How NT Handles I/O Completion

Previously we described how to "roll your own IRPs" for performing I/O in the kernel. This is a very powerful technique and one which is highly useful for those building device drivers, file system drivers, and filter drivers. In this issue we talk about how the I/O Manager completes IRPs.

While this might seem like a simple topic, it turns out there are a number of subtle issues involved. If you don t fully understand I/O completion, proper operation of your driver could be at risk. For example, one problem we see very frequently in driver code that we review involves setting and clearing I/O completion routines. Understanding how the I/O Manager performs completion processing will ensure that you understand why I/O completion routines must be set up "just so."

In this article we describe the I/O stack as it relates to I/O completion processing, how to set up the I/O stack when calling a lower level driver, how the I/O Manager unrolls the I/O stack for completion processing, and how you should set up your I/O completion routine to work properly.

The I/O Stack

Figure 1 shows an I/O Stack with three stack locations, as well as the contents of the I/O completion routine within each stack location.

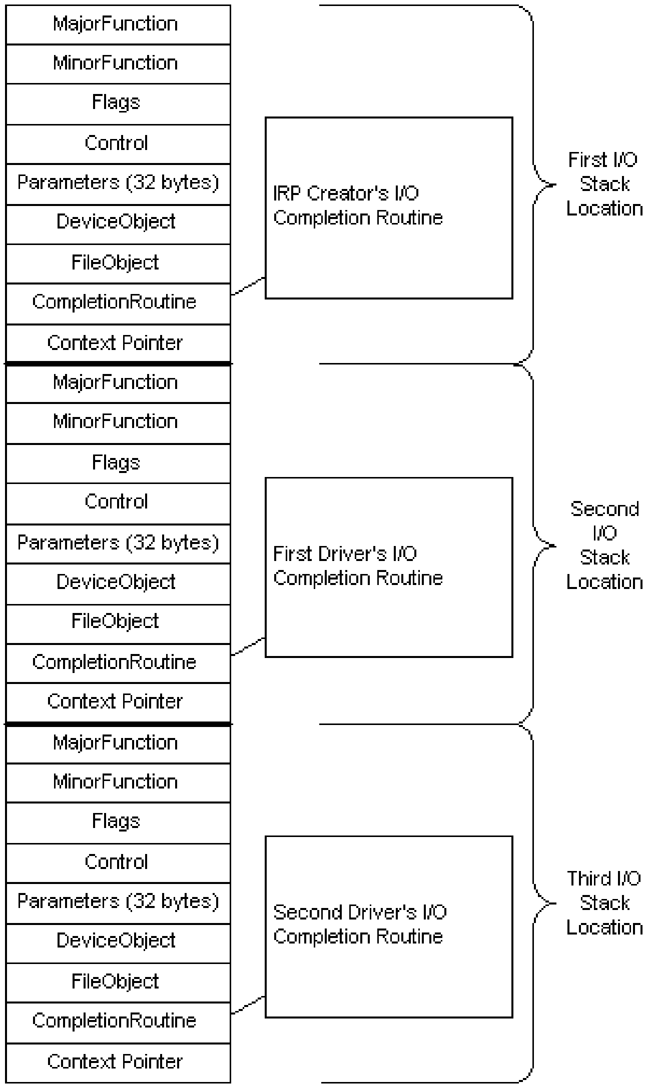

Figure 1

The stack locations are used from the top to the bottom (i.e., the I/O stack grows "down" in memory). You can see this by examining the implementation of the I/O Manager function IoSetNextIrpStackLocation (a macro inside of ntddk.h):

```
#define IoSetNextIrpStackLocation( Irp ) { \

(Irp)->CurrentLocation--; \

(Irp)->Tail.Overlay.CurrentStackLocation--; }
```

When the IRP is first created, the current IRP stack location is not valid and calling **IoGetCurrentIrpStackLocation()** will return a pointer to a location inside of the IRP structure itself. However, the next IRP stack location, retrieved by using **IoGetNextIrpStackLocation()** is valid and in fact is used when setting up the parameters for the first driver to be called (the mechanics of setting up the first I/O stack location were described in the article "Rolling Your Own" , *The NT Insider V4N1*).

