# General I/O Concepts

This section explores I/O concepts and provides performance guidelines pertaining to I/O of any kind. This advice is applicable to networking applications, heavy disk-accessing processes, and even software designed to access a custom high-bandwidth hardware device.

## Synchronous and Asynchronous I/O

With synchronous I/O, the I/O transfer function (e.g. `ReadFile`, `WriteFile`, or `DeviceIoControl` Win32 API functions) blocks until the I/O operation completes. Although this model is convenient to use, it is not very efficient. During the time between issuing successive I/O requests, the device may be idle and, therefore, is potentially under-utilized. Another problem with synchronous I/O is that a thread is "wasted" for each concurrently pending I/O request. For example, in a server application servicing many clients concurrently, you may end up creating a thread per session. These threads, which are essentially mostly idle, are wasting memory and may create a situation called *thread thrashing* in which many threads wake up when I/O completes and compete with each other for CPU time, resulting in many context switches and poor scalability.

The Windows I/O subsystem (including device drivers) is internally asynchronous – execution of program flow can continue while an I/O operation is in progress. Almost all modern hardware is asynchronous in nature as well and does not need polling to transfer data or to determine if an I/O operation is complete. Most devices instead rely on Direct Memory Access (DMA) controllers to transfer data between the device and the computer RAM, without requiring the CPU's attention during the transfer, and then raise an interrupt to signal completion the data transfer. It is only at the application level that Windows allows synchronous I/O that is actually asynchronous internally.

In Win32, asynchronous I/O is called *overlapped I/O* (see Figure 7-1 comparing synchronous and overlapped I/O). Once an application issues an overlapped I/O, Windows either completes the I/O operation immediately or returns a status code indicating the I/O operation is still pending. The thread can then issue more I/O operations, or it can do some computational work. The programmer has several options for receiving a notification about the I/O operation's completion:

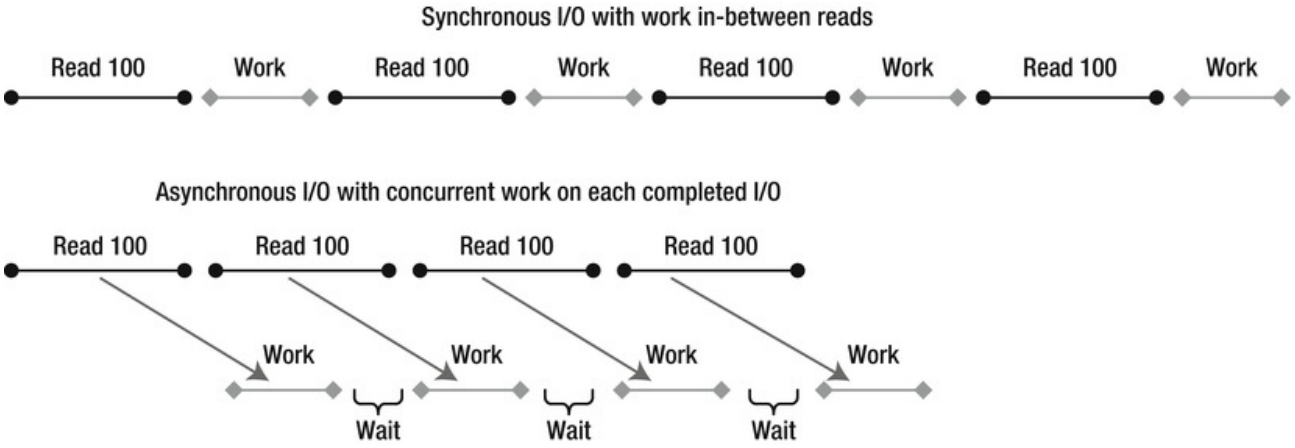


*Figure 7-1. Comparison between synchronous and overlapped I/O*

- Signaling of a Win32 event: A wait operation on this event will complete when the I/O completes.
- Invocation of a user callback routine via the Asynchronous Procedure Call (APC) mechanism: The issuing thread must be in a state of *alertable wait* to allow APCs.
- Notification via I/O Completion Ports: This is usually the most efficient mechanism. We explore I/O completion ports in detail later in this chapter.

