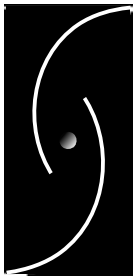
Filter Driver Discussion Paper

Microsoft

July 2001



OSR OPEN SYSTEMS RESOURCES, INC.
105 Route 101A, Suite 19
Amherst, New Hampshire 03031-2277
(603) 595-6500 ♦ FAX: (603) 595-6503

Printed in the United States of America

LIMITED WARRANTY

OSR Open Systems Resources, Inc. (OSR) expressly disclaims any warranty for the information presented herein. This material is presented “as is” without warranty of any kind, either express or implied, including, without limitation, the implied warranties of merchantability or fitness for a particular purpose. The entire risk arising from the use of this material remains with you. OSR’s entire liability and your exclusive remedy shall not exceed the price paid for this material. In no event shall OSR or its suppliers be liable for any damages whatsoever (including, without limitation, damages for loss of business profit, business interruption, loss of business information, or any other pecuniary loss) arising out of the use or inability to use this information, even if OSR has been advised of the possibility of such damages. Because some states/jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

U.S. GOVERNMENT RESTRICTED RIGHTS

This material is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Right in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software--Restricted Rights 48 CFR 52.227-19, as applicable. Manufacturer is OSR Open Systems Resources, Inc. Amherst, New Hampshire 03031.

Table of Contents

1 Introduction

1.1 Executive Overview

This document has been prepared for Microsoft Corporation by OSR Open Systems Resources, Inc. to describe the various issues that arise during the development of file system filter drivers in the Windows XP operating system. In addition, we also discuss these issues with respect to Windows 2000 as appropriate.

1.2 Document Status

This document is a DRAFT document provided to Microsoft for review. Review comments are due back to OSR (via the bugs@osr.com alias) by 15 March 2001.

2 Tracking System Context

Typically, file system filter drivers must track context information on both a per *file* basis as well as a per *file object* basis. Windows XP now provides a mechanism for associating context information on a per file basis using ***FsRtlInitializePerStreamContext,*** ***FsRtlInsertPerStreamContext,*** and ***FsRtlRemovePerStreamContext***. Support for these routines is restricted to file systems that use the **FSRTL_ADVANCED_FCB_HEADER** and indicate their support for filter contexts via the **FSRTL_FLAG2_SUPPORTS_FILTER_CONTEXTS** bit in the ***Flags2*** field of the FCB header. Because this support is tied to the file control block, this context is tracked per-file (not to be confused with "per file object.")

Unfortunately, this technique is not generally useful for file system filter drivers because it is not available in earlier versions (Windows 2000) although it is expected to be available in all standard Windows XP file systems.

2.1 Tracking File State

File state is normally tracked by utilizing two levels of data structures. One data structure is created and is used to track each file object that is being managed by the file system filter driver. The file object only represents a single open instance of a file, however, so frequently a file system filter driver will need to use a *per file* structure as well, using the FsContext value as the key for tracking this per file state.

Note: if your filter needs to track per file state, it will be able to use the context tracking mechanism present in Windows XP. For those filter drivers supporting Windows 2000, it will be necessary to construct your own mechanism for achieving this.

In *Figure 1: Key File System Data Structure Associations* a graphical description of the relationship between the key file systems data structures is portrayed. These relationships provide a fundamental key to the organization of data structures within a typical file system filter driver. Specifically, a filter will normally track both *per file* data structures as well as *per file object* data structures. The association between these two is made within the filter driver by using the address of the file control block for the per-file data structures, and the file object itself for the other.

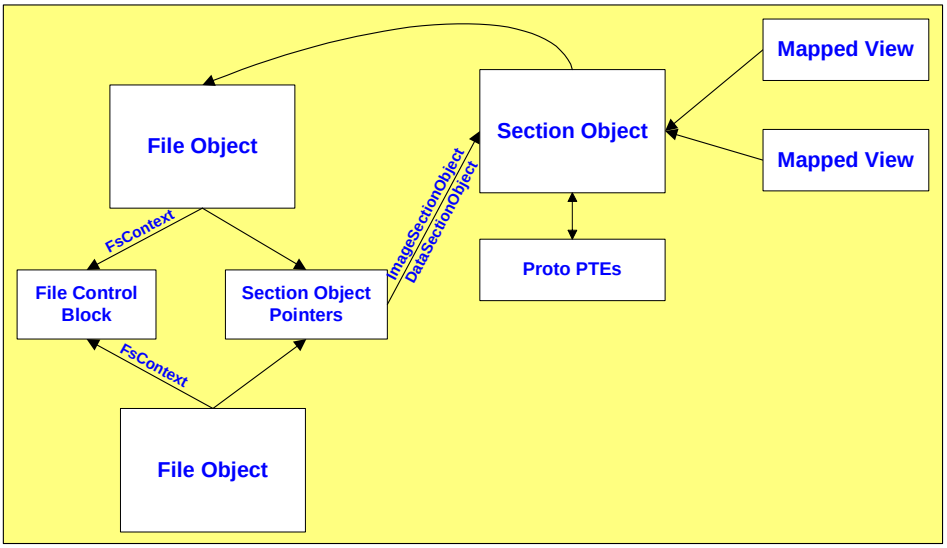


Figure 1: Key File System Data Structure Associations

A key reason this is important is because many operations that are performed through the filter driver may in fact use one specific file object. This is the case for paging I/O, for example, because the Virtual Memory system is maintaining a reference to the file object from the section object. Thus, it is frequently necessary for a filter driver to track information for both.

2.1.1 Tracking Per File Object Context

Normally, tracking per file object context is done utilizing either a driver internal data structure tracking mechanism (such as a hash table) or an operating system provided mechanism (such as a generic table.) In either case, the *key* for lookup is normally the address of the file object data structure.

The lifetime of this data structure is normally tied to the lifetime of the file object itself. Thus, when the **IRP_MJ_CREATE** is observed for the file object, the filter creates a tracking structure for that file object. Similarly, when the **IRP_MJ_CLOSE** is observed for the file object, the filter deletes the tracking structure. Unfortunately, this technique does not work for a certain class of file objects created by the file systems themselves. This issue is discussed in more detail in Section 2.1.3.

Normally a file system filter driver will define a per file object data structure containing information associated with this particular file object. Here is a sample from a typical filter:

```
typedef struct {

    //
    // Standard magic number trick to verify data pointer
    // correctness.
    //
    ULONG MagicNumber;

    //
    // We use the Resource to synchronize data structure access.  Note
    // that this is unnecessary during creation/deletion typically
    // since there are no other references to it.
    //

    ERESOURCE Resource;

    //
    // We build the fully qualified name during create and
    // store it here.
    //
    UNICODE_STRING FileName;

    //
    // We track our per file instance (FCB) based data structure and
    // store IT here.
    //
    PTRACK_FCB_HEADER FcbHeader;

    //
    // This list entry is used for tracking the file objects relative
    // to the FCB.
    //
    LIST_ENTRY FcbListEntry;

    //
    // This points back to the file object for which we constructed
    // this tracking structure.
    //
    PFILE_OBJECT FileObject;

    //
    // This points to our data for the tracking structure.
    //
    PTRACK_DEVICE_INFORMATION DeviceInfo;

} TRACK_FILE_OBJECT_HEADER, *PTRACK_FILE_OBJECT_HEADER;
```

This data structure refers to several other data structures that we frequently find in a file system filter driver: those used for tracking per file data structures, keyed off the file control block, and those used for tracking the device specific information.

These are then, normally maintained in some lookup structure, such as a generic table (using ***RtlInsertElementGenericTable***) or comparable mechanism. This allows the entry to be located while processing individual I/O operations within the dispatch entry point of the file system filter driver.

Note that even if a filter driver can take advantage of the new context tracking mechanism in Windows XP, this mechanism is tied to the per file instance data, and not

the per file data. Thus, it will still be necessary to track this per file object data. Still, a filter using the new context tracking mechanism would be well-advised to track per file object state relative to this file context tracking mechanism.

2.1.2 Tracking Per File Instance Context

Similarly, it is normally useful to track per file data within the file system filter driver. This allows association of information with the file, rather than only with the open instance of the file. As we noted earlier, this is particularly important because some operations are all performed using only a specific file object.

In Windows XP, a file system filter driver working with a file system that supports the FSRTL_ADVANCED_FCB_HEADER, such as all of the standard file systems, can use the new context tracking mechanism (***FsRtlInitializePerStreamContext,*** ***FsRtlInsertPerStreamContext,*** and ***FsRtlRemovePerStreamContext)***. For file system filter drivers supporting Windows 2000, or if it is necessary to support any file system that does not support this new feature, it is necessary to use a table driven lookup mechanism, such as we described in Section 2.1.1.

2.1.3 Handling Stream File Objects

A particularly problematic case for file system filter drivers is that it is often necessary to process I/O operations on behalf of file system created internal file objects (referred to as *stream file objects* because they are created using the functions ***IoCreateStreamFileObject*** and ***IoCreateStreamFileObjectLite***.) These two functions are used by a file system to create an auxiliary file object that is in turn used by the file system for its own internal processing (not necessarily the processing of *streams* in spite of the name.) The problem for a file system filter driver is that a file object may arrive in the file system filter driver without any preceding **IRP_MJ_CREATE** operation. In this case, the file system filter driver must be prepared to dynamically create the necessary state information for its own internal tracking.

The ***IoCreateStreamFileObject*** call causes an **IRP_MJ_CLEANUP** to be sent to the filter driver. The ***IoCreateStreamFileObjectLite*** call does not cause an **IRP_MJ_CLEANUP** and the first operation observed by the filter driver will be one of those operations allowed between **IRP_MJ_CLEANUP** and **IRP_MJ_CLOSE** (notably, virtual memory I/O operations):

- **IRP_MJ_READ** – paging I/O read on this file
- **IRP_MJ_WRITE** – paging I/O write on this file
- **IRP_MJ_QUERY_INFORMATION** – retrieving file size information from the file system
- **IRP_MJ_SET_INFORMATION** – setting file size information prior to a paging I/O write operation.

Thus, the filter driver must be able to handle creation of the new file object state while processing any one of these four operations, as well as **IRP_MJ_CLEANUP**.

3 Complexities of Filtering Redirector

A file system filter driver interacting with the CIFS redirector will experience somewhat different behavior than a file system filter driver concerned with the behavior of the physical media file systems. In this section we will discuss some of the key issues involved in filtering the redirector.

3.1 Tracking File State

A common technique in tracking file state is to utilize either the file context tracking mechanisms in Windows XP (***FsRtlInitializePerStreamContext,*** ***FsRtlInsertPerStreamContext,*** and ***FsRtlRemovePerStreamContext***) or to construct a driver specific lookup mechanism. The Windows XP context tracking mechanism should be available for Redirector (or any other file system utilizing the **RDBSS** mini-redirector model) and thus a filter driver may use that technique. For a filter driver that must support Windows 2000, it must rely upon its own tracking mechanisms for associating context with the specific file.

One important issue to keep in mind is that for a redirector, even if two file objects have different FCB values, they may still represent the same file. This can occur because of the details of how the redirector is implemented and on aliasing issues, because a file can be reached via multiple paths, whether it is via one (or more) exported shares, or via the UNC name mechanism.

3.2 Security and Access

A common problem when filtering redirector is related to accessing the file. For example, a file system filter driver frequently attempts to open and access the file, but these operations will fail if the correct security context is not used. Thus, it is frequently the case that a file system filter driver cannot call ***ZwCreateFile*** or ***IoCreateFile*** in an arbitrary process context (such as a worker thread context) because the system credentials are insufficient to access the file.

There are typically four ways to resolve this problem:

- Restrict the operation to the context of the correct process
- Force the thread into the context of the correct process

- Use impersonation
- Use worker threads within a captive service

Each of these mechanisms has relative costs and benefits – and thus the ultimate choice of *which* one will depend upon the circumstances of the particular driver's implementation.

3.2.1 Restricting Operations to Correct Process Context

Perhaps the simplest mechanism to manage context is to simply perform any context-sensitive operation within the context of the original thread. For example, I/O operations can be performed in process context in their entirety. This is often a good choice for an unusual event (e.g., reading a configuration file once during initialization) or for an operation where performance is not critical.

Unfortunately, this technique often has some overhead associated with it that makes it unsuitable for performance-sensitive processing. For example, it is possible to perform file I/O by opening the file (***ZwCreateFile***) performing the I/O (***ZwReadFile***, ***ZwWriteFile***, etc.) and then closing the file (***ZwClose***). This technique includes the overhead associated with creating a file object, initializing it (which requires an **IRP** be sent to the FSD,) performing the I/O operation in question, and then tearing down the file object. This entails three additional **IRP**s be sent to the file system (one for the ***ZwCreateFile*** call, two for the ***ZwClose*** call) in addition to the **IRP**s used to perform the underlying I/O operation.

3.2.2 Switching Process Context

Another mechanism to manage context is to switch to the correct thread context as needed. This can be done by using the ***KeStackAttachProcess*** operation, which attaches the current thread to a different process. Once in the correct process context, the driver may perform normal operations, such as I/O. Handles can of course be used, provided they were created within the same process context. Security credentials for the attached process are used, which is particularly important when working with the CIFS redirector.

Once the operation has been completed, the filter can detach using ***KeStackDetachProcess***. The thread is then returned to the previous process context.

3.2.3 Impersonation

With respect to security issues, one powerful technique to accomplish this is impersonation. In this technique, the credentials of the original caller are used, even though the operation is being performed in a separate thread context. The steps for doing this are:

- Save the impersonation token (if any) of the current thread – this allows you to restore it when you have finished impersonating. This is typically accomplished by using *PsReferencePrimaryToken*. Note that since this creates a reference to the token, it must subsequently be dereferenced.
- Call ***PsImpersonateClient*** using the security token that should be used for the impersonation.
- Perform the necessary operation
- Call ***PsImpersonateClient*** using the original impersonation token of the thread; this will restore the previous security context of the operation.

Normally, it is open/create operations that require the correct credentials. This can include opening files, named pipes, registry keys, or any other system resource which is protected from general access.

3.2.4 Worker Threads within a Captive Service

One specific mechanism for ensuring correct context is related to using a set of worker threads – either system managed or driver created and managed – and having those worker threads perform the details of the operation.

For example, handle-related operations can be enqueued to the worker thread (or threads) so that the worker may process it. Since the worker threads are all created within the same process address space, they share the security credentials of that process. Thus, if the service utilizes the ability to "log on" – and hence has its own security credentials distinct from the system credentials (this can be controlled from the ***CreateService*** API to the Service Control Manager.)

Thus, the steps to create such a set of worker threads:

- Install the service so that it logs on using a service account (ideally, this account would have credentials within the security domain.)