Setting Up The I/O Stack For Completion

One thing that can be done when processing up an IRP is the establishment of a completion routine. A completion routine is simply a routine that is called whenever the I/O described by the IRP is completed by the next lower driver. A completion routine can be specified for the case of the I/O request completing with success, error, or cancellation, as well as any combination of these. Completion routines for a particular IRP are established using the function **IoSetCompletionRoutine()**. If your driver does not wish to be notified on I/O completion, it should also indicate that before passing the IRP down to the next lower caller. This can be done with the following code:

```
IoSetCompletionRoutine(Irp, NULL, NULL, FALSE, FALSE, FALSE);
```

IoSetCompletionRoutine() is a macro and its definition is in ntddk.h (See *Figure 2*).

```
#define IoSetCompletionRoutine( Irp, Routine, CompletionContext, Success, Error, Cancel ) {
PIO_STACK_LOCATION irpSp;
ASSERT( (Success) | (Error) | (Cancel) ? (Routine) != NULL : TRUE );
irpSp = IoGetNextIrpStackLocation( (Irp) );
irpSp->CompletionRoutine = (Routine);
irpSp->Context = (CompletionContext);
irpSp->Control = 0;
if ((Success)) { irpSp->Control = SL_INVOKE_ON_SUCCESS; }
if ((Error)) { irpSp->Control |= SL_INVOKE_ON_ERROR; }
if ((Cancel)) { irpSp->Control |= SL_INVOKE_ON_CANCEL; } }
```

Figure 2

One common instance where a completion routine is used is when a driver creates and manages its own IRP pool. In this case, the I/O completion routine traps the IRP, returns it to the driver s private IRP pool, and then returns**STATUS_MORE_PROCESSING**

_REQUIRED to the I/O Manager. This causes the I/O Manager to immediately cease processing the IRP completion and leaves the IRP with the driver.

The important point to notice about I/O completion routines, looking at the *Figure 1*, is that your driver s completion routine is not in*your* I/O stack location, but in the I/O stack location of the *next driver*. Why is it set up this way? One reason is that the last driver called does not need an I/O completion routine; After all, it is ultimately the driver performing the I/O completion (by calling **IoCompleteRequest()**). There is therefore no need for the I/O Manager to notify the lowest level driver that it just completed the I/O.

Another reason a driver s I/O completion routine is in the next I/O stack location concerns the way the IRP is initially built. The kernel mode component, such as a driver, that initially allocates the IRP does not require an I/O stack location. However, it is possible that it might require an I/O completion routine (to allow it to return the IRP to its private pool, for example). Recall that the last driver in the calling chain does not require a completion routine but does require an I/O stack location. Thus, for space efficiency, the N-1^{st} driver s completion routine is stored in the N^{th} driver s I/O stack location.

Since the I/O Manager does not use I/O completion routines itself, if you are developing a highest level driver you will almost never see a completion routine in your I/O stack location. Some system components, however, create their own IRPs and pass them along to highest level drivers, specifying their own I/O completion routines. For example, SRV (the Lan Manager File Server) and NTVDM (the MS-DOS emulation layer) both build their own IRPs and pass them to highest-level drivers. And these components both set I/O completion routines into the IRPs they create.

