C#

IOCP Thread Pooling in C# - Part II
Contributed by William Kennedy
2003-04-11

In part 2, William will continue to explain how the create a class that will handle threads using a IOCP Thread Pool.

Defining the Solution

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.

Component Design and Coding

Start by adding a new class to your C# project. Remove all of the code provided by the Visual Studio .NET wizard. Then add the following namespaces. The System.Runtime.InteropServices namespace is required to access the Win32 API methods from the Kernel32 DLL.

```
using System;
using System.Threading;
using System.Runtime.InteropServices;
```

Don't forget to change the project properties to allow unsafe code blocks. This can be done by opening the project properties and selecting the Configuration Properties. Under the Build / Code Generation section you will see "Allow unsafe code blocks". Set this to true.

Next add the namespace. You will notice that I have defined a two level namespace. This is great when you are building a class library with many different classes.

```
namespace Continuum.Threading
{
```

The PostQueuedCompletionStatus and GetQueuedCompletionStats Win32 API methods both require a pointer to the Win32 OVERLAPPED structure. Because this structure will be used by the unsafe Win32 API call, we need to make sure the structure is aligned exactly the same way it would be in our C++ applications. This can be accomplished by using the StructLayout attribute. By setting the attribute to "LayoutKind.Sequential", the structure will be aligned based on the same rules as the C++ compiler.

The structure requires members that are pointers. The only way to add pointers to this structure is to use the unsafe keyword. We can still use the FCL types when defining the structure. This is very important because we can make sure the structure is identical to the C++ version.

```
// Structures
  //=========================================
  /// <summary> This is the WIN32 OVERLAPPED structure </summary>
  [StructLayout(LayoutKind.Sequential, CharSet=CharSet.Auto)]
  public unsafe struct OVERLAPPED
  {
    UInt32* ulpInternal;
    UInt32* ulpInternalHigh;
    Int32   lOffset;
    Int32   lOffsetHigh;
    UInt32  hEvent;
  }
```

Now it is time to define the IOCP thread pool class. I am using the keyword sealed in the definition of this class. The sealed keyword tells the compiler that this class can not be inherited. If you know there is no reason for a class to be inherited, use the sealed keyword. Certain run-time optimizations are enabled for the class when the sealed keyword is used.

```
// Classes
  //=========================================
  /// <summary> This class provides the ability to create a thread pool to manage work.  The
  ///           class abstracts the Win32 IOCompletionPort API so it requires the use of
  ///           unmanaged code.  Unfortunately the .NET framework does not provide this functionality
</summary>
  public sealed class IOCPThreadPool
  {
```

The first section of the IOCP thread pool class is the Win32 function prototypes. These are the same ones described earlier.

```
  // Win32 Function Prototypes
    /// <summary> Win32Func: Create an IO Completion Port Thread Pool </summary>
    [DllImport("Kernel32", CharSet=CharSet.Auto)]
    private unsafe static extern UInt32 CreateIoCompletionPort(UInt32 hFile, UInt32
hExistingCompletionPort, UInt32* puiCompletionKey, UInt32 uiNumberOfConcurrentThreads);

    /// <summary> Win32Func: Closes an IO Completion Port Thread Pool </summary>
    [DllImport("Kernel32", CharSet=CharSet.Auto)]
    private unsafe static extern Boolean CloseHandle(UInt32 hObject);

    /// <summary> Win32Func: Posts a context based event into an IO Completion Port Thread Pool
</summary>
    [DllImport("Kernel32", CharSet=CharSet.Auto)]
    private unsafe static extern Boolean PostQueuedCompletionStatus(UInt32 hCompletionPort,
UInt32 uiSizeOfArgument, UInt32* puiUserArg, OVERLAPPED* pOverlapped);

    /// <summary> Win32Func: Waits on a context based event from an IO Completion Port Thread
Pool.
    ///           All threads in the pool wait in this Win32 Function </summary>
    [DllImport("Kernel32", CharSet=CharSet.Auto)]
    private unsafe static extern Boolean GetQueuedCompletionStatus(UInt32 hCompletionPort,
UInt32* pSizeOfArgument, UInt32* puiUserArg, OVERLAPPED** ppOverlapped, UInt32
uiMilliseconds);
```