- When the service starts, one or more threads call the filter, specifying a custom IOCTL code. Inside the filter, this IOCTL causes the thread to call the filter's service loop.

The "service loop" is typically a simple loop that looks for new work items to process. If it finds an entry in the queue, it removes it. If the queue is empty, the thread blocks on a dispatcher object; the dispatcher object is signaled when an element is inserted into the work queue.

Using this mechanism, the filter can then insert a "work item" into the work queue. The thread initiating the I/O operation calls into the work queue. It can then either wait for the I/O operation to complete, or it can return control back to the caller.

4 Handling MDL Read/Write operations

Typical file system filter drivers are often tested initially using Win32 applications, such as explorer. Because of this there are large categories of usage that remain untested within the file system filter driver. One such category is the area of MDL-based file I/O operations. These operations are used extensively by the CIFS file server (which is implemented largely by the kernel mode **srv.sys** component.)

Specifically, in order to allow more rapid transmission of files for the network stack, the file systems implement a direct-to-cache mechanism that relies upon the **IRP_MN_MDL** option for **IRP_MJ_READ** and **IRP_MJ_WRITE**. In this technique, the caller is returned an MDL that normally describes the data within the file system data cache. The caller then utilizes the data buffer directly, rather than using its own buffer. The primary advantage of this technique is that it eliminates an additional data copy.

Thus, this technique involves not one, but two distinct operations: the first operation retrieves an **MDL** that describes the region in the cache where the data is stored. When the caller has completed its own operations, a second call (**IRP_MN_COMPLETE_MDL**) is made to indicate that the caller has completed the I/O operation on the cache buffer.

For a *read* operation, the data is contained within the buffer upon completion of the initial **IRP_MJ_READ** call. Further, it has been "locked down" so that accessing the data buffer will not generate any additional page faults (although it may have been necessary to fault in the data when the **IRP_MJ_READ** operation was performed initially.) For a *write* operation, the data is not contained within the buffer until after the caller has placed it there. Thus, a file system filter driver cannot rely upon the data in the buffer until the **IRP_MN_COMPLETE_MDL**.

A common source of **IRP_MJ_READ/IRP_MN_MDL** operations is the ***TransmitFile*** API (which is actually exposed as part of the Windows Socket API for user mode applications.) This API is used by Internet Information Service (IIS) Version 4.0 and more recent. Normally, this is not difficult for a file system filter driver to handle because it is sufficient to ignore the subsequent

IRP_MJ_READ/IRP_MN_COMPLETE_MDL request, because it only represents a release of the buffer.

If your file system filter driver must handle **IRP_MJ_WRITE/IRP_MN_MDL** (which is *not* used by the TransmitFile API) it is important to keep in mind that the data within the cache is "pinned" (it cannot be written back to disk and the virtual to physical memory translation is locked down) but it is not *present* until after the caller releases the MDL by making the **IRP_MJ_WRITE/IRP_MN_COMPLETE_MDL**. Only on this second operation can the caller actually rely upon the data contents of the MDL.

In addition to the IRP path, the MDL path is also available via the Fast I/O mechanism. The function of these Fast I/O routines is comparable to the function of the IRP mechanism described earlier in this section, except that the I/O operations are done via the fast I/O vector table.

Finally, there are special "compressed" versions of each of these calls and operations. These calls are not used at present, but they may be used in future releases and versions of Windows.

5 Completion Processing

File system filter drivers (in particular) manipulate IRPs both prior to delivery to the underlying file system as well as during completion processing. Since completion routines can be called at IRQL<= DISPATCH_LEVEL, it is sometimes necessary for them to post the I/O operations to worker threads in order to perform additional processing on the IRP.

Traditionally, the technique for most layered drivers is to implement a synchronous call-and-wait model, where an event is used to "hand off" control of the IRP between the completion routine and the mainline (dispatch) entry point (this can be done easily using ***IoForwardAndCatchIrp***.) Unfortunately, this technique is not sufficient for file system filter drivers because they must handle operations that *cannot* be properly implemented synchronously.

However, once the filter begins to mix synchronous and asynchronous I/O models together there are a number of problems that can easily occur. While it is possible to write a correctly functioning file system filter driver without understanding I/O completion processing, it is helpful to understand it to ensure that the file system filter driver is handling it correctly. Further, the interactions between synchronous and asynchronous I/O in the filter driver give rise to a number of subtle issues. Unfortunately, failing to master this topic can compromise the proper operation of your filter driver.

5.1 The I/O Stack

Figure 2: IRP Stack Locations shows an I/O Stack with three stack locations, as well as the contents of the I/O completion routine within each stack location.

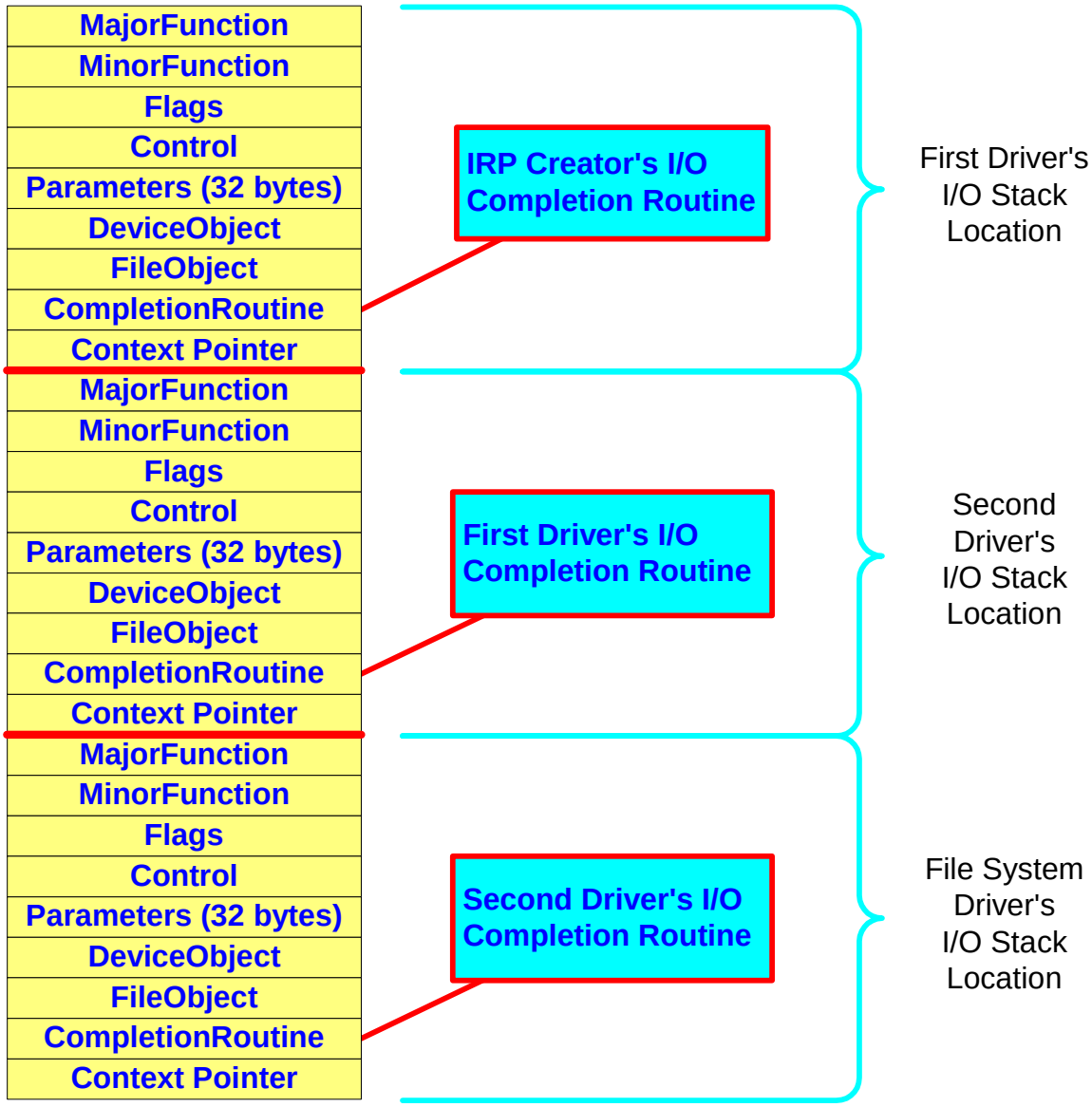


Figure 2: IRP Stack Locations

The stack locations are used from the top to the bottom (i.e., the I/O stack grows "down" in memory, although they are displayed in the opposite order in the kernel debuggers when using the **!irp** command). You can see this by examining the implementation of the I/O Manager function IoSetNextIrpStackLocation (a macro inside of ntifs.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 – be careful of this because it is a frequent error for drivers constructing their own **IRP**s.

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.

5.2 Setting Up The I/O Stack For Completion

When setting up an IRP, one thing the driver may do is establish a completion routine. This allows the driver to process the I/O operation after the driver below has completed it, and before the completion routine of the driver above has been called. The driver may specify that its completion routine should be called in the case where the I/O request is completing with success or 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 ntifs.h:

```
#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; } }
```

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.

It is important to note that the only values that should be returned from your filter driver's completion routine is **STATUS_SUCCESS** or **STATUS_MORE_PROCESSING_REQUIRED**. The actual completion status of the IRP is in the **IoStatus** field. A filter driver should not return the contents of this field as their return value. Indeed, a filter driver should not return any value besides

STATUS_MORE_PROCESSING_REQUIRED or **STATUS_SUCCESS**. Otherwise, the behavior of the I/O Manager is undefined.

The important point to notice about I/O completion routines, looking at Figure 2, 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. A 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-1st driver's completion routine is stored in the Nth 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.

Because of this, it is a mistake for your driver to copy all fields of the current I/O stack location to the next driver's I/O stack location (even though there are still many examples that use this technique!)

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. Here is an example of this classic approach:

```
PIO_STACK_LOCATION IrpSp;
PIO_STACK_LOCATION NextIrpSp;

IrpSp = IoGetCurrentIrpStackLocation(Irp);
NextIrpSp = IoGetNextIrpStackLocation(Irp);

*NextIrpSp = *IrpSp;

return IoCallDriver(NextDeviceObject, Irp);
```

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 completion routine 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!

Note that Windows provides the macro ***IoCopyCurrentIrpStackLocationToNext***. This macro should be used, rather than copying the I/O stack location manually. Using this function avoids copying the completion routine field.

A recurring problem we have seen in file system drivers is where the underlying file system does not properly call ***IoCompleteRequest()*** for the I/O operation. For synchronous I/O operations this works correctly – until a file system filter driver is attached on top of the file system. In that case, the symptom of the problem is that the filter driver's completion routine is never called. A filter driver can detect – and work around – this problem when it does arise. We note that this work-around relies upon an understanding of how I/O completion processing works. Any file system that exhibits this behavior is in error. Fortunately, the file systems provided as part of current versions of Windows do not exhibit this particular problem.

The following code fragment demonstrates a mechanism for detecting and working around this problem.

```
IoCopyCurrentIrpStackLocationToNext(Irp);

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;
```

This is then used with a completion routine that sets the event:

```
OurCloseCompletion(PDEVICE_OBJECT Device Object, PIRP Irp, PVOID Context)
{
Event = (PKEVENT)Context;
KeSetEvent(Event, 0, FALSE);
return STATUS_MORE_PROCESSING_REQUIRED;
}
```

As it turns out, for most I/O operations, if a driver returns anything except **STATUS_PENDING** in its dispatch routine, the I/O manager will "complete" the IRP being dispatched. 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.

5.3 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 (that processing resumes when that driver calls **IoCompleteRequest** with the **IRP** again.)

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 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 3 we demonstrate an example of the flow of control in this case (the "synchronous I/O case"). This is a very common case!

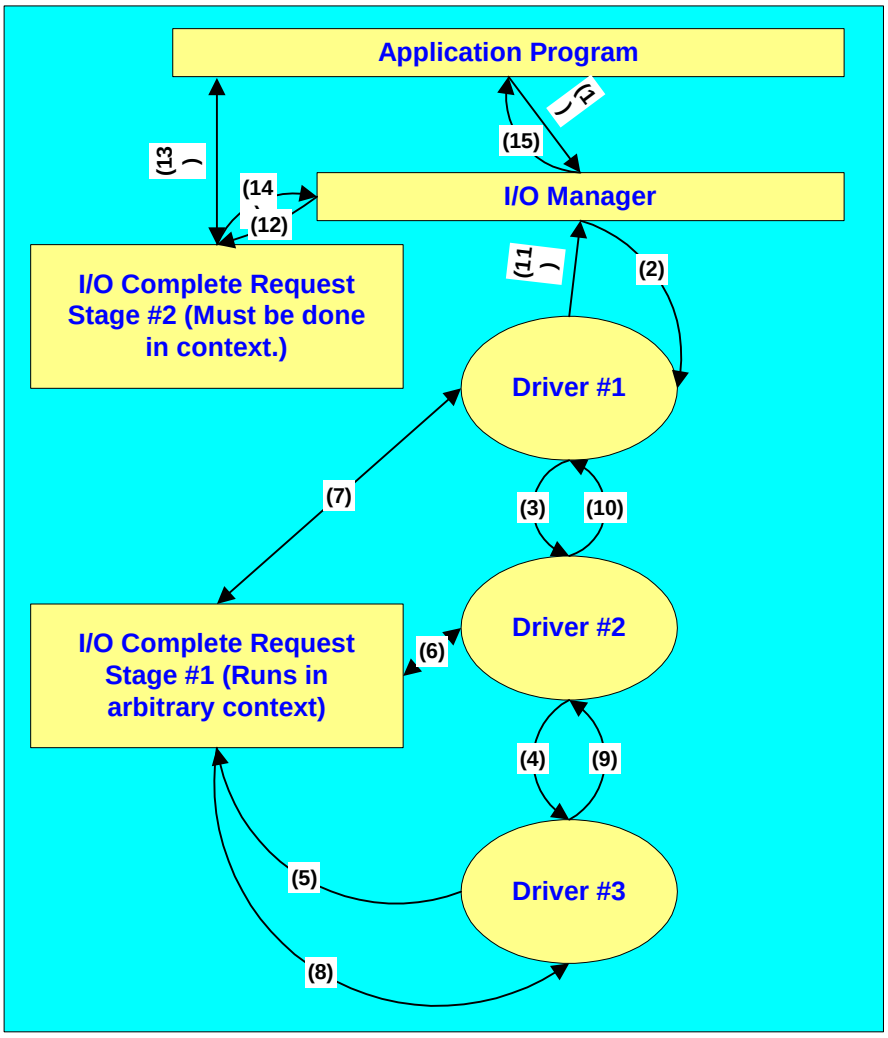


Figure 3: Prototypical Synchronous I/O with Completion

Thus, the steps in this example are:

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 problem of a driver not calling ***IoCompleteRequest()*** described earlier, where it 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!

