In this chapter:

- *I/O Revisited: Who Called?*
- *Asynchronous I/O Processing*
- *Dispatch Routine: File Information*
- *Dispatch Routine: Directory Control*
- *Dispatch Routine: Cleanup*
- *Dispatch Routine: Close*

10

Writing A File System Driver II

In this chapter, we'll continue to discuss how a file system driver can be conceived and implemented. First, discuss the read and write dispatch routines that you were introduced to in the previous chapter, focusing on the different ways in which these two entry points can be invoked. When you design a file system driver, knowing the different ways in which a particular dispatch routine can be invoked is essential to creating a robust design. I intend to help you understand better the logic described by the code and comments presented in the previous chapter, as well as to plug in the gaps left by the sample code presented earlier. In order to understand the context in which these two routines can be invoked, you must first understand the concept of the *top-level component* for any I/O request dispatched to an FSD. I discuss this concept at length here.

Next I look at some of the issues that you must deal with in providing support for asynchronous I/O, including the file information dispatch routines (both query and set file information) and the directory control, cleanup, and close entry points. By this time, you should have a very good understanding of the issues involved in providing some of the basic functionality expected from a Windows NT file system driver.

I/O Revisited: Who Called?

Throughout the course of this book, I have repeatedly mentioned that the FSD read and write dispatch routines can be invoked by all sorts of different components on a Windows NT system and that these invocations can occur due to different direct or indirect actions initiated by processes. Here is a formal list of

the different ways in which the read and write entry points for a file system driver can be invoked:

- From a user- or kernel-mode thread that requests I/O using one of the NT system services, e.g., NtReadFile(), NtWriteFile(), ZwReadFile(), ZwWriteFile(), or NtFlushBuffers()
- From a user- or kernel-mode thread as a result of a page fault on a byte range that is part of a mapped view of a named file stream (i.e., page faults on virtual addresses backed by named file objects):
  - From the NT Cache Manager as a result of asynchronous read-ahead operations being performed
  - Recursion into the FSD dispatch routines, due to page faults incurred by the NT Cache Manager when servicing a buffered I/O request
  - Due to page faults in files mapped by user application processes (typically page faults on mapped-in, executable files)
- From the NT Virtual Memory Manager as a result of servicing a page fault that was incurred by some user-mode or kernel-mode process for allocated buffers (i.e., page faults on virtual addresses backed by paging files)
- From the NT Virtual Memory Manager as a result of asynchronous flushing of modified pages (modified page write operations)*
- From the NT Cache Manager as a result of asynchronous flushing of Cache Manager buffers (lazy-write of data)

Regardless of the caller and of the situation leading up to the invocation of the read/write entry points, the implementation of these two important dispatch entry points should try to achieve the following goals:

- Satisfy cached (buffered) I/O requests by forwarding the I/O request to the NT Cache Manager
- Satisfy nonbuffered I/O requests by directly accessing secondary storage devices
- Return a consistent view of file stream data, regardless of whether the request is a buffered or nonbuffered I/O request
- Try to maintain consistency between views of file data mapped in as an executable and as a regular data stream

---

* For the purposes of discussions in this book, there is conceptually no difference between VMM-initiated flushing of pages belonging to a page file and VMM-initiated flushing of pages belonging to a named on-disk (mapped) file.

- Ensure correct synchronization by following a strict, well-defined resource acquisition hierarchy

Figure 10-1 illustrates the manner in which the NT file system drivers, the NT Cache Manager, and the NT VMM interact.* This figure also serves to demonstrate the various ways in which an FSD read/write dispatch entry point can be invoked. In order to better understand how the FSD achieves the goals listed earlier, you should understand the concept of a top-level component for an IRP.

Top-Level Componentfor an IRP

From the figure, you can see that an I/O request in Windows NT can be one of the following three types:

- The I/O request is directly issued to an FSD.
- The I/O request either originates in the Cache Manager component or is handed directly to the Cache Manager by the I/O Manager (bypassing the FSD).
- The I/O request originates in the VMM component, or is directly handled by the VMM in the kernel in the case of a page fault.

Depending on which category a given I/O request falls into, the FSD always identifies a top-level component that is associated with the IRP representing the request. The top-level component is defined as the kernel-mode component that initiates the processing for a specific I/O request.t

Note carefully that identification of a top-level component is not restricted to read/write I/O requests. Rather, your FSD must consistently be aware of the top-level component associated with any functionality invoked in your FSD implementation; either the FSD itself, or the NT VMM could be a top-level component for query/set file size requests.

According to our definition, therefore, the FSD will identify itself as the top-level component when a user read request is directly forwarded by the NT I/O Manager to the read dispatch routine in the FSD, because all of the processing for that IRP is initiated in the FSD dispatch routine. If instead, the I/O request for a read operation originates in the NT Cache Manager (due to read-ahead being

---

' Note that the shaded areas represent modules that initiate asynchronous I/O in the context of system worker threads or dedicated kernel-mode worker threads.

t Microsoft Windows NT developers have previously defined a top-level component as the kernel-mode component that directly receives the user I/O request. I believe that this definition is not complete, since lead-ahead and lazy-write calls originating in the NT Cache Manager clo not originate as a result of any particular user request, yet the Cache Manager should be considered the top-level component for these 1/0 operations. We will therefore use the definition presented here to identify top-level components.

