C#

IOCP Thread Pooling in C# - Part I
Contributed by William Kennedy
2003-04-08

This is the first part of William's two part series on thread pooling in C#. By importing a dll file for IOCP thread support.

When building server based applications in C#, it is important to have the ability to create thread pools. Thread pools allow our server to queue and perform work in the most efficient and scalable way possible. Without thread pooling we are left with two options.

The first option is to perform all of the work on a single thread. The second option is to spawn a thread every time some piece of work needs to be done. For this article, work is defined as an event that requires the processing of code. Work may or may not be associated with data, and it is our job to process all of the work our server receives in the most efficient and fastest way possible.

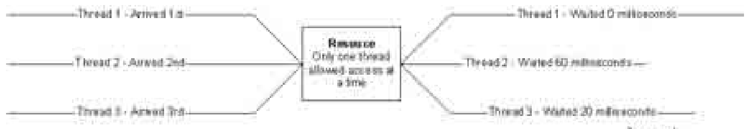
As a general rule, if you can accomplish all of the work required with a single thread, then only use a single thread. Having multiple threads performing work at the same time does not necessarily mean our application is getting more work done, or getting work done faster. This is true for many reasons.

For example, if you spawn multiple threads which attempt to access the same resource bound to a synchronization object, like a monitor object, these threads will serialize and fall in line waiting for the resource to become available. As each thread tries to access the resource, it has the potential to block, and wait for the thread that owns the resource to release the resource.

At that point, these waiting threads are put to sleep, and not getting any work done. In fact, these waiting threads have caused more work for the operating system to perform. Now the operating system must task another thread to perform work, and then determine which thread, waiting for the resource, may access the resource next, once it becomes available.

If the threads that need to perform work are sleeping, because they are waiting for the resource to become available, we have actually created a performance problem. In this case it would be more efficient to queue up this work and have a single thread process the queue.

Threads that start waiting for a resource before other threads, are not guaranteed to be given the resource first. In diagram A, thread 1 requests access to the resource before thread 2, and thread 2 requests access to the resource before thread 3. The operating system however decides to give the resource to thread 1 first, then thread 3, and then thread 2. This scenario causes work to be performed in an undetermined order. The possible issues are endless when dealing with multi-threaded applications.


Thread 1 - Arrived 1st
Thread 2 - Arrived 2nd
Thread 3 - Arrived 3rd
Resource
Only one thread allowed access at a time
Thread 1 - Waited 0 milliseconds
Thread 2 - Waited 60 milliseconds
Thread 3 - Waited 20 milliseconds
Diagram A

If work received can be performed independent of each other, we could always spawn a thread for processing that piece of work. The problem here is that an operating system like Windows has severe performance problems when a large number of threads are created or running at the same time, waiting to have access to the CPU.

The Windows operating system needs to manage all of these threads, and compared to the UNIX operating system, it just doesn't hold up. If large amounts of work are issued to the server, this model will most likely cause the Windows operating system to become overloaded. System performance will degrade drastically.<>

This article is a case study comparing thread performance between Windows NT and Solaris.

http://www.usenix.org/publications/library/proceedings/usenix-nt98/full_papers/zabatta/zabatta_html/zabatta.html

In the .NET framework, the "System.Threading" namespace has a ThreadPool class. Unfortunately, it is a static class and therefore our server can only have a single thread pool. This isn't the only issue. The ThreadPool class does not allow us to set the concurrency level of the thread pool.

The concurrency level is the most important setting when configuring a thread pool. The concurrency level defines how many threads in the pool may be in an "active state" at the same time. If we set this parameter correctly, we will have the most efficient, performance enhanced thread pool for the work being processed.

Imagine we have a thread pool with 4 threads and a concurrency level of 1. Then, three pieces of work are queued up for processing in the pool. Since the concurrency level for the thread pool is 1, only a single thread from the pool is activated and given work from the queue. Even though there are two pieces of work queued up, no other threads are activated. This is because the concurrency level is set to 1. If the concurrency level was set to 2, then another thread would have been activated immediately and given work from the queue. In diagram B we have thread 1 running and all of the other threads sleeping with two pieces of work queued.

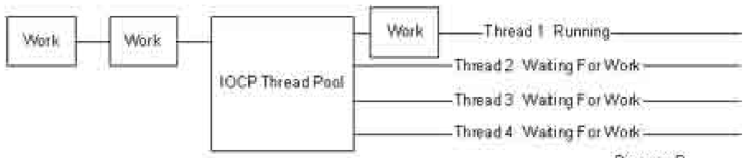

Work
Work
IOCP Thread Pool
Work
Thread 1 Running
Thread 2 Waiting For Work
Thread 3 Waiting For Work
Thread 4 Waiting For Work

Diagram B

So the question exists, why have more than 1 thread in the pool if the concurrency level is set to 1? If thread 1 in diagram B ever goes to sleep before it completes its work, another thread from the pool will be activated. When thread 1 goes to sleep, there are 0 threads "active" in the pool and it is ok to activate a new thread based on the concurrency level. In diagram C, we now have thread 1 sleeping and thread 4 running with one piece of work queued.