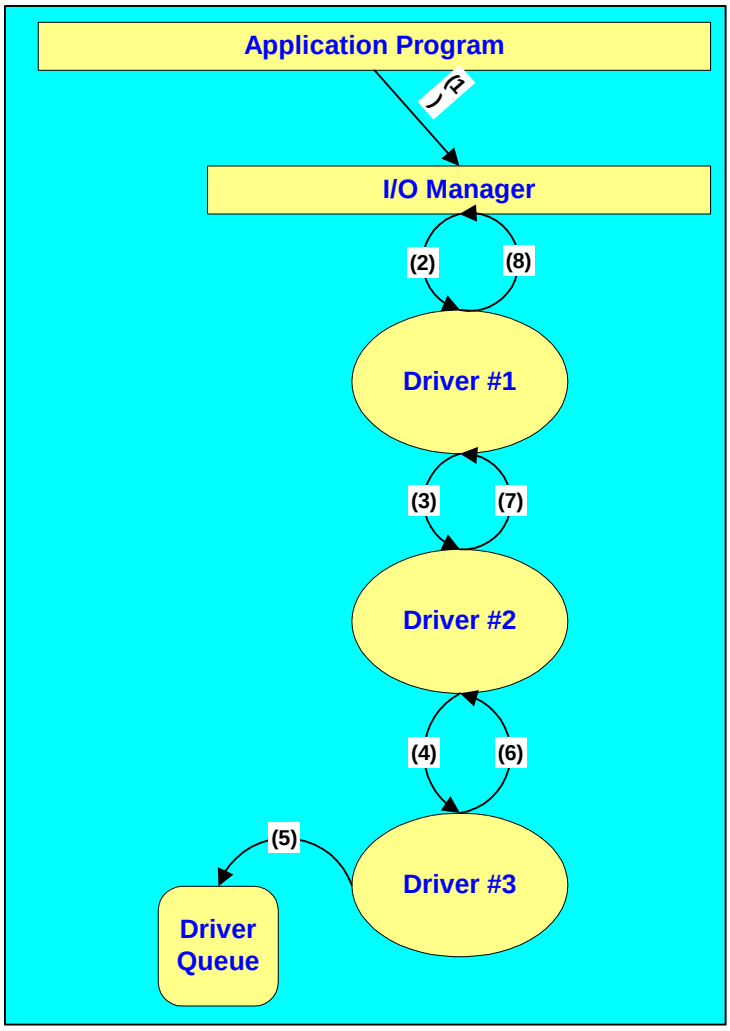


Figure 4: Initiation of an Asynchronous I/O Operation

The asynchronous case is much more complicated. To describe this, we use three separate diagrams. The first (Figure 4) represents initiation of the I/O operation. The second figure (Figure 5) represents processing of the I/O operation by an arbitrary thread of control, and shows it up to the point of I/O completion. The third figure (Figure 6) represents the final I/O completion processing that occurs in the original thread context.

We note that these operations occur in a very different order than the operations did for the synchronous example earlier.

So, as we did before, we walk through the steps and describe what is happening (and again we note that this is just one possible scenario of many.) In Figure 4 we have the following 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 marks the IRP as pending and enqueues the I/O request.
6. The third driver returns **STATUS_PENDING** to the second driver, indicating that the I/O operation has been enqueued and will be processed separately.
7. The second driver returns **STATUS_PENDING** to the first driver.
8. The first driver returns **STATUS_PENDING** to the I/O Manager.
9. The I/O operation is in progress so the I/O Manager blocks the thread to await its completion.

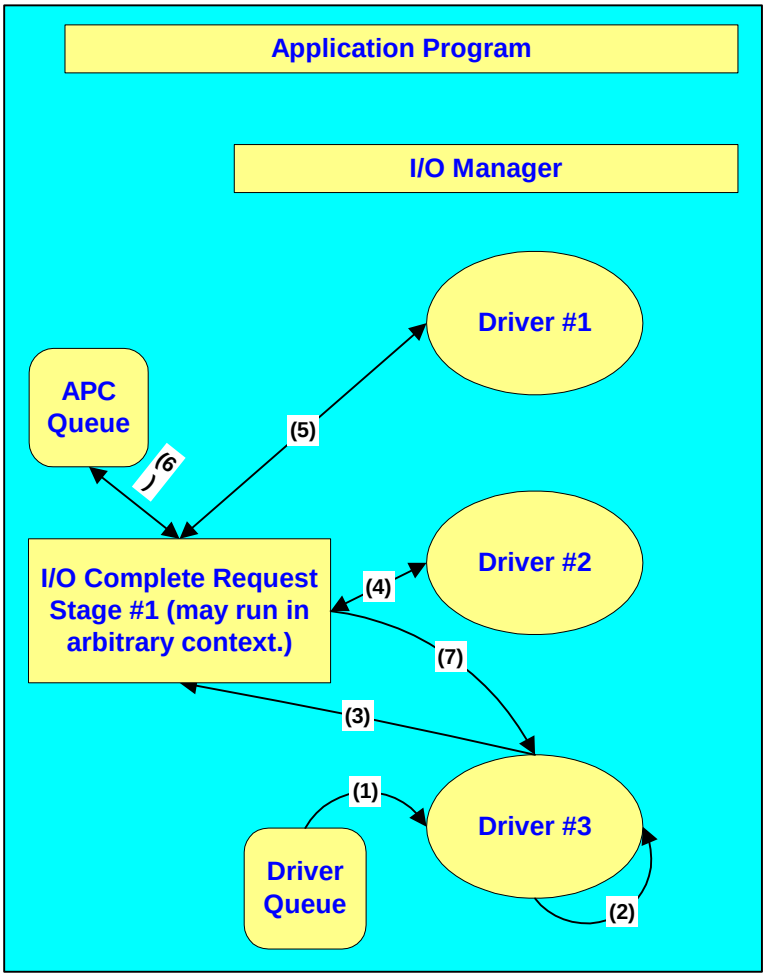


Figure 5: Asynchronous I/O Completion - Return to Thread Context

Independently, the I/O request is removed from the queue. This is pictured in Figure 5 and demonstrates the following steps:

1. The independent thread removes the I/O operation from the queue;
2. The driver processes the request internally;
3. The third driver completes the I/O operation. It calls **IoCompleteRequest()** and initiates first stage completion.
4. During first stage completion the second driver's completion routine (if any) gets called back.
5. During first stage completion the first driver's completion routine (if any) gets called back.

6. The I/O Manager identifies that the first driver returned **STATUS_PENDING** and thus enqueues an APC.
7. Finally, **IoCompleteRequest()** returns control back to the third driver. First stage I/O completion is done.

The I/O Manager has thus performed all of the "context independent" processing for I/O completion. Thus, **IoCompleteRequest** returns to the caller when:

- A completion routine has returned **STATUS_MORE_PROCESSING_REQUIRED**; or
- All completion routines have been processed and the "context independent" portion of I/O completion processing has been completed by the I/O Manager.

If a driver suspends processing by returning **STATUS_MORE_PROCESSING_REQUIRED**, it must resume that processing by calling **IoCompleteRequest** against the IRP. When all the completion routines have been called and the I/O manager has finished "context independent" processing, it will enqueue an APC. The purpose of the APC is to allow the final steps of I/O completion to occur in the context of the original call thread.

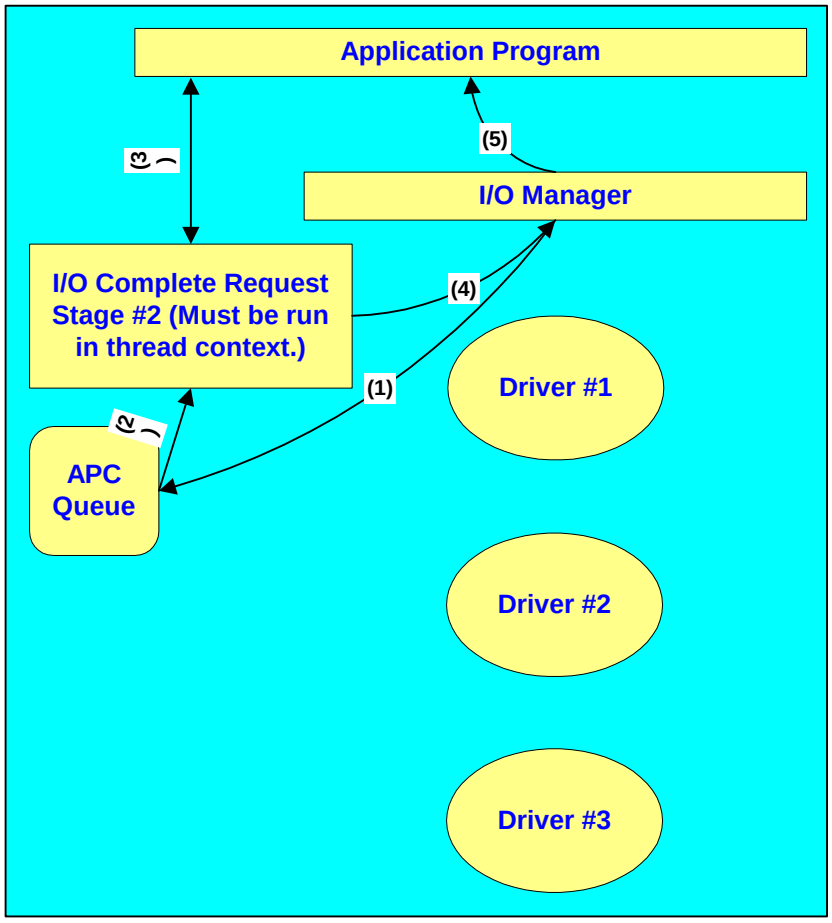


Figure 6: I/O Operation Processing through Completion

In Figure 6 we show the remaining steps the I/O Manager performs as part of processing this I/O operation. These steps are:

1) The thread that initiated the I/O, blocked waiting for the I/O operation to complete, awakens because an entry has been placed in its APC queue;
2) The APC routine is called (it is specified in the APC object);
3) The APC routine copies any data and the I/O status information into the address space of the thread. It frees the IRP and any associated resources and then it terminates, as well as setting the event specified in the IRP – which awakens any threads waiting for this I/O.
4) Because the APC ran, the OS must verify that the wait condition has been satisfied for the thread. If it has not been satisfied, the thread will block again.

However, because the event was set in step (3) the wait is satisfied and the thread returns from its wait.

5) Since the wait is satisfied, the thread returns back to user mode.

Of course, this example focused on synchronous I/O – the flow for asynchronous I/O would not have blocked the thread! That is how Win32 applications can utilize "overlapped I/O" for example.

5.4 Setting Up Your Completion Routine

Because the use of completion routines within certain types of drivers – especially file system filter drivers - is fairly common, understanding the model used within Windows 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:

```
DbgPrint("Createfor%Z\n",&IrpSp>FileObject>FileName;
return IoCallDriver(NextDeviceObject, Irp);
```

And in their completion routine do something like:

```
// Queue a work item for additional processing
IoQueueWorkItem(&WorkItem, WorkRoutine, DelayedWorkQueue, &WorkItem);
return STATUS_MORE_PROCESSING_REQUIRED;
```

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. 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 (e.g. **STATUS_SUCCESS**) 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.

This seems like it should fail, but in fact most of the time it works. This is because 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.

What this means is that 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:

- Either return STATUS_PENDING; or
- 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 this:

```
DbgPrint("Createfor%Z\n",&IrpSp->FileObject->FileName);
IoMarkIrpPending(Irp);
(void) IoCallDriver(NextDeviceObject, Irp);
return STATUS_PENDING;
```

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 enqueue 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 then the mainline code would look something like:

```
DbgPrint("Create for %Z\n", &IrpSp->FileObject->FileName);
Status = IoCallDriver(NextDeviceObject, Irp);
KeWaitForSingleObject(&Event, Executive, KernelMode, FALSE, 0);
```

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. Of course, the filter driver is still responsible for completing the request (using **IoCompleteRequest**) once processing is completed.

While making I/O synchronous 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 is 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, always making I/O asynchronous 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 is 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 (although, if you use driver verifier this particular bug will show up much more quickly.) 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!

5.5 *Beware IRQL APC_LEVEL*

As we discussed earlier in this section, much of I/O completion processing is tied to the use of asynchronous procedure calls to clean up from I/O operations. One important point here to keep in mind is that if a thread is at IRQL **APC_LEVEL** and it waits for an I/O completion, the thread will hang indefinitely because the APC cannot be delivered. Thus, the thread is waiting on the event (which indicates completion of the I/O) and the APC (which will set the event) cannot run because the thread is at **APC_LEVEL**. Thus, an important rule is that if your driver is going to call any of the **Zw*** API functions, it must be done at **PASSIVE_LEVEL** and not at **APC_LEVEL**.

We sometimes see drivers run afoul of this rule by using *fast mutexes*. While we applaud the desire of driver writers to use efficient locking mechanisms (and, since they are called "fast mutexes" we know they must be efficient!) a side-effect of fast mutexes is that they raise the IRQL to **APC_LEVEL**.

5.6 Rules 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. 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 copy I/O Stack locations using ***IoCopyCurrentIrpStackLocationToNext***. This avoids the potential problems of inadvertently copying a completion routine. Unfortunately, there is a lot of older sample code that still copies the stack location “manually”.
2. If you need to process the IRP after it has been handed off to a lower driver, you must set your I/O Completion routine.
3. If you do not need to process the IRP after it has been handed off to a lower driver, you should not an I/O Completion. You may also use **IoSkipCurrentIrpStackLocation** (but beware! You should **never** use both **IoSkipCurrentIrpStackLocation** and **IoMarkIrpPending** on the same IRP.)
4. 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.
5. In your completion routine you can be called at IRQL <= **DISPATCH_LEVEL**. Thus, you must either perform only those operations that are acceptable at **DISPATCH_LEVEL** or you must use a worker thread (running at **PASSIVE_LEVEL**.)
6. If you are writing a lowest level driver, always complete the request properly, via **IoCompleteRequest()**. Otherwise, you will cause problems for higher-level drivers.
7. Use driver verifier! It will test many of these assumptions to ensure correct operation of your driver. While this won’t guarantee there are no bugs, it will certainly ensure there are fewer bugs.

6 Re-entrancy Issues

Few issues seem to plague the development of file system filter drivers more than issues with respect to reentrancy of the various I/O operations from the filter back through the storage stack. Typically, this happens because the filter driver attempts to utilize the ***ZwXxx*** functions to access the underlying file system driver. While simple, the **Zw** functions always re-enter the storage stack with the top file system filter driver.