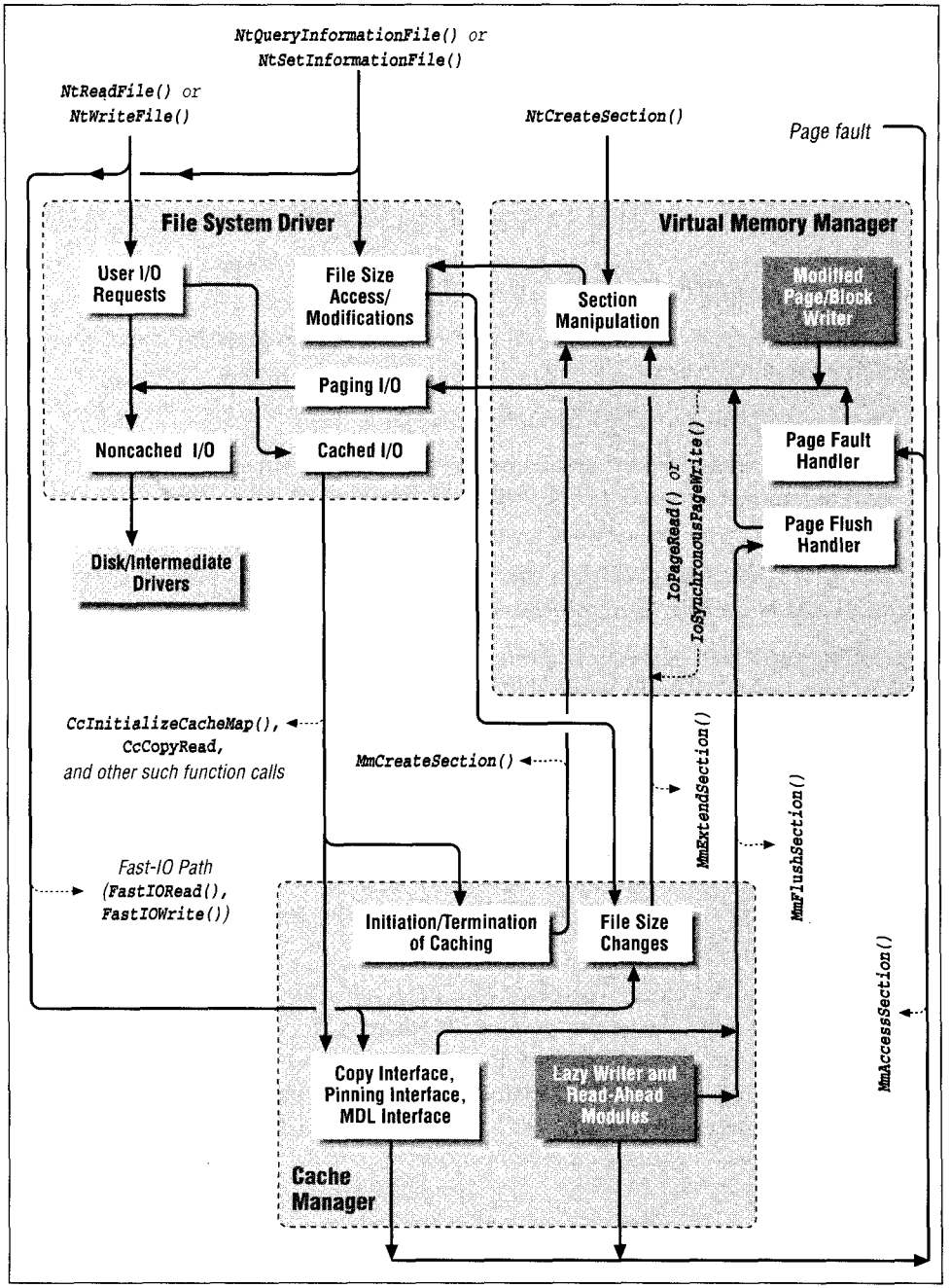

Figure 10-1. Interaction between FSDs, the VMM, and the Cache Manager

performed), the FSD identifies the Cache Manager as being the top-level component handling the particular IRP. Again, the rationale for classifying the Cache Manager as the top-level component is that all of the processing for the particular I/O operation originates in the Cache Manager. Finally, requests that originate in the VMM for flushing modified pages to secondary storage are identified by the FSD by noting that the VMM is the top-level component associated with the request.

Setting and querying the top-level component value

To identify the top-level component for a particular I/O request, the FSD, the NT Cache Manager, and the NT VMM use Thread-Local Storage (TLS). A thread is represented within the Windows NT Executive by a structure called ETHREAD. Although the structure is opaque to most of the NT Executive components, including FSDs, you should note that this structure contains a field called TopLevellrp. The TopLevellrp field is large enough to store a pointer value. An FSD typically stores the pointer to the current IRP being processed in the context of a particular thread in this field. This, however, is only done if the FSD is the top-level component for the IRP.

There are a few constant values that are used to identify the fact that some other component may be top-level for a particular IRP. For example, the fact that the NT Cache Manager is top-level for a particular I/O request is noted by storing a constant value defined as FSRTL_CACHE_TOP_LEVEL_IRP in this field. Here is a list of the constant values that could be stored in the top-level IRP field:

```
#define     FSRTL_FSP_TOP_LEVEL_IRP             (0x01)
(tdefine    FSRTL_CACHE_TOP_LEVEL_IRP           (0x02)
#define     FSRTL_MOD_WRITE_TOP_LEVEL_IRP       (0x03)
#define     FSRTL_FAST_IO_TOP_LEVEL_IRP         (0x04)
#define     FSRTL_MAX_TOP_LEVEL_IRP_FLAG         (0x04)
```

The constant value FSRTL_FSP_TOP_LEVEL_IRP is stored by an FSD in the TLS only when an IRP has been posted to be processed in the context of a worker thread and only if some other component (other than the FSD) happens to be top-level for that particular I/O request. In other words, when processing a request for deferred processing in the context of a worker thread, the FSD performs the following tests:

Was the FSD the top-level component for the original request? If so, set the IRP pointer in TLS, since the FSD will still continue to be the top-level component even while processing the request in the context of the current worker thread.

- Otherwise, set the constant FSRTL_FSP_TOP_LEVEL_IRP in the TLS for the worker thread, to indicate that some other component is actually top-level for this particular IRP.

The constant value FSRTL_CACHE_TOP_LEVEL_IRP is stored in the TLS by the FSD when a callback is received by the FSD to preacquire FSD resources for read-ahead, lazy-write and/or flush operations initiated by the Cache Manager. You have already been introduced to the Cache Manager callbacks provided by a file system driver in Chapter 7, *The NT Cache Manager II.*

When the FSD receives the callback to preacquire resources, it should set the FSRTL_CACHE_TOP_LEVEL_IRP constant flag value in the TLS. Later, when the IRP is received by the FSD, it can easily identify that the Cache Manager happens to be the top-level component for the request.

The constant value FSRTL_MOD_WRITE_TOP_LEVEL_IRP is stored in the TLS by the modified/mapped page writer threads themselves at thread creation time. This is because the modified/mapped page writer threads are dedicated worker threads that initiate write-behind requests and are therefore always top-level components for IRPs that result from their actions. The FSD can check for the existence of this value, but does not need to set the value itself.

The constant value FSRTL_FAST_IO_TOP_LEVEL_IRP is set in the TLS by the File System run-time library (FSRTL) fast I/O routines. The FSRTL routines are not typically exported in the DDK. You have to buy a separate IPS Kit license from Microsoft to get header files that define all of the FSRTL routines that Microsoft wishes to export.