The next section is the constants section. Here we need to define the Win32 constants required for the Win32 API calls we are going to make later.

```
  // Constants
    /// <summary> SimTypeConst: This represents the Win32 Invalid Handle Value Macro
</summary>
    private const UInt32 INVALID_HANDLE_VALUE = 0xffffffff;

    /// <summary> SimTypeConst: This represents the Win32 INFINITE Macro </summary>
    private const UInt32 INIFINITE = 0xffffffff;

    /// <summary> SimTypeConst: This tells the IOCP Function to shutdown </summary>
    private const Int32 SHUTDOWN_IOCPTHREAD = 0x7fffffff;
```

The delegate function type section is where we define any delegate functions. We need one delegate function type to define the signature of the function we will call when work needs to be processed.

```
  // Delegate Function Types
    /// <summary> DelType: This is the type of user function to be supplied for the thread pool
</summary>
    public delegate void USER_FUNCTION(Int32 iValue);
```

These private properties are required to maintain the application developer's settings. The most interesting property is the GetUserFunction property. This property contains a reference to a method supplied by the application developer. We will use this property to call the application developers method.

```
  // Private Properties
    private UInt32 m_hHandle;
      /// <summary> SimType: Contains the IO Completion Port Thread Pool handle for this instance
</summary>
      private UInt32 GetHandle { get { return m_hHandle; } set { m_hHandle = value; } }

    private Int32 m_uiMaxConcurrency;
      /// <summary> SimType: The maximum number of threads that may be running at the same
time </summary>
      private Int32 GetMaxConcurrency { get { return m_uiMaxConcurrency; } set {
m_uiMaxConcurrency = value; } }

    private Int32 m_iMinThreadsInPool;
      /// <summary> SimType: The minimal number of threads the thread pool maintains
</summary>
      private Int32 GetMinThreadsInPool { get { return m_iMinThreadsInPool; } set {
m_iMinThreadsInPool = value; } }

    private Int32 m_iMaxThreadsInPool;
      /// <summary> SimType: The maximum number of threads the thread pool maintains
</summary>
      private Int32 GetMaxThreadsInPool { get { return m_iMaxThreadsInPool; } set {
m_iMaxThreadsInPool = value; } }

    private Object m_pCriticalSection;
      /// <summary> RefType: A serialization object to protect the class state </summary>
      private Object GetCriticalSection { get { return m_pCriticalSection; } set { m_pCriticalSection
= value; } }

    private USER_FUNCTION m_pfnUserFunction;
```

```
    /// <summary> DelType: A reference to a user specified function to be call by the thread pool
</summary>
    private USER_FUNCTION GetUserFunction { get { return m_pfnUserFunction; } set {
m_pfnUserFunction = value; } }

  private Boolean m_bDisposeFlag;
    /// <summary> SimType: Flag to indicate if the class is disposing </summary>
    private Boolean IsDisposed { get { return m_bDisposeFlag; } set { m_bDisposeFlag = value; }
}
```

These public properties are used to determine if new threads need to be added to the thread pool. These properties also provide statistical data about the thread pool. Here we use the Interlocked class to provide serialization when we increment or decrement these properties. This is the least expensive way to perform serialization.

```
 // Public Properties
  private Int32 m_iCurThreadsInPool;
    /// <summary> SimType: The current number of threads in the thread pool </summary>
    public Int32 GetCurThreadsInPool { get { return m_iCurThreadsInPool; } set {
m_iCurThreadsInPool = value; } }
    /// <summary> SimType: Increment current number of threads in the thread pool
</summary>
    private Int32 IncCurThreadsInPool() { return Interlocked.Increment(ref m_iCurThreadsInPool);
}
    /// <summary> SimType: Decrement current number of threads in the thread pool
</summary>
    private Int32 DecCurThreadsInPool() { return Interlocked.Decrement(ref
m_iCurThreadsInPool); }
  private Int32 m_iActThreadsInPool;
    /// <summary> SimType: The current number of active threads in the thread pool
</summary>
    public Int32 GetActThreadsInPool { get { return m_iActThreadsInPool; } set {
m_iActThreadsInPool = value; } }
    /// <summary> SimType: Increment current number of active threads in the thread pool
</summary>
    private Int32 IncActThreadsInPool() { return Interlocked.Increment(ref m_iActThreadsInPool);
}
    /// <summary> SimType: Decrement current number of active threads in the thread pool
</summary>
    private Int32 DecActThreadsInPool() { return Interlocked.Decrement(ref m_iActThreadsInPool);
}
  private Int32 m_iCurWorkInPool;
    /// <summary> SimType: The current number of Work posted in the thread pool </summary>
    public Int32 GetCurWorkInPool { get { return m_iCurWorkInPool; } set { m_iCurWorkInPool =
value; } }
    /// <summary> SimType: Increment current number of Work posted in the thread pool
</summary>
    private Int32 IncCurWorkInPool() { return Interlocked.Increment(ref m_iCurWorkInPool); }
    /// <summary> SimType: Decrement current number of Work posted in the thread pool
</summary>
    private Int32 DecCurWorkInPool() { return Interlocked.Decrement(ref m_iCurWorkInPool); }
```