6.1 Create Reentrancy

Perhaps the most common form of reentrancy is when a file system filter driver attempts to open the file as part of another operation (frequently a create operation.) Windows XP provides the ***IoCreateFileSpecifyDeviceObjectHint*** function that simplifies one major aspect of this operation: this call "circumvents" reentrancy into the calling stack For Windows 2000 drivers, this call is not available, and they may be better served by utilizing the alternate device path technique described in Section 6.2.2.

6.1.1 Oplocks

Another common issue with respect to reentrancy in the storage stack that is not resolved by the ***IoCreateFileSpecifyDeviceObjectHint*** is the handling of files for which an oplock break is required. An *oplock* is a mechanism used by the CIFS file system protocol to allow caching of file system data on remote clients. The risk in allowing such caching is that any time there are multiple copies of the data one or more of those copies might be "out of date" with respect to the underlying original file. The simplest solution then is to prohibit multiple copies. However, prohibiting caching seriously degrades performance. Indeed, caching data is a standard operating system technique for improving performance. This approach works well because data that has been recently accessed is more likely to be accessed again. For a system where there is only ever one program accessing data, caching works ideally. However, once a second program (on a different computer system) attempts to access the same data the issue of cache consistency becomes critical. This is because there are now potentially many copies of the data – each one in a different cache on a different computer. Thus, there must be some mechanism to keep these copies in sync with one another. If not different users of the data will have copies that differ from one another.

When accessing files across a network, the simplest scheme is to always store all data back on the file server. Thus, whenever an application program reads data that read request is satisfied from the file server – across the network. This ensures that there is a consistent view of data since there is only a single copy of the data in existence – on the file server.

Unfortunately, the performance characteristics of such systems are not ideal. While file servers can be built to transfer data quickly there are numerous bottlenecks between the client accessing the data and the file server storing the data, not to mention the added latency necessary to fetch the data from the file server each time.

Studying this problem at length reveals that the majority of data being retrieved from the file server is never modified – it is only read. Normally a single program is responsible for modifying the data of a single file. For example, a word processor would modify the data within a specific document. Data that is being modified by multiple programs represent a small amount of the total data traffic. In spite of this, users do expect their file systems to ensure that any data they access is correct – not most of the time, but all of the time.

The typical access characteristics for such data make network file system clients ideal candidates to cache data – and many of them do so, using a variety of different techniques to ensure cache consistency. For example, the venerable NFS file system protocol utilizes a scheme of checking file system time stamps on the remote file server to detect when data in its cache may become stale. This solution is not a perfect one since there is a window in which the NFS client may cache old data; however, it has worked acceptably well for many years.

In Windows XP (and earlier versions of Windows NT) the CIFS redirector (the “client”) and the CIFS file server (the “server”) implement a file caching protocol between them. In order to ensure correctness of the cached data, LanManager implements a basic cache consistency scheme that covers the entire file contents. Where files are being simultaneously accessed across the network by multiple users for both read and write access, caching is disabled – clients must fetch data from the file server each time it is read, and must store it back immediately each time it is written. However, because such shared-write access is unusual, in the vast majority of cases, the client will cache data

locally. This minimizes network traffic and vastly improves performance for most file access on Windows NT.

CIFS implements this cache consistency scheme by using a protocol known as opportunistic locking. An opportunistic lock is known as an "oplock" in the parlance of Windows file systems. Further, the implementation of oplocks by Microsoft impacts both their network and local file systems. Because the details of the local implementation are tightly coupled to how oplocks are used by network file systems, we describe the network implementation initially and then return to discussing issues associated with their local implementation for NT file systems.

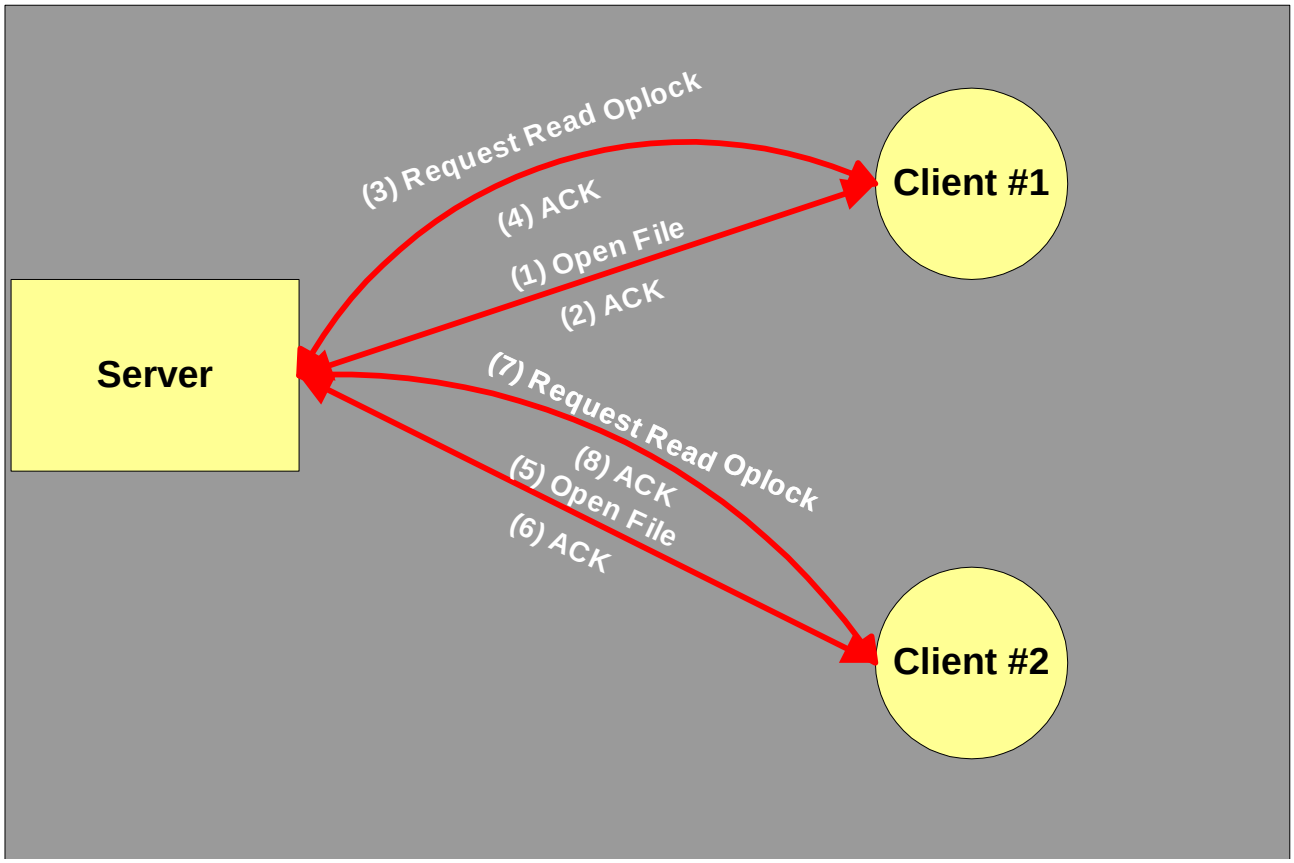


Figure 7 Level 2 Oplocks

Figure 7 shows a simplified transaction between the server, which is responsible for managing the oplocks, and two separate clients, which are using the oplocks to ensure the data they are caching remains consistent with the original copy. These steps are:

1. Client # 1 opens the file, which is stored on the server;
2. The server opens the file and indicates the success of the operation back to the client;
3. Client # 1 requests a Level 2 (read) oplock from the server;
4. The server grants the oplock request to the client;
5. Client # 2 opens the file.
6. This is consistent with the oplock that has already been granted and so the server grants the open request;
7. Client # 2 requests a Level 2 (read) oplock from the server;
8. The server grants the oplock request to the client.

This process allows many clients to cache the data. If a client modifies the data the server must then advise all systems to which it has granted a Level 2 oplock that their oplock is broken.

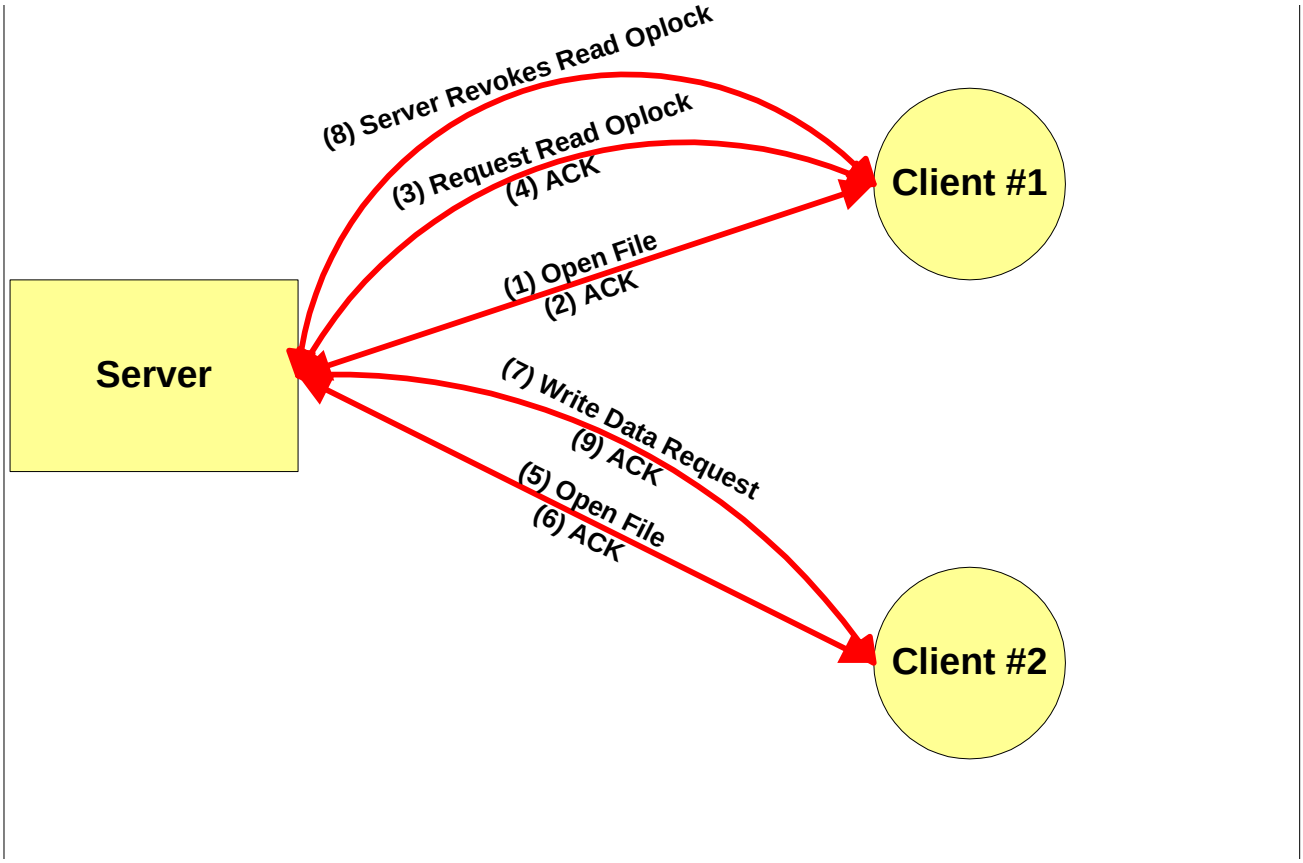


Figure 8: Breaking Level 2 Oplock

In Figure 8 we demonstrate the calling sequence for the oplock break. The steps from this are:

1. Client # 1 opens the file, which is stored on the server;
2. The server opens the file and indicates the success of the operation back to the client;
3. Client # 1 requests a Level 2 (read) oplock from the server;
4. The server grants the oplock request to the client;
5. Client # 2 opens the file.
6. This is consistent with the oplock that has already been granted and so the server grants the open request;
7. Client # 2 modifies the file;

8. The server, receiving the modification request, notes that there is an outstanding oplock on the file and sends a message to each client holding a Level 2 (read) oplock revoking the oplock;
9. Having notified all of the Level 2 oplock holders that their oplocks have been revoked, the server processes Client # 2's write request and acknowledges it.

This mechanism works quite well if the data is never modified and allows clients to "fall back" to the always-perform-I/O-to-the-server model if the data is modified. Thus, this provides a convenient mechanism to balance between correctness and performance.

In addition to the Level 2 oplock, there are three other types of oplocks. Thus, the function of the four oplocks are:

- Level 1 – these oplocks allow a client to make, and cache, local modifications. This includes both the data contents and attributes of the file. Such oplocks can only be granted/held so long as there is only a single open handle against the file. Any subsequent open on the file (except for file attribute access ONLY) will require breaking this oplock. The level 1 oplock check in this case is done after the file sharing check is done;
- Level 2 – these oplocks allow a client to cache file data, although modifications are to be written back to the server. Such oplocks can be granted/held by many clients so long as no client modifies the data;
- Batch – these oplocks are held across individual open operations on the file. They are named because they were added in order to improve the performance of "batch files", which were an important performance benchmark. Batch oplocks function similarly to Level 1 oplocks, although they can be held (by the redirector) even when the file is not opened by an application running on the client. The batch oplock check is done prior to the file sharing check (unlike the level 1 oplock case);
- Filter – these oplocks are held on a file to detect changes to its attributes. Normally, a filter oplock starts when the file is opened and protects the data and attributes of the file. Many clients may hold a filter oplock.

Each of these oplocks is used to protect the correctness of the data stored within the file.

Some oplock breaks can be more complicated than the earlier example. For instance, a Level 1 (write) oplock break can require that the client write data back to the server before acknowledging the oplock break. Indeed, unlike a Level 2 oplock, where the client does not need to "acknowledge" the oplock break, a Level 1 oplock requires that the client acknowledge the oplock break – and before it can do so, it must ensure that all of the modified file information (attributes and data) have been flushed back to the server. In Figure 9 we show a simplified model for the Level 1 oplock break.

