitations.

Listing 2-20. Using the thread pool

<!-- File: AsyncPool.ashx -->

<%@ WebHandler Language="C#"

Class="Apress.Threading.HttpPipeline.AsyncHandler" %>

namespace Apress.Threading.HttpPipeline

{

public class AsyncHandler : IHttpAsyncHandler

{

static DevelopMentor.ThreadPool \_threadPool;

static AsyncHandler()

{

\_threadPool =

new DevelopMentor.ThreadPool(2, 25, "AsyncPool");

\_threadPool.PropogateCallContext = true;

\_threadPool.PropogateThreadPrincipal = true;

\_threadPool.PropogateHttpContext = true;

\_threadPool.Start();

}

public void ProcessRequest(HttpContext ctx)

{

// not used

}

public bool IsReusable

{

get { return false;}

}

public IAsyncResult BeginProcessRequest(HttpContext ctx,

AsyncCallback cb, object obj)

{

AsyncRequestState reqState =

new AsyncRequestState(ctx, cb, obj);

\_threadPool.PostRequest( new Apress.Threading.CustomThreadPool. ⮰

WorkRequestDelegate(ProcessRequest),

reqState);

return reqState;

}

public void EndProcessRequest(IAsyncResult ar)

{

}

void ProcessRequest(object state, DateTime requestTime)

{

AsyncRequestState reqState = state as AsyncRequestState;

// Take some time to do it

Thread.Sleep(2000);

reqState.\_ctx.Response.Output.Write(

"AsyncThreadPool, {0}",

AppDomain.GetCurrentThreadId);

// tell asp.net you are finished processing this request

reqState.CompleteRequest();

}

}

}

In theory, this thread pool will never run out of threads. However, there are some limitations such as available memory, CPU power, and operating system restrictions. This is why the pool was designed: so that you can optionally specify limits. The code shown above is a good platform for experimenting with threads and thread pools and for monitoring the number of threads for your application. If you want to replace the internal thread pool, try out the custom pool to learn how ASP.NET requests threads and find out where to improve performance.

### Asynchronous Pages

So far, all of the examples of asynchronous request handling have involved building custom handlers. In chapter 3, I’ll cover such handlers in more detail.

There may be many \*.aspx pages in your ASP.NET applications that are also good candidates for asynchronous execution because they perform non-CPU-bound tasks that take significant amounts of time. Such pages are also potential bottlenecks for the ASP.NET thread pool.

Because pages are also managed by a handler, it seems easy to let pages execute asynchronously. This is already implemented by the framework for us and there is nothing to do but set a property.

#### Define the Right Page Handler

Building asynchronous pages is pretty simple. Begin by including an attribute in the page’s @Page directive:

<%@ Page Async="true" ... %>

Behind the scenes, this tells ASP.NET to implement IHttpAsyncHandler in the page instead of IHttpHandler used regularly. Next, call the AddOnPreRenderCompleteAsync method early in the page’s lifetime. For example, in Load, this is early enough. Register a Begin method and an End method, as shown in the following code:

AddOnPreRenderCompleteAsync (

new BeginEventHandler(MyBeginMethod),

new EndEventHandler (MyEndMethod)

);

The page runs through its normal processing lifecycle until shortly after the PreRender event fires. Then ASP.NET calls the Begin method that you registered using AddOnPreRenderCompleteAsync. The Begin method launches an asynchronous operation and returns immediately. This is a lengthy operation which might require more time, such as a database query or a webservice call to another server. At this point, the thread assigned to the request returns to the thread pool. Furthermore, the Begin method returns an IAsyncResult that allows ASP.NET to determine when the asynchronous operation has been completed. ASP.NET then extracts a thread from the thread pool and calls your End method. After End returns, it executes the remaining portion of the page’s lifecycle. This might sound confusing—getting the thread back from thread pool would again block the thread. But between the time Begin returns and End gets called, the request-processing thread is free to service other requests. Keep in mind that the page processing usually takes only a few milliseconds. The time-consuming operation in the asynchronous handler might run for seconds. The process of freeing the thread pool thread for this time allows ASP.NET to process hundreds if not thousands of regular pages on this very same thread. However, until End is called, the rendering of the current asynchronous page is delayed. This is the same as in the previous example. It improves the situation not only for one user, but for all users. In the sample code, you might look for the IAsyncResult implementation. Instead of implementing our own version, we take one that the Framework implements for us.

