The Windows XP IRP Completion Primer

By Enrico Martignetti

First edition, June 2009

Table of Contents

Introduction

This document explains how the I/O manager and a stack of layered drivers interact to carry out completion of an I/O operation. It is completely based on two articles published on The NT Insider ([1], [2]) and does not add new information to what is presented in them. Rather, it tries to present a synthesis of their content in a way which may result more understandable to a less experienced reader.

Both [1] and [2] contain more information than what is presented here, but may be easier to understand after having read this document.

Test code is provided, albeit minimal, which comprises a driver and a client program. The client program opens an handle to the driver's device and executes an asynchronous read operation. By commenting and uncommenting various portions of code, the behavior of the I/O manager seen by the client can be examined.

The test driver includes all the test cases as part of its DispatchRead() routine. Again, by commenting/uncommenting portions of code its possible to examine the result of different completion behaviors from the driver.

Caution: the purpose of many tests is to observe how I/O operations may be left unfinished inside the system or how bad behavior can cause bug checks. You'll need an "expendable" test machine for that.

See Building The Sample Code for details on the build process.

The tests described in this document have been performed on Windows XP SP 1.

Effect of The Dispatch Routine Returning STATUS_PENDING

To begin with a simple scenario, we will consider for now a single driver, without lower level drivers beneath itself and we will analyze how the I/O manager reacts to the NTSTATUS value returned by the driver Dispatch routines.

In this section we will discuss what happens when the I/O manager receives STATUS_PENDING.

The I/O manager returns this information to the program which initiated the I/O operation.

To better understand how, we can use the test code. In order to perform this test, DispatchRead() inside the driver code must not do anything except returning STATUS_PENDING; all the rest must be commented out. In TestClt.cpp, it's better to have the call to GetOverlappedresult() and the code below it, which tests its return status, uncommented.

If we execute the test client, we see that ReadFile() returns with ERROR_IO_PENDING and GetOverlappedResult() blocks. If we had tried to execute a synchronous read, ReadFile() would have blocked.

It's worth to note a few points here:

- for this to happen it is enough that the driver is returning STATUS_PENDING, regardless of whether it has called IoMarkIrpPending().
- if the client is making asynchronous I/O, control is returned to it.
- in a highest level driver called directly by the I/O manager, the dispatch routine is called in the context of the thread which initiated the I/O operation (see, for instance, the section titled *Dispatch Routines and IRQLs* in the DDK Help). This means that only when the dispatch routine returns, the user mode caller may get something back. For instance, calling IoMarkIrpPending() does not mean the user mode thread returns from the I/O API with ERROR_IO_PENDING. Actually, the user mode thread is the same one who is living in kernel mode and executing the dispatch routine.

If the dispatch routine just returns STATUS_PENDING, without doing anything else, the I/O never completes. Even an attempt to terminate the process blocked on GetOverlappedResult() fail: the process does not close.

Furthermore, if the thread does not call GetOverlappedResult() so that it does not block, and tries to go on until it terminates, its termination is suspended, because of the unfinished IRP.

It's possible to observe this with the test code by commenting the call to GetOverlappedResult() in the test client: the program executes until it returns from wmain(), but then does not close.

It turns out that to allow the process to terminate, we should handle cancellation of IRPs. An excellent explanation of how to do this can be found in [3].

Proper completion of a pending IRP

Everybody has probably been told too many times that, in order to properly handle a pending IRP we should:

- call IoMarkIrpPending()
- queue the IRP for some later processing (StartIo(), etc.)
- return STATUS_PENDING
- the code doing the later processing will call IoCompleteRequest() for the IRP.

The call to IoCompleteRequest() may happen in a different thread from the one executing the dispatch routine. It may happen after the dispatch routine returns with STATUS_PENDING, or even before it does.

[1] explains that IoCompleteRequest() does the following:

- calls an eventual completion routine attached to the IRP
- checks inside the current I/O stack location for this IRP, looking into the Control field. If the SL_PENDING_RETURNED bit inside this field is set, it enqueues an APC for the thread which initiated the I/O operation (I am not beign 100% accurate here, but we will return on this later). This is the thread which executed the dispatch routine and which may have already returned STATUS_PENDING or may lie pre-empted inside the dispatch routine.

  The APC will execute I/O manager code which will return the result of the I/O operation to the calling process, copying it into its address space. Then this code will destroy the IRP. This is the code which finalizes the I/O operation.