Note that the FSRTL exports certain routines that your FSD can use to service the fast I/O calls to your FSD. For example, the FSRTL exports a function called FsRtlCopyRead(), to which Microsoft I/O designers recommend your fast I/O read function pointer should be initialized. This function provides the expected preamble before passing the fast I/O request directly to the NT Cache Manager and bypassing the FSD in the process. We will discuss the fast I/O path in greater detail later in the next chapter, but note for now that the FsRtlCopyRead() helper routine and others like it automatically set the FSRTL_FAST_IO_TOP_LEVEL_IRP flag in the TLS for the thread performing fast I/O. This flag indicates to the FSD that some other component (in this case, the NT Cache Manager), is top-level, since the fast path bypassed the FSD dispatch routines.

An FSD uses the following two routines to access and/or modify the contents of this field in the TLS:

- `IoSetTopLevelIrp()`

```
VOID
loSetTopLevellrp(
    IN PIRP    Irp
);
```

Resource Acquisition Constraints:

None.

Parameters:

Irp

This is either a pointer to an IRP structure or a constant value. If the FSD happens to be the top-level component for the IRP, it supplies the pointer to the IRP as an argument to this routine.

Functionality Provided:

This routine will simply set the passed-in value into the TopLevellrp field in the thread structure for the currently executing thread in whose context the routine is invoked.

- `IoGetTopLeve1Irp()`

```
PIRP
loGetTopLevellrp(
    VOID
);
```

Resource Acquisition Constraints:

None.

Parameters:

None

Return Value:

An IRP pointer or the constant value that was stored in the TLS. You can always identify whether or not this is a valid IRP pointer by checking whether the returned value, cast to an unsigned long, is less than the constant *FSRTL_MAX_TOP_LEVEL_IRP_FLAG*.

Functionality Provided:

This routine returns the contents of the TopLevellrp field in the thread structure for the currently executing thread in whose context the routine is invoked.

Code sample

Here is a code fragment from the sample FSD that illustrates how an FSD would check and/or set the top-level component field in the TLS.

```
NTSTATUS SFsdRead(
PDEVICE_OBJECT       DeviceObject,     // the logical volume device object
PIRP                    Irp)           // I/O Request Packet
{
    NTSTATUS               RC = STATUS_SUCCESS;
    PtrSFsdlrpContext      PtrlrpContext = NULL;
    BOOLEAN                AreWeTopLevel = FALSE;

    FsRtlEnterFileSystemO;
    ASSERT(DeviceObject);
    ASSERT(Irp);

    // set the top level context
    AreWeTopLevel = SFsdlsIrpTopLevel(Irp);

    try {

        // get an IRP context structure and issue the request
        PtrlrpContext = SFsdAllocatelrpContext(Irp, DeviceObject);
        ASSERT(PtrlrpContext);

        RC = SFsdCommonRead(PtrlrpContext, Irp);

    } except (SFsdExceptionFilter(PtrlrpContext, GetExceptionInformation()))
    {

        RC = SFsdExceptionHandler(PtrlrpContext, Irp);

        SFSdLogEvent(SFSD_ERROR_INTERNAL_ERROR, RC);
    }

    if (AreWeTopLevel) {
        loSetTopLevellrp(NULL) ;
    }

    FsRtlExitFileSystemO;

    return(RC) ;
}

BOOLEAN SFsdlsIrpTopLevel(
PIRP           Irp)           // the IRP sent to our dispatch routine
{
    BOOLEAN            ReturnCode = FALSE;

    if (IoGetTopLevelIrp() == NULL) {
        //OK, so we can set ourselves to become the "top level" component.
        loSetTopLevellrp(Irp) ;
        ReturnCode = TRUE;
    }

    return(ReturnCode);
}
```

Notes

The code fragment illustrates the processing performed in the FSD read dispatch routine entry point before the FSD invokes the SFsdCommonRead ( ) routine, shown in the previous chapter. Here, you can see that the FSD invokes a routine to determine whether the FSD can be the top-level component for the current IRP. The invoked routine is called SFsdIsIrpTopLevel (). This routine simply checks the current value of the TopLevelIrp field in the TLS to determine whether it has already been set. If the field contains a nonzero value, the FSD assumes that some other component is top-level for the current request; otherwise, the FSD sets the IRP pointer value in the TLS to indicate that the FSD itself is top-level for the current request.

Although this processing is adequate for the sample FSD (and for that matter, the FAT and CDFS file system implementations in NT also do pretty much the same thing), NTFS and other more sophisticated file systems may manipulate the TLS storage area differently. Fundamentally though, the above concepts can be used to determine the top-level component for any I/O request dispatched to the FSD.

How information about the top-level component is used

This concept of identifying the top-level component for each request is used as follows:

In determining the flow of execution when processing a request. Consider a file stream mapped by some thread that has the file stream opened for nonbuffered I/O. Write operations performed by this thread will eventually be dispatched to the FSD write routine via the NT VMM in the form of paging I/O write operations. If such a modification performed by the user extends beyond current valid data length for the file, most FSD implementations will attempt to zero the range between the current valid data length and the start of the new write operation.

Figure 10-2 illustrates the byte range being modified in such a situation.

The reason an FSD might wish to zero the "hole" represented by the byte range between the current valid data length and the starting offset for the current request is to avoid returning old data that might be present on disk for the sectors backing this range.

To ensure consistency between cached and buffered data for a file stream, the FSD should use CcZeroData ( ) to zero the resulting hole. Upon receiving the request to zero data, the Cache Manager checks for a write-through file object, and directly flushes data to disk in such a scenario. This flush is performed synchronously by the Cache Manager.

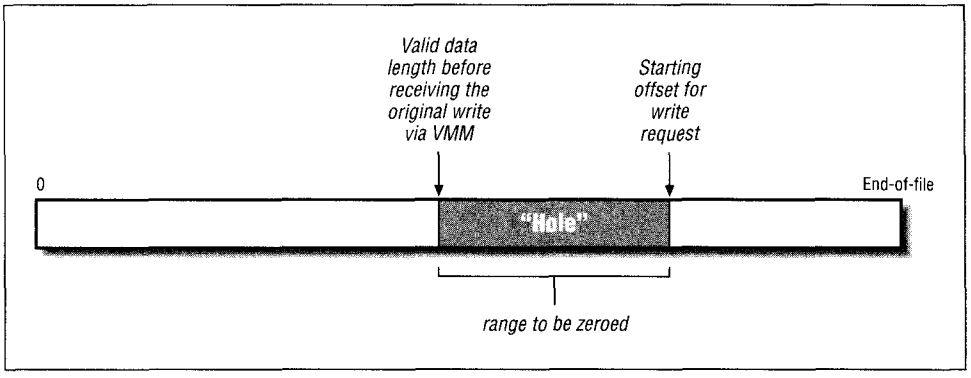

Figure 10-2. Range to be zeroedfor write extending beyond valid data length

The flush operation is also dispatched to the FSD write routine as a synchronous, paging I/O write operation. Now, the FSD must distinguish this synchronous flush for the original write-through request from the original request itself. The only reliable method to do this is to check the top-level component for the new request. Since the flush is a recursive call in the context of the thread performing the original write-through operation, the FSD can identify that it cannot be top-level with respect to the flush IRP. This identification allows the FSD to perform the right processing in response to the flush, and the FSD will now not attempt to change the valid data length again (it was already done when the original write was received), nor will the FSD try to recursively invoke the Cache Manager to zero any holes (since that would lead to a deadlock, and/or infinite recursive loop condition).