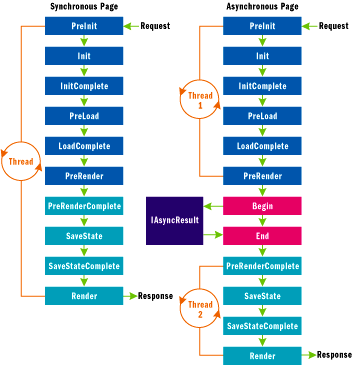


Figure 2-12. Synchronous vs. Asynchronous Page Processing

Figure 2-12 illustrates the difference between a synchronous page and an asynchronous page. When a synchronous page is requested, ASP.NET assigns the request a thread from the thread pool and executes the page on that thread.

That’s enough theory for now. It’s time to look at some samples to get a better appreciation of how to implement asynchronous pages. The two examples address two common problems:

\* Call a database operation asynchronously

\* Call a webservice—a current exchange rate service that converts between US$ and Euro.

#### Asynchronous Data Binding

It is a common task to query databases and data bind the results. Using asynchronous pages to perform asynchronous data binding seems like a perfect symbiosis. The code behind class in Listing 2-21 shows one way to go.

Listing 2-21. Asynchronous binding to a data source (AsyncDataBind.aspx.cs)

using System;

using System.Data;

using System.Data.SqlClient;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

using System.Web.Configuration;

public partial class AsyncDataBind : System.Web.UI.Page

{

private SqlConnection \_connection;

private SqlCommand \_command;

private SqlDataReader \_reader;

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

// Hook PreRenderComplete event for data binding

this.PreRenderComplete +=

new EventHandler(Page\_PreRenderComplete);

// Register async methods

AddOnPreRenderCompleteAsync(

new BeginEventHandler(BeginAsyncOperation),

new EndEventHandler(EndAsyncOperation)

);

}

}

IAsyncResult BeginAsyncOperation (object sender, EventArgs e,

AsyncCallback cb, object state)

{

string connect = WebConfigurationManager.ConnectionStrings

["PubsConnectionString"].ConnectionString;

\_connection = new SqlConnection(connect);

\_connection.Open();

\_command = new SqlCommand(

"SELECT title\_id, title, price FROM titles", \_connection);

return \_command.BeginExecuteReader (cb, state);

}

void EndAsyncOperation(IAsyncResult ar)

{

\_reader = \_command.EndExecuteReader(ar);

}

protected void Page\_PreRenderComplete(object sender, EventArgs e)

{

Output.DataSource = \_reader;

Output.DataBind();

}

public override void Dispose()

{

if (\_connection != null) \_connection.Close();

base.Dispose();

}

}

The AsyncDataBind class uses the AddOnPreRenderCompleteAsync pattern. In its BeginAsyncOperation method, it calls SqlCommand.BeginExecuteReader to perform an asynchronous database query. When the call is completed, EndAsyncOperation calls SqlCommand.EndExecuteReader to get a SqlDataReader, which it stores in a private field. In an event handler for the PreRenderComplete event, which fires after the asynchronous operation completes but before the page is rendered, it binds the SqlDataReader to the Output GridView control. On the outside, the page looks like a normal synchronous page that uses a GridView to render the results of a database query. On the inside, this page is much more scalable because it doesn't tie up a thread-pool thread waiting for the query to return.

#### Calling Web Services Asynchronously

Another I/O-related task commonly performed by ASP.NET Web pages is calling to a Web service. Since Web service calls can take a long time to return, pages that execute them are ideal candidates for asynchronous processing.

Listing 2-22 shows one way to build an asynchronous page that calls a Web service. It uses the same AddOnPreRenderCompleteAsync mechanism featured in this section. The page’s Begin method launches an asynchronous Web service call by calling the Web service proxy’s asynchronous Begin method. The page’s End method caches in a private field a reference to the DataSet returned by the Web method, and the PreRenderComplete handler binds the DataSet to a GridView. For reference, the Web method targeted by the call is shown in the following code:

Listing 2-22. Calling a webservice from an ASP.NET page asynchronously

[WebMethod]

public DataSet GetTitles ()

{

string connect = WebConfigurationManager.ConnectionStrings

["PubsConnectionString"].ConnectionString;

SqlDataAdapter adapter = new SqlDataAdapter

("SELECT title\_id, title, price FROM titles", connect);

DataSet ds = new DataSet();

adapter.Fill(ds);

return ds;

}

Listing 2-23. AsyncWSInvoke1.aspx.cs

using System;

using System.Data;

using System.Configuration;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

public partial class AsyncWSInvoke1 : System.Web.UI.Page

{

private WS.PubsWebService \_ws;

private DataSet \_ds;

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

// Hook PreRenderComplete event for data binding

this.PreRenderComplete +=

new EventHandler(Page\_PreRenderComplete);

