Inside I/O Completion Ports

Last updated July 30, 1998

Introduction

Writing a high-performance server application requires implementing an efficient threading model. Having either too few or too many server threads to process client requests can lead to performance problems. For example, if a server creates a single thread to handle all requests clients can become starved since the server will be tied up processing one request at a time. Of course, a single thread could simultaneously process multiple requests, switching from one to another as I/O operations are started, but this architecture introduces significant complexity and cannot take advantage of multiprocessor systems. At the other extreme a server could create a big pool of threads so that virtually every client request is processed by a dedicated thread. This scenario usually leads to thread-thrashing, where lots of threads wake-up, perform some CPU processing, block waiting for I/O and then after request procesing is completed block again waiting for a new request. If nothing else, context-switches are caused by the scheduler having to divide processor time among multiple active threads.

The goal of a server is to incur as few context switches as possible by having its threads avoid unnecessary blocking, while at the same time maximizing parallelism by using multiple threads. The ideal is for there to be a thread actively servicing a client request on every processor and for those threads not to block if there are additional requests waiting when they complete a request. For this to work correctly however, there must be a way for the application to activate another thread when one processing a client request blocks on I/O (like when it reads from a file as part of the processing).

Windows NT 3.5 introduced a set of APIs that make this goal relatively easy to achieve. The APIs are centered on an object called a *completion port*. In this article I'm going to provide an overview of how completion ports are used and then go inside them to show you how Windows NT implements them.

Using I/O Completion Ports

Applications use completion ports as the the focal point for the completion of I/O associated with multiple file handles. Once a file is associated with a completion port any asynchronous I/O operations that complete on the file result in a completion packet being queued to the port. A thread can wait for any outstanding I/Os to complete on multiple files simply by waiting for a completion packet to be queued on the completion port. The Win32 API provides similar functionality with the **WaitForMultipleObjects** API, but the advantage that completion ports have is that concurrency, or the number of threads that an application has actively servicing client requests, is controlled with the aid of the system.

When an application creates a completion port it specifies a concurrency value. This value indicates the maximum number of threads associated with the port that should be running at any given point in time. As I stated earlier, the ideal is to have one thread active at any given point in time for every processor in the system. The concurrency value associated with a port is used by NT to control how many threads an application has active - if the number of active threads associated with a port equals the concurrency value then a thread that is waiting on the completion port will not be allowed to run. Instead, it is expected that one of the active threads will finish processing its current request and check to see if there's another packet waiting at the port - if there is then it simply grabs it and goes off to process it. When this happens there is no context switch, and the CPUs are utilized to near their full capacity.

Figure 1 below shows a high-level picture of completion port operation. Incoming client requests cause completion packets to be queued at the port. A number of threads, up to the concurrency limit for the port, are allowed by NT to process client requests. Any additional threads associated with the port are blocked until the number of active threads drops, as can happen when an active thread blocks on file I/O. I'll discuss this further a little later.

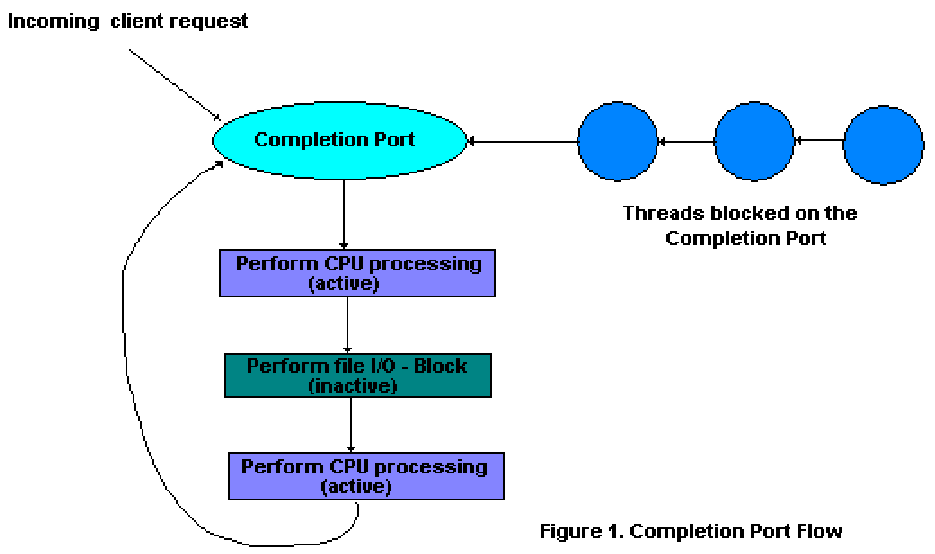
Incoming client request
Completion Port
Threads blocked on the Completion Port
Perform CPU processing (active)
Perform file I/O - Block (inactive)
Perform CPU processing (active)

Figure 1. Completion Port Flow

A completion port is created with a call to the Win32 API **CreateIoCompletionPort**:

HANDLE CreateIoCompletionPort(
    HANDLE *FileHandle*,
    HANDLE *ExistingCompletionPort*,
    DWORD *CompletionKey ,*
    DWORD *NumberOfConcurrentThreads*
    );

To create the port an application passes in a NULL for the *ExistingCompletionPort* parameter and indicates the concurreny value with the *NumberOfConcurrentThreads* parameter. If a *FileHandle* parameter is specified then the file handle becomes associated with the port. When an I/O request that has been issued on the file handle completes a completion packet is queued to the completion port. To retrieve a completion packet and possibly block waiting for one to arrive a thread calls the **GetQueuedCompletionStatus** API:

BOOL GetQueuedCompletionStatus(
    HANDLE *CompletionPort*,
    LPDWORD *lpNumberOfBytesTransferred*,
    LPDWORD *CompletionKey ,*
    LPOVERLAPPED **lpOverlapped,*
    DWORD *dwMiillisecondTimeout*
    );

Threads that block on a completion port become associated with the port and are woken in LIFO order so that the thread that blocked most recently is the one that is given the next packet. Threads that block for long periods of time can have their stacks swapped out to disk, so if there are more threads associated with a port then there is work to process the in-memory footprints of threads blocked the longest are minimized.

A server application will usually receive client requests via network endpoints that are represented as file handles. Examples include Winsock2 sockets or named pipes. As the server creates its communications endpoints it associates them with a completion port and its threads wait for incoming requests by calling **GetQueuedCompletionStatus** on the port. When a thread is given a packet from the completion port it will go off and start processing the request, becoming an active thread. Many times a thread will block during its processing, like when it needs to read or write data to a file on disk, or when it synchronizes with other threads. Windows NT is clever enough to detect this and recognize that the completion port has one less active thread. Therefore, when a thread becomes inactive because it blocks, a thread waiting on the completion port will be woken if there is packet in the queue.

Microsoft's guidelines are to set the concurrency value roughly equal to the number of processors in a system. Note that it is possible for the number of active threads for a completion port to exceed the concurrency limit. Consider a case where the limit is specified as 1. A client request comes in and a thread is dispatched to process the request, becoming active. A second requests comes in but a second thread waiting on the port is not allowed to proceed because the concurrency limit has been reached. Then the first thread blocks waiting for a file I/O so it becomes inactive. The second thread is then released and while it is still active the first thread's file I/O is completes, making it active again. At that point in time, and until one of the threads blocks, the concurrency value is 2, which is higher than the limit of 1. Most of the time the active count will remain at or just above the concurrency limit.

The completion port API also makes it possible for a server application to queue privately defined completion packets to a completion port using **PostQueuedCompletionStatus**. Servers typically use this function to inform its threads of external events such as the need to shut down gracefully.