As a result of misunderstanding the location of completion routines in the I/O stack, a very common mistake occurs when passing an IRP from your driver to an underlying driver. This mistake, illustrated in *Figure 3*, is to simply copy the current I/O stack location to the next driver s I/O stack location. The problem is that copying the contents of the current stack location to the next stack location also copies the I/O completion routine along with it. This works fine*, as long as there is no completion routine in the current stack location.* If a completion routine *was* supplied, it will now appear in two different I/O stack locations, with the obvious result of it being called *twice*. This is generally not good, and results in an eventual blue screen or other unstable system behavior. Of course, the caller s code*could* be written to handle this case and everything will still work fine. But few driver writers anticipate that their completion routines will be called incorrectly!

```
PIO_STACK_LOCATION IrpSp;
PIO_STACK_LOCATION NextIrpSp;
```

```
IrpSp = IoGetCurrentIrpStackLocation(Irp);

NextIrpSp = IoGetNextIrpStackLocation(Irp);


*NextIrpSp = *IrpSp;


return IoCallDriver(NextDeviceObject, Irp);
```

Figure 3

For file system filter drivers, this problem often first surfaces when you begin filtering drives being served by SRV. SRV provides its own I/O completion routines and does not protect against having its completion routine called with a different driver s I/O stack location. The fix for this is to be sure that you always call **IoSetCompletionRoutine()** after copying the I/O stack location in your driver.

Of course, even if you write your filter driver code correctly, there might still be problems. Once example of the problems you can see is present in NTFS in NT 4.0. As late as SP2, NTFS did not clear the I/O completion routine when copying the I/O stack during **IRP_MJ_DEVICE_CONTROL** function processing. Thus, there were circumstances under which our completion routine was being called twice (once with NTFS device object!).

The defense against this particular problem is straight-forward. In your device object s extension area, you should keep some identifying signature: A unique value that you can use to determine (with high probability, at least) that this is your device object and not some other driver s device object being passed to your completion routine. If your completion routine is called with a device object that s not yours, you simply need to emulate the logic the I/O Manager would use had there been no completion routine. See the code fragment in *Figure 4* that demonstrates this. Now, if your completion routine is called with some other driver s device object, everything will work correctly.

```
if
(!DeviceObject>Extension||(POUR_DEVICE_EXTENSION)(DeviceObject>ExtensionMagicNumber)!=OUR_EXTENSION_MAGI
{

if (Irp->PendingReturned) {

IoMarkIrpPending(Irp);

}
```

```
return STATUS_SUCCESS;

}
```

Figure 4

A less common, but equally daunting problem we found recently was that prior to NT 4.0 Service Pack 2, there were certain cases in CDFS for **IRP_MJ_CLOSE** where the IRP was not being properly completed. Because of the mechanics of the I/O Manager, things seem to work correctly, even with the checked build (more on why this is the case in a moment). For a filter driver, however, the symptom of the problem is that the filter driver s completion routine is never called. While this was fixed in Service Pack 2, for those people who must support their product on earlier NT 4.0 versions or wish to handle this problem correctly should it reappear in the future, there is a work-around that relies upon the I/O completion processing. The code fragment in *Figure 5* describes a mechanism for detecting and working around this problem. This mainline code is then combined with a completion routine (*Figure 6*) which sets the event.

```
RtlCopyMemory(NextIrpStack, CurrentIrpStack, sizeof(IO_STACK_LOCATION));

KeInitializeEvent(&Event, NotificationEvent, FALSE);

IoSetCompletionRoutine(Irp,&OurCloseCompletion,&Event,TRUE,TRUE,TRUE);

IoMarkIrpPending(Irp);

status = IoCallDriver(Extension->FilteredDeviceObject, Irp);

if (status != STATUS_PENDING) {

        if (KeReadStateEvent(&Event) == 0) {

                IoCompleteRequest(Irp, IO_NO_INCREMENT);

        }

} else {

        KeWaitForSingleObject(&Event, Executive, KernelMode, FALSE, 0);

}


return STATUS_PENDING;
```

Figure 5

```
OurCloseCompletion(PDEVICE_OBJECT Device Object, PIRP Irp, PVOID Context)

{

        Event = (PKEVENT)Context;
```

```
        KeSetEvent(Event, 0, FALSE);

        return STATUS_MORE_PROCESSING_REQUIRED;

    }
```

Figure 6

The problem was that CDFS was returning **STATUS_SUCCESS** without ever calling **IoCompleteRequest()**. Indeed, it turns out that for most I/O operations, if a driver returns anything except **STATUS_PENDING** in its dispatch routine, the IRP being dispatched will be completed by the I/O manager. Of course, *you should never write your code to do this*. Completing IRPs without calling **IoCompleteRequest()** is a logic error in the driver that does it, and can cause all sorts of nasty problems in higher level drivers.