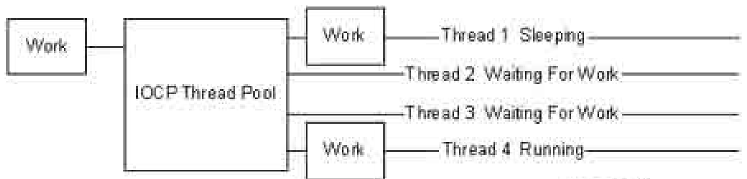


Diagram C

Eventually, thread 1 will wake up, and it is possible for thread 4 to still be active. We have 2 threads active in the pool, even though the concurrency level is set to 1. In diagram D, we now have thread 1 and thread 4 running and one piece of work still queued.

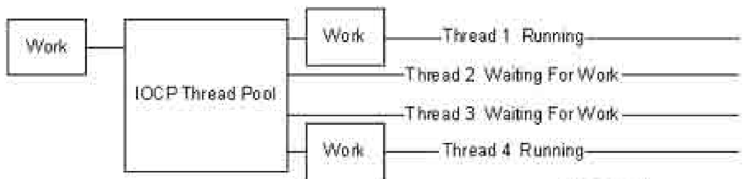


Diagram D

The last piece of work in the queue will need to wait until both threads return to a sleeping state. This is because the concurrency level is set to 1. As we can see, even though the concurrency level restricts the number of active threads in the pool at any given time, we could have more active threads then the concurrency level allows. It all depends on the state of the threads in the pool and how fast the threads can complete the work they are processing.

A good rule of thumb is to set the concurrency level to match the number of CPU's in the system. If the machine our server is running on only has one CPU, then only one thread can be executing at any given time. It will require a task swap to have another thread get CPU time. We want to reduce the number of active threads at any given time to maximize performance. This also leads to scalability. As the number of CPU's increase, we can increase the concurrency level because there is a CPU to execute that thread. This is a general rule and is always a good starting point for configuring our thread pools.

The bottom line is, if the CPU is available, and there is work to perform, activate a thread. If the CPU is not available, do not activate a thread. One other thing, we need to be careful that we don't cause a situation where the threads in the pool are constantly being put to sleep for long periods of time during the processing of work. This may cause all of the threads in the pool to constantly be in an active state, defeating the efficiency of the pool and the performance of the server.

The remaining scope of this article will show you how to add IOCP thread pools to your C# server based applications. How to configure the thread pools for your specific application will not be covered. It is suggested to use the general rules as discussed.

System Requirements

A basic understanding of C# is required to follow through the examples and the classes. Basic concepts of type, properties, threading, synchronization, and delegates are required.

Defining the Problem

IOCP thread support has not been made available to C# developers through the "System.Threading" namespace. We need to access the Win32 API calls from the Kernel32.dll. This requires us to write unsafe code. This is really not a problem, but something that needs to be discussed. Let's take a look at the Win32 API calls we need to implement an IOCP thread pool.

```
[DllImport("Kernel32", CharSet=CharSet.Auto)]
private unsafe static extern UInt32 CreateIoCompletionPort(UInt32 hFile, UInt32 hExistingCompletionPort, UInt32* puiCompletionKey, UInt32 uiNumberOfConcurrentThreads);
```

This Win32 API call is used to create an IOCP thread pool. The first argument will always be set to INVALID_HANDLE_VALUE, which is 0xFFFFFFFF. This tells the operating system this IOCP thread pool is not linked to a device. The second argument will always be set to 0. There is no existing IOCP thread pool because we are creating this for the first time. The third argument will always be null.

We do not require a key because we have not associated this IOCP thread pool with a device. The last argument is the important argument. Here we define the concurrency level of the thread pool. If we pass a 0 for this argument the operating system will set the concurrency level to match the number of CPU's in the machine.