There are other file systems (e.g., distributed file systems) that might be required to perform some special processing when the FSD is top-level for a request, and they could safely avoid such processing in the case of recursive requests. These FSD implementations also find it useful to identify the top-level component for an IRP, and they modify their processing appropriately.

Last but not least, an FSD must always be careful about dealing with asynchronous I/O requests if some other component is top-level for a request. As discussed below, the top-level component for the IRP ensures that resource acquisition hierarchies across the FSD, VMM, and Cache Manager are maintained. Therefore, when an FSD receives an I/O request for which it is not top-level, and when it is a recursive I/O request or an I/O request that requires synchronous processing, the FSD should never post the request to be handled in the context of some other worker thread, since this could lead to a deadlock situation. This issue is addressed once again in the discussion on asynchronous I/O processing below.

In performing synchronization. Earlier in this book, we discussed the resource acquisition hierarchy that must be maintained by FSD implementations, the NT Cache Manager, and the NT VMM in order to avoid deadlocks. The hierarchy follows:

- File system driver resources must always be acquired first.
- The NT Cache Manager resources are acquired next, if required.
- The NT VMM resources are acquired last.

To help maintain this hierarchy, the Cache Manager, as well as the VMM, are careful to preacquire FSD resources in situations when they are top-level for an I/O operation. This ensures that the resource acquisition hierarchy is always maintained. Earlier in Chapter 7, as well as in Chapter 8, *The NT Cache Manager III,* we discussed the four Cache Manager callbacks that an FSD should be cognizant of. In the next chapter, we'll see a sample implementation of the callback routines that the FSD is expected to provide.

For now, note that since the top-level component for any IRP operation is careful to preacquire FSD resources before sending the request down to the FSD, the file system implementation must always be careful of which resources it then tries to acquire recursively (remember that ERESOURCE type synchronization objects, as well as normal KMUTEX objects, can be recursively obtained) and of the manner in which it then tries to acquire such resources. If your FSD uses other resources that are not recursively acquirable (e.g., FAST_MUTEX structures), the FSD must be careful not to attempt such acquisition when it is not top-level for the IRP (since the top-level component must have presumably preacquired the resource).

The bottom line is that the FSD should be aware that the top-level component typically has a whole bunch of resources acquired before sending the request to the FSD dispatch routine entry point, and it is therefore the FSD's responsibility to proceed carefully.

In ***file size modifications.*** There are two rules you must always follow with respect to file size modifications:

- The valid data length for a file stream can only be extended by the top-level component for any I/O request, and only directly in response to user modifications of file stream data.
- The end-of-file value cannot be extended or changed by paging I/O operations. Chapter 6, *The NT Cache Manager I,* explains in greater detail the behavior expected from an FSD in this regard.

The first rule is straightforward if you understand the concept of a *top-level writer,* which can be defined as the component that is performing a write operation

ılt of a user thread modification of file stream data. Now, it o restate the above rule as only the top-level writer component lid data length for a file stream.

write request extending the valid data length is received by the em is the top-level writer and therefore extends the valid data /O write requests, the FSRTL package is typically the top-level •equest is not a recursive request, and therefore the valid data d. If a user maps in a file, and modifies that mapped view for ; the valid data length in the process, the FSD once again is the nd extends the value appropriately when the modifications are >D via paging I/O.

hat Cache Manager lazy-write operations can never cause the to be extended, because lazy-write operations are never directly odifications. However, VMM-initiated modified page write opera- cause the valid data length to be extended by the FSD, just as it ile for an FSD to receive other paging I/O write operations beyond valid data length. The reason for this is that paging I/O :oupled from normal user thread synchronization.

ıall problem in determining whether or not it should extend the for a paging I/O write operation. As I mentioned earlier, if the is due to a user modification of a mapped view of a file, the FSD g I/O write request and must extend the valid data length. ɔaging I/O write beyond the current valid data length value is ıg by the Cache Manager, the FSD does not have to extend the since this will be done by the Cache Manager itself at some Unfortunately, it is difficult to distinguish between these two fore, the native Windows NT FSD implementations, as well as FSDs, typically use the following workaround. When the FSD :k from the lazy-writer thread requesting resources be acquired ɛrations via the `AcquireForLazyWrite()` callback function, ıe lazy-writer thread ID (using the `PsGetCurrentThread()` call) in the FCB for the file stream. Later, when a paging I/O eceived, the FSD checks the current thread ID with the stored ndicates that the write is due to lazy-writing performed by the

• in this book when discussing the NT Cache Manager, it is extremely important for 3ache Manager informed when any file size value changes. Therefore, if the FSD lid data length should be extended, it must inform the Cache Manager, using the routine (invoked by the FSD write dispatch routine processing the user write re- transferring modified data from the user-supplied buffer either to the system cache or directly to disk.

Cache Manager, and Subsequently, wher Manager via anothe ID value is zeroed.

In reporting unrecou transfer cannot be cc level component fo support routine (exp the user. For examp flushing modified p `DATA` as the error Manager lazy-writer when trying to lazily

Achieving I/O-

In Chapter 9, *Writin* IRP processing den forwarded to the C `Write()` function c

As mentioned, the F I/O to the same file file data, given the stream, if it is mappe There are two kinds of I/O operation:

- If the caller atte should try to av been cached in ters that the NT stream mapped FSD must, theref different section image section ob

* As discussed in Chapte section objects leads to c mapped in both as an ex being executed, returning ecutable to crash anyway. data has dubious benefits

Cache Manager, and the FSD knows that it must not modify the valid data length. Subsequently, when preacquired FSD resources are released by the Cache Manager via another callback (ReleaseFromLazyWrite ()), the stored thread ID value is zeroed.