IRPs get completed in the case where anything except **STATUS_PENDING** is returned because the lower level driver should have already called **IoCompleteRequest()**, which in turn would have called your driver s I/O completion routine. This leads us to the most complicated part of I/O completion   understanding how the I/O Manager actually*does* I/O completion.

Role Of The I/O Manager

It turns out that the I/O Manager implements I/O completion processing in two separate and distinct stages. The first stage is thread context-independent, meaning that it can be performed in any arbitrary thread context. The second stage is thread context-dependent this means the step must be performed in the context of a specific thread, in our case the context of the thread that originally requested the I/O.

In the first stage of I/O completion processing, the IRP is "unrolled" and the I/O completion routines of each driver are notified that the I/O operation has completed. During this stage of processing, any driver that provided an I/O completion routine will be called back and will have a second opportunity to look at the IRP.

In the second stage of I/O completion, information from the IRP is copied back into the thread s user-space memory buffers. Thus, this operation must be done in the correct process context. Of course, even for calls where the bulk of data need not be copied, such as in the case for direct I/O, there is always some information that needs to be copied, such as the I/O status block. Once this is done, the IRP itself is torn down, which includes discarding any MDLs associated with the IRP and finally the memory of the IRP itself is freed.

The first stage of I/O completion is done when a driver calls **IoCompleteRequest()**. Note that **IoCompleteRequest()** is a void function, so the caller is provided with no additional information about the I/O completion. Indeed, the caller does not even know if the first stage of I/O completion is done, as a higher-level driver might have returned **STATUS_MORE_PROCESSING_REQUIRED** which suspended further first stage I/O completion handling.

The second stage of I/O completion happens at an arbitrary time after **IoCompleteRequest()** is called. This could mean that second stage I/O completion is done by the time **IoCompleteRequest()** returns, or it could even happen at some later time, depending upon other activities ongoing in the system.

The actual ordering of events is based upon the status of the IRP after the last I/O completion routine has been called. At that point, the I/O manager looks at the IRP to determine if **STATUS_PENDING** was returned (or will be returned since we don t know the exact order of operations at this point) from the highest level driver. If **STATUS_PENDING** was returned from the highest level driver, then the I/O manager queues an Asynchronous Procedure Call (APC) to perform the secondary I/O completion. It performs this check by looking at the PendingReturned field within the IRP itself.

Here s where it gets interesting. If**STATUS_PENDING** was not returned from the highest level driver, then the I/O manager simply returns control to the caller (i.e., it returns back to the driver that made the **IoCompleteRequest()** call). Recall that at this stage, the driver that called **IoCompleteRequest()** has no knowledge of the state of the IRP.

So where does the second stage I/O completion occur in this case? Why, in the I/O Manager, of course! In *Figure 7* we provide a graphical description of the flow of control in this case (the "synchronous I/O case").