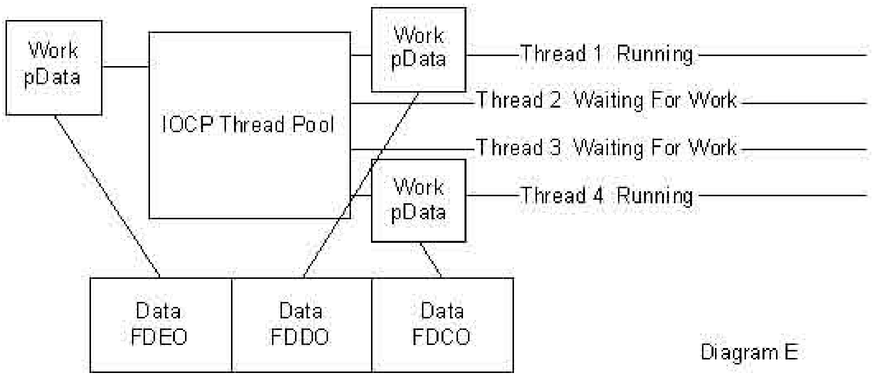
This option gives us our best chance to be scalable and take advantage of the number of CPU's present in the machine. This API call will return a handle to the newly created IOCP thread pool. If the API call fails, it will return null.

```
[DllImport("Kernel32", CharSet=CharSet.Auto)]
private unsafe static extern Boolean CloseHandle(UInt32 hObject);
```

This Win32 API call is used to close our thread pool. The only argument is the handle to the IOCP thread pool. This API call will return TRUE or FALSE if the handle can not be closed.

```
[DllImport("Kernel32", CharSet=CharSet.Auto)]
private unsafe static extern Boolean PostQueuedCompletionStatus(UInt32 hCompletionPort, UInt32 uiSizeOfArgument, UInt32* puiUserArg, OVERLAPPED* pOverlapped);
```

This Win32 API call is used to post work in the IOCP thread pool queue. Other threads in our application will make this Win32 API call. The first argument is the handle to the IOCP thread pool. The second argument is the size of the data we are posting to the queue. The third argument is a value or a reference to an object or data structure we are posting to the queue. The last argument will always be null. The following diagram shows how the data is associated with the posted work.



In diagram E, we have two threads actively processing posted work and one piece of work on the queue waiting for its data to be processed. The thing to note here is that each piece of work was given a reference to its specific data. I am calling this variable pData to help describe what is happening in the IOCP thread pool. The actual name or structure of this variable is undocumented.

When we make this API call in a C++ application, we can pass the address of any object in memory we wish, as in diagram E. In C#, we don't have the same luxury because of the managed heap. The managed heap is a contiguous region of address space that contains all of the memory allocated for reference variables. The heap maintains a pointer that indicates where the next object is to be allocated, and all allocations are contiguous from that point. This is much different from the C-runtime heap.

The C-runtime heap uses a link list of data structures to reference available memory blocks. For the C-runtime heap to allocate memory, it must walk through the link list until a large enough block of free memory is found. Then the free block of memory must be resized, and the link list adjusted.

If objects are allocated consecutively in a C++ application, those objects could be allocated anywhere on the heap. This can never happen with the managed heap. Objects that are allocated consecutively in a C# application will always be allocated consecutively on the managed heap. The catch is that the managed heap must be compacted to guarantee the heap does not run out of memory. That is the job of garbage collection.

For more information on garbage collection, try these links:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconautomaticmemorymanagement.asp

http://msdn.microsoft.com/msdnmag/issues/1100/GCI/default.aspx

http://msdn.microsoft.com/msdnmag/issues/1200/GCI2/default.aspx

In diagram F, we have allocated four objects on the managed heap. Imagine that the managed heap has allocated memory for these objects at address FDEO, FDDO, FDCO, and FDBO. This would mean the value of pClass1 is FDEO, the value of pClass2 is FDDO, the value of pClass3 is FDCO, and the value of pClass4 is FDBO.

```
MyClass pClass1 = new MyClass();
MyClass pClass2 = new MyClass();
MyClass pClass3 = new MyClass();
MyClass pClass4 = new MyClass();
```

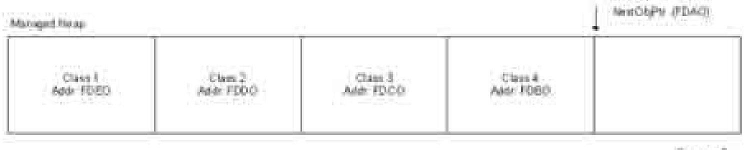


Diagram F

Now we write the following code.

```
pClass2 = null;
```

Diagram G shows what happens to the managed heap after garbage collection takes place and the managed heap is compacted.

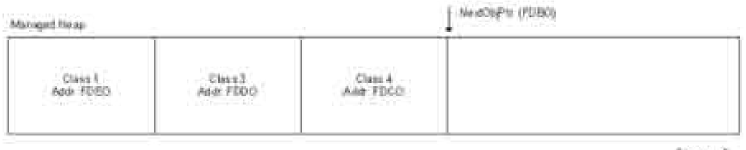


Diagram G

The Class 2 object has been removed from the managed heap and the Class3 and Class 4 objects have been moved. Now the value of pClass3 is FDDO and the value of pClass4 is FDCO. The value that the pointer points to has changed. The garbage collection process changes the values of all reference variables to make sure they are pointing to the correct objects after the managed heap is compacted.