**Note** Some I/O devices (e.g. a file opened in unbuffered mode) benefit (by increasing device utilization) if an application can keep a small amount of I/O requests pending. A recommended strategy is to pre-issue a certain number of I/O requests and, for each request that completes, re-issue another. This ensures the device driver can initiate the next I/O as quickly as possible, without waiting for the application to issue the next I/O in response. However, do not exaggerate with the amount of pending data, since it can consume limited kernel memory resources.

## I/O Completion Ports

Windows provides an efficient asynchronous I/O completion notification mechanism called the *I/O Completion Port* (IOCP). It is exposed through the .NET `ThreadPool.BindHandle` method. Several .NET types dealing with I/O utilize this functionality internally: `FileStream`, `Socket`, `SerialPort`, `HttpListener`, `PipeStream`, and some .NET Remoting channels.

An IOCP (see Figure 7-2) is associated with zero or more I/O handles (sockets, files, and specialized device driver objects) opened in overlapped mode and with user-created threads. Once an I/O operation of an associated I/O handle completes, Windows enqueues the completion notification to the appropriate IOCP and an associated thread handles the completion notification. By having a thread pool that services completions and by intelligently controlling thread wakeup, context switches are reduced and multi-processor concurrency is maximized. It is no surprise that high-performance servers, such as Microsoft SQL Server, use I/O completion ports.