The constructor method does several things. The class state is initialized and then the IOCP thread pool is created with a call to the CreateIoCompletionPort method. Notice the method call is within the scope of the unsafe keyword. This is required because we are passing pointers into the Win32 API call.

The last thing we do is create the minimal number of threads specified by the application developer. Notice we use the .NET threading classes to create the threads. We do not need to use the unsafe CreateThread method. One might think we need to because these threads will be calling the GetQueuedCompletionStatus Win32 API method.

```
   // Constructor, Finalize, and Dispose
   //*******************************************
   /// <summary> Constructor </summary>
   /// <param name = "iMaxConcurrency"> SimType: Max number of running threads allowed
</param>
   /// <param name = "iMinThreadsInPool"> SimType: Min number of threads in the pool
</param>
   /// <param name = "iMaxThreadsInPool"> SimType: Max number of threads in the pool
</param>
   /// <param name = "pfnUserFunction"> DelType: Reference to a function to call to perform work
</param>
   /// <exception cref = "Exception"> Unhandled Exception </exception>
   public IOCPThreadPool(Int32 iMaxConcurrency, Int32 iMinThreadsInPool, Int32
iMaxThreadsInPool, USER_FUNCTION pfnUserFunction)
   {
     try
     {
       // Set initial class state
       GetMaxConcurrency   = iMaxConcurrency;
       GetMinThreadsInPool = iMinThreadsInPool;
       GetMaxThreadsInPool = iMaxThreadsInPool;
       GetUserFunction     = pfnUserFunction;
       // Init the thread counters
       GetCurThreadsInPool = 0;
       GetActThreadsInPool = 0;
       GetCurWorkInPool    = 0;
       // Initialize the Monitor Object
       GetCriticalSection = new Object();
       // Set the disposing flag to false
       IsDisposed = false;
       unsafe
       {
         // Create an IO Completion Port for Thread Pool use
         GetHandle = CreateIoCompletionPort(INVALID_HANDLE_VALUE, 0, null, (UInt32)
GetMaxConcurrency);
       }
       // Test to make sure the IO Completion Port was created
       if (GetHandle == 0)
         throw new Exception("Unable To Create IO Completion Port");
       // Allocate and start the Minimum number of threads specified
       Int32 iStartingCount = GetCurThreadsInPool;
       ThreadStart tsThread = new ThreadStart(IOCPFunction);
       for (Int32 iThread = 0; iThread < GetMinThreadsInPool; ++iThread)
       {
         // Create a thread and start it
         Thread thThread = new Thread(tsThread);
         thThread.Name = "IOCP " + thThread.GetHashCode();
         thThread.Start();
         // Increment the thread pool count
         IncCurThreadsInPool();
```

```
      }
    }
    catch
    {
      throw new Exception("Unhandled Exception");
    }
  }
```

The finalize method is only required to guarantee the IOCP thread pool handle is closed. As a general rule, if a class allocates a resource outside the scope of the .NET framework, a finalize method is required else do not add a finalize method. A finalize method will cause the garbage collection process to spend more time trying to release the memory for the object.

```
  //*********************************************
  /// <summary> Finalize called by the GC </summary>
  ~IOCPThreadPool()
  {
    if (!IsDisposed)
      Dispose();
  }
```

The dispose method will not return until all of the threads in the pool have been terminated. We can't use the Abort method to kill the threads in the pool because any thread blocked, via the call to the GetQueuedCompletionStatus Win32 API method, will not respond to the Abort message.