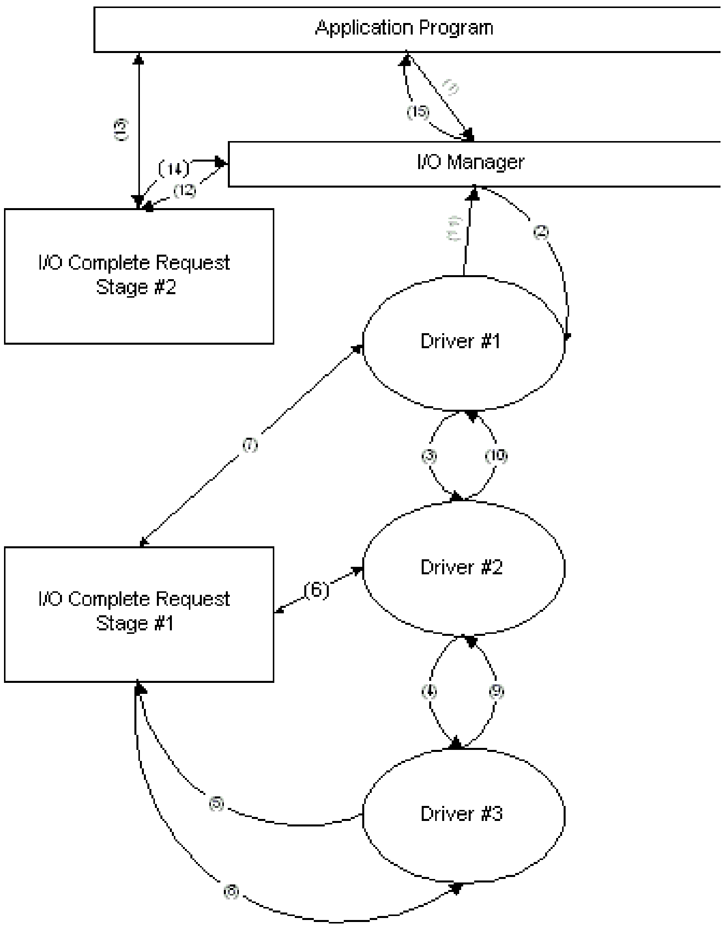

Figure 7

Before analyzing I/O completion further, it is a good idea to review each of these individual steps:

1. The application program issues an I/O request (read, write, etc.) to the I/O Manager.

2. The I/O Manager dispatches the I/O request by building an IRP and passing it to the first driver in the chain.

3. The first driver continues processing by sending the I/O request to the second driver.

4. The second driver continues processing by sending the I/O request to the third driver.

5. The third driver completes the I/O operation. It calls **IoCompleteRequest()** and initiates first stage completion.

6. During first stage completion the second driver s completion routine (if any) gets

called back.

7. During first stage completion the first driver s completion routine (if any) gets called back.

8. Next, **IoCompleteRequest()** returns control back to the third driver. First stage I/O completion is done.

9. The third driver returns a status value (e.g., **SUCCESS**) to the second driver.

10. The second driver returns a status value to the first driver.

11. The first driver returns a status value to the I/O Manager

12. The I/O Manager notes that second stage processing must be performed and initiates second stage I/O completion.

13. The second stage I/O completion routine returns the status of the I/O operation and copies any data out to the application s address space. Once that is done, the IRP is dismantled and discarded.

14. The second stage I/O completion routine returns control to the I/O Manager.

15. The I/O operation is now complete and control returns to the caller.

The interesting point in this figure is step 12 where the I/O Manager initiates the second stage I/O completion processing. This works correctly because the call completed synchronously and the I/O Manager "knows" that it is in the correct thread context AND the first stage I/O completion processing has been properly completed.

Recall the CDFS problem in NT 4.0 we described earlier, where in one particular situation CDFS failed to call **IoCompleteRequest()**, and instead just returned **STATUS_SUCCESS** in its dispatch function? Well, the reason that it worked correctly was because the I/O Manager was performing the stage two completion processing, thus making sure that the information was being transferred back to the application program as well as freeing the IRP. Of course, any intermediate drivers never learned about that I/O completion. Stage one processing (steps 5 through 8 in the diagram) was never performed!

In the asynchronous case, these events occur in a *very* different order. This is shown in *Figure 8*.