Completion Port Internals

A call to the Win32 API **CreateIoCompletionPort** with a NULL completion port handle results in the execution of the native API function **NtCreateIoCompletion**, which invokes the corresponding kernel-mode system service of the same name. Internally, completion ports are based on an undocumented executive synchronization object called a *Queue*. Thus, the system service creates a completion port object and initializes a queue object in the port's allocated memory (a pointer to the port also points to the queue object since the queue is at the start of the port memory). A queue object has (coincidentally) a concurrency value that is specified when a thread initializes one, and in this case the value that is used is the one that was passed to **CreateIoCompletionPort**. **KeInitializeQueue** is the function that **NtCreateIoCompletion** calls to initialize a port's queue object.

When an application calls **CreateIoCompletionPort** to associate a file handle with a port the Win32 API invokes the native function **NtSetInformationFile** with the file handle as the primary parameter. The information class that is set is **FileCompletionInformation** and the completion port's handle and the *CompletionKey* parameter from **CreateIoCompletionPort** are the data values. **NtSetInformationFile** dereferences the file handle to obtain the file object and allocates a completion context data structure, which is defined in NTDDK.H as:

```
typedef struct _IO_COMPLETION_CONTEXT {
      PVOID Port;
      ULONG Key;
} IO_COMPLETION_CONTEXT, *PIO_COMPLETION_CONTEXT;
```

Finally, **NtSetInformationFile** sets the *CompletionContext* field in the file object to point to the context structure. When an I/O operation completes on a file object the point at the context structure. When an I/O operation completes on a file object the internal I/O manager function **IopCompleteRequest** executes and, if the I/O was asynchronous, checks to see if the *CompletionContext* field in the file object is non-NULL. If its non-NULL the I/O Manager allocates a completion packet and queues it to the completion port by calling **KeInsertQueue** with the port as the queue on which to insert the packet (remember that the completion port object and queue object are synonymous).

When **GetQueuedCompletionStatus** is invoked by a server thread, it calls the native API function **NtRemoveIoCompletion**, which transfers control to the **NtRemoveIoCompletion** system service. After validating parameters and translating the completion port handle to a pointer to the port, **NtRemoveIoCompletion** calls **KeRemoveQueue**.

As you can see, **KeRemoveQueue** and **KeInsertQueue** are the engine behind completion ports and are the functions that determine whether a thread waiting for an I/O completion packet should be activated or not. Internally, a queue object maintains a count of the current number of active threads and the maximum active threads. If the current number equals or exceeds the maximum when a thread calls **KeRemoveQueue**, the thread will be put (in LIFO order) onto a list of threads waiting for a turn to process a completion packet. The list of threads hangs off the queue object. A thread's control block data structure has a pointer in it that references the queue object of a queue that it is associated with; if the pointer is NULL then the thread is not associated with a queue.

So how does NT keep track of threads that become inactive because they block on something other than the completion port? The answer lies in the queue pointer in a thread's control block. The scheduler routines that are executed in response to a thread blocking (**KeWaitForSingleObject**, **KeDelayExecutionThread**, etc.) check the thread's queue pointer and if its not NULL they will call **KiActivateWaiterQueue**, a queue-related function. **KiActivateWaiterQueue** decrements the count of active threads associated with the queue, and if the result is less than the maximum and there is at least one completion packet in the queue then the thread at the front of the queue's thread list is woken and given the oldest packet. Conversely, whenever a thread that is associated with a queue wakes up after blocking the scheduler executes the function **KiUnwaitThread**, which increments the queue's active count.

Finally, the **PostQueuedCompletionStatus** Win32 API calls upon the native function **NtSetIoCompletion**. As with the other native APIs in the completion port group, this one invokes a system service bearing the same name, which simply inserts that packet onto the completion port's queue using **KeInsertQueue**.

Not Exported

Windows NT's completion port API provides an easy-to-use and efficient way to maximize a server's performance by minimizing context switches while obtaining high-degrees of parallelism. The API is made possible with support in the I/O Manager, Kernel, and system services. While the Queue object is exported for use by device drivers (it is undocumented but its interfaces are relatively easy to figure out), the completion port APIs are not. However, if the queue interfaces are derived it is possible to mimick the completion port interfaces by simply using the queue routines and manually associating file objects with queues by setting the *CompletionContext* entry.

Back to Top