The GetQueuedCompletionStatus Win32 API method will cause the thread to run outside the scope of the CLR and the .NET framework will lose access to the thread. So what we do is post work into the IOCP thread pool. We pass the SHUTDOWN_IOCPTHREAD data when we post the work. This will tell the thread to terminate. Then, we wait in a spin lock, until all of the threads have terminated. The last thing is to close the IOCP thread pool.

```
  //*********************************************
  /// <summary> Called when the object will be shutdown. This
  ///         function will wait for all of the work to be completed
  ///         inside the queue before completing </summary>
  public void Dispose()
  {
    try
    {
      // Flag that we are disposing this object
      IsDisposed = true;
      // Get the current number of threads in the pool
      Int32 iCurThreadsInPool = GetCurThreadsInPool;
      // Shutdown all thread in the pool
      for (Int32 iThread = 0; iThread < iCurThreadsInPool; ++iThread)
      {
        unsafe
        {
          bool bret = PostQueuedCompletionStatus(GetHandle, 4, (UInt32*)
SHUTDOWN_IOCPTHREAD, null);
        }
      }
      // Wait here until all the threads are gone
      while (GetCurThreadsInPool != 0) Thread.Sleep(100);
      unsafe
      {
```

```
        // Close the IOCP Handle
        CloseHandle(GetHandle);
      }
    }
    catch
    {
    }
  }
```

The only private method is the IOCPFunction method. This method is spawned as a thread and is made part of the IOCP thread pool by calling the GetQueuedCompletionStatus Win32 API method. When the GetQueuedCompletionStatus Win32 API method returns, we check to make sure we are not being asked to shutdown the thread. The third argument is the data associated with the posted work. If the data is not SHUTDOWN_IOCPTHREAD, then real work has been posted into the IOCP thread pool and this thread has been chosen to process the work.

The application developer's supplied user function is called since the application developer is the only one who knows what needs to be done. Once that is complete, the method checks if a new thread should be added to the pool. This is done by reviewing the number of active threads in the pool.

```
  // Private Methods
  //*****************************************
  /// <summary> IOCP Worker Function that calls the specified user function </summary>
  private void IOCPFunction()
  {
    UInt32 uiNumberOfBytes;
    Int32  iValue;
    try
    {
      while (true)
      {
        unsafe
        {
          OVERLAPPED* pOv;
          // Wait for an event
          GetQueuedCompletionStatus(GetHandle, &uiNumberOfBytes, (UInt32*) &iValue, &pOv,
INIFINITE);
        }
        // Decrement the number of events in queue
        DecCurWorkInPool();
        // Was this thread told to shutdown
        if (iValue == SHUTDOWN_IOCPTHREAD)
          break;
        // Increment the number of active threads
        IncActThreadsInPool();
        try
        {
          // Call the user function
          GetUserFunction(iValue);
        }
        catch
        {
        }
        // Get a lock
        Monitor.Enter(GetCriticalSection);
```

```
        try
        {
          // If we have less than max threads currently in the pool
          if (GetCurThreadsInPool < GetMaxThreadsInPool)
          {
            // Should we add a new thread to the pool
            if (GetActThreadsInPool == GetCurThreadsInPool)
            {
              if (IsDisposed == false)
              {
                // Create a thread and start it
                ThreadStart tsThread = new ThreadStart(IOCPFunction);
                Thread thThread = new Thread(tsThread);
                thThread.Name = "IOCP " + thThread.GetHashCode();
                thThread.Start();
                // Increment the thread pool count
                IncCurThreadsInPool();
              }
            }
          }
        }
        catch
        {
        }
        // Relase the lock
        Monitor.Exit(GetCriticalSection);
       // Increment the number of active threads
        DecActThreadsInPool();
      }
    }
    catch
    {
    }
    // Decrement the thread pool count
    DecCurThreadsInPool();
  }
```

