# **ASP.NET Thread Model**

I'd like to briefly explain how ASP.NET uses threads when hosted on IIS 7.0 and IIS 6.0, as well as the configuration changes that you can make to alter the defaults. Please take a quick look at the “Threading Explained” section in [Chapter 6](http://www.google.com/url?q=http%3A%2F%2Fmsdn2.microsoft.com%2Fen-us%2Flibrary%2Fms998549.aspx&sa=D&sntz=1&usg=AFQjCNFSf2jc0yAu6WJ5ck-Yf4tyFb8NhQ) of “Improving .NET Application Performance and Scalability”. Prior to v2.0 of the .NET Framework, it was necessary to tweak the processModel/maxWorkerThreads, processModel/maxIoThreads, httpRuntime/minFreeThreads, httpRuntime/minLocalRequestFreeThreads, and connectionManagement/maxconnection configuration. The v2.0 .NET Framework attempted to simplify this by adding a new [processModel/autoConfig](http://www.google.com/url?q=http%3A%2F%2Fmsdn2.microsoft.com%2Fen-us%2Flibrary%2F7w2sway1%2528vs.80%2529.aspx&sa=D&sntz=1&usg=AFQjCNGicoTdJ-fbq0KEfwMO98ONJaN7iw) configuration, which made the changes for you at runtime. And with the introduction of IIS 7.0 and the ASP.NET integrated pipeline, we've introduced yet another element to the mix, a registry key named **MaxConcurrentRequestsPerCPU**. Lets start with a discussion of how things worked on IIS 6.0 before discussing the changes made in IIS 7.0.

**When ASP.NET is hosted on IIS 6.0, the request is handed over to ASP.NET on an IIS I/O thread. ASP.NET immediately posts the request to the CLR ThreadPool and returns HSE\_STATUS\_PENDING to IIS. This frees up IIS threads, enabling IIS to serve other requests, such as static files. Posting the request to the CLR Threadpool also acts as a queue.** The CLR Threadpool automatically adjusts the number of threads according to the workload, so that if the requests are high throughput there will only be 1 or 2 threads per CPU, and if the requests are high latency there will be potentially far more concurrently executing requests than 1 or 2 per CPU. The queuing provided by the CLR Threadpool is very useful, because while the requests are in the queue there is only a very small amount of memory allocated for the request, and it is all native memory. It’s not until a thread picks up the request and begins to execute that we enter managed code and allocate managed memory.

The CLR Threadpool is not the only queue used by ASP.NET when hosted in IIS 6.0. There are also queues at the application level, within each AppDomain. If there is a lot of latency, the CLR Threadpool will grow and inject more active threads. At some point we would either run out of threads, not have enough threads left over for performing other tasks, or the memory associated with all the concurrently executing requests would be too much, so ASP.NET imposes a cap on the number of threads concurrently executing requests. This is controlled by the httpRuntime/minFreeThreads and httpRuntime/minLocalRequestFreeThreads settings. If the cap is exceeded, the request is queued in the application-level queue, and executed later when the concurrency falls back down below the limit. The performance of these application-level queues is really quite miserable. If you observe that the “ASP.NET Applications\Requests in Application Queue” performance counter is non-zero, you definitely have a performance problem. These queues were implemented to prevent thread exhaustion and contention related to web service requests. The problem was first described in [KB 821268](http://www.google.com/url?q=http%3A%2F%2Fsupport.microsoft.com%2Fkb%2F821268&sa=D&sntz=1&usg=AFQjCNEMPIUaSVQMHP67AJ5oBOEFdKT2AA), which I had published many years ago. The KB article has been re-written a few times since it was originally published, and I hope nothing has been lost during the translations.

For most usage scenarios, the changes recommended in the KB article are not necessary because v2.0 introduced [processModel/autoConfig](http://www.google.com/url?q=http%3A%2F%2Fmsdn2.microsoft.com%2Fen-us%2Flibrary%2F7w2sway1%2528vs.80%2529.aspx&sa=D&sntz=1&usg=AFQjCNGicoTdJ-fbq0KEfwMO98ONJaN7iw). However, the autoConfig setting may not work for everyone--it limits the number of concurrently executing requests per CPU to 12. An application with high latency may want to allow higher concurrency than this, in which case you can disable autoConfig and make the changes yourself. If you do allow higher concurrency, keep an eye on your working set. I believe the default works for about 90% of the applications out there. I do wish we had the foresight to name that setting maxConcurrentRequestsPerCPU, and allow it to be used to control concurrency, since that would be much easier to configure. I guess this is just another example of when business was just a little bit faster than the speed of thought.

When ASP.NET is hosted on IIS 7.0 in integrated mode, the use of threads is a bit different. First of all, the application-level queues are no more. Their performance was always really bad, there was no hope in fixing this, and so we got rid of them. But perhaps the biggest difference is that in IIS 6.0, or ISAPI mode, ASP.NET restricts the number of **threads** concurrently executing requests, but in IIS 7.0 integrated mode, ASP.NET restricts the number of concurrently executing **requests**. The difference only matters when the requests are asynchronous (the request either has an asynchronous handler or a module in the pipeline completes asynchronously). Obviously if the reqeusts are synchronous, then the number of concurrently executing **requests** is the same as the number of **threads** concurrently executing requests, but if the requests are asynchronous then these two numbers can be quite different as you could have far more reqeusts than threads. So how do things work for IIS 7.0 integrated mode? Similar to IIS 6.0, the request is still handed over to ASP.NET on an IIS I/O thread. And ASP.NET immediately posts the request to the CLR Threadpool and returns pending. We found this thread switch was still necessary to maintain optimal performance for static file requests. So although you will take a performance hit if you’re only executing ASP.NET requests, if you have a mix of dynamic and static files, as we see with many large corporate workloads, this thread switch will actually free up threads for retrieving the static files. Finally, once the request is picked up by a thread from the CLR Threadpool, we check to see how many requests are currently executing. If the count is too high, the request is queued in a global (process-wide) queue. This global, native queue performs much better than the application-level queues used when we’re running in ISAPI mode (same as on IIS 6.0). There is very little memory associated with a queued request, and we have not entered managed code yet so there is no managed memory associated with it. And we respect the FIFO aspect of a queue, something we didn’t do with the application-level queues--if there was more than one application, there was no simple way to globally manage the individual queues. We did however have a difficult time trying to come up with a good configuration story for the IIS 7.0 changes.