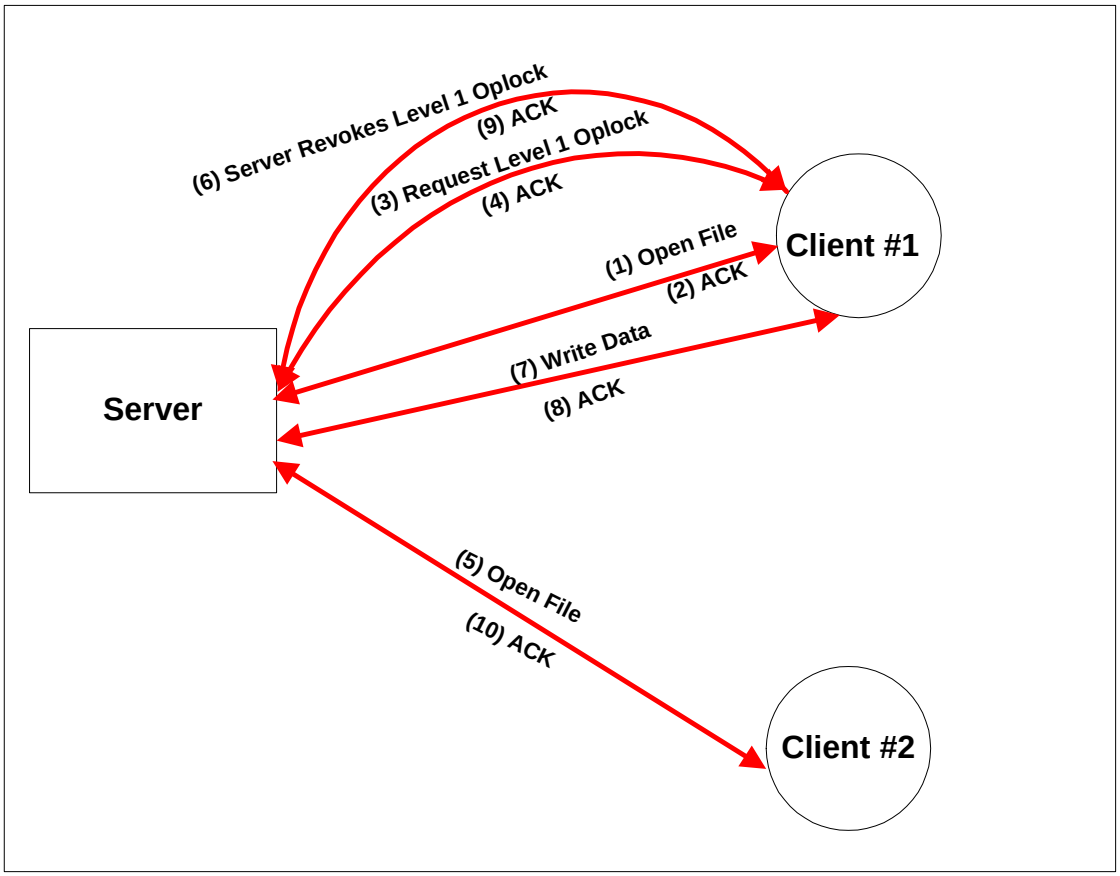


Figure 9 Level 1 (write) oplock break

The steps in this example are:

1. Client # 1 opens the file;
2. The server grant's client #1's request to open the file;
3. Client #1 requests a Level 1 oplock on the file;
4. The server grants the request;

5. Client # 2 opens the file;
6. The server initiates an oplock break from client # 1. This is necessary since client # 1 may have modified the data and attributes of this file. Thus, the server cannot even process the open request while the oplock is outstanding;
7. Client # 1 begins writing data (and attributes) back to the server;
8. The server acknowledges the data – note that this process (write by client/ack by server) may iterate many times;
9. Client # 1 acknowledges the Level 1 oplock break;
10. The server acknowledges Client # 2's request to open the file.

The time required to process a Level 1 oplock break can be substantial as it may entail many write operations from the client back to the server.

A batch oplock protects the cache coherency of the file, even when it is closed by the client. For example, batch files are processed one line at a time and between processing each line of the batch file the file is closed and then re-opened. If the contents of the file do not change from one open instance to the next, the batch oplock allows the client to use the cached contents of the file – even though it was closed between the two lines. However, if the file changes, the batch oplock must be broken. An odd complication here is that batch oplocks also protect the path to the file – and thus a rename of the protected file or of any ancestral directory causes the oplock to be broken. This can be quite complicated, as can be seen in the rename code within the FastFat file system code (which is part of the IFS Kit.)

The oplock protocol itself is sufficient to ensure cache consistency between clients anywhere on the network. There is one case, however, that is not covered by this mechanism – the case of local application access to the local file system access. In this case, the application will call directly into the FSD without using either the server (SRV) or client (RDR) components. Thus, to "resolve" this potential inconsistency, the Windows developers decided that oplocks needed to be handled by the FSD directly because it was the point of commonality between the network file server (CIFS) and the local applications. Thus, an inherently network activity (remote caching of data) has an

important impact on file systems – and it directly impacts file system filter drivers because of their role in processing oplock requests.

The subsequent subsections detail the oplock protocol.

6.1.1.1 FSCTL_REQUEST_OPLOCK_LEVEL_1

A level 1 oplock is an exclusive oplock on the file. That is, it gives the holder of the lock the right to cache the data and to modify the data in its cache. Essentially, no other computer anywhere in the network may be accessing the file.

An FSD will grant such an oplock when the file is only opened once. Thus, if two or more clients already have the file open when a request for a level 1 lock is made, the request will be denied. Note that "clients" in this context could be both local applications, as well as remote (across-the-network) users of the file.

Similarly, if the remote client already holds a level 1 lock and a second client opens the file, the level 1 lock previously granted must be revoked. This will trigger a write-back of any dirty data stored by the first client before the oplock break is completed.

An interesting requirement of the way that the oplock protocol is implemented on Windows is that the interface must be implemented asynchronously. The oplock is granted when STATUS_PENDING is returned to the IRP containing the oplock request. Thus, an FSD must complete the processing of the original IRP synchronously because returning STATUS_PENDING would indicate the oplock grant was successful to the caller.

Once an oplock has been granted, the IRP representing that oplock is queued and held. The oplock break processing is implemented by completing the original IRP that requested the oplock. The IRP must be completed by setting the Information field of the IoStatus field to either FILE_OPLOCK_BROKEN_TO_LEVEL_2 or FILE_OPLOCK_BROKEN_TO_NONE.

However, the oplock break at this stage has not completed. Instead, the owner of the oplock must do any internal processing required. Once that processing has completed, the oplock owner must then acknowledge the oplock break. If FILE_OPLOCK_BROKEN_TO_LEVEL_2 is returned, the owner of the oplock may either indicate FSCTL_OPLOCK_BREAK_ACKNOWLEDGE, in which case the

acknowledgment IRP is treated as a request for a level 2 oplock (c.f., Section 6.1.1.2.) Alternatively, the oplock owner may acknowledge the IRP but decline the offer of a level 2 oplock (c.f., Section 6.1.1.2) by indicating **FSCTL_OPLOCK_BREAK_ACK_NO_2**.

The principal reason a level 1 oplock is broken is because another caller opens the file. Normally a caller who wishes to open the file will block until the oplock break is completed. However, SRV (the LanManager file server) requires, for internal deadlock prevention reasons, that a create operation be completed before the oplock break is completed. The caller indicates this special requirement to the FSD by setting (in the create request) the **FILE_COMPLETE_IF_OPLOCKED** bit in the option flags. If an oplock break is required, a special status code is returned to the caller. This tells the caller that before it can use the file object thus created, it must verify that the oplock break has completed – once it is "safe" to do so. It does this by making a subsequent call to the FSD to wait until the oplock break on the given file is completed (c.f., Section 6.1.1.6).

6.1.1.2 FSCTL_REQUEST_OPLOCK_LEVEL_2

A level 2 oplock is a shared oplock on the contents of the file. It allows a network client to cache the data in memory. If the data changes on the server, the server will send the client holding the level 2 oplock a message indicating that its cached data is "stale" and must be refreshed from the server.

As with a level 1 oplock, the oplock is requested via an IRP and the oplock is granted when the FSD returns STATUS_PENDING. Unlike a level 1 oplock, however, a level 2 oplock may be granted when a file has previously been opened. Further, a level 2 oplock may be granted even when other opens of the file allow write access. This point is really very important because many applications will open a file for write access, even if they never intend on modifying the contents of the file.

Thus, an FSD must check when a write operation is being performed to ensure that no level 2 oplocks have been granted against the file. If there are level 2 oplocks outstanding on the file they need to be revoked, which invalidates the client's cached copy of the data.

The FSD breaks the oplock by completing the pending IRP. In the case of a level 2 oplock nothing is set in the Information field - the IRP is simply completed with STATUS_SUCCESS. This ensures that the oplock holder has received notification that their cached data is now stale and must be refreshed prior to subsequent use.

6.1.1.3 FSCTL_REQUEST_BATCH_OPLOCK

A batch oplock is an exclusive oplock against a file's contents and against changes in the attributes of the file (notably, but not exclusively, its name.) It allows a network client to keep a file "oplocked" even though the application on the remote client is opening and closing the file repeatedly.

A batch oplock can only be granted under the same circumstances as a level 1 oplock (c.f., Section 6.1.1.1.) The oplock itself is requested via an IRP. Returning STATUS_PENDING for that IRP indicates the oplock itself has been granted.

A batch oplock must be broken any time a Level 1 oplock would be broken. In addition, a batch oplock must also be broken whenever the name of the file changes. This is because a batch oplock covers the file even thought the client may be opening and closing the file repeatedly. Were this the case and a rename occurrs, the client needs to be advised that the file's data and attributes are no longer valid.

One interesting side-effect to using batch oplocks is that certain CREATE operations may fail with the Information field set to FILE_OPBATCH_BREAK_UNDERWAY. This occurs when the caller indicated they were unwilling to wait for the oplock break to complete by setting the FILE_COMPLETE_IF_OPLOCKED options flag, as is typically the case for SRV, the LanManager file server. In this case the create operation will fail (typically with STATUS_SHARING_VIOLATION) to indicate to the caller that the problem is with a batch oplock presently held on the file and that a blocking call to CREATE would not necessarily fail.

6.1.1.4 FSCTL_REQUEST_FILTER_OPLOCK

When a client opens a file, it may specify that it wishes to initiate a "filter" oplock against the file. This is done by setting the **FILE_RESERVE_OPFILTER** option as part of the file open operation. In order for the filter oplock to be granted the following must be true:

- This must be the only open instance of the file; and
- The caller must ask only for FILE_READ_ATTRIBUTES; and
- The caller is sharing read, write, and delete access.

A subsequent open of the file will cause the filter oplock to break if the caller asks for write access or fails to share read access to the file. If this subsequent open requests a filter oplock or specifies a destructive disposition for the file (**FILE_SUPERSEDE**, **FILE_OVERWRITE**, or **FILE_OVERWRITE_IF**) then the filter oplock is completely broken. Otherwise, it is demoted to a Level 2 oplock. Of course, other destructive operations (renaming the file, certain file system operations, etc.) will also cause a filter oplock break as well.

6.1.1.5 FSCTL_OPLOCK_BREAK_ACKNOWLEDGE

Once an exclusive (level 1 or batch) oplock has been broken, other file system requests cannot continue until the oplock break is acknowledged. This can be done one of two ways – either by a subsequent call to the FSD indicating a control code of **FSCTL_OPLOCK_BREAK_ACKNOWLEDGE** or by closing the file handle..

6.1.1.6 FSCTL_OPLOCK_BREAK_NOTIFY

When SRV opens a new file, indicating that it does not wish to wait for the oplock break to complete (c.f. Section 6.1.1.1) it must subsequently make a call to the underlying FSD to ensure that the oplock break has successfully completed. A filter driver may need to perform further processing on the file once this operation has completed, if it deferred it from the earlier create request.

A client indicates that it is awaiting completion of an oplock break by using **FSCTL_OPLOCK_BREAK_NOTIFY** as the control code in the IRP. This IRP will then block waiting for any oplock break activity to complete on the file. Once this call returns (**STATUS_SUCCESS**) the FSD may use the file object safely.

For SRV, proper implementation of these semantics by the FSD (and any attached filters) is essential to correct behavior. If a normally asynchronous CREATE operation by SRV is forced to be synchronous (perhaps by a filter driver) SRV will experience internal deadlock conditions.

A common problem for file system filter drivers is when they attempt to open a file that triggers an oplock break. Since the oplock break will block the create operation – and the create must return to the caller to process the oplock break, this yields a "classic" deadlock problem. To resolve this, the filter driver must also use the same processing model. Thus, the filter driver must specify **FILE_COMPLETE_IF_OPLOCKED** and may not perform I/O operations against the file object until the file object is ready for normal use.

6.1.1.7 FSCTL_OPBATCH_ACK_CLOSE_PENDING

Earlier we mentioned that an oplock break could be acknowledged by closing the file. The oplock owner uses this control code to indicate the oplock break has been acknowledged and a close of the file is imminent.

In this case, a level 2 oplock is not necessary. No further use should be made of this file object except to close the file.

6.1.1.8 FSCTL_OPLOCK_BREAK_ACK_NO_2

This control code is a variation on the general acknowledgment operation. In this instance, the owner of the oplock is declining the offer (by the FSD) of a level 2 oplock. This is typically because the owner of the oplock does not use or support level 2 oplocks.

6.2 I/O Reentrancy

A filter driver that performs its own I/O must deal with the potential problems of re-entrancy of those I/O operations. For example, if a filter driver does a **ZwCreateFile** or **IoCreateFile** call, the I/O request will initiate processing at the top of the device stack. Similarly, any other **Zw** API call would also cause the call to start processing at the top of the device stack.

Windows XP introduced **IoCreateFileSpecifyDeviceObjectHint** that allows a filter driver to direct that a specific create operation start at the specified location in the device stack. This avoids the re-entrancy problems inherent in **ZwCreateFile** or **IoCreateFile**. For other I/O operations, and for opening files on Windows 2000, it is necessary to use a different technique to avoid re-entrancy. Note that the handle returned by **IoCreateFileSpecifyDeviceObjectHint** is a normal file handle and may be used in other **Zw** API routines.

6.2.1 Building IRPs

Perhaps the most common mechanism for avoiding re-entrancy is to perform operations directly to the target device by either using an existing IRP, or constructing a new IRP, and sending it to the target device.

If the filter driver does not have an existing I/O request packet, it must allocate and initialize one prior to sending it to another driver. Allocating of IRPs can take one of three different forms:

- A driver may use I/O Manager functions to allocate and initialize an I/O request packet (**IoBuildSynchronousFsdRequest**, **IoBuildAsynchronousFsdRequest**, **IoBuildDeviceIoControlRequest**); or
- A driver may use I/O Manager functions to allocate (**IoAllocateIrp**) an IRP and initialize it within the driver itself; or
- A driver may manage its own pool of IRPs and allocate them from that pool. In this case, the IRP is formatted using **IoInitializeIrp**.