In reporting unrecoverable hard error conditions. It is possible that a data transfer cannot be completed due to some unrecoverable error condition. The top-level component for an IRP uses the loRaiseInformationalHardError () support routine (explained in the DDK) to report an appropriate error message to the user. For example, the modified page writer component will report failures in flushing modified pages to disk by specifying STATUS_LOST_WRITEBEHIND_DATA as the error code when invoking this routine. Similarly, the NT Cache Manager lazy-writer thread will use the same error code if it received an error when trying to lazily-write modified cached data for a file stream.

Achieving I/O-Related Goals

In Chapter 9, *Writing a File System Driver /,* the code samples for read and write IRP processing demonstrated how I/O requests for cached data transfer are forwarded to the Cache Manager via the CcCopyRead ( ) and the CcCopy-Write {) function calls.

As mentioned, the FSD must ensure consistency between cached and noncached I/O to the same file stream. The FSD must also maintain a consistent view of the file data, given the fact that two separate sections can possibly exist for a file stream, if it is mapped both as an executable and as a regular data section object. There are two kinds of consistency problems that arise depending upon the type of I/O operation:

- If the caller attempts to read file data requesting nonbuffered I/O, the FSD should try to avoid returning stale data to the user if the file stream has also been cached in system memory. Also, you probably recall from earlier chapters that the NT VMM maintains separate section objects for the same file stream mapped in both as an executable and as a data section object. Your FSD must, therefore, also attempt to maintain consistency between these two different section objects; if a thread modifies the data for the file stream, the image section object should also get the most recent modifications.*

* As discussed in Chapter 5, *The NT Virtual Memory Manager,* this policy of maintaining two different section objects leads to considerable headaches for FSD designers. If the same file stream is indeed mapped in both as an executable and as a data section object, and is modified while the executable is being executed, returning the latest modifications when servicing page faults will probably cause the executable to crash anyway. Therefore, you could legitimately argue that providing a consistent view of the data has dubious benefits in this case.

When the FSD receives a noncached read request on a file stream that is currently being cached by the Cache Manager, most FSD implementations simply perform a flush operation on the accessed byte range using the CcFlushCache() call. However, the invocation of CcFlushCache() is typically done by NT FSD implementations before acquiring any resources in the context of the thread requesting nonbuffered read access. The implication here is that no guarantees are made by the FSD in this case to always return the latest data—it is still theoretically possible for some other thread to quickly modify the accessed byte range in the system cache between completion of the flush operation and the instant when the FSD acquires FCB resources shared to satisfy the noncached read.

If your FSD needs to guarantee that the most recently modified data is always returned, it can do so either by preacquiring resources exclusively before initiating the flush operation or by purging data from the system cache, as in the noncached write access described below.

In the code sample presented in the previous chapter for the read dispatch entry point, you would have to add the following code to achieve the flush:

```
// The test below flushes the data cached in system memory if the
// current request mandates noncached access (file stream must be
// cached) and
// (a) the current request is not paging I/O, which indicates it is not
//     a recursive I/O operation OR originating in the Cache Manager
// (b) OR the current request is paging I/O BUT it did not originate
//     via the Cache Manager (or is a recursive I/O operation) and we
//     do have an image section that has been initialized.

// Note that the MmlsRecursiveloFault() macro below is defined in the
// IPS Kit as follows:
// #define MmlsRecursiveloFault()                                    \
//       ((PsGetCurrentThread()->DisablePageFaultClustering)  |     \
//          (PsGetCurrentThread()->ForwardClusterOnly))
//
#define    SFSD_REQ_NOT_VIA_CACHE_MGR(ptr)                          \
    (!MmlsRecursiveloFault() && ((ptr)->ImageSectionObject != NULL))

if (NonBufferedlo &&
     (PtrReqdFCB->SectionObject.DataSectionObject != NULL)) {
    if    (!Paginglo ||
          (SFSD_REQ_NOT_VIA_CACHE_MGR(&(PtrReqdFCB->SectionObject)))) {
          CcFlushCache(&(PtrReqdFCB->SectionObject),
                                       &ByteOffset, ReadLength,
                                       &(PtrIrp->IoStatus)) ;
        // If the flush failed, return error to the caller
        if (!NT_SUCCESS(RC = PtrIrp->IoStatus.Status)) {
            try_return(RC);
        }
    }
}
```

- If the caller requests a noncached write operation on a file stream that is also currently being cached, the FSD must ensure that the cached data is consistent with the new (to-be-written) on-disk information.

  The FSD has to avoid the situation where it writes new information to disk, and subsequently, the older information, when flushed by the lazy-writer or modified page writer threads overwrites the latest data.

  To prevent such problems from occurring, I would suggest that your FSD implementation flush the currently cached information for the affected byte range and also purge it from the system cache, thereby forcing the Cache Manager to reload the latest information from secondary storage. This is also the approach followed by most existing Windows NT file system drivers.

  One point that you must be aware of is that the NT VMM will fail a purge request if any process has the file stream mapped in its virtual address space. This will result in stale data being returned to the caller, but unfortunately, given the current design of the NT VMM, all FSD implementations have to learn to live with this restriction.

  Finally, be careful about how you acquire file control block resources when performing such a purge operation. The Cache Manager requires that the FCB resources be acquired exclusively when requesting a purge. However, if you acquire FCB resources exclusively, perform the purge, and then release the FCB resources, you still run the risk of having another thread sneak in and perform another cached write on the file stream data, thereby invalidating all you just tried to achieve via the purge.

  The following code fragment demonstrates how the cache flush and subsequent purge can be achieved:

```
if (NonBufferedIo && !PagingIo &&
     (PtrReqdFCB->SectionObject.DataSectionObject !=NULL)) {
    // Flush and then attempt to purge the cache
    CcFlushCache(&(PtrReqdFCB->SectionObject)       ,
                              &ByteOffset, WriteLength,
                              &(PtrIrp->IoStatus));
    // If the flush failed, return error to the caller
    if ( !NT_SUCCESS(RC = PtrIrp->IoStatus.Status) ) {
        try_return(RC) ;
    }

    // Attempt the purge and ignore the return code
    CcPurgeCacheSection(&(PtrReqdFCB->SectionObject),
                                 (WritingAtEndOfFile ?
                                 &(PtrReqdFCB->
                                      CommonFCBHeader.FileSize) :
                                 &(ByteOffset) ),
                                 WriteLength, FALSE);
    // We are finished with our flushing and purging
}
```

Resource acquisition hierarchies across the NT Cache Manager, the VMM, and the FSD are maintained by the presence of FSD callbacks, which are invoked by the Cache Manager and the NT VMM, to preacquire FSD resources before they initiate an I/O operation. Later in the next chapter, you will see sample code for such a callback operation. Similarly, when the I/O Manager uses the fast I/O method to bypass the FSD and directly request data from the NT Cache Manager, either the FSRTL routine or the FSD fast I/O routine must ensure that the correct file system resources are acquired before passing the request on to the Cache Manager.