The last two public methods are the PostEvent methods. The first method takes an integer as an argument and the second version takes no argument at all. The integer is the data the application developer wishes to pass with the work posted into the IOCP thread pool. In the PostQueuedCompletionStatus Win32 API call, we can see that the third argument is where we pass the data value. Since this value is always an integer we set the size of the data to four, as seen in the second argument. Like in the IOCPFunction, we check to see if we need to add a new thread to the pool.

```
  // Public Methods
  //*****************************************
  /// <summary> IOCP Worker Function that calls the specified user function </summary>
  /// <param name="iValue"> SimType: A value to be passed with the event </param>
  /// <exception cref = "Exception"> Unhandled Exception </exception>
  public void PostEvent(Int32 iValue)
  {
    try
    {
      // Only add work if we are not disposing
      if (IsDisposed == false)
```

```
    {
      unsafe
      {
        // Post an event into the IOCP Thread Pool
        PostQueuedCompletionStatus(GetHandle, 4, (UInt32*) iValue, null);
      }
      // Increment the number of item of work
      IncCurWorkInPool();
      // Get a lock
      Monitor.Enter(GetCriticalSection);
      try
      {
        // If we have less than max threads currently in the pool
        if (GetCurThreadsInPool < GetMaxThreadsInPool)
        {
          // Should we add a new thread to the pool
          if (GetActThreadsInPool == GetCurThreadsInPool)
          {
            if (IsDisposed == false)
            {
              // Create a thread and start it
              ThreadStart tsThread = new ThreadStart(IOCPFunction);
              Thread thThread = new Thread(tsThread);
              thThread.Name = "IOCP " + thThread.GetHashCode();
              thThread.Start();
              // Increment the thread pool count
              IncCurThreadsInPool();
            }
          }
        }
      }
      catch
      {
      }
      // Release the lock
      Monitor.Exit(GetCriticalSection);
    }
  }
  catch (Exception e)
  {
    throw e;
  }
  catch
  {
    throw new Exception("Unhandled Exception");
  }
}
//*****************************************
/// <summary> IOCP Worker Function that calls the specified user function </summary>
/// <exception cref = "Exception"> Unhandled Exception </exception>
public void PostEvent()
{
  try
  {
    // Only add work if we are not disposing
    if (IsDisposed == false)
```

```
        {
          unsafe
          {
            // Post an event into the IOCP Thread Pool
            PostQueuedCompletionStatus(GetHandle, 0, null, null);
          }
          // Increment the number of item of work
          IncCurWorkInPool();
          // Get a lock
          Monitor.Enter(GetCriticalSection);
          try
          {
            // If we have less than max threads currently in the pool
            if (GetCurThreadsInPool < GetMaxThreadsInPool)
            {
              // Should we add a new thread to the pool
              if (GetActThreadsInPool == GetCurThreadsInPool)
              {
                if (IsDisposed == false)
                {
                  // Create a thread and start it
                  ThreadStart tsThread = new ThreadStart(IOCPFunction);
                  Thread thThread = new Thread(tsThread);
                  thThread.Name = "IOCP " + thThread.GetHashCode();
                  thThread.Start();
                  // Increment the thread pool count
                  IncCurThreadsInPool();
                }
              }
            }
          }
          catch
          {
          }
          // Release the lock
          Monitor.Exit(GetCriticalSection);
        }
      }
      catch (Exception e)
      {
        throw e;
      }
      catch
      {
        throw new Exception("Unhandled Exception");
      }
    }
  }
}
```

We have now completed the implementation of the IOCP thread pool class. Now it is time to test it.

Start by adding a new class to your C# project. Remove all of the code provided by the Visual Studio .NET wizard. Then add all of the following code. In Main, an IOCP thread pool is created, and a single piece of work is posted to the IOCP thread pool. We pass the data value of 10 along with the posted work.

The main thread is then put to sleep. This gives the IOCP thread function time to wake up to process the work posted. The last thing in main is to dispose the IOCP thread pool. The IOCP thread function displays the value of the data passed into the IOCP thread pool.

```
using System;
using System.Threading;  // Included for the Thread.Sleep call
using Continuum.Threading;
namespace Sample
{
 //=============================================
 /// <summary> Sample class for the threading class </summary>
 public class UtilThreadingSample
 {
  //*****************************************
  /// <summary> Test Method </summary>
  static void Main()
  {
   // Create the MSSQL IOCP Thread Pool
   IOCPThreadPool pThreadPool = new IOCPThreadPool(0, 5, 10, new
IOCPThreadPool.USER_FUNCTION(IOCPThreadFunction));
    pThreadPool.PostEvent(10);
    Thread.Sleep(100);
    pThreadPool.Dispose();
  }
  //*****************************************
  /// <summary> Function to be called by the IOCP thread pool.  Called when
  ///         a command is posted for processing by the SocketManager </summary>
  /// <param name="iValue"> The value provided by the thread posting the event </param>
  static public void IOCPThreadFunction(Int32 iValue)
  {
   try
   {
    Console.WriteLine("Value: {0}", iValue);
   }
   catch (Exception pException)
   {
    Console.WriteLine(pException.Message);
   }
  }
 }
}
```

This is what you should see when you run the sample application. On your own change the main function to call the PostEvent method several times and see how the IOCP thread pool performs.

This class provides the IOCP thread pooling support your server based software requires. By using this class, you can build efficient and robust servers that can scale as the amount of work increases. Though there are some limitations due to the nature of the managed heap and unsafe code, these limitations can easily be overcome with a little ingenuity.