Regardless of the different techniques, the basic mechanism is similar. In the following example, we demonstrate using an I/O manager allocated IRP to query for the size of a file stored within an FSD.

```
VOID GetFileStandardInformation(PFILE_OBJECT FileObject,
                                        PFILE_STANDARD_INFORMATION StandardInformation,
                                        PIO_STATUS_BLOCK IoStatusBlock)
{
    PIRP irp;
    PDEVICE_OBJECT fsdDevice = IoGetRelatedDeviceObject(FileObject);
    KEVENT event;
    PIO_STACK_LOCATION ioStackLocation;

    //
    // Start off on the right foot - zero the information block.
    //
    RtlZeroMemory(StandardInformation, sizeof(FILE_STANDARD_INFORMATION));

    //
    // Allocate an irp for this request.  This could also come from a
    // private pool, for instance.
    //
    irp = IoAllocateIrp(fsdDevice->StackSize, FALSE);

    if (!irp) {

        //
        // Failure!
        //
```

```
            return;
        }

        irp->AssociatedIrp.SystemBuffer = StandardInformation;
        irp->UserEvent = &event;
        irp->UserIosb = IoStatusBlock;
        irp->Tail.Overlay.Thread = PsGetCurrentThread();
        irp->Tail.Overlay.OriginalFileObject = FileObject;
        irp->RequestorMode = KernelMode;

        //
        // Initialize the event
        //
        KeInitializeEvent(&event, SynchronizationEvent, FALSE);

        //
        // Set up the I/O stack location.
        //
        ioStackLocation = IoGetNextIrpStackLocation(irp);
        ioStackLocation->MajorFunction = IRP_MJ_QUERY_INFORMATION;
        ioStackLocation->DeviceObject = fsdDevice;
        ioStackLocation->FileObject = FileObject;
        ioStackLocation->Parameters.QueryFile.Length = sizeof(FILE_STANDARD_INFORMATION);
        ioStackLocation->Parameters.QueryFile.FileInformationClass = FileStandardInformation;

        //
        // Set the completion routine.
        //
        IoSetCompletionRoutine(irp, KfcIoCompletion, 0, TRUE, TRUE, TRUE);

        //
        // Send it to the FSD
        //
        (void) IoCallDriver(fsdDevice, irp);

        //
        // Wait for the I/O
        //
        KeWaitForSingleObject(&event, Executive, KernelMode, TRUE, 0);

        //
        // Done!
        //
        return;
}
```

In general, constructing IRPs can be a difficult process because of the number of options and permutations that can arise during their construction. In addition, there are some fields within the IRP that have reserved meanings (e.g., a filter driver should never set the **OriginalFileObject** field.) It is certainly quite possible that your filter driver might construct a combination of parameters that causes the FSD you are calling to exercise code paths that have never been used before because the request is using some previously unused combination of IRP options.

6.2.2 Alternate Device Path

A completely different mechanism for avoiding re-entrancy is for a filter driver to create two device objects for each device it is filtering. In Figure 10 we provide a simplified diagram of this model.

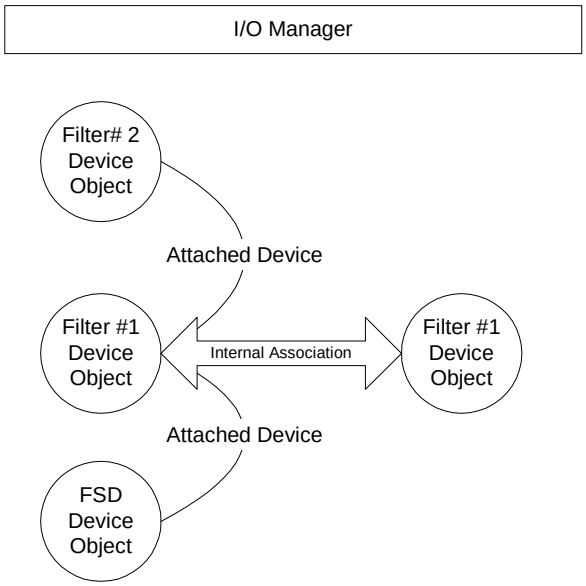


Figure 10 Using a Second Device in a Filter

In this model, a filter driver attaches one of the device objects to the device it is filtering. The second device object is associated with the first device object internally by the filter driver (typically, this would be done via information stored in the device extension field.)

This allows the filter to detect the re-entrancy of its own operations because they can be made against the second device object, rather than the first device object.

For example, suppose the filter needs to open a file "\foo\bar\test.txt". It would do so by calling **ZwCreateFile** specifying its second device's name plus the name of the file it intends to open – thus something like "\Device\MySecondDevice\foo\bar\test.txt". When the IRP arrives in the filter, the second device object will be passed into the appropriate dispatch call. The filter can then send the IRP to the FSD device object (a pointer to which is presumably stored in the first device object's device extension.)

This mechanism avoids re-entrancy, works with both Windows 2000 and Windows XP, and works with all I/O operations.

7 Data Modifying Filter Issues

Any file system filter driver that wishes to modify the data being returned from the file system to the application, must be prepared to handle some key issues. While any type of data that can be returned can be modified by the file system filter driver, this section of the document focuses on commonly modified data flows within the I/O path:

- I/O for user data (read/write, including paging I/O)
- Directory Enumeration Information
- File Information

7.1 Modifying File Data

In constructing a file system filter driver that modifies data there are several issues that must be handled appropriately by the filter including:

- The distinctions between cached, non-cached, and paging I/O operations;
- The manipulation of MDLs
- Modifications to data within the cache (presented to the filter via MDLs)

Cached I/O is the type of I/O that is normally performed by application programs. While many filter drivers initially attempt to base their data manipulations on cached I/O, this seldom works well in the Windows 2000 or Windows XP environments because applications may utilize alternative interfaces into the cache including the Fast I/O path, memory mapped files, and direct-to-cache interfaces (either via fast I/O or via the **IRP_MN_MDL** options to **IRP_MJ_READ** and **IRP_MJ_WRITE**.) Since these mechanisms are not performed using normal user-level read and write operations, a filter driver typically migrates to managing only non-cached I/O (whether application or paging I/O.)

Many of the issues involved in handling **IRP_MJ_READ** and **IRP_MJ_WRITE** operations were discussed earlier in Section 4 and thus we do not cover them here. An important issue for filter drivers that do modify the data is that they must substitute their own buffer for the caller-provided buffer.

For example, an encryption filter driver writer will usually find that they cannot encrypt the data “in place” for paging I/O calls. That is because the pages described by the MDL

in the I/O request are actually from the cache. Thus, modifying the data in-place will cause the data in the cache to be encrypted. An application that is using this information will see encrypted versions of the data that typically causes incorrect application behavior.

To resolve this problem, the filter driver writer will normally allocate their own separate buffer (from non-paged pool) and encrypt the data from the original location into the separate buffer. The filter driver then updates the IRP so that it refers to the separate (replacement) buffer. A step that is frequently omitted in updating the IRP is that the **UserBuffer** field of the IRP must be updated so that it corresponds to the address in the MDL. Normally, a driver does this by using **MmGetMdlVirtualAddress**. While this might seem odd, since the MDL does not have a user mode address, it turns out that the value of this field is used to compute offsets into the MDL – thus, it is used as a numerical value and is not dereferenced as a virtual address. If the file system filter driver does not set it up properly it can lead to unpredictable file system behavior.

7.2 Directory Enumeration Information

Some types of file system filter drivers need to modify information that is being returned as part of a directory enumeration. This is the case when the filter modifies the size of the data and wishes to control the size reported back to the user.

There are four distinct formats for the information returned from an FSD regarding directory enumeration; the specific format required is specified as part of the **IRP_MJ_QUERY_DIRECTORY** operation's parameters (in the I/O stack location) and is one of **FileDirectoryInformation**, **FileFullDirectoryInformation**, **FileBothDirectoryInformation**, or **FileNamesInformation**. The first three types are quite similar with **FileBothDirectoryInformation** being the most common format used by Windows applications. The other three types are rarely used.

The mechanism the underlying FSD uses to pack the return information can be a bit confusing, but it is essential for a file system filter driver that manipulates these values to understand it. Here are issues to keep in mind when manipulating directory information buffers:

- Directory information is placed directly into a user buffer, described by **Irp->UserBuffer**. As such, the operation must be done in thread context or it must be done using a system mapping of an MDL describing the user's buffer.
- For a successful return, the user's buffer will contain one or more entries. The last entry will indicate there are no further entries by setting the **NextEntryOffset** field to zero;
- If a name does not fit into the user's buffer, it will be truncated.
- Directory enumerations are frequent. Applications routinely enumerate some (or all) of the entries in the directory, often to obtain ancillary information (e.g., retrieve the correct case name information during a data copy.)
- Directory enumeration requests can contain wildcard characters. There are five such characters (<,>,",* and ?) with specific interpretation as to how they match in a regular expression (this is done by the function **FsRtlIsNameInExpression**.)

None of these are terribly difficult, but they all must be properly handled within your file system filter driver.

In the extreme case, your filter driver might even retrieve the entire contents of the directory and then present only a subset of the information to the caller. For example, your filter driver might be hiding the knowledge of the existence of certain files from application programs, in which case it would remove those entries from the directory. In that case, it would become your filter driver's responsibility to implement directory enumeration, and we would strongly suggest using the FastFat or CDFS examples in the IFS Kit in implementing this logic.

8 Naming

For an FSD (and hence a file system filter driver) most naming issues are handled when the file is first opened. There are also issues when a file is renamed. We discuss those issues in this section.

8.1 File Open (IRP_MJ_CREATE) Processing

Typically, a file system filter driver will handle many of its naming issues when first opening the file.

8.1.1 General Issues

Most names presented to the file system are "normal" names and because of this, many file system filter drivers are written to assume that they will only be presented with such normal names. However, the file systems allow a broad range of "special cases" for names. Further, Windows does not restrict the range of valid names, so that it is possible for one file system to support names that would be illegal for another file system.

For example, the NTFS file system allows a rather broad range of names. It is acceptable to use colons in the names of files – NTFS uses this mechanism to distinguish separate streams (the default data stream is ::$DATA. If an application does not specify a stream name, this is the stream that is used.) Also, because the POSIX definition allows the * character to be a legitimate file name character, it isn't uncommon for a file system filter driver to observe requests to open the '*' file.

Thus, it is generally the file system that determines whether or not a given file name is legal, or not. NTFS is the most permissive file system in this regard and those developing file system filter drivers should be warned to expect a broader range of naming issues when filtering NTFS.

8.1.1.1 Short File Names

One of the issues that is typically handled by a file system (and hence a file system filter driver) is that of short names. This is because, for each file, there can be two distinct names. One is the long file name and has minimal restrictions upon its format. The other is the short file name and is required to be compatible with the MS-DOS 8.3 name

format. Of course, if the file's long name is 8.3 compliant, then there is no separate short name.

Thus, whenever an FSD is attempting to find a file, it must use both the short and long file name. This is an important consideration for file system filter drivers, because many times they implement a policy based (in part) on file names and paths. The complexity is introduced because any directory can have a short name as well as any file. Thus, if a path to a given file has five different directories there are 64 ways to specify the name of the file – every possible permutation of the path name and the file name.

8.1.1.2 Relative Paths

A request to open a file can be specified relative to an existing open file; the "related file" is thus found by starting with the specified file and then, as necessary, parsing the balance of the related name.

We would note that the "related file" is normally a directory, but there is no requirement that this be the case. The related file could be an alternate stream of the file or it could even be the same stream of the same file (in which case the "file name" is a zero-length UNICODE string.) Sometimes a file is re-opened in order to perform a specific security check against the credentials of the caller.

8.1.1.3 Object/File Ids

Another special case example of names is the object ID. When an object ID is specified it is always done relative to some already opened file or directory; the precise file or directory is unimportant.

An object ID is a 128-bit GUID that is assigned by an application. A GUID can be assigned to a file to simplify finding – and opening – the file by those same applications. Thus, not all applications have object IDs assigned to them.

However, when a file is opened-by-ID, there is no guarantee that the underlying FSD can even provide the name of the file (although we note that typically NTFS will provide some name.) Indeed, for NTFS this is the most complicated because a single file might even have multiple paths to it (via hard links.) Thus, there is not necessarily any single name for a file – there may be many such names. File system filter drivers that rely upon name-based policy must be aware of this and handle it as appropriate for their filter.

8.1.1.4 Computing File Path Names

A frequent operation for file system filter drivers is to construct a "canonical" or "fully qualified" name for a file. There are a number of issues that must be considered for any file system filter driver that seeks to construct such canonical names:

- A file system filter driver will encounter circumstances when a name for the file cannot be derived;
- Names are not unique. Even in the common case where a name can be derived, it is probably not the only name for that file;
- Files may have both long and short names; so may each ancestral directory of that file;
- Network "shares" may overlap and thus network names cannot be directly compared.

Despite these limitations, it is normally possible to construct a path name for a given file (or directory.) In doing this, a file system filter driver will typically rely upon one or more different techniques:

- Using the **ZwQueryInformationFile** API. The name returned in this fashion will be a complete Windows NT name (provided that one can be returned) that will include the full path name of the file, including the device object on which the file resides;
- Using the **ObQueryNameString** API. The name returned in this case is the same as the name returned by **ZwQueryInformationFile**;
- Using **IRP_MJ_QUERY_INFORMATION**. In this case, the file system will return its portion of the name (thus, the name will exclude the device object portion returned by **ZwQueryInformationFile** and **ObQueryNameString**.)
- Using the name encoded in the FileObject that is passed to the file system as part of an **IRP_MJ_CREATE** operation.

In all of these operations, it is important to note that they cannot be used until after an **IRP_MJ_CREATE** has been processed by the underlying FSD. For example, a common file system filter driver error is to attempt to query the name (e.g., via

IRP_MJ_QUERY_INFORMATION) in their dispatch entry point for **IRP_MJ_CREATE**. Unfortunately, the file system cannot process the request because it has yet to construct its own internal data structures for the file object! Thus, this processing must be done after the filter driver's completion routine has been called (again, note that it may not be possible to perform this processing in the completion routine, since it might be called at IRQL **DISPATCH_LEVEL**.)

Assuming that it is possible for the filter driver to construct (or extract) the name of the file from the file system, it is also generally necessary to ensure some "canonical" form of the name for management purposes. Thus, it may be necessary to change short names to long names, or preserve the original case of the file name, or some other special characteristic. In that case, the normal approach is to examine the entry in the containing directory and use the data from that directory. You can simplify this by caching information (e.g., the canonical name information for containing directories, or for files that are frequently opened and closed) rather than recomputing this information on each new access to the file.

Given that it is important to ensure the correct behavior for your filter driver, a test case to consider building to ensure your filter's logic for extracting the canonical name is to ensure that you can handle an "open by file ID" (on an NTFS volume) where the caller does not have traverse privilege. For this case, there is no simple mechanism for extracting the name of the file.

8.2 Rename Processing

This section discusses how to decode one of the more complex File System IRPs: a rename request. In particular, we will show you how you can filter an NT system API **NtSetInformationFile()** operation for the case *FileInformationClass* equals *FileRenameInformation*. Specifically, we will look at the IRP that represents this operation.