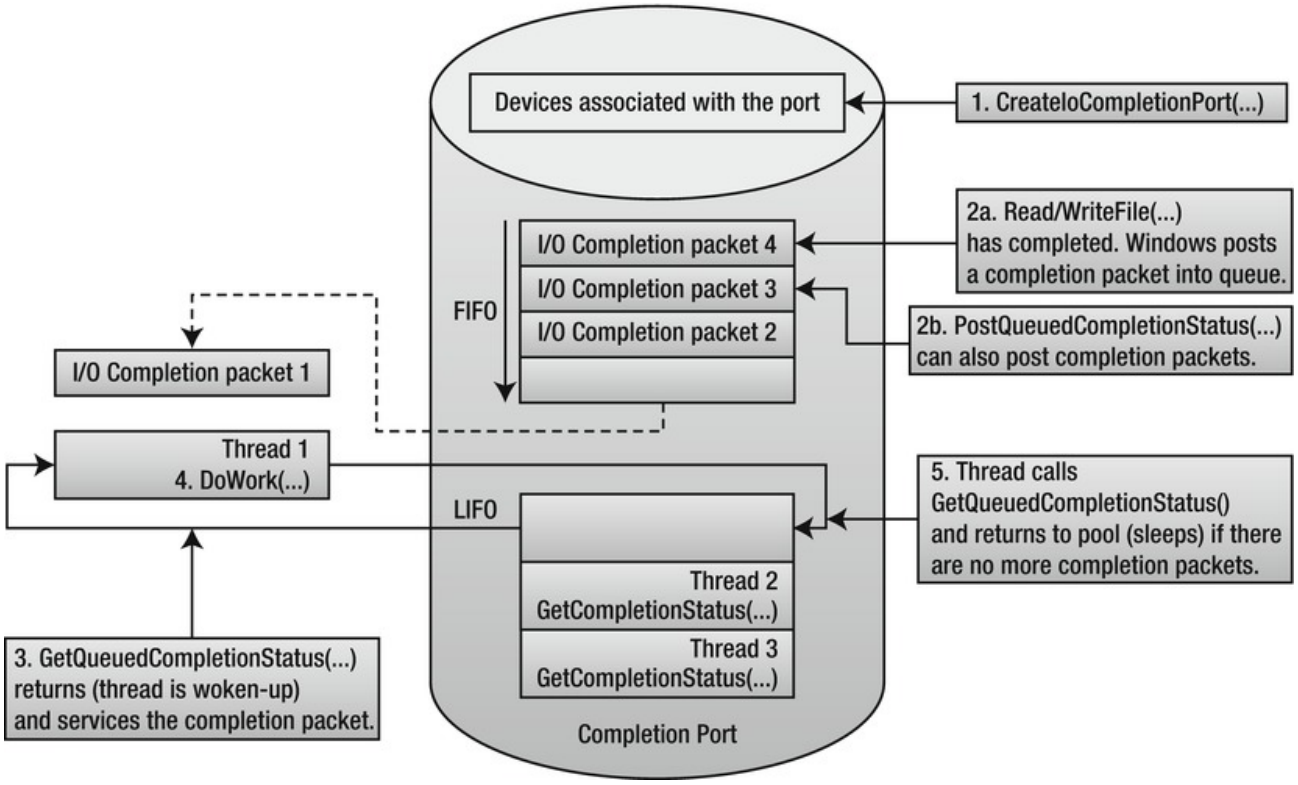


*Figure 7-2. Structure and operation of an I/O Completion Port*

A completion port is created by calling the `CreateIoCompletionPort` Win32 API function, passing a maximum concurrency value, a completion key and optionally associating it with an I/O-capable handle. A completion key is a user-specified value that serves to distinguish between different I/O handles at completion. More I/O handles can be associated with the same or a different IOCP by again calling `CreateIoCompletionPort` and specifying the existing completion port handle.

User-created threads then call `GetCompletionStatus` to become bound to the specified IOCP and to wait for completion. A thread can only be bound to one IOCP at a time. `GetQueuedCompletionStatus` blocks until there is an I/O completion notification available (or a timeout period has elapsed), at which point it returns with the I/O operation details, such as the number of number of bytes transferred, the completion key, and the overlapped structure passed during I/O. If another I/O completes while all associated threads are busy (i.e., not blocked on `GetQueuedCompletionStatus`), the IOCP wakes another thread in LIFO order, up to the maximum concurrency value. If a thread calls `GetQueuedCompletionStatus` and the notification queue is not empty, the call returns immediately, without the thread blocking in the OS kernel.

---

**Note** The IOCP is aware if one of the "busy" threads is actually doing a synchronous I/O or wait and will wake additional associated threads (if any), potentially exceeding the concurrency value. A completion notification can also be posted manually without involving I/O by calling `PostQueuedCompletionStatus`.

---

The following code listing shows an example of using `ThreadPool.BindHandle` on a Win32 file handle. Start by looking at the `TestIOCP` method. Here, we call `CreateFile`, which is a P/Invoke'd Win32 function used to open or create files or devices. We must specify the `EFileAttributes.Overlapped` flag in the call to use any kind of asynchronous I/O. `CreateFile` returns a Win32 file handle if it succeeds, which we then bind to .NET's I/O completion port by calling `ThreadPool.BindHandle`. We create an auto-reset event, which is used to temporarily block the thread issuing I/O operations if there are too many such operations in progress (the limit is set by the `MaxPendingIos` constant).

We then begin a loop of asynchronous write operations. At each iteration, we allocate a buffer that contains the data to be written. We also allocate an `Overlapped` structure that contains the file offset (here, we always write to offset 0), an event handle signaled when I/O completes (not used in I/O Completion Ports) and an optional user-created `IAsyncResult` object that can be used to carry state to the completion function. We then call `Overlapped` structure's `Pack` method, which takes the completion function and data buffer as parameters. It allocates an equivalent native overlapped structure from unmanaged memory and pins the data buffer. The native structure has to be manually freed to release the unmanaged memory it occupies and to unpin the managed buffer.