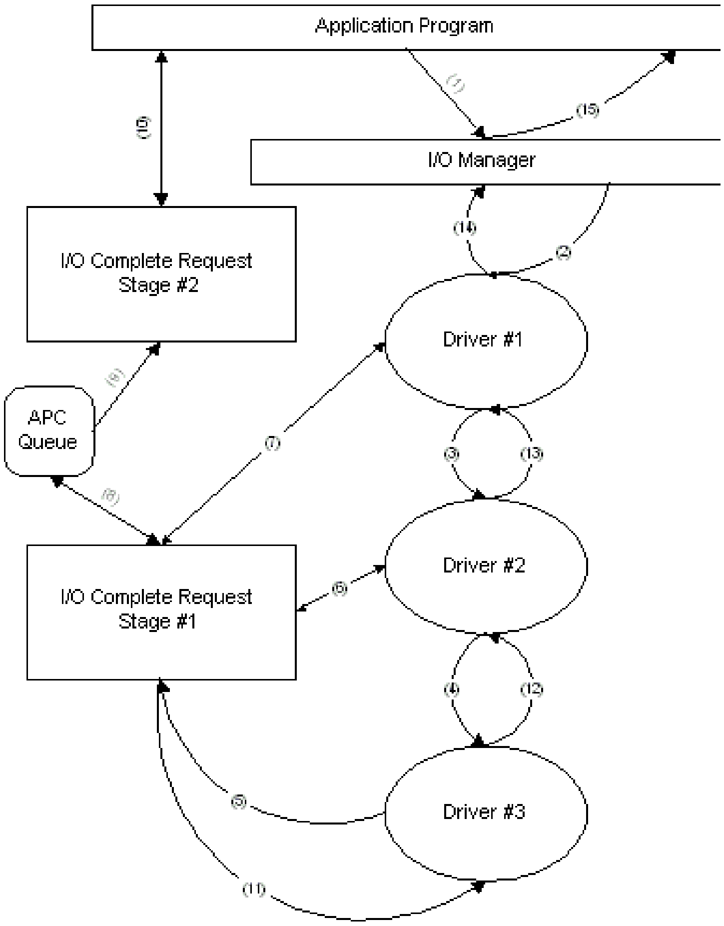

Figure 8

Again, here is a description of each of the labeled steps:

1. The application program issues an I/O request (read, write, etc.) to the I/O Manager.

2. The I/O Manager dispatches the I/O request by building an IRP and passing it to the first driver in the chain.

3. The first driver continues processing by sending the I/O request to the second driver.

4. The second driver continues processing by sending the I/O request to the third driver.

5. The third driver completes the I/O operation. It calls **IoCompleteRequest()** and initiates first stage completion.

6. During first stage completion the second driver s completion routine (if any) gets

called back.

7. During first stage completion the first driver s completion routine (if any) gets called back.

8. Next, **IoCompleteRequest()** queues the I/O completion APC to handle second stage I/O completion. First stage I/O completion is done.

9. The I/O completion APC is dequeued and second stage I/O completion starts. Note that this is done in the context of the thread which originated the I/O.

10. The I/O status and data is copied to the user address space.

11. **IoCompleteRequest()** returns back to the third driver.

12. The third driver returns a status value (e.g., **SUCCESS** or **PENDING**) to the second driver.

13. The second driver returns a status value to the first driver.

14. The first driver returns a status value to the I/O Manager.

15. The I/O operation is now complete and control returns to the caller.

In this case (the "asynchronous I/O" case) the second stage I/O completion processing (steps (9) and (10)) occur in an arbitrary order relative to the return of control from **IoCompleteRequest()** back to the application program (steps (11) through (14)).

The actual return to the application (step (15)) may be serialized by the I/O Manager, such as in the case where the original calling application asked for synchronous I/O behavior. Of course, the I/O Manager does this the same way any other Windows NT kernel mode code would do it   by waiting on an event (in this case, an event in the IRP). When the stage two completion processing is done, that event will be set and the status of the I/O operation is returned to the calling application (step (15)).

Also note that the steps of calling from driver to driver, and the subsequent return, could even each be performed asynchronously. Thus, if the first driver in the chain wishes to queue the request and process it asynchronously, it would return **STATUS_PENDING** to the I/O Manager (step (14)).