// Register async methods

AddOnPreRenderCompleteAsync(

new BeginEventHandler(BeginAsyncOperation),

new EndEventHandler(EndAsyncOperation)

);

}

}

IAsyncResult BeginAsyncOperation (object sender, EventArgs e,

AsyncCallback cb, object state)

{

\_ws = new WS.PubsWebService();

// Fix up URL for call to local VWD-hosted Web service

\_ws.Url = new Uri(Request.Url, "Pubs.asmx").ToString();

\_ws.UseDefaultCredentials = true;

return \_ws.BeginGetTitles (cb, state);

}

void EndAsyncOperation(IAsyncResult ar)

{

\_ds = \_ws.EndGetTitles(ar);

}

protected void Page\_PreRenderComplete(object sender, EventArgs e)

{

Output.DataSource = \_ds;

Output.DataBind();

}

public override void Dispose()

{

if (\_ws != null) \_ws.Dispose();

base.Dispose();

}

}

This is one way to do it, but it’s not the only way. The Web service proxy supports two mechanisms for placing asynchronous calls to Web services. One is the per-method Begin and End . The other is the new MethodAsync methods and MethodCompleted events. Even if the Begin/End pattern seems to be easy to read and adopt, because it is similar to the asynchronous pattern of the page’s handler, there are a few advantages to using the other pattern.

If a Web service has a method named Foo, then in addition to having methods named Foo, BeginFoo, and EndFoo, a Web service proxy includes a method named FooAsync and an event named FooCompleted. You can call Foo asynchronously by registering a handler for the FooCompleted event and calling FooAsync, like this:

proxy.FooCompleted += new FooCompletedEventHandler (OnFooCompleted);

proxy.FooAsync (...);

...

void OnFooCompleted (Object source, FooCompletedEventArgs e)

{

// Called when Foo completes

}

The asynchronous call begins when FooAsync completes, and a FooCompleted event fires then, causing your FooCompleted event handler to be called. Both the delegate wrapping the event handler (FooCompletedEventHandler) and the second parameter passed to it (FooCompletedEventArgs) are generated with the Web service proxy. You can access Foo’s return value through FooCompletedEventArgs.Result.

Listing 2-24 presents a code behind class that calls a Web service's GetTitles method asynchronously using the MethodAsync pattern. Functionally, this page is identical to the one shown in Listing 2-23. Internally, it’s quite different. AsyncWSInvoke2.aspx includes a Page directive with property Async set to true. But it doesn’t call AddOnPreRenderCompleteAsync; it registers a handler for GetTitlesCompleted events and calls GetTitlesAsync on the Web service proxy. ASP.NET still delays rendering the page until GetTitlesAsync completes. Under the hood, it uses an instance of System.Threading.SynchronizationContext, a class that receives notifications when the asynchronous calls begin and when they complete.

Listing 2-24. AsyncWSInvoke2.aspx.cs

using System;

using System.Data;

using System.Configuration;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

public partial class AsyncWSInvoke2 : System.Web.UI.Page

{

private WS.PubsWebService \_ws;

private DataSet \_ds;

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

// Hook PreRenderComplete event for data binding

this.PreRenderComplete +=

new EventHandler(Page\_PreRenderComplete);

// Call the Web service asynchronously

\_ws = new WS.PubsWebService();

\_ws.GetTitlesCompleted += new

WS.GetTitlesCompletedEventHandler(GetTitlesCompleted);

\_ws.Url = new Uri(Request.Url, "Pubs.asmx").ToString();

\_ws.UseDefaultCredentials = true;

\_ws.GetTitlesAsync();

}

}

void GetTitlesCompleted(Object source,

WS.GetTitlesCompletedEventArgs e)

{

\_ds = e.Result;

}

protected void Page\_PreRenderComplete(object sender, EventArgs e)

{

Output.DataSource = \_ds;

Output.DataBind();

}

public override void Dispose()

{

if (\_ws != null) \_ws.Dispose();

base.Dispose();

}

}

There are two advantages to using MethodAsync rather than AddOnPreRenderCompleteAsync to implement asynchronous pages. Firstly, MethodAsync supports forwarding impersonation, culture, and HttpContext.Current to the MethodCompleted event handler. Secondly, if the page makes multiple asynchronous calls and must delay rendering until all calls have been completed, using AddOnPreRenderCompleteAsync requires you to compose an IAsyncResult that remains unsignaled until all the calls are finished. This is not necessary in MethodAsync; simply place the calls, as many of them as you like, and the ASP.NET engine will delay the rendering phase until the final call returns.