If there are not too many I/O operations in progress, we call `WriteFile`, while specifying the native overlapped structure. Otherwise, we wait until the event becomes signaled, which indicates the pending I/O operations count has dropped below the limit.

The I/O completion function `WriteComplete` is invoked by .NET I/O Completion thread pool threads when I/O completes. It receives a pointer to the native overlapped structure, which can be unpacked to convert it back to the managed `Overlapped` structure.

```
using System;
using System.Threading;
using Microsoft.Win32.SafeHandles;
using System.Runtime.InteropServices;

[DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Auto)]
internal static extern SafeFileHandle CreateFile(
  string lpFileName,
  EFileAccess dwDesiredAccess,
  EFileShare dwShareMode,
  IntPtr lpSecurityAttributes,
  ECreationDisposition dwCreationDisposition,
  EFileAttributes dwFlagsAndAttributes,
  IntPtr hTemplateFile);

[DllImport("kernel32.dll", SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
static unsafe extern bool WriteFile(SafeFileHandle hFile, byte[] lpBuffer,
```

```
  uint nNumberOfBytesToWrite, out uint lpNumberOfBytesWritten,
  System.Threading.NativeOverlapped *lpOverlapped);

[Flags]
enum EFileShare : uint {
  None = 0x00000000,
  Read = 0x00000001,
  Write = 0x00000002,
  Delete = 0x00000004
}

enum ECreationDisposition : uint {
  New = 1,
  CreateAlways = 2,
  OpenExisting = 3,
  OpenAlways = 4,
  TruncateExisting = 5
}

[Flags]
enum EFileAttributes : uint {
  //Some flags not present for brevity
  Normal = 0x00000080,
  Overlapped = 0x40000000,
  NoBuffering = 0x20000000,
}

[Flags]
enum EFileAccess : uint {
  //Some flags not present for brevity
  GenericRead = 0x80000000,
  GenericWrite = 0x40000000,
}

static long _numBytesWritten;
static AutoResetEvent _waterMarkFullEvent; // throttles writer thread
static int _pendingIosCount;

const int MaxPendingIos = 10;

//Completion routine called by .NET ThreadPool I/O completion threads
static unsafe void WriteComplete(uint errorCode, uint numBytes, NativeOverlapped* pOVERLAP) {
  _numBytesWritten + = numBytes;
  Overlapped ovl = Overlapped.Unpack(pOVERLAP);

  Overlapped.Free(pOVERLAP);
  //Notify writer thread that pending I/O count fell below watermark
  if (Interlocked.Decrement(ref _pendingIosCount) < MaxPendingIos)
  _waterMarkFullEvent.Set();
}

static unsafe void TestIOCP() {
    //Open file in overlapped mode
    var handle = CreateFile(@"F:\largefile.bin",
     EFileAccess.GenericRead | EFileAccess.GenericWrite,
     EFileShare.Read | EFileShare.Write,
     IntPtr.Zero, ECreationDisposition.CreateAlways,
     EFileAttributes.Normal | EFileAttributes.Overlapped, IntPtr.Zero);

     _waterMarkFullEvent = new AutoResetEvent(false);
  ThreadPool.BindHandle(handle);
  for (int k = 0; k < 1000000; k++) {
    byte[] fbuffer = new byte[4096];
```

```
    //Args: file offset low & high, event handle, IAsyncResult object
    Overlapped ovl = new Overlapped(0, 0, IntPtr.Zero, null);
    //The CLR takes care to pin the buffer
    NativeOverlapped* pNativeOVL = ovl.Pack(WriteComplete, fbuffer);
    uint numBytesWritten;

    //Check if too many I/O requests are pending
    if (Interlocked.Increment(ref _pendingIosCount) < MaxPendingIos) {
  if (WriteFile(handle, fbuffer, (uint)fbuffer.Length, out numBytesWritten,
  pNativeOVL)) {
  //I/O completed synchronously
  _numBytesWritten + = numBytesWritten;
  Interlocked.Decrement(ref _pendingIosCount);
  } else {
  if (Marshal.GetLastWin32Error() ! = ERROR_IO_PENDING) {
  return; //Handle error
  }
  }
  } else {
    Interlocked.Decrement(ref _pendingIosCount);
    while (_pendingIosCount > = MaxPendingIos) {
    _waterMarkFullEvent.WaitOne();
   }
    }
   }
}
```