If the caller did not request synchronous behavior, then at step (15) the I/O Manager returns control back to the caller, indicating that the I/O is in progress. During stage two I/O completion processing the I/O is then completed and the status of the operation is copied back into the appropriate address space. The application can then determine that the I/O has completed by using one of the existing mechanisms, such as an APC routine, waiting on an event, or by polling to see if the I/O is done. In the interim, the thread that initiated the I/O can continue handling normal processing, or even start additional I/O operations.

Setting Up Your Completion Routine

Because the use of completion routines within certain types of drivers is fairly common, understanding the model used within Windows NT for I/O completion processing can help developers build more robust drivers and locate problems within their own driver. One common example of this is code we have seen used numerous times in an attempt to trap **IRP_MJ_CREATE** calls above a file system. The new filter driver writer will attempt to write something like *Figure 9*, and in their completion routine do something like *Figure 10*.

```
DbgPrint("Createfor%Z\n",&IrpSp>FileObject>FileName;

return IoCallDriver(NextDeviceObject, Irp);
```

Figure 9

```
// Build a work item for additional processing

ExQueueWorkItem(&WorkItem);

return STATUS_MORE_PROCESSING_REQUIRED;
```

Figure 10

A reading of the DDK documentation leaves one with the belief that this will work correctly. Unfortunately, it turns out it does not. The I/O Manager treats **IRP_MJ_CREATE** as a synchronous I/O operation (recall the flow of control in this situation is shown in *Figure 7*.) The fact that your completion routine returned **STATUS_MORE_PROCESSING _REQUIRED** (at step 7) is NOT indicated back to the lowest level driver (**IoCompleteRequest()** is a void function, so it is not indicating this back to the caller). However, the mainline code then returns the results of the create operation to the calling driver, which in turn is returned to the I/O Manager. The I/O

Manager concludes that because this is the synchronous I/O case and the return value was something other than **STATUS_PENDING,** stage two I/O completion processing must be executed.

Here s the rub   if your driver s work item has already completed, everything is fine since you called **IoCompleteRequest()** from within your work routine. If your driver s work item has not yet been processed, when it *is* processed and you call **IoCompleteRequest()**, the system crashes with the **MULTIPLE_IRP_COMPLETIONS** blue screen.

It is insufficient to just return **STATUS_MORE_ PROCESSING_REQUIRED** from your completion routine. In addition to that you must do one of two things in your dispatch entry point:

A. Either return STATUS_PENDING; or

A. Make the I/O request synchronous.

Doing either one of these is sufficient to then allow your completion routine to safely return **STATUS_MORE_PROCESSING_REQUIRED**. Thus, if the driver returns **STATUS_PENDING** (case (A)) then the mainline code would look something like *Figure 11*:

```
DbgPrint("Createfor%Z\n",&IrpSp->FileObject->FileName);

IoMarkIrpPending(Irp);

(void) IoCallDriver(NextDeviceObject, Irp);

return STATUS_PENDING;
```

Figure 11

Recall earlier we noted that the I/O Manager queues an APC to perform second stage I/O processing if **STATUS_PENDING** was returned from the highest level driver. However, there is no ordering between when the I/O Manager determines if **STATUS_PENDING** was returned and when **STATUS_PENDING** is actually returned from the driver. Indeed, all we know is that at some point (past, present, or future) **STATUS_PENDING** will be returned from the highest level driver.

This information (that the highest level driver will return **STATUS_PENDING**) is stored in the I/O stack location for that driver. This is the **SL_PENDING_RETURNED** bit, which is stored in the Control field. In your driver, you set this bit by calling **IoMarkIrpPending()**. As the I/O Manager processes each stack location in turn, it sets the Irp->PendingReturned field to indicate if the **SL_PENDING_RETURNED** bit was set in the next lower driver s Control field. After the last driver s completion routine has been called, the I/O Manager uses information in the Irp->PendingReturned field to determine if it needs to queue an APC for the stage two I/O completion, which is why your driver must both call **IoMarkIrpPending()** *and* return **STATUS_PENDING**. If your driver does only one or the other, the system will misbehave in various ways.