The Cache Manager and the VMM are also extremely careful not to invoke routines exported by the respective modules in any manner that could lead to deadlock.

Asynchronous I/O Processing

FSD dispatch routines can be invoked either for synchronous or for asynchronous processing. Synchronous processing implies that the I/O request can be processed and completed in the context of the requesting thread, even if the requesting thread must be made to block, awaiting completion of processing of the request. Asynchronous processing, on the other hand, requires that the request either be completed in the context of the thread that invoked the FSD dispatch routine entry point, or if processing requires blocking of the original thread, be processed asynchronously in the context of some worker thread.

Two situations can result in a thread being blocked when processing an I/O request:

- When a thread tries to acquire some synchronization resource (e.g., a mutex or a read/write lock)

  The thread requesting the resource may be put into the blocked state, awaiting release of the resource by another thread that already has this resource acquired.

- When transferring data to/from secondary storage

  Most lower-level disk drivers queue I/O requests for subsequent, asynchronous processing if they are actively processing other I/O requests when the new request is received.

Although you can design an FSD that always performs synchronous processing, this can lead to system stability problems, especially in the case when your FSD tries to synchronously service asynchronous paging I/O requests from the VMM.

WARNING It is important that your FSD honor requests for asynchronous processing. As was explained in Chapter 5, the NT VMM modified page writer and mapped page writer threads aggressively try to write out modified pages when the system is running low on available physical memory. To achieve their objectives of flushing out pages quickly, each of these routines sends asynchronous paging I/O requests to the different FSDs in the system. If your FSD attempts to process such I/O requests synchronously, you are essentially thwarting the memory manager's attempts to respond quickly to the system's requirements for free pages. Not only do you prevent additional write requests from being queued to your FSD for processing, you also prevent write requests from being queued to any other FSD in the system. Worse, if your FSD were to block for a long time, it is almost certain that the VMM would eventually bugcheck the system.

Note that you -will never block while attempting to acquire FSD resources for asynchronous mapped page -writer requests, since these will have been preacquired by the NT VMM via a callback to the FSD before issuing the write request.

To provide support for asynchronous processing, your FSD must perform the following operations:

- Determine whether the caller has requested synchronous or asynchronous processing.

  Your FSD can use the loIsOperationSynchronous () routine to find out whether an operation should be performed synchronously. This routine is defined as follows:

```
BOOLEAN
loIsOperationSynchronous(
    IN PIRP         Irp
);
```

  Resource Acquisition Constraints:

  None.

  Parameters:

  Irp

  Pointer to the I/O Request Packet sent by the I/O Manager to the FSD.

  Return Value:

  TRUE if the current request should be processed synchronously; FALSE if the request is an asynchronous I/O request.

Functionality Provided:

The NT I/O Manager checks the following conditions to determine whether the operation is synchronous. If this is not an asynchronous paging I/O operation,* and one of the following is true, the operation is synchronous.

— The file object used in the IRP specifies that the file was opened for synchronous access.

— The NT I/O Manager API is an inherently synchronous API (e.g., the "create/open" operation is inherently synchronous).

— The IRP indicates that this is a synchronous paging I/O operation.

If the above checks evaluate to TRUE, the I/O Manager returns TRUE to indicate that the operation should be performed synchronously; otherwise, the I/O Manager returns FALSE, indicating that this I/O request should not be processed synchronously.

- If the caller is not prepared to block, always attempt to acquire resources in a nonblocking manner only. If resources cannot be acquired without blocking, post the request to a queue to be picked up later and processed in the context of a worker thread routine.
- When invoking the Cache Manager for accesses to buffered data, always inform the Cache Manager of whether the caller is prepared to block. Often, the Cache Manager may not be able to satisfy the request immediately and for nonblocking callers will return a FALSE value from the function call, indicating that the request processing should be deferred and retried later.

Most Windows NT file system drivers do not create dedicated worker threads to process asynchronous requests. Rather, the FSDs use the services of a pool of global system worker threads. The Windows NT Executive provides a set of supporting structure definitions and utilities that allow the FSD to initialize a work queue item for deferred processing and post the request to an appropriate queue supplying a callback function that can subsequently be invoked in the context of the worker thread.

Earlier in this chapter, we saw some sample code for a typical read dispatch routine entry point in the FSD. The SFsdRead() routine allocates an IrpContext structure. This IrpContext structure serves as an encapsulation of the current I/O request, and turns out to be useful when preparing the IRP for deferred processing and the subsequent posting of the IRP. Here is a sample IrpContext structure as defined by the FSD:

* Even if the file object was opened specifying synchronous I/O operations, the modified/mapped page writer will try to write data out asynchronously. Therefore, the last clause in the list of checks performed above is important.

```
typedef struct _SFsdIrpContext {
    SFsdIdentifier                  NodeIdentifier;
    uint32                          IrpContextFlags;
    // copied from the IRP
    uint8                           MajorFunction;
    // copied from the IRP
    uint8                           MinorFunction;
    // to queue this IRP for asynchronous processing
    WORK_QUEUE_ITEM                 WorkQueueItem;
    // the IRP for which this context structure was created
    PIRP                            Irp;
    // the target of the request (obtained from the IRP)
    PDEVICE_OBJECT                  TargetDeviceObject;
    // if an exception occurs, we will store the code here
    NTSTATUS                        SavedExceptionCode;
} SFsdIrpContext, *PtrSFsdIrpContext;
#define         SFSD_IRP_CONTEXT_CAN_BLOCK          (0x00000001)
#define         SFSD_IRP_CONTEXT_WRITE_THROUGH      (0x00000002)
#define         SFSD_IRP_CONTEXT_EXCEPTION          (0x00000004)
#define         SFSD_IRP_CONTEXT_DEFERRED_WRITE     (0x00000008)
#define         SFSD_IRP_CONTEXT_ASYNC_PROCESSING   (0x00000010)
#define         SFSD_IRP_CONTEXT_NOT_TOP_LEVEL      (0x00000020)
#define         SFSD_IRP_CONTEXT_NOT_FROM_ZONE      (0x80000000)
```

The IrpContext structure is used by the sample FSD implementation to encapsulate the current I/O request. Your FSD can utilize a similar structure, if it proves to be convenient. Notice that the IrpContext structure has a flag, SFSD_IRP_CONTEXT_CAN_BLOCK, that indicates to the FSD if the current caller of the dispatch routine can block during I/O processing. This flag is set when the IrpContext structure is allocated, to indicate whether synchronous processing can be performed. Furthermore, the WorkQueueItem field in the IrpContext structure is used by the FSD to post the request for deferred processing in the context of a system worker thread.