So what does this mean for our IOCP thread pool implementation? If we pass the reference of a managed object as the data for the work, there is a chance the reference is no longer valid when a thread in the pool is chosen to work on the data.

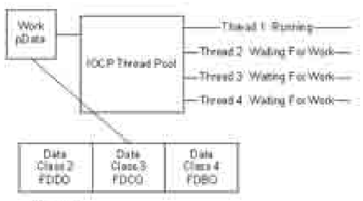


Diagram H

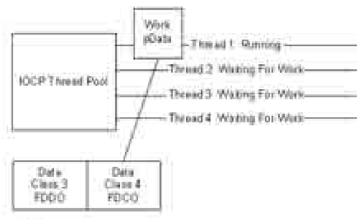


Diagram I

In diagram H, we have passed a reference to the Class 3 object as the data for the work posted to the IOCP thread pool. This object is at address FDCO. Before the work is given to thread 1, the Class 2 object is marked for deletion. Then the garbage collection process runs, and the managed heap is compacted. Now in diagram I, the work has been given to thread 1 for processing. The value of pData is still FDCO, but Class 3 is no longer at address FDCO, it is at address FDDO. The thread will perform the work, but using Class 4 instead of Class 3.

The garbage collection process can not change the value of pData, as it does with other variables, because this variable is not a managed variable. It is a variable owned by the IOCP thread pool and exists outside the scope of the CLR. The garbage collector has no knowledge of this variable or access to this variable. The variable is set during the unsafe call to PostQueuedCompletionStatus.

Unfortunately, pinning the objects we want to pass as the data for the work posted to the IOCP thread pool is not a possible solution. Pinning provides the ability to prevent an object from being moved on the manage heap during the garbage collection process. We can not pin these objects because there is no way to pin an object in one thread and unpin the object in a different thread. To pin an object, we need to use the fixed keyword. This keyword can only be used in the context of a single method. Here is a quick example of pinning.

```
Int32 iArray = new Int32[5] {12, 34, 56, 78, 90};
unsafe
{
 fixed (Int32* piArray = iArray)
 {
  // Do Something
 }
}
```

The safest thing we can do is pass a value to the IOCP thread pool. This value could be the index from a managed array, containing a reference to an object on the managed heap. If the garbage collection process does compact the heap, the index values of the array will not change. In diagram J and K, we can see one way to properly pass data for the work posted to the IOCP thread pool. After the garbage collection process compacts the heap, the values of pData change, but the index positions to the pData variables do not change.

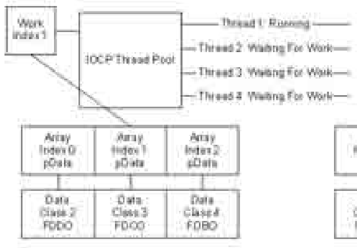


Diagram J

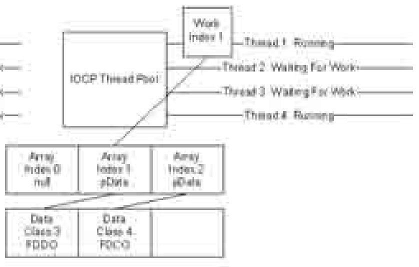


Diagram K

```
[DllImport("Kernel32", CharSet=CharSet.Auto)]
private static extern Boolean GetQueuedCompletionStatus(UInt32 hCompletionPort, UInt32* pSizeOfArgument, UInt32*
puiUserArg, OVERLAPPED** ppOverlapped, UInt32 uiMilliseconds);
```

The final Win32 API call is used to add threads to the IOCP thread pool. Any thread that makes this Win32 API call will become part of the IOCP thread pool. This is a blocking call and the method will return when the IOCP thread pool chooses the thread to perform work.

The first argument is the handle to the IOCP thread pool. The second argument is the size of the data associated with the work. This value was provided when the work was posted. The third argument is the data value or data reference associated with the work. This value was provided when the work was posted. The forth argument is the address to a pointer of type OVERLAPPED.

This address is returned after the call. The last argument is the time in milliseconds the thread should wait to be activated to perform work. We will always pass INFINITE or 0xFFFFFFFF.

These are the Win32 API calls we need to add IOCP thread pool support to our C# server based applications. We need to encapsulate these Win32 API calls using .NET threads and minimize the sections of unsafe code. We need to prevent the application developer from passing a reference variable into the IOCP thread pool, by restricting them to passing only integer values.

In part II of this article, we will build a class that encapsulates a single IOCP thread pool. The application developer will be able to instantiate as many thread pools as he wishes.

During construction, the application developer will be able to: set the concurrency level of the thread pool, set the minimum and maximum number of threads in the pool, and will be able to provide a method to be called when work posted to the thread pool needs to be processed. The application developer will also be able to post work with data into the IOCP thread pool.