When I discuss how to configure thread usage for ASP.NET/IIS 7.0 integrated mode, please remember that we have a lot of pre-existing code and configuration, and you can’t just create something new the way you would like to without introducing backward compatibility issues. In this new mode, the CLR Threadpool is still controlled by the processModel configuration settings (autoConfig, maxWorkerThreads, maxIoThreads, minWorkerThreads, and minIoThreads). And autoConfig is still enabled, but its modifications to httpRuntime/minFreeThreads and httpRuntime/minLocalRequestFreeThreads do nothing, since the application-level queues do not exist. Perhaps we should have tried to use them to configure the global (process-wide) queue limits, but they have application scope (httpRuntime configuration is application specific), not process scope, not to mention being too difficult to understand. And because of some issues with using the configuration system that I won’t go into right now, we decided to use a registry key to control concurrency. So for IIS 7.0 integrated mode, a DWORD named **MaxConcurrentRequestsPerCPU** within HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\ASP.NET\2.0.50727.0 determines the number of concurrent **requests** per CPU. By default, it does not exist and the number of requests per CPU is limited to 12. If you’re curious to see how much faster ASP.NET requests execute without the thread switch, you can set the value to 0. This will cause the request to execute on the IIS I/O thread, without switching to a CLR Threadpool thread. I don’t recommend this primarily because dynamic requests take a long time to execute relative to static requests, and I believe the overall performance of the system is better with the thread switch. However, and this is important, if your application consists of primarily or entirely asynchronous requests, it is likely that the default **MaxConcurrentReqeustsPerCPU** limit of 12 may be too restrictive for you, especially if the requests are very long running. In this case, I do recommend either setting **MaxConcurrentRequestsPerCPU** to 0 or increasing it to allow the concurrency that you would like. There's nothing wrong with setting MaxConcurrentRequestsPerCPU to a very high number, except that this will allow more requests to enter into managed code and execute concurrently, and this will consume more memory. In fact, in v4.0, we have changed the default for MaxConcurrentRequestsPerCPU to 5000. There's nothing special about 5000, other than it is a very large number, and will therefore allow plenty of async requests to execute concurrently. The CLR ThreadPool will still do a great job maintaining the number of threads in the ThreadPool, so there should be no concern about this adversly impacting synchronous requests. I know there are people out using ASP.NET 2.0 and developing [Comet](http://www.google.com/url?q=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FComet_%2528programming%2529&sa=D&sntz=1&usg=AFQjCNGYFbw7rm3ne5bQfCusbm4q9LIsKw) or Comet-like applications on WS08 x64 servers, and they set MaxConcurrentRequestsPerCPU to 5000 and increase the HTTP.sys kernel queue limit.

As a final remark, please note that the processModel/requestQueueLimit configuration limits the maximum number of requests in the ASP.NET system for both IIS 6.0 and IIS 7.0. This number is exposed by the "ASP.NET/Requests Current" performance counter, and when it exceeds the limit (default is 5000) we reject requests with a 503 status (Server Too Busy).

-Thomas

**UPDATE (Aug-18-2008)**: .NET Framework v3.5 SP1 released earlier this week and it includes an update to the v2.0 binaries that supports configuring IIS application pools via the aspnet.config file. This configuration applies to integrated mode only (Classic/ISAPI mode ignores these settings). The new aspnet.config config section with default values is:

<system.web>

<applicationPool maxConcurrentRequestsPerCPU="12" maxConcurrentThreadsPerCPU="0" requestQueueLimit="5000"/>

</system.web>

There is a corresponding IIS change in Windows7 which allows different aspnet.config files to be specified for each application pool (this change has not been ported to WS08). With this, you can configure each application pool differently. The maxConcurrentRequestsPerCPU setting is the same as the registry key described above, except that the setting in aspnet.config will override the registry key value. The maxConcurrentThreadsPerCPU setting is new, and allows concurrency to be gated by the number of threads, similar to the way it was done in Classic/ISAPI mode. By default maxConcurrentThreadsPerCPU is disabled (has a value of 0), in favor of gating concurrency by the number of requests, primarily because maxConcurrentRequestsPerCPU performs better (gating the number of threads is more complicated/costly to implement). Normally you'll use request gating, but you now have the option of disabling it (set maxConccurrentRequestsPerCPU=0) and enabling maxConccurentThreadsPerCPU instead. You can also enable both request and thread gating at the same time, and ASP.NET will ensure both requirements are met. The requestQueueLimit setting is the same as processModel/requestQueueLimit, except that the setting in aspnet.config will override the machine.config setting. All of this may be a little confusing, but for nearly everyone, my recommendation is that for ASP.NET 2.0 you should use the same settings as the defaults in ASP.NET v4.0; that is, set maxConcurrentRequestsPerCPU = "5000" and maxConcurrentThreadsPerCPU="0".

-Thomas