The following code fragment demonstrates the implementation of a simple (though typical) IrpContext allocation routine:

```
PtrSFsdIrpContext SFsdAllocateIrpContext(
PIRP                Irp,
PDEVICE_OBJECT      PtrTargetDeviceObject)
{
    PtrSFsdIrpContext           PtrIrpContext = NULL;
    BOOLEAN                     AllocatedFromZone = TRUE;
    KIRQL                       CurrentIrql;
    PIO_STACK_LOCATION          PtrIoStackLocation = NULL;

    // first, try to allocate out of the zone
    KeAcquireSpinLock(&(SFsdGlobalData.ZoneAllocationSpinLock),
                                ScCurrentIrql);
    if (!ExIsFullZone(&(SFsdGlobalData.IrpContextZoneHeader))) {
        // we have enough memory
        PtrIrpContext =
```

```
            (PtrSFsdlrpContext)ExAllocateFromZone
                        (&(SFsdGlobalData.IrpContextZoneHeader));

        // release the spin lock
        KeReleaseSpinLock(&(SFsdGlobalData.ZoneAllocationSpinLock),
                                        Currentlrql) ;
    } else {
        // release the spin lock
        KeReleaseSpinLock(&(SFsdGlobalData.ZoneAllocationSpinLock),
                                        Currentlrql);

        // if we failed to obtain from the zone, get it directly from the
        // VMM
        PtrlrpContext = (PtrSFsdlrpContext) ExAllocatePool (NonPagedPool,
                                    SFsdQuadAlign(sizeof(SFsdlrpContext)));
      AllocatedFromZone = FALSE;
    }

    // if we could not obtain the required memory, bugcheck.
    // Do NOT do this in your commercial driver, instead handle
    // the error gracefully (e.g., by returning STATUS_INSUFFICIENT_
    // RESOURCES to the caller and also logging the error condition) .
    if (!PtrlrpContext) {
        SFsdPanic ( STATUS_INSUFFICIENT_RESOURCES ,
                SFsdQuadAlign(sizeof(SFsdlrpContext)), 0);
    }

    // zero-out the allocated memory block
    RtlZeroMemory(PtrlrpContext, SFsdQuadAlign(sizeof(SFsdlrpContext)));

    // set up some fields . . .
    PtrIrpContext->NodeIdentifier.NodeType    = SFSD_NODE_TYPE_IRP_CONTEXT;
    PtrIrpContext->NodeIdentifier.NodeSize    =
                        SFsdQuadAlign(sizeof(SFsdlrpContext));


    PtrIrpContext->Irp = Irp;
    PtrIrpContext->TargetDeviceObject = PtrTargetDeviceObject;

    // copy over some fields from the IRP and set appropriate flag values
    if (Irp) {
        PtrloStackLocation = loGetCurrentlrpStackLocation(Irp);
        ASSERT(PtrloStackLocation);

        PtrIrpContext->MajorFunction = PtrloStackLocation->MajorFunction;
        PtrIrpContext->MinorFunction = PtrIoStackLocation->MinorFunction;

        // Often, an FSD cannot honor a request for asynchronous processing
        // of certain critical requests. For example, a "close" request on
        // a file object can typically never be deferred. Therefore, do not
        // be surprised if sometimes your FSD (just like all other FSD
        // implementations on the Windows NT system) has to override the
        // flag below.
        if (IoIsOperationSynchronous(Irp)) {
```

```
            SFsdSetFlag ( PtrIrpContext->IrpContextFlags ,
                    SFSD_IRP_CONTEXT_CAN_BLOCK)  ;
        }
    }

    if (!AllocatedFromZone) {
        SFsdSetFlag (PtrIrpContext->IrpContextFlags,
                        SFSD_IRP_CONTEXT_NOT_FROM_ZONE ) ;
    }

    // Are we top-level? This information is used by the dispatching code
    // later (and also by the FSD dispatch routine)
    if (loGetTopLevellrpO != Irp) {
        // We are not top-level. Note this fact in the context structure
        SFsdSetFlag (PtrIrpContext->IrpContextFlags,
                        SFSD_I RP_CONTEXT_NOT_TOP_LEVEL ) ;
    }

    return(PtrlrpContext);
}
```

The IrpContext allocation routine determines whether the FSD can be considered top-level for the original invocation of the FSD dispatch routine and remembers this fact by setting an appropriate flag value.

A work queue item must be initialized by the FSD to post a request for deferred processing. This initialization can be performed by using the Exlnitialize-Workltem( ) Executive support function, which accepts the following arguments:

- A pointer to the work item to be initiated

  You can pass in a pointer to the WorkQueueltem field in the IrpContext structure.

- A pointer to the callback function

  Note that the sample FSD implementation uses a common callback function called SFsdAsyncDispatch( ) , shown later.

- A context for which you should simply pass in the pointer to the IRP context structure itself

The following expanded code fragment from the SFsdCommonRead ( ) function, originally presented in the previous chapter, illustrates how the FSD can post an item for subsequent (deferred) processing:

```
NTSTATUS    SFsdCommonRead (
PtrSFsdlrpContext            PtrlrpContext,
PIRP                         Ptrlrp)
{
    // Declarations go here …

    try {
```

```
        II Chapter 9 has more information on processing performed here.

        ....

        // Acquire the appropriate FCB resource shared.
        if (Paginglo) {
            // Try to acquire the FCB PagingloResource shared
            if ( !ExAcquireResourceSharedLite(&(PtrReqdFCB->
                                                PagingloResource)   ,
                                                CanWait) ) {
                Completelrp = FALSE;

                // This is one instance where we have decided to defer
                // processing . . .
                PostRequest = TRUE;
                try_return(RC = STATUS_PENDING) ;
            }
            // Remember the resource that was acquired.
         PtrResourceAcquired = &(PtrReqdFCB->PagingIoResource) ;
        } else {
            // Try to acquire the FCB MainResource shared.
            if ( !ExAcquireResourceSharedLite(&(PtrReqdFCB->MainResource),
                          CanWait) ) {
                Completelrp = FALSE;

                // Defer processing . . .
                PostRequest = TRUE;
                try_return(RC = STATUS_PENDING) ;
            }
            // Remember the resource that was acquired.
         PtrResourceAcquired = &(PtrReqdFCB->MainResource) ;
        }

        // There are other situations that could require us to post
        // the request.
        ...

        try_exit:   NOTHING;

    } finally {
        // Post IRP if required.
        if (PostRequest) {

            // Release any resources acquired here ...
            if (PtrResourceAcquired) {
                SFSdReleaseResource(PtrResourceAcquired);
            }

            // Implement a routine that will queue up the request to be
            // executed later (asynchronously) in the context of a system
            // worker thread.

            // Lock the caller's buffer here. Then invoke a common routine
            // to perform the post operation.
```

```
            if (!(PtrIoStackLocation->MinorFunction & IRP_MN_MDL)) {
                RC = SFsdLockCallersBuffer(Ptrlrp, TRUE, ReadLength);
                ASSERT(NT_SUCCESS(RC));
            }

            // Perform the post operation, which will mark the IRP pending
            // and will return STATUS_PENDING back to us.
            RC = SFsdPostRequest(PtrlrpContext, Ptrlrp);

        } else if (Completelrp && !(RC == STATUS_PENDING)) {

            // More information in Chapter 9 ...

        } // can we complete the IRP?
    } // end of "finally" processing.

    return(RC);
}
```