8.2.1 Identifying an NtSetInformationFile Operation

Well of course a Rename operation in Windows is not its own separate type of file operation, as it is for example in the UNIX VFS architecture. Instead, it is one of many file operations that are all possible from a single Windows API call (We are not talking

about Win32. We are talking about the *real* system service API.). That API function is **NtSetInformationFile,** and its prototype is shown below:

```
NTSetInformationFile(
            HANDLE FileHandle,
            PIO_STATUS_BLOCK IoStatusBlock,
            PVOID FileInformation,
            ULONG Length,
            FILE_INFORMATION_CLASS FileInformationClass);
```

The careful observer may notice that this function looks exactly like **ZwSetInformationFile()**, except for the substitution of Nt for Zw. Just so. They are in fact the same. Of course, because it is not germane to device drivers, the DDK documentation for **ZwSetInformationFile()** contains absolutely no reference to rename operations, although this may change in a future version of the DDK.

Since the purpose of this discussion is to describe rename operations, we do not discuss other options about **NtSetInformationFile()**. Our goal is to assist you in understanding how a rename request is transformed from a procedure call in an application into an IRP. The IRP of course has a major function code, and it turns out that the major function code for file system "set information" operations is **IRP_MJ_SET_INFORMATION**. A file system driver, or file system filter driver on top of such a file system, which finds rename operations of interest must provide a dispatch entry point for **IRP_MJ_SET_INFORMATION**.

You can find out what *type* of the many different types of SetInformation operations you have in a particular IRP by looking at the IRP's *IoStack.Parameters.SetFile.FileInformationClass* field because that is where the I/O Manager has placed the value of FileInformationClass from the original request. In the case of a rename operation only the *FileRenameInformation* class operations are interesting.

In summary, to filter rename operations you must have a dispatch entry point in your filter driver for **IRP_MJ_SET_INFORMATION**, and in that dispatch entry point you have to further decode the IRP from the *IoStack.Parameters.SetFile.FileInformationClass* field in order to identify a rename operation.

The C code for routing from **IRP_MJ_SET_INFORMATION** to a *FileRenameInformation* operation might look like:

```
PIO_STACK_LOCATION irpSp = IoGetCurrentIrpStackLocation(Irp);
BOOLEAN IsRename = (irpSp->Parameters.SetFile.FileInformationClass ==
                    FileRenameInformation);
if (IsRename) {
                ntStatus = FilterProcessRenameOperation(Irp, irpSp);
}
```

8.2.1.1 What Is A File System Rename Operation?

It may seem obvious that a rename operation changes the name of a file. For example you could change the file C:\frob\nicate.txt to C:\frob\etacin.txt. It is less obvious that you could also change the name of C:\frob\nicate.txt to C:\frobnicate.txt. Which is to say you can *move* a file from one location to another by renaming it.

A rename operation has four pieces of pathname data to deal with:

- The source directory or pathname
- The source filename
- The target directory or pathname
- The target filename

In addition, there are three different types of renaming that can occur:

- Rename, change the name of the source file, *not* its location
- Move - change a file's location, *not* its name
- Move and rename – change the name of a file *and* its location

In addition, there may or may not be a conflict with an existing file that currently uses the rename target filename. A filter driver or a file system driver might have to understand what is the correct thing to do if a conflict occurs.

Windows NT file system documentation refers to three types of rename operations called **Simple Rename**, **Fully Qualified Rename**, and **Relative Rename**. These only roughly correspond to the 3 cases above. What these three NT types of rename really describe is how the target of the rename operation is described by the SetInformation operation, and therefore how the file system driver, and by extension our file system filter driver, can understand it.

In the case of a Simple Rename, the source file object and the target file object are in fact the same objects. Only the name (not the directory) of the file is being changed.

In the Fully Qualified Rename case, an absolute (fully qualified) pathname identifies the target of the rename operation. In this case the target filename may or may not be the same as the source filename, but we can be sure that the fully qualified source pathname is not the same as the fully qualified target pathname.

In the case of a Relative Rename, the target file name is specified as a filename relative to a directory. Once again we can be sure that the two fully qualified pathnames are not the same.

8.2.1.2 Filter Drivers and Rename Operations

A File System Filter Driver that is performing replication services might wish to track all rename operations so that these operations can be replayed on another file system, perhaps even on another system. To do so the filter driver would have to record the name of the file being renamed (the source file) and the new name of the file (the target file.)

For example, if I were to rename C:\frob\nicate.txt to C:\frobnicate.txt, my replication filter driver would create the log record:

```
RENAME: C:\frob\nicate.txt C:\frobnicate.txt
```

And my *replication agent*, in communication with my filter driver, would read this log record and then perform whatever miracle is required to replicate the rename elsewhere.

All the filter driver has to do is produce a simple record containing three values:

- The operation (RENAME)
- The fully qualified pathname of the source file
- The fully qualified pathname of the target file

While this seems to be something simple, it is unfortunately more complicated, as we discuss in the next section.

8.2.1.3 Getting at the parameters

The IFS Kit defines the parameters relevant to an **IRP_MJ_SET_INFORMATION** in one of the fields in the union *IO_STACK.Parameters* (shown below).

```
//
// System service parameters for: NtSetInformationFile
//
struct {

        ULONG Length;
        FILE_INFORMATION_CLASS FileInformationClass;
        PFILE_OBJECT FileObject;
```

```
        union {
        struct {
        BOOLEAN ReplaceIfExists;
        BOOLEAN AdvanceOnly;
        };
        ULONG ClusterCount;
        HANDLE DeleteHandle;
        };
} SetFile;
```

In addition, the parameters for a rename operation are described there as well. These parameters are located in the system buffer (*Irp->AssociatedIrp.SystemBuffer*). The data structure describing this information is:

```
typedef struct {
    BOOLEAN Replace;
    HANDLE RootDir;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_RENAME_INFORMATION, *PFILE_RENAME_INFORMATION;
```

8.2.1.4 Putting The Pieces Together

Now that you now know where all the pieces are, all that remains is to put them into the context of a rename operation so that your filter driver can do something useful.

We need to look at the two structures from the **SET_INFORMATION** IRP, the *IO_STACK_LOCATION.Parameters.SetFile* structure and the **FILE_RENAME_INFORMATION** structure found at *IRP->AssociatedIrp.SystemBuffer*. The goal of this exercise, then, is to produce a log record of the rename operation. Of course, the details depend upon what type of rename is being performed:

- *Simple Rename:* SetFile.FileObject is NULL.
- *Fully Qualified Rename*: SetFile.FileObject is non-NULL and FILE_RENAME_INFORMATION.RootDir is NULL.
- *Relative Rename*: SetFile.FileObject and FILE_RENAME_INFORMATION.RootDir are both non-NULL.

Expressed as a C algorithm, these rules look like the code segment below.

```
//
// assume that Irp is a pointer to an IRP and that
// irpSp is a pointer to our IO_STACK location in that IRP.
//
PFILE_RENAME_INFORMATION renameInfo;
renameInfo = (PFILE_RENAME_INFORMATION) Irp->AssociatedIrp.SystemBuffer;
if (!irpSp->Parameters.SetFile.FileObject) {
        //
        // simple rename case – call some function that handles this case
        //
        status = processSimpleRename(Irp, irpSp, renameInfo);
} else {
```

```
        //
        //
        if (renameInfo->RootDirectory == NULL) {
                //
                //fully qualified rename case – call a function for this
                //
                status = processFQRename(Irp, irpSp, renameInfo);
        } else {
                //
                // relative rename case – call a function for this case.
                //
                status = processRelativeRename(Irp, irpSp, renameInfo);
        }
}
```

Earlier we mentioned that you might need to know what to do if a rename target already exists. The SetFile.ReplaceIfExists field informs the driver about the correct action. This is a BOOLEAN, and if it has the value TRUE, a rename target can be deleted if it exists, otherwise, if it is FALSE, an existing rename target causes the rename operation to fail.

Starters

Build a fully qualified filename for the source file object. Having done that, you still need the fully qualified target pathname in order to record the file operation. How to get the target fully qualified pathname depends on which type of rename operation being handled: simple, fully qualified or relative. The remainder of this article demonstrates how, for each rename type, the target pathname can be constructed.

8.2.1.5 Simple Rename

In the case of a simple rename, constructing the fully qualified filename is straightforward. You can just copy the pathname component of the fully qualified source filename and append the target filename. The target filename is the WCHAR string at FILE_RENAME_INFORMATION.FileName.

The code for a simple rename might look like that below(note that the following code segments, while based on a functional filter driver are pseudo code only and will not compile without errors).

```
//
// simple rename case
// FileNameBuffer is the fully qualified source file name
//
WCHAR * FileNameBuffer;
//
// renameInfo is a pointer to the FILE_RENAME_INFORMATION structure
//
PFILE_RENAME_INFORMATION renameInfo;
//
// NewNameBuffer will contain the fully qualified target file name
//
WCHAR * NewNameBuffer;
//
// prefix is a pointer to the end of the pathname
// component of FileNameBuffer
//
```

```
WCHAR * prefix;
//
// prefixLength is the length of prefix
//
ULONG prefixLength;
//
// length is the string length of NewNameBuffer
//
ULONG length;
//
// First find out just how big a string we need:
//
length = 0;
//
// find the last pathname component of the source file
//
prefix = wcsrchr(FileNameBuffer, (int) L'\\');
if (prefix == NULL) {
        //
        // just the file system will fail too.
        //
        return (STATUS_SUCCESS);
} else {
        //
        // set prefixlength and initalize length
        //
        length = prefixLength = ((prefix - FileNameBuffer) + 1)* sizeof(WCHAR);
}
//
// Now add the prefix length to the length of the target FileName
//
length += (wcslen(renameInfo->FileName) + 1) * sizeof(WCHAR);
//
// Great! Now allocate the buffer for this thing
//
NewNameBuffer = ExAllocatePoolWithTag(PagedPool, length, 'tset');
if (!NewNameBuffer) { // ? no memory of any type?
    return (STATUS_INSUFFICIENT_RESOURCES);
}
//
// Ok we have the memory and the length, now construct the string
//
RtlZeroMemory(NewNameBuffer, length);
(void) wcsncpy(NewNameBuffer, FileNameBuffer, prefixLength);
wcscat(NewNameBuffer, renameInfo->FileName);
//
// That's it – now just create the log record
//
```

8.2.1.6 Fully Qualified Rename

In the Fully Qualified Rename case, RenameInfo.FileName is the fully qualified pathname of the rename target. This makes the task relatively simple.

However, one other complexity here is retrieving the drive specification of the fully qualified pathname. In the simple rename case we ignored the drive specification portion of the fully qualified pathname. In other words we just left out the fact that to fully log a file system operation we need the fully qualified pathname to include the drive (e.g. "C:"). This was to simplify our discussion of rename, since dealing with naming is an independently complex topic.

Since we are not just copying the source file name, this drive letter has to come from somewhere. In the following example, how we obtain the drive specification is undefined, and a global variable DriveLetter just magically has the correct value.

```
//
// fully qualified rename
//
// renameInfo is a pointer to the FILE_RENAME_INFORMATION structure
//
PFILE_RENAME_INFORMATION renameInfo;
//
// NewNameBuffer will contain the fully qualified target file name
//
WCHAR * NewNameBuffer;
//
// startOffset is used to locate the beginning of the fully qualified pathname
// we want to use from the string that is passed into us at renameInfo.FileName
//
ULONG startOffset = 0;
//
// If we need to add a drive spec, prependChars is used to account for the space
// we need to do that.
//
ULONG prependChars = 0;
//
// finally target length is going to be set to the total size of the WCHAR string
// we are going to construct.
//
ULONG targetlength = 0;
//
// DriveLetter has been set to the correct value for this operation.
//
extern WCHAR DriveLetter;
//
// For this case the string at renameInfo.FileName MUST start with a '\'
// and could start with "\DosDevices\". If we find "\DosDevices\" then
// we can assume that this string already has a drive specifier, otherwise we
// will provide one based on the value of DriveLetter.
//
// If the string does not start with a '\' note this and stop
// trying to process this operation.
//
if (renameInfo->FileName[0] != L'\\') {
      return (STATUS_SUCCESS);
}
//
// See if the prefix of FileName is "\DosDevices\"
// if it is, strip off the \DosDevices\ part, leaving the FQpathname.
// else prepend the Device specification.
//
if (0 == wcsnicmp(L"\\DosDevices\\", renameInfo->FileName, 12)) {
            startOffset = 12;
            // Copy from end of \DosDevices\
} else {
       // put in the device spec (e.g. D:)
       prependChars = 2;
}
//
// figure out how many WCHARS we need
//
targetLength = (renameInfo->FileNameLength/sizeof(WCHAR)) –
            startOffset + prependChars + 1;
            // make room for NULL too
if (targetLength < 4) {
            // must be at least "D:\"
       return (STATUS_SUCCESS);
}
//
// Allocate the storage for the target file name
//
NewNameBuffer = ExAllocatePool(PagedPool, targetLength * sizeof(WCHAR), 'tset');
//
// if there is no paged pool available fail the operation
//
if (NewNameBuffer == NULL) {
return (STATUS_INSUFFICIENT_RESOURCES);
}
//
// now construct the string by prepending the drive letter if needed
// and appending the filename in renameInfo.
//
if (prependChars) {
```

```
            //
            // set the device spec, just assume that DriveLetter is a WCHAR
            // that contains the correct value.
            //
            NewNameBuffer[0] = DriveLetter;
            NewNameBuffer[1] = L':';
    }
    wcsncpy(&NewNameBuffer[prependChars], &renameInfo->FileName[startOffset],
                    targetLength );
    //
    // that's all – go log the operation
    //
```

8.2.1.7 Relative Rename

The last case is the most complicated – the relative rename operation. In this case the source file is being moved to a new location relative to the directory referenced at FILE_RENAME_INFORMATION.RootDir. One can easily construct the fully qualified pathname from a file object, but we are given a handle. So we have to call into the object manager to get the object (i.e. the file object) referenced by the handle at RootDir. This can only work if we are in the same process context as the process that issued the original I/O request. For file systems that support streams, a rename on a stream always appears as a relative rename.