  The SL_PENDING_RETURNED bit is set when IoMarkIrpPending()is called for our IRP. It is set in the current I/O stack location. This can be verified by inspecting wdm.h, because IoMarkIrpPending() is actually a macro defined there.

  If the SL_PENDING_RETURNED bit is not set, the I/O manager does not schedule the APC.
- In both cases, control is then returned to the caller of IoCompleteRequest().

It is therefore important to understand that, in this scenario, the SL_PENDING_RETURNED bit must have been set before calling IoCompleteRequest(). Otherwise, the request will never complete, even though IoCompleteRequest is called, because its execution will not trigger the APC which finalizes the I/O.

You can verify this by setting up the code in DispatchRead() so that it does the following:

- calls IoCompleteRequest()
- returns STATUS_PENDING

The result is again a hung I/O operation, as it was when we did not call IoCompleteRequest().

On the other hand, if you set up DispatchRead() so that it:

- calls IoMarkIrpPending()
- calls IoCompleteRequest()
- returns STATUS_PENDING

The I/O works: ReadFile() returns ERROR_IO_PENDING, then GetOverlappedResult() finds the I/O completed soon thereafter (at last! - we have actually performed a test which does not mess up the test machine).

It's worth noting that the I/O manager decides wether to enqueue the APC or not solely on the basis of information stored inside the I/O stack location (i. e. the SL_PENDING_RETURNED bit). It does not rely on the STATUS_PENDING value returned by the dispatch routine. This makes sense because the call to IoCompleteRequest() can take place even before the dispatch routine returns (as in our test).

It's also important to understand that, with the sequence of calls above, we must return STATUS_PENDING, even though we have already called IoCompleteRequest. This is necessary because we called IoMarkIrpPending(), so the execution of IoCompleteRequest() took note of the fact that the dispatch routine was returning STATUS_PENDING and enqueued the APC. If we return STATUS_SUCCESS now, we are not being consistent with what is going on inside the I/O manager and the system crashes. More on this later.

This is what happens when a dispatch routine returns STATUS_PENDING, now we will see what happens when it returns STATUS_SUCCESS.

Effect of The Dispatch Routine Returning STATUS_SUCCESS

We already know that when we return STATUS_SUCCESS we must not call IoMarkIrpPending(), so how does the I/O complete in this case?

[1] explains that, because of the STATUS_SUCCESS returned, the I/O manager executes the completion code (copy of the outcome into user mode address region, destruction of the IRP) in the thread which initiated the I/O, before returning to the caller. In other words, when the dispatch routine returns to the I/O manager, the latter calls the completion code because the return value from the dispatch routine is STATUS_SUCCESS; then it returns to the caller.

So this is why the I/O complete even though IoCompleteRequest() does not enqueue the APC.

Actually, [1] states that this is the behavior for most NTSTATUS values returned by the dispatch routine, as long as the returned value is not STATUS_PENDING.

For instance, when we return an NTSTATUS indicating an error from a dispatch routine, we must not call IoMarkIrpPending(), but just setup the fields of IoStatus, call IoCompleteRequest() and return our NTSTATUS. In this case also, the outcome is synchronously copied into the user mode address region.

Why Does The I/O Manager Handle The Returned NTSTATUS Like This

As explained in [1], the concept behind this design is that, when the dispatch routine returns something different from STATUS_PENDING, the driver is telling the I/O manager that it has completely processed the request and already called IoCompleteRequest(). So the I/O manager can count on having at his disposal the outcome of the operation in the IRP (and the eventual data to copy, in case of buffered I/O) and completes the I/O immediately.

The I/O manager must execute in the context of the thread which started the I/O, because it must copy data into the user mode address region of the calling process. The thread which called the dispatch routine is exactly the right one so the I/O manager code starts doing the job, as soon as it notices the return value is not STATUS_PENDING.

On the other hand, when the dispatch routine returns STATUS_PENDING, it is telling the I/O manager that, at some other point in time it will perform the I/O and call IoCompleteRequest(), but this can happen at any time after the dispatch routine has returned, of even before it does. So the I/O manager does nothing upon receiving control from the dispatch routine and just returns the pending status to the caller.

The I/O manager will enqueue the APC when the driver has completed the request, but only if the driver notifies it by setting the SL_PENDING_RETURNED inside the I/O stack location.

The reason why an APC is used is also explained in [1]: APCs execute in the context of a specific thread, so this one is targeted to the thread which started the I/O, allowing the I/O manager to copy data into the user mode address region.