To summarize, when working with high-throughput I/O devices, use overlapped I/O with completion ports, either by directly creating and using your own completion port in the unmanaged library or by associating a Win32 handle with .NET's completion port through `ThreadPool.BindHandle`.

## NET Thread Pool

The .NET Thread Pool is used for many purposes, each served by a different kind of thread. Chapter 6 showed the thread pool APIs that we used to tap into the thread pool's ability to parallelize a CPU-bound computation. However, there are numerous kinds of work for which the thread pool is suited:

- *Worker threads* handle asynchronous invocation of user delegates (e.g. `BeginInvoke` or `ThreadPool.QueueUserWorkItem`).
- *I/O completion threads* handle completions for the global IOCP.
- *Wait threads* handle registered wait. A registered wait saves threads by combining several waits into one wait (using `WaitForMultipleObjects`), up to the Windows limit (`MAXIMUM_WAIT_OBJECTS` = 64). Registered wait is used for overlapped I/O that is not using I/O completion ports.
- *Timer thread* combines waiting on multiple timers.
- *Gate thread* monitors CPU usage of thread pool threads, as well as grows or shrinks the thread counts (within preset limits) for best performance.

---

**Note** You can issue an I/O operation that may appear asynchronous, although it really is not. For example, calling `ThreadPool.QueueUserWorkItem` on a delegate and then doing a synchronous I/O operation does not make it truly asynchronous and is no better than doing so on a regular thread.

---

## Copying Memory

Commonly, a data buffer received from a hardware device is copied over and over until the application finishes processing it. Copying can become a significant source of CPU overhead and, thus, should be avoided for high throughput I/O code paths. We now survey some scenarios in which you copy data and how to avoid it.

### Unmanaged Memory

In .NET, an unmanaged memory buffer is more cumbersome to work with than a managed `byte[]`, so programmers often take the easy way out and just copy the buffer to managed memory.

If your API or library lets you specify your own memory buffer or has a user-defined allocator callback, allocate a managed buffer and pin it so that it can be accessed through both a pointer and as a managed reference. If the buffer is so large (>85,000 bytes) it is allocated in the Large Object Heap, try to re-use the buffer. If re-using is non-trivial because of indeterminate object lifetime, use memory pooling, as described in Chapter 8.

In other cases, the API or library insists on allocating its own (unmanaged) memory buffer. You can access it directly with a pointer (requires unsafe code) or by using wrapper classes, such as `UnmanagedMemoryStream` or `UnmanagedMemoryAccessor`. However, if you need to pass the buffer to some code that only works with `byte[]` or `string` objects, copying may be unavoidable.

Even if you cannot avoid copying memory, if some or most of your data is filtered early on (e.g. network packets), it is possible to avoid unnecessary memory copying by first checking if the data is useful without copying it.

### Exposing Part of a Buffer

As Chapter 8 explains, programmers sometime assume that a `byte[]` contains only the desired data and that it spans from the beginning until the end, forcing the caller to splice the buffer (allocate a new `byte[]` and copy only the desired portion). This scenario often comes up in parsing a protocol stack. In contrast, equivalent unmanaged code would take a pointer, will have no knowledge whether it points to the beginning of the allocation or not, and will have to take a length parameter to tell it where the data ends.

To avoid unnecessary memory copying, take an offset and length parameters wherever you take a `byte[]` parameter. The length parameter is used instead of the array's `Length` property, and the offset value is added to the index.