An example of processing a relative rename operation in order to produce a log record of the operation from a file system filter driver is shown below.

```
//
// relative rename
//
// renameInfo is a pointer to the FILE_RENAME_INFORMATION structure
//
PFILE_RENAME_INFORMATION renameInfo;
//
// NewNameBuffer will contain the fully qualified target file name
//
WCHAR * NewNameBuffer;
//
// status is use to collect the return value from standard DDK functions.
//
NTSTATUS status;
//
// We need to convert from the RootDir HANDLE to a FILE_OBJECT
//
PFILE_OBJECT dirObject;
//
// we will call ConstructFQPname() and this function will magically produce
// a fully qualified pathname given a file object and a drive specification
// dirPath will contain the unicode string for the fully qualified pathname
//
UNICODE_STRING dirPath;
//
// as usual we need to index into WCHAR strings
//
ULONG wcharOffset;
//
// Our friend the mysterious drive specification
//
extern WCHAR DriveLetter;
//
// We need the fully qualified pathname
// of the directory referenced by Buffer->RootDir
// Once we have that we should then be able to
// append the contents of Buffer->FileName to
// construct the fully qualified target pathname
//
```

```
status = ObReferenceObjectByHandle(renameInfo->RootDirectory,
                                   STANDARD_RIGHTS_REQUIRED,
                                   NULL,
                                   KernelMode,
                                   (PVOID *)&dirObject,
                                   NULL);
if (!NT_SUCCESS(status) ) {
        //
        // assume that the file system driver will fail as well
        //
        return (STATUS_SUCCESS);
}
//
// the function ConstructFQPname takes a file object and a driveletter
// as input and produces a UNICODE_STRING containing the fully
// qualified pathname of the input file object as output.
// The buffer for the UNICODE_STRING is allocated by this function, we
// must remember to free it up when we are done
//
if (! ConstructFQPname(dirObject, &dirPath, DriveLetter) ) {
        //
        // assume the file system driver will fail
        //
        return (STATUS_SUCCESS);
}
ObDereferenceObject(dirObject); // we don't need this anymore
//
// figure out how many WCHARs we need
//
targetLength = ((dirPath.Length + renameInfo->FileNameLength ) /
                sizeof(WCHAR)) + 2; // add in a L'\' and a NULL
//
// Allocate the storage for the target file name
//
NewNameBuffer = ExAllocatePool(PagedPool, targetLength * sizeof(WCHAR), 'tset');
if (NewNameBuffer == NULL) {
        //
        // Or we can stop a file operation right here
        //
        return (STATUS_INSUFFICIENT_RESOURCES);
}
//
// copy the relative directory fully qualified pathname to our buffer
//
wcharOffset = dirPath.Length/sizeof(WCHAR);
//
// if we need a path separator, then add it it now
//
if (NewNameBuffer[wcharOffset-1] != L'\\') {
        NewNameBuffer[wcharOffset] = L'\\';
        wcsncpy(&NewNameBuffer[wcharOffset+1],
                renameInfo->FileName,
                renameInfo->FileNameLength/sizeof(WCHAR));
} else {
        //
        // we don't need a path separator
        //
        wcsncpy(&NewNameBuffer[wcharOffset],
                renameInfo->FileName,
                renameInfo->FileNameLength/sizeof(WCHAR));
}
//
// since constructFQPname allocated storage, it would be nice to free it up
//
ExFreePool(dirPath.Buffer);
//
// that's all – go log the operation
//
```

8.2.1.8 Conclusion

Thus, to handle rename operations here is what your filter driver needs to do:

- Filter IRP_MJ_SET_INFORMATION operations.

- Decode rename operations by examining the IO_STACK.Parameters.SetFile.FileInformationClass field.
- Compute the fully qualified pathnames for the source and target files of the rename operation. To do that you have to understand the rules for deciding if you have a simple rename, a fully qualified rename, or a relative rename, and derive the target file name correctly based on which type of rename you have.

That's all that there really is to logging a rename operation.

9 Mounting/Handling Removable Media

Very few things can cause more problems than removable media, and this problem is no better in Windows 2000 than it is in earlier versions. Fortunately, this problem has been resolved in Windows XP.

For example, one problem that we continue to observe is a bug where the filter driver deletes its device object prematurely. For this to occur, the filter driver must use a completion handler. The call arrives in the filter driver. The filter in turn passes the call down to the underlying file system. The underlying filesystem device then deletes its device object, which in turn causes the filter's fast I/O detach device entry point to be called. The filter driver then detaches and deletes its own device object.

The file system then completes the IRP (normally an **IRP_MJ_CLOSE**, but also some of the **IRP_MJ_PNP** operations related to removable devices.) The I/O Manager then calls the filter's completion routine. The completion routine, if it attempts to access the device object, can experience various problems – since it has already deleted that device object!

For example, if we look at the FileSpy code in the Windows 2000 IFS Kit, we note that in their completion routine they access their device object (in SpyPassThroughCompletion):

```
if (SHOULD_LOG(DeviceObject))
```

And of course the corresponding (and necessary, to trigger this problem,) code in DetachDevice:

```
IoDetachDevice( TargetDevice );
IoDeleteDevice( SourceDevice );
```

(Indeed, you can find comparable code in the sfilter example as well.)

Now, if we review FastFat, we note that the routine FatCheckForDismount is called when there are no longer any open files on the volume (that is, this is the *last* close on the volume.) This is in close.c, near the end of FatCommonClose:

```
if ( (Vcb->OpenFileCount == 0) &&
     ((Vcb->VcbCondition == VcbNotMounted) ||
      (Vcb->VcbCondition == VcbBad)) &&
     FatCheckForDismount( &IrpContext, Vcb, FALSE ) ) {

    //
    //  If this is not the Vpb "attached" to the device, free it.
```

And then in FatCheckForDismount:

```
FatDeleteVcb( IrpContext, Vcb );

Vpb->DeviceObject = NULL;

IoDeleteDevice( (PDEVICE_OBJECT)
                CONTAINING_RECORD( Vcb,
                                   VOLUME_DEVICE_OBJECT,
                                   Vcb ) );

VcbDeleted = TRUE;
```

Then (back in FatCommonClose) the processing finishes and (in FatFsdClose) the file system calls FatCompleteRequest. This then calls into the filter driver's completion routine, where it references its now deleted device object's device extension.

For a file system filter driver that only deals with non-removable media, this is not a problem. For a file system filter driver that deals with removable media, however, there is (at present) no good solution to this problem.

The correct solution is to implement a reference counting scheme, so that each time the filter sends an IRP down to the underlying file system, it bumps a reference count in the device object extension. When the filter receives the device back in the completion routine it decrements the reference count. It also would then maintain one reference for the attachment. Thus, when the device is both detached AND there are no outstanding IRPs, then the filter driver can delete the device object. To fix this in Windows XP, the I/O Manager now holds an additional reference, so that the device object cannot "disappear" while the IRP is being processed.

Implementing this within your filter driver is an expensive solution, but it is correct – and it eliminates crashes that can be caused by manipulating your device extension after the device object has been deleted.

10 User/Kernel Communications

A common issue for file system filter drivers is the need to communicate between a user-mode service and its kernel mode driver. Such communications are a standard driver problem, not restricted to file system filter drivers, and we would note that solutions designed to work with other drivers will also work with file system filter driver.

10.1 Shared Memory

Perhaps the simplest (and most common) mechanism to use for communicating between user and kernel components is to utilize a "shared memory" region. In this model, the application and the kernel driver utilize the same physical memory (although usually with different virtual mappings to that memory) to exchange information.

10.1.1 Application-provided Memory Buffer

If the shared memory is provided by the application, it would normally send an address to the driver via a device I/O control call. The driver then must ensure that the application-provided memory has been locked down. If the memory is uni-directional (that is, it will either be written or read, but not both,) this can also be done by the I/O Manager by specifying either **METHOD_IN_DIRECT** or **METHOD_OUT_DIRECT** in the IOCTL code. However, if the memory is bi-directional, there is no documented mechanism to properly probe the memory for both read and write access. In that case, to utilize a documented mechanism, the driver would use **METHOD_NEITHER**. The user buffer (Irp->UserBuffer) would then be probed and locked by the driver, being careful to catch any exceptions (using structured exception handling via try and except.) Note that a driver cannot use **METHOD_BUFFERED** using this technique, since buffered I/O requires copying data between user and kernel buffers, which undermines the very purpose of sharing memory.

Thus, the normal sequence to lock down memory by a driver would be followed in this case as well:

- Allocate an MDL (**IoAllocateMdl**)
- Probe and Lock the physical memory (**MmProbeAndLockPages**)
- Map the MDL to a system virtual address (**MmGetSystemAddressForMdlSafe**)

In addition, the driver and application must agree upon the signaling mechanism (normally an event object) to use between them as well as the data format of the shared memory region.

In the DDK, the **async** example demonstrates how to share an event between the user and kernel components. Because user components are typically written to use the Win32 API, the kernel components must be aware that the Win32 functions restrict access to named events – they must be located in the **BaseNamedObjects** directory. Again, this is demonstrated in the DDK async example.

The driver is responsible for ensuring proper clean-up of the memory that has been locked down using this mechanism. A driver typically does this by monitoring the application. The application could send an explicit request (via an IOCTL operation) or the application could terminate abnormally, in which case the driver would detect this by the receipt of an **IRP_MJ_CLEANUP** request against the file object being used by the application. At this point, the driver must unmap (**MmUnmapLockedPages**) and unlock (**MmUnlockPages**) the memory it previously mapped and locked. Then it would free the MDL (**IoFreeMdl**).

10.1.2 Application-provided Section

Another possible technique for sharing memory between the application and the driver is to share a memory section, backed either by the paging file or by a regular file system file.

In this model, instead of locking down memory, the driver would access the section using either the name of the section, the name of the file, the handle of the section or the handle of the file, one of which would be provided by the application to the driver, presumably via an IOCTL call. For named objects, it would also be possible to accomplish this by utilizing the registry to store the necessary configuration information.

The driver must take the following steps to obtain a valid mapping to the same memory:

- First, it must obtain a pointer to the section object. If the application provided a section handle, section name, file name, or file handle, the driver must perform additional work:

  - For a section handle, the driver must convert the handle to an object using **ObReferenceObjectByHandle**;
  - For a section name, the driver must open the section using **ZwOpenSection**, which will provide a section handle (which then must be converted to a section object);
  - For a file handle, the driver must open the section using **ZwCreateSection**, which takes the file handle, and returns a section handle (which then must be converted to a section object);
  - For a file name, the driver must open the file using **ZwOpenFile** (or **ZwCreateFile**, or **IoCreateFile**, or **IoCreateFileSpecifyDeviceObjectHint**.) This then returns a file handle, which can be passed to **ZwCreateSection**.

- With the section object, the driver would map the section into kernel memory using **MmMapViewInSystemSpace**.

Because this is shared memory between the application and the driver, any changes by one are visible by the other.

This second technique can be quite useful for sharing memory between multiple applications and drivers, because each application and driver locate the shared memory segment using a well-known name (the name of the section or file) rather than using explicit addresses.

Again, the application(s) and driver(s) must agree on a serialization mechanism, which is normally accomplished using a named event.

10.1.3 Driver-provided Memory

The driver can implement the same basic techniques described earlier for applications. For example, a memory buffer can be mapped from the system address space into the application's address space using **MmMapLockedPages** and specifying **UserMode** access rather than kernel mode access.

Similarly, a named section can be created by the driver, rather than by the application, although in such a case the driver must use **ZwCreateSection** (since **ZwOpenSection** will fail if the section already exists.)

10.2 Hanging IRP

Another model for user/kernel communications is to utilize asynchronous I/O operations. In this model, the user mode application sends an IOCTL call to the filter driver. Upon receiving this IRP, the filter driver:

- Marks the IRP as pending (using **IoMarkIrpPending**);
- Places the IRP on a queue (typically stored in the device extension for the filter);
- Sets an event, indicating there is something on the queue;
- Returns **STATUS_PENDING** to the caller

When the driver must indicate an event to the application:

- It checks the queue:
    - If there is an element on the queue it removes it; or
    - If the queue is empty, it resets the event, checks the queue one last time and then waits on the event;
- It builds the response information for the application program and stores it in the output buffer (Irp->AssociatedIrp.SystemBuffer, typically, but that depends upon the transfer method specified in the IOCTL code.)
- It completes the IRP, specifying the return information size in the Information field, and **STATUS_SUCCESS** in the status field.

This model can be combined with shared memory, in which case the return information is stored in the shared memory region. It can also utilize overlapped I/O (for Win32 applications,) which allows a single user mode thread to issue many such requests to the driver, or the application may create several user mode threads, each of which issues this same IOCTL.

11 File System Behavior Differences

Frequently, a single file system filter driver is expected to handle and process requests for multiple file systems. Each of these file systems behaves somewhat differently from the others. In this section we discuss some of those differences and how to anticipate them in the development of our own filter driver.

11.1 NTFS

In filtering NTFS we have found a large number of interesting issues and differences that are unique to NTFS. Each of these can pose a potential problem to a file system filter driver. The primary differences we have observed are:

- It optionally honors the FILE_NO_INTERMEDIATE_BUFFERING option; a file that is compressed or encrypted will be cached by the NTFS file system even when this option is chosen;
- It utilizes "stream file objects" (which are not to be confused with "streams") extensively, and because of this the filters must be able to handle them (see Section 2.1.3 for more information on Stream File Objects);
- It supports alternative "streams" of information, in addition to the default ::$DATA stream;
- It is the only FSD that truly supports POSIX naming semantics. While few applications use these semantics, "few" does not equate to "none" and thus this is a case that must be handled properly within a filter driver;
- It is one of two FSDs that support Services for Macintosh (SFM) and SFM is only available on server versions of Windows;
- It still supports OS/2 style extended attributes;
- It offers numerous special options: reparse points, sparse files, USN journal, hard links, etc. Each of these must be properly handled by a file system filter driver
- It does not handle traditional removable media. By this, we mean that it does not handle the

  IRP_MJ_FILE_SYSTEM_CONTROL/IRP_MN_VERIFY_VOLUME

  operation. This is different than FAT and CDFS. While it is possible to dismount media, NTFS does not handle external ejection of the media.

- NTFS is the only file system provided with Windows that supports security descriptors; file system filter drivers may not care about such, but again must be designed to handle them properly.

We often find that a file system filter driver will have more issues when working with NTFS than with any other file system. Thus, we believe it is imperative that any file system filter driver must be thoroughly tested with NTFS, across a broad range of actual uses.

11.2 FAT

Fortunately, filtering the FAT file system is often straightforward. It is immensely helpful that the FAT file system source code is included in the IFS Kit – and anyone writing file system filter drivers will find that having a build of FAT with full symbols is of tremendous value when trying to debug even the simplest of file system problems. Even FAT has some minor differences, however:

- FAT supports removable media, including media that can be externally removed and must be "discovered" by the FSD.
- FAT will "re-mount" volumes. In this case, when a previously dismounted volume is "re-discovered", the volume device object from the previous mount will be re-used.
- FAT does not support many of the "advanced" features of NTFS. Thus, for instance, FAT does not support the ability to open a file by ID (See Section 8.1.1.3);
- FAT is the only file system (besides RAW) that supports floppy media.

Fortunately, FAT and NTFS have many common elements and because of this, a filter that works with one will largely work with the other.