It's worth noting that if the user mode code performed the I/O asynchronously, the thread may be doing something else when the APC interrupt is acknowledged. The thread will then be diverted by the kernel to the I/O manager finalization code and the user mode code will see the effect of the completed I/O (e. g. event signaled inside the OVERLAPPED structure).

From what we have seen so far, we know that the I/O manager determines its behavior from two pieces of information: the return value from the dispatch routine and the SL_PENDING_RETURNED bit in the I/O

stack location, when IoCompleteRequest is called. So far we are considering a stack made up of a single driver, so we have only one stack location.

Not Following The IRP Completion Rules

Now we can understand what happens if we misbehave.

We have already seen that a driver which

- does not call IoMarkIrpPending()
- returns STATUS_PENDING
- at some point in time (may be before returning STATUS_PENDING) calls IoCompleteRequest

cause the I/O operation to never finish.

Another case is

- call IoMarkIrpPending()
- call IoCompleteRequest()
- return STATUS_SUCCESS

The result is the I/O is completed by the APC code, because of the IoMarkIrpPending() call and then again when the dispatch routine returns and the I/O manager sees STATUS_SUCCESS. Actually, these two events could also take place in different order: first the dispatch routine returns and the IRP is completed, then the APC executes and the IRP is completed again.

In both cases, this results in a bugcheck 0x44 (MULTIPLE_IRP_COMPLETE_REQUESTS), because an IRP cannot be completed twice.

It's even possible that the memory used by the IRP has already been reused by the time the second completion is attempted, so instead of the double completion bugcheck there could be every sort of strange behavior due to corruption of kernel memory.

The scenario can be tested in two ways with the test code.

First, it's possible to set up DispatchRead so that it performs the sequence IoMarkIrpPending(), IoCompleteRequest(), return STATUS_SUCCESS. This usually results in bugcheck 0x44 while the client thread is still inside ReadFile().

Second, DispatchRead() also allows you to use a timer to schedule the call to IoCompleteRequest with a 10'' delay. This ensures the dispatch routine returns first and the bugcheck occurs when the timer routine calls IoCompleteRequest().

This scenario is much more interesting, because the client process actually has time to terminate and report that ReadFile() executed just fine, because of the returned STATUS_SUCCESS. Then, without doing anything else, after about 10'' the system crashes, apparently by itself (Who? Me? I wasn't even running when it happened, your honor!).

This leads to an interesting question: IoCompleteRequest() should enqueue an APC targeted at the thread which initiated the I/O, which, at the time IoCompleteRequest() is called, has terminated. So why do we get a bugcheck 0x44, as if the (vanished) thread were completing the IRP for the second time and not a different one, say a were-the-heck-is-my-thread bugcheck?

If you take the time to step through IoCompleteRequest() with the debugger, you'll discover that the first thing it does (inside a function named IopfCompleteRequest()) is to discover that the IRP has already been completed and raise bugcheck 0x44, way before enqueuing the APC. Thus, it is IoCompleteRequest() itself which checks for the double completion and not the code which would be executed by the APC, should we make it to it. At least this is what happens as of Windows XP SP 1.

Extension to A Stack of Layered Drivers

I/O Manager Behavior

So far the scenario was: a single driver called by the I/O manager. Now let's see what happens when we have a driver stack, with IRPs passed down the stack with IoCallDriver().

The IoManager sees only the NTSTATUS value returned by the top level driver, so this is what determines wether the I/O must be completed immediately or not, as explained in [1]. So, as before, if the returned value is not STATUS_PENDING, the I/O is completed immediately, otherwise the pending status is returned to the caller.

What needs more explanations is the behavior of IoCompleteRequest().

For now, let's assume every layer in the stack has installed an I/O completion routine.

It is a known fact that IoCompleteRequest causes all the completion routines attached to the different I/O stack locations to execute. [1] explains that, when each completion routine returns, IoCompleteRequest checks the SL_PENDING_RETURNED bit for the current stack location. If it's set, it sets the Irp->PendingReturned field inside the IRP.

Afterwards, there are two cases.

1) The completion routine which returned was not the one of the topmost driver: IoCompleteRequest moves the stack pointer up one level and calls the completion routine for the next upper layer. This next completion routine can know wether SL_PENDING_RETURNED was set in the next lower stack location by looking at Irp->PendingReturned.