### Asynchronous Tasks

MethodAsync is a convenient way to make multiple asynchronous Web service calls from an asynchronous page and delay the rendering phase until all the calls complete. Calling a Web service is not the only task that requires asynchronous programming. I/O-operations can be slow, too, especially when the server comes under pressure.

#### Register Asynchronous Tasks

The Page class introduces another method of facilitating asynchronous operations: RegisterAsyncTask. RegisterAsyncTask has several advantages. In addition to Begin and End methods, RegisterAsyncTask lets you register a timeout method that’s called if an asynchronous operation takes too long to complete. You can set the timeout declaratively by including an AsyncTimeout attribute in the page’s Page directive. For example, to set the timeout to ten seconds:

AsyncTimeout="10"

The second advantage is that you can call RegisterAsyncTask several times in one request to register several asynchronous operations. As with MethodAsync and all other solutions, ASP.NET delays rendering the page until all operations have been completed. Thirdly, you can use the fourth parameter of RegisterAsyncTask to pass the state to your Begin methods. Finally, RegisterAsyncTask forwards impersonation, culture, and HttpContext.Current to the End and Timeout methods for easy and transparent access. As mentioned earlier, the same is not true of an End method registered with AddOnPreRenderCompleteAsync.

In other respects, an asynchronous page that relies on RegisterAsyncTask is similar to one that relies on AddOnPreRenderCompleteAsync. The attribute in the page directive Async="true" is still required, and it still executes as normal through the PreRender event. Listing 2-25 demonstrates RegisterAsyncTask in a short example.

Listing 2-25. AsyncPageTask.aspx.cs

using System;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

using System.Net;

using System.IO;

using System.Text;

using System.Text.RegularExpressions;

public partial class AsyncPageTask : System.Web.UI.Page

{

private WebRequest \_request;

protected void Page\_Load(object sender, EventArgs e)

{

PageAsyncTask task = new PageAsyncTask(

new BeginEventHandler(BeginAsyncOperation),

new EndEventHandler(EndAsyncOperation),

new EndEventHandler(TimeoutAsyncOperation),

null

);

RegisterAsyncTask(task);

}

IAsyncResult BeginAsyncOperation(object sender, EventArgs e,

AsyncCallback cb, object state)

{

\_request = WebRequest.Create("http://msdn.microsoft.com");

return \_request.BeginGetResponse(cb, state);

}

void EndAsyncOperation(IAsyncResult ar)

{

string text;

using (WebResponse response = \_request.EndGetResponse(ar))

{

using (StreamReader reader =

new StreamReader(response.GetResponseStream()))

{

text = reader.ReadToEnd();

}

}

Regex regex = new Regex("href\\s\*=\\s\*\"([^\"]\*)\"",

RegexOptions.IgnoreCase);

MatchCollection matches = regex.Matches(text);

StringBuilder builder = new StringBuilder(1024);

foreach (Match match in matches)

{

builder.Append(match.Groups[1]);

builder.Append("<br/>");

}

Output.Text = builder.ToString();

}

void TimeoutAsyncOperation(IAsyncResult ar)

{

Output.Text = "Data temporarily unavailable";

}

}

The primary advantage of RegisterAsyncTask is that it allows asynchronous pages to fire off multiple asynchronous calls and delays rendering until all the calls have been completed. It works equally well for one asynchronous call, and it offers a timeout option that AddOnPreRenderCompleteAsync does not provide. If you build an asynchronous page which makes just one asynchronous call, you can use AddOnPreRenderCompleteAsync or RegisterAsyncTask. But for asynchronous pages that place two or more such calls, RegisterAsyncTask simplifies your life considerably.

#### Setting Properties

There are a number of relevant properties that can be set programmatically. (It was not possible to demonstrate all aspects in the sample code.) Please refer to the documentation to see what else is possible and how to modify its behavior. The intention of this chapter is to show ways to improve the overall performance of a web server through understanding the basics of thread pooling and request handling.

# Summary

In this chapter you got in-depth information about the internal request processing and how to tweak the worker process and the threading. You got an idea how the thread pool works and how to extend the behavior. Using asynchronous process might increase performance and handle high workload. A custom thread pool was introduced to demonstrate how to change internal parts of the processing pipeline transparently. In a anticipation to Chapter 3 you got a first look into asynchronous handlers and there usage.

1. Before the appearance of .NET 3.5 SP1, the settings can be found in the Registry under the path HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\ASP.NET\2.0.50727.0. A DWORD value named MaxConcurrentRequestsPerCpu limits the number of concurrent requests per CPU. The default is 12. [↑](#footnote-ref-1)