If the driver makes the I/O synchronous (case (B)) then the mainline code would look something like *Figure 12*. The event in question is then set in the driver s completion routine (normally, this is passed as part of the context information to the completion routine) or from a work routine queued by the completion routine.

```
DbgPrint("Create for %Z\n", &IrpSp->FileObject->FileName);

Status = IoCallDriver(NextDeviceObject, Irp);

KeWaitForSingleObject(&Event, Executive, KernelMode, FALSE, 0);
```

Figure 12

While making I/O synchronous, as we do in case (B), is the easiest option of the two, it cannot be used in all cased because it interferes with any I/O operation that requires asynchronous behavior, and there are a set of such operations. For one of these asynchronous I/O operations, attempting to make the I/O synchronous will cause it to break.

Similarly, making I/O asynchronous, as we do in case (A), will cause certain asynchronous operations to work improperly because they treat a return value of **STATUS_PENDING** as a form of success. One example of this are file system "oplocks". The oplock protocol treats a return value of **STATUS_PENDING** as a grant of the oplock. Unfortunately, if a driver (such as a file system filter driver) returns **STATUS_PENDING** in such a case it can lead to cache consistency problems and data corruption for network clients.

One thing we mentioned before, but that deserves a second mention is that if your driver returns **STATUS_PENDING** it must also call **IoMarkIrpPending()**. So, consider a driver that does the following:

```
return IoCallDriver(NextDeviceObject, Irp);
```

If the driver below returns **STATUS_PENDING**, this driver must call **IoMarkIrpPending()**. The next thing many driver writers have tried is the following:

```
Status = IoCallDriver(NextDeviceObject, Irp);

if (Status == STATUS_PENDING) IoMarkIrpPending(Irp);

return Status;
```

This will work *most* of the time. The remainder of the time it will cause the system to crash, threads to hang, or other unsavory behavior. Keep in mind, that once you pass the IRP on to the next driver (via **IoCallDriver()**) it is gone. Your driver has *no idea* what the status of the I/O request is after **IoCallDriver()** returns. You do, however, get one more chance to look at the IRP in your completion routine.

So, what you can do instead of the above code, is to modify your completion routine to include the following:

```
if (Irp->PendingReturned) IoMarkIrpPending(Irp);
```

It turns out that if your driver does not have a completion routine, this step is done for you by the I/O Manager. Once you begin providing your own completion routine, however, the I/O Manager gets out of the way and allows you to do anything you want including doing it wrong!

Summary

You might be asking yourself now if completion routines are worth all the trouble. The answer is a resounding "yes". Given the rich I/O model provided by Windows NT for both synchronous and asynchronous I/O, the completion model allows your driver to monitor and handle a variety of events associated with I/O. For example, the ability to handle error cases is essential to building software RAID solutions (such as FTDISK). In this way the driver can transparently recover from I/O errors. In a filter driver, using a completion routine allows the completion status to be "matched up" with the original I/O.

To summarize the discussion in this article, we have compiled a list of "rules" you should follow anytime you will be passing an IRP to another driver.

1. Always set your I/O Completion routine. If you don t want to be notified upon I/O completion, at a minimum you should clear it out, using: **IoSetCompletionRoutine(Irp, NULL, NULL, FALSE, FALSE, FALSE);**

2. If you do provide an I/O Completion routine, you are responsible for propagating the pending status of the IRP back up the call chain. What this means is that if your driver returns **STATUS_PENDING**, you must also mark the IRP as pending.

3. In your completion routine you can be called at IRQL <= **DISPATCH_LEVEL**. Thus, you must either perform only those operations which are acceptable at **DISPATCH_LEVEL** or you must use a worker thread (running at **PASSIVE_LEVEL**.)

4. If you are writing a lowest level driver, always complete the request properly, via **IoCompleteRequest()**. Otherwise, you will cause problems for higher-level drivers.

Return to Previously Published Articles

Return to OSR’s Home Page

consulting& developing/ kits / seminars / NT insider/ resources / client area / about OSR