It's worth noting that the currently executing completion routine cannot look at the I/O stack location of the next lower driver, so it makes sense that the information represented by SL_PENDING_RETURNED is duplicated by setting PendingReturned inside the IRP, which is accessible by all layers.

2) The completion routine which returned was the one of the topmost driver: IoCompleteRequest() looks at Irp->PendingReturned to decide wether to enqueue or not the completion APC. So, to be more accurate, IoCompleteRequest() does not look directly at the SL_PENDING_RETURNED bit to decide wether to schedule the APC or not, but rather at Irp->PendingReturned.

Anyway, when IoCompleteRequest() reaches the topmost driver, SL_PENDING_RETURNED causes the APC to be enqueued, just as we saw in the one layer case.

Returning an IRP from The Next Lower Level

To understand the implications of this, consider a stack of two layers and suppose the upper driver has installed a completion routine.

Now suppose for some I/O operation the topmost driver calls the lowest one and then wants to return the result to the I/O manager, without modifications. The dispatch routine for the topmost driver will be something like this example, copied from [2]:

```
NTSTATUS
YourDispatchRead(PDEVICE_OBJECT DeviceObject,
                 PIRP Irp) {

        PIO_STACK_LOCATION ioStack;
        PYOUR_DEV_EXT devExt;

        ioStack = IoGetCurrentIrpStackLocation(Irp);
        devExt = DeviceObject->DeviceExtension;

        //
        // you do some validation here
        //
        IoCopyCurrentIrpStackLocationToNext(Irp);

        //
        // Maybe you manipulate some of the IRP
        // stack parameters here.
        //
        return(IoCallDriver(devExt->LowerDriver, Irp));
}
```

Now suppose the lower driver returned STATUS_PENDING. The topmost driver is returning the same value to the I/O manager. This means it must also set the SL_PENDING_RETURNED bit inside its I/O stack location. In the end, it is *the topmost* stack location which will cause the I/O manager to schedule the APC which completes the I/O. This comes from the behavior of IoCompleteRequest() outlined above (explained in [1]): when, and only when, the *topmost* completion routine returns, the I/O manager checks Irp->PendingReturned to see if it has to enqueue the APC.

So the upper driver must call IoMarkIrpPending(). As explained both in [1] and in [2], it cannot do it this way (example copied from [2]):

```
status = IoCallDriver(devExt->LowerDriver, Irp);

        if (status == STATUS_PENDING) {
                IoMarkIrpPending(Irp);
        }

        return(status);
```

Because, after the call to IoCallDriver, the code cannot touch the IRP anymore. For instance, the call to IoCompleteRequest from the lower driver could already have happened.

But, as [1] and [2] explain, inside its completion routine the upper driver can:

1) legally access its own stack location.

2) know wether the lower level returned STATUS_PENDING, because it can check Irp->PendingReturned.

So the completion routine could be like this (sample copied from [2]):

```
        [...]
        If(Irp->PendingReturned) {
                IoMarkIrpPending(Irp);
        }

        return(STATUS_SUCCESS);
```

This passes the I/O manager a consistent set of information: on one hand, the topmost dispatch routine returns STATUS_PENDING. On the other hand, the topmost I/O stack location has SL_PENDING_RETURNED set by the time the topmost completion routine returns.

The information seen by the I/O manager is consistent also for return values other than STATUS_PENDING: the SL_PENDING_RETURNED bit will not be set.

The upper driver can safely pass the IRP up to the I/O manager, regardless of how the lower driver completes it: synchronously or asynchronously.

Now consider a stack made of an arbitrary number of layers, suppose our upper one is somewhere in the middle and analyze the STATUS_PENDING case. In this scenario, our upper driver is receiving these two pieces of information from the one below itself:

- the dispatch routine returns STATUS_PENDING
- by the time our completion routine is called, Irp->PendingReturned is set.

By behaving like it does, it is returning the very same information to the level above itself: it's returning STATUS_PENDING and its SL_PENDING_RETURNED bit set will cause Irp->PendingReturned to be set when the upper level completion routine will be called.

The upper driver is being "transparent": the driver above it receives the same status it would have received, if it had called the lower driver directly.

So we can have an arbitrary number of driver behaving this way and the result is the completion status correctly bubbles up to the I/O manager, which will complete the IRP in a proper way.

It is worth noting how, in this scenario, *every* driver in the stack calls IoMarkIrpPending(). This is different from IoCompletRequest(), which is called *only once* by the lowest driver.

General Rules for IRP Completion

Now consider a more generic scenario, where drivers in the stack does not necessarily pass up IRPs transparently, but manipulate them, maybe create new ones, etc.

Suppose each driver in the stack observe these rules:

- if the dispatch routine returns to the upper level STATUS_PENDING, SL_PENDING_RETURNED must be set in the driver's stack location *by the time* IoCompleteRequest() examines it.

  Note that:

  - if the driver is the one calling IoCompleteRequest(), this means it must call IoMarkIrpPending() before the former call.
  - otherwise, IoCompleteRequest() looks at the stack location for the driver when its completion routine returns (if any; for now we go on assuming there is one). So, this completion routine must call IoMarkIrpPending() before returning.

  This rule means: by the time IoCompleteRequest() gets to the upper level, Irp->PendingReturned will be set.
- if the dispatch routine returns any other value, SL_PENDING_RETURNED is cleared by the time IoCompleteRequest() examines it.

These two rules imply that:

- if the layer above the driver is the I/O manager, it will properly complete the operation
- otherwise the upper layer is another driver, which has enough information to decide what to do (may be it created the IRP itself and will wait for its completion).

The code we saw, which passed the IRP result transparently to its upper level, is written to satisfy these rules, both when the lower level returns STATUS_SUCCESS and when it returns STATUS_PENDING. It is the implementation of these rules when we want to just pass the IRP to whoever is above us.

[2] explains how drivers must actually conform to a set of rules which imply the two outlined above.

Indeed, since the NT architecture is based on layered drivers and allows for the insertion of filter drivers in the middle, there must be a set of rules each layer follows, so that a new driver inserted somewhere know what to expect from the driver beneath itself. The rules outlined here make sense because, to the core, they amount to this: *each layer must behave as if it had the I/O manager above itself*.

Up until now, we assumed every driver in the stack installed a completion routine. Both [1] and [2] explains that when a driver does not, IoCompleteRequest() does the following: it behaves as if there was a completion routine like this (code fragment copied from [2]):

```
[...]
If(Irp->PendingReturned) {
        IoMarkIrpPending(Irp);
}

return(STATUS_SUCCESS);
```

So IoCompleteRequest() automatically "propagates" the pending status if a completion routine is not set. This allows drivers who don't set completion routines and do

```
return(IoCallDriver(devExt->LowerDriver, Irp));
```

to work correctly.

The Sample Code

The sample code package contains a Visual C++ Express solution.

You don't need to have VS installed to edit the code and build the driver. The solution projects can just be regarded as subfolders of the overall source tree and all the building is done with the DDK build utility or the SDK nmake tool. VS can be used as a text editor and to access the various files through the Solution Explorer window, but this is not required.

Building The Sample Code

The test driver comes with the scripts needed to build it with the *build* utility, executed in one of the DDK build environment windows. Just move to the directory Driver\WxpBuild and run build. A “clean” target is provided as well, so that running

```
build –clean
```

will throw away everything from a previous build.

As for the test client, a makefile can be found in the directory named TestClt, which can be used to build it with nmake, from a standard SDK build environment which must include the settings for Visual C++.

Loading And Running The Driver

The test client does not load the driver by itself, so some loading utility is needed.

The one I use and I find very handy is w2k_load.exe by Sven B. Schreiber, originally found on the companion CD of his book, Undocumented Windows 2000 Secrets. The CD image can be downloaded from Mr. Schreiber site at:

http://undocumented.rawol.com/

In spite of having been written for Windows 2000, w2k_load still works like a breeze under XP and Vista.

Loading the driver is as simple as entering

```
W2k_load IoCompl.sys
```

And to unload it:

```
W2k_load IoCompl.sys /unload
```

References

[1] Secrets of the Universe Revealed! - How NT Handles I/O Completion; The NT Insider, Vol 4, Issue 3, May-Jun 1997; available at http://www.osronline.com/article.cfm?id=83 (registration required)

[2] Properly Pending IRPs - IRP Handling for the Rest of Us; The NT Insider, Vol 8, Issue 3, May-Jun 2001; available at http://www.osronline.com/article.cfm?id=21 (registration required)

[3] The Truth About Cancel - IRP Cancel Operations (Part I/II), The NT Insider, Vol 4/5, Issue 6/2, Nov-Dec 1997/ Mar-Apr 1998; available at http://www.osronline.com/article.cfm?id=78 (registration required)