In this code fragment, you can see that the request is posted for deferred processing if appropriate resources cannot be acquired without blocking and if the caller had specified no blocking. Before sending the request to be queued, the FSD is careful to create a memory descriptor list (MDL), describing the caller-supplied buffer and also to lock the pages comprising this MDL. Then, the FSD invokes the SFsdPostRequest() routine to post the request. This routine is shown below:

```
NTSTATUS SFsdPostRequest(
PtrSFsdlrpContext            PtrlrpContext,
PIRP                         Ptrlrp)
{
    NTSTATUS            RC = STATUS_PENDING;

    // mark the IRP pending; a flag SL_PENDING_RETURNED is set in the
    // current stack location.
    loMarklrpPending(Ptrlrp);

    // queue up the request.
    ExInitializeWorkItem(&(PtrIrpContext->WorkQueueItem),
                         SFsdCommonDi spatch ,
                         PtrlrpContext);

    ExQueueWorkItem(&(PtrIrpContext->WorkQueueItem), CriticalWorkQueue);

    // return status pending.
    return(RC);
}
```

The SFsdPostRequest() function shown here is very simple; it marks the I/O Request Packet pending, queues the request for processing by a system worker thread, and returns the STATUS_PENDING code to the caller. Each of these steps

is important to successfully process the request asynchronously. Here's what each of the steps in the SFsdPostRequest ( ) routine achieves:

- The I/O Manager checks for the presence of the SL_PENDING_RETURNED flag when the IRP is eventually completed.

  This flag is an indicator to the I/O Manager that your driver must have returned STATUS_PENDING to the caller of a dispatch routine, and that the IRP could have been processed asynchronously.

  If this flag is set in the current stack location (the stack location of the driver that invokes loCompleteRequest ( )), the I/O Manager remembers not to take the shortcut method, described in Chapter 4, *The NT I/O Manager,* of performing IRP completion postprocessing directly in the context of the thread that originated the I/O request; instead, the I/O Manager queues a kernel asynchronous procedure call to the originating thread and performs the requisite postprocessing when the APC is delivered.

- Invoking ExQueueWorkItemO queues the request in a global system queue for asynchronous handling by an available worker thread.

- Returning STATUS_PENDING informs the caller that your driver will process the request asynchronously.

  When the caller receives this return status, it knows that the request will be completed asynchronously and that the caller can wait for the request completion immediately, or after performing some concurrent processing. It is quite possible that the request can be completed even before your STATUS_PENDING gets returned to the caller if the thread that invoked your FSD dispatch routine is preempted. However, that is not a race condition that you have to worry about. The pseudocode below demonstrates how FSD dispatch routines are invoked:

```
// The FSD dispatch routine is invoked as shown here:
RC = loCallDriver(PtrDeviceObject, Ptrlrp);
if (RC != STATUS_PENDING) {
    // Check the return status and react appropriately, since the
    // request has been processed synchronously.
    ...
} else {
    //We received a STATUS_PENDING. Optionally, perform some
processing
    // and then wait for the request completion.
    ...
    KeWaitForSingleObject(...);
    // Now, the wait was completed; therefore loCompleteRequest()
    // must have been invoked on the IRP.
    ...
}
```

In this pseudocode, the caller simply waits for an event object to be signaled when STATUS_PENDING is returned. The worst that could happen if the request gets completed before the caller begins the wait is that the event object may have already been signaled (when loCompleteRequest ( ) was processed), and the caller will find the event object in the signaled state in the KeWaitForSingleObject ( ) function call; this will result in no wait actually being performed.

Once the request has been posted by the FSD, a system worker thread picks up the request from the appropriate queue. There is a fixed pool of system worker threads, and they exist for the sole purpose of performing work for the different Windows NT Executive components. When your FSD initializes the work item for subsequent queuing, it specifies a function that the worker thread must execute. The sample FSD supplies a pointer to the SFsdCommonDispatchO routine. Also note that the sample FSD uses a pointer to the IrpContext structure as the context to be supplied to the callback routine. This is convenient, since the IrpContext structure contains a pointer to the original IRP, and also additional information, such as whether the FSD was top-level for the IRP request to be processed. The following code fragment demonstrates a typical callback dispatch routine that your FSD could implement:

```
void  SFsdCommonDispatch(
void                          *Context)     // actually a SFsdlrpContext
                                            // structure
{
    NTSTATUS                            RC = STATUS_SUCCESS ;
    PtrSFsdlrpContext                    PtrlrpContext = NULL;
    PIRP                                 Ptrlrp = NULL;

    // The context must be a pointer to an IrpContext structure.
    PtrlrpContext = (PtrSFsdlrpContext) Context;
    ASSERT(PtrlrpContext);

    // Assert that the context is legitimate.
    if ( (PtrIrpContext->NodeIdentifier.NodeType !=
            SFSD_NODE_TYPE_IRP_CONTEXT)
          || (PtrIrpContext->NodeIdentifier.NodeSize !=
                SFsdQuadAlign(sizeof(SFsdlrpContext)))) {
        // This does not look good!
        SFsdPanic(SFSD_ERROR_INTERNAL_ERROR,
                PtrIrpContext->NodeIdentifier.NodeType,
                PtrIrpContext->NodeIdentifier.NodeSize) ;
    }

    // Get a pointer to the IRP structure.
    Ptrlrp = PtