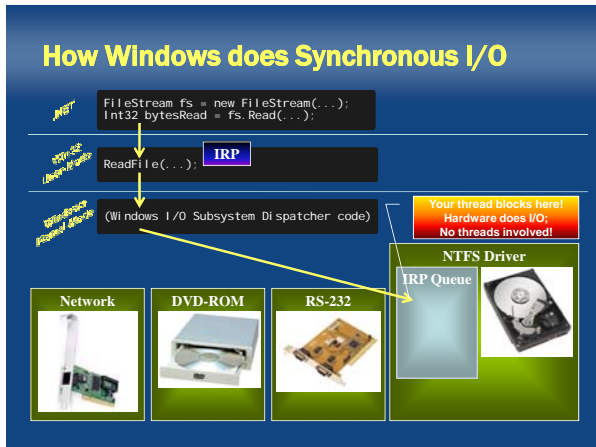


The CLR's Asynchronous Programming Model

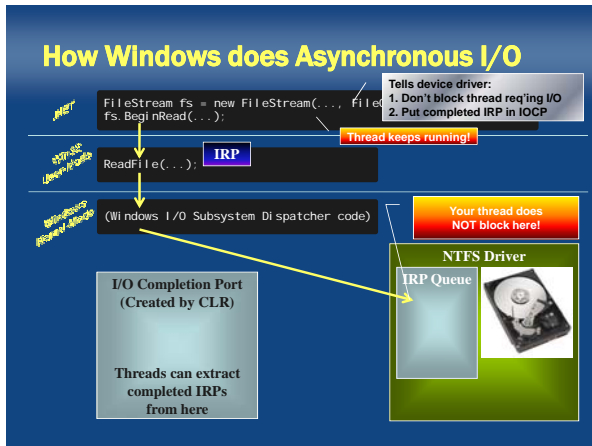


How Windows does Synchronous I/O



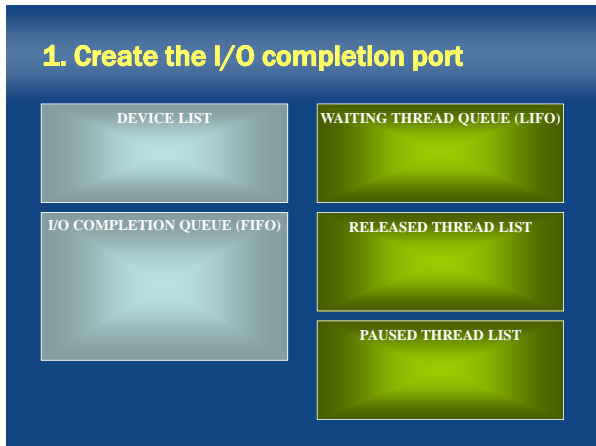
The Thread Pool is Architected Around Windows' I/O Completion Port

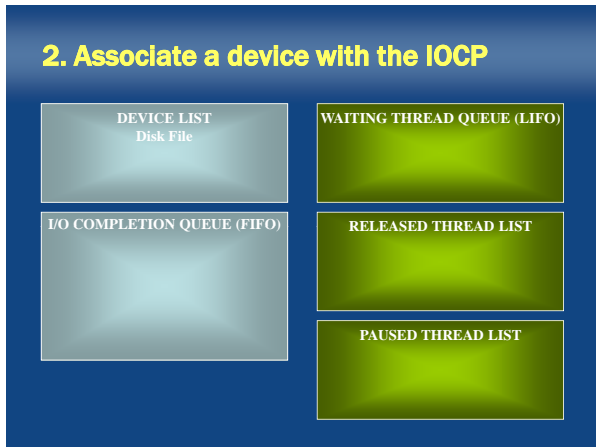
- It's all about 2 benefits:
 - Not blocking threads requesting I/O
 - Keeping CPUs (no more than) saturated (reduce context switching)
- The CLR creates IOCP when initializing and has threads waiting on IOCP which wake "intelligently"
 - Completed IRPs are processed FIFO; threads process them LIFO
- You open device specifying 2 things (via flag)
 - Driver shouldn't suspend thread req'ing I/O
 - Driver should put completed IRPs in CLR's IOCP
 - Done internally via ThreadPool.BindHandle(SafeHandle osHandle)
- For each I/O, you give method to process each completed IRP

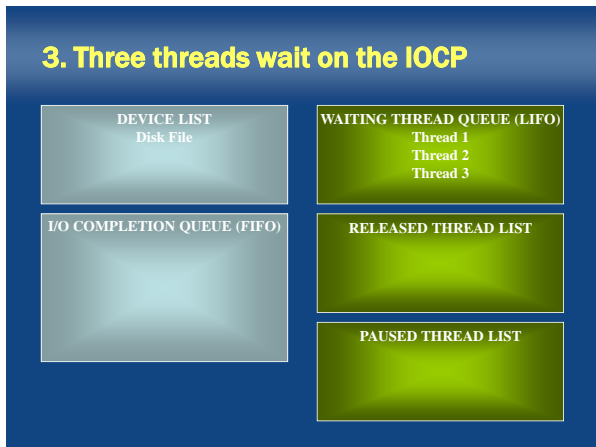


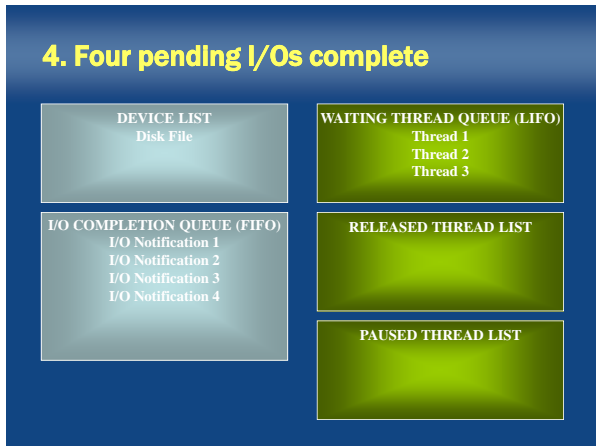
- ### Asynchronous I/O Notes
- Remember that I/Os from ALL processes queue to the appropriate device driver(s)
 - Windows documentation explicitly states that device drivers can perform the I/O requests in any order they want
 - For example, when hard-disk driver is ready to perform an I/O request, it scans the set of IRPs and takes IRPs that are close to where the disk head is currently to reduce drive seeking (which is very slow)
 - This increases overall system throughput
 - Vista and Windows Server 2008 support I/O priorities
 - Each IRP specifies priority; driver does normal priority first
 - Low priority used for long-running background tasks
 - Anti-virus, content indexer, defragmenter, etc.

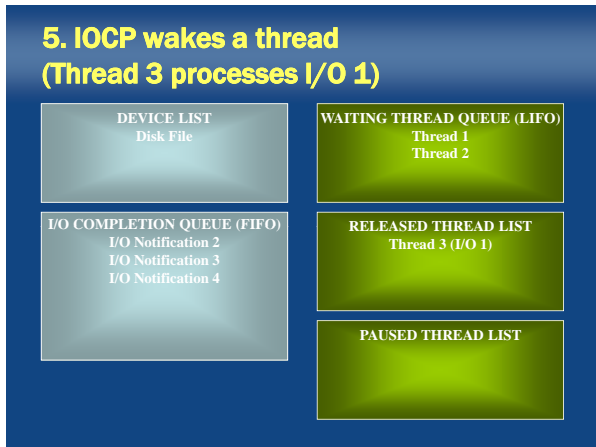
- ### Inside the Windows' I/O Completion Port
- The CLR creates 1 IOCP per process
 - The CLR creates thread pool threads that wait on the IOCP
 - The CLR/IOCP work together to intelligently create/destroy thread pool threads
 - The CLR threads call your methods when IRPs complete
 - Let's walk through an example
 - Assume you've opened a file for async I/O
 - Associates file with CLR-created IOCP
 - Assume you've queued 4 asynchronous I/Os against the file
 - Assume thread pool has 3 threads in it
 - Assume computer has dual-core CPU in it

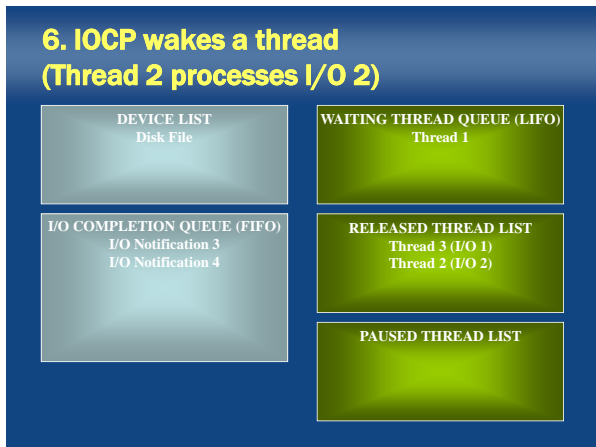


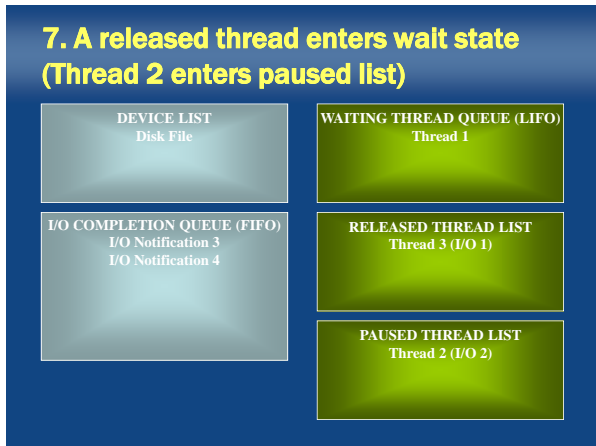


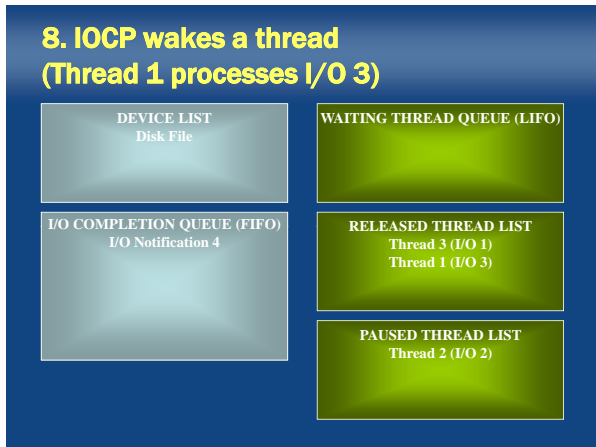


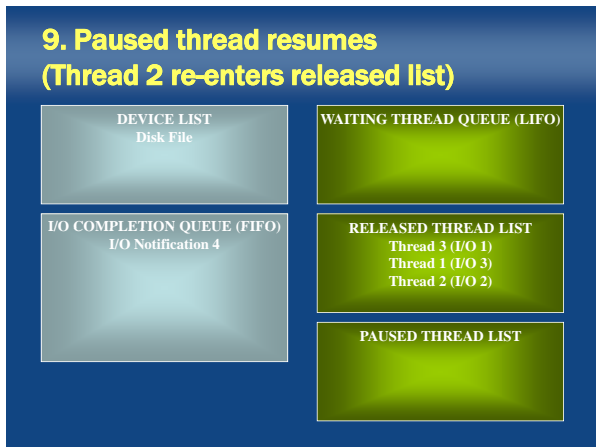


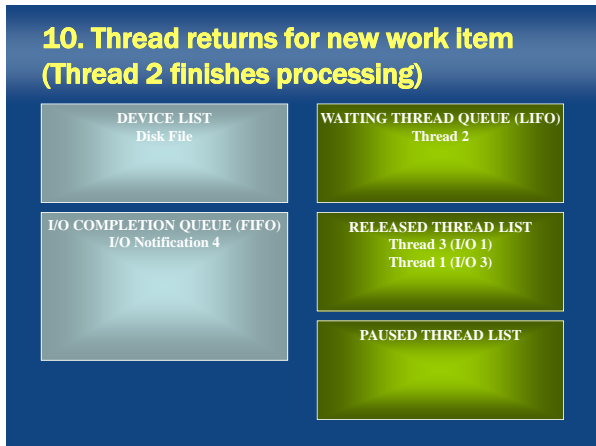


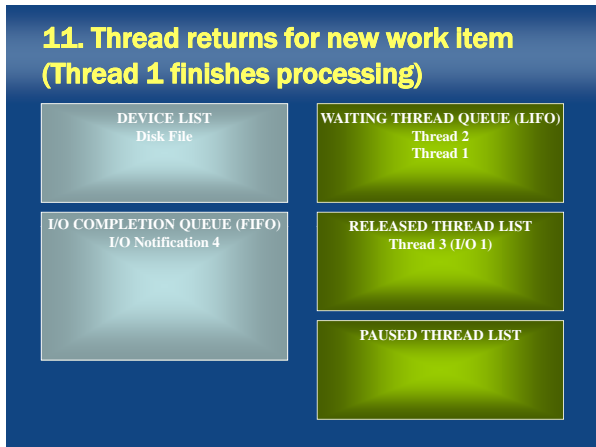


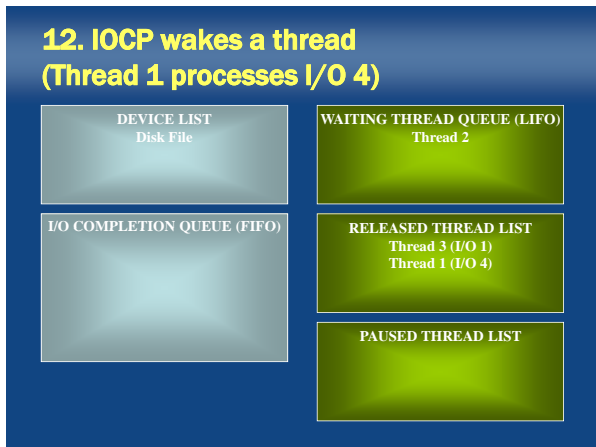


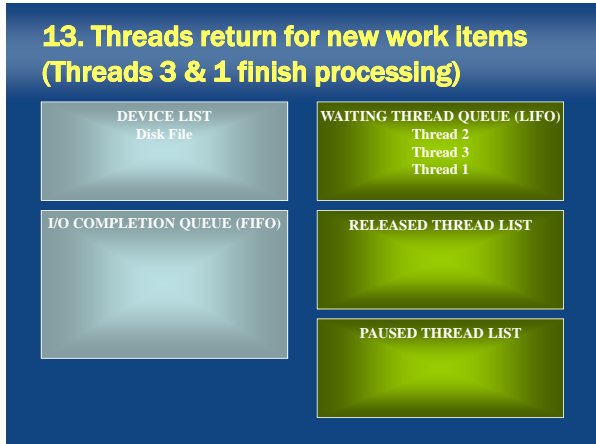














- Introducing the Asynchronous Programming Model**
- Standard pattern to execute asynchronous operations
 - Asynchronous I/O-bound operations
 - Stream types: `BeginRead`, `BeginWrite`
 - Dns: `BeginGetHostByName`, `BeginResolve`
 - Socket: `BeginAccept`, `BeginConnect`, `BeginReceive`, `BeginSend`, ...
 - WebRequest types: `BeginGetRequestStream`, `BeginGetResponse`
 - SqlCommand: `BeginExecuteNonQuery`, `BeginExecuteReader`, ...
 - Tools producing WS proxies (WSDL.exe & SvcUtil.exe): `BeginXxx`
 - Asynchronous Compute-bound operations
 - Delegate types: `BeginInvoke`
 - All `BeginXxx` methods have a corresponding `EndXxx` method
 - The APM supports 3 rendezvous techniques
 - Wait until done, Polling, Callback method

Example: Performing Synchronous I/O with a FileStream

- Call Read to read synchronously
 - Thread is suspended until data has been read
 - You don't know when (or if) the call will return
 - Data could be in cache; Read returns immediately
 - Data could be on network server; Read could never return
- Synchronous I/O is inefficient & hangs an application
 - Waiting thread: wastes resources & stops responding to user input

```
public class FileStream : Stream, ... {  
    // Method to perform synchronous I/O operation  
    public Int32 Read(Byte[] array, Int32 offset, Int32 count);  
}
```

Example: Performing Asynchronous I/O with a FileStream

- Construct FileStream with FileOptions.Asynchronous flag
- BeginRead queues read request to Windows device driver
 - Begin method creates an IAsyncResult object as a receipt
 - Calling thread returns immediately (with receipt) & keeps running
- Possible outcomes
 - Data is cached, operation might be done before Begin returns
 - Data might come in shortly
 - Data might not come in at all

```
public class FileStream : Stream, ... {  
    // Method to perform synchronous I/O operation  
    public Int32 Read(Byte[] array, Int32 offset, Int32 count);  
  
    // Methods to perform asynchronous I/O operation  
    public IAsyncResult BeginRead(Byte[] array, Int32 offset, Int32 count,  
        AsyncCallback userCallback, Object stateObject);  
  
    public Int32 EndRead(IAsyncResult ar);  
}
```

FileStream Notes

- Windows & .NET programming models clash
 - Windows: on device open, you indicate sync or async I/O
 - .NET: on device operation, you indicate sync or async I/O
- If you don't specify the FileOptions.Asynchronous flag
 - Windows performs all operations synchronously
 - BeginRead uses another thread to emulate async I/O
 - This is very inefficient
- If you specify the FileOptions.Asynchronous flag
 - Windows performs all operations asynchronously
 - Read queues async op and blocks thread until I/O completes
 - This is inefficient too but better than above
- Summary: Specify the FileOptions.Asynchronous flag

Windows Always Performs Some File I/O Operations Synchronously

- Compression
 - NTFS compression; not ZIP or cabinet files
- Extending a file's length by writing (appending) to the file
 - Fix this by setting the length first and then writing
 - Or, call Win32's SetFileValidData function before writing
- Cached data
 - If data is already in a cache, the driver just returns it synchronously
 - Use FILE_FLAG_NO_BUFFERING to avoid cache; forcing async I/O
 - Flag not supported by .NET; requires sector-aligned data buffer
 - This is good because there is no I/O; compute bound
- For more info, see this knowledgebase article
 - <http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B156932>

APM Usage Notes



APM Usage Notes

- Outstanding operations cannot be canceled
 - I/O: How can you tell a device/server to stop?
 - Compute: You can roll your own cancel or abort the thread
- Constructing objects hurts performance
 - IAsyncResult (& AsyncWaitHandle): .NET 3.5 Socket has new methods to avoid 1 object per I/O request; used by WCF
 - <http://msdn2.microsoft.com/en-us/library/bb968780.aspx>
 - AsyncCallback delegates: Minimize these by explicitly initializing delegate field(s) & pass field(s) to BeginXxx methods
- Many Win32 APIs don't offer async versions
 - Opening a file, Get files in a dir, Get subdirectories, Changing attributes of a file/dir, accessing registry/event log, many more



The APM and Windows Forms

- Windows requires that the thread that created a window process all messages for that window
 - A TP thread cannot access System.Windows.Forms.Control-derived object directly
- The Control class offers methods to help

```
public class Control : ... {  
    // Invokes method on GUI thread & waits until done processing (avoid)  
    public Object Invoke(Delegate method);  
    public Object Invoke(Delegate method, params Object[] args);  
  
    // Invokes method on GUI thread & returns immediately (use)  
    public IAsyncResult BeginInvoke(Delegate method);  
    public IAsyncResult BeginInvoke(Delegate method, params Object[] args);  
  
    // Waits until invoked method is done processing (optional: avoid)  
    public Object EndInvoke(IAsyncResult asyncResult);  
}
```

The APM and Windows Presentation Foundation

- WPF requires that the thread that created a DispatcherObject-derived object process all messages for that object
- The Dispatcher class offers methods to help

```
public class Dispatcher : ... {  
    // Invokes method on GUI thread & waits until done processing (avoid)  
    // Other overloads not shown  
    public Object Invoke(DispatcherPriority priority, Delegate method, Object arg);  
  
    // Invokes method on GUI thread & returns immediately (use)  
    // Other overloads not shown  
    public DispatcherOperation BeginInvoke(DispatcherPriority priority, Delegate method, Object arg);  
}
```

Using the APM to Perform Asynchronous Compute-Bound Operations

- Not as efficient as an I/O op because a thread must run
- To call any method async, define a matching delegate
 - BeginInvoke queues work item to TP
 - TP thread calls method
 - When method returns, TP thread calls your callback method
 - Call EndInvoke to get result

```
UInt64 Sum(UInt64 n, UInt32 by) { /* compute-bound operation */ }  
  
delegate UInt64 AsyncSum(UInt64 n, UInt32 by);  
  
internal sealed class AsyncSum : MulticastDelegate {  
    public AsyncSum(Object object, IntPtr method);  
    public UInt64 Invoke(UInt64 n, UInt32 by);  
    public IAsyncResult BeginInvoke(UInt64 n, UInt32 by, AsyncCallback callback, Object object);  
    public UInt64 EndInvoke(IAsyncResult result);  
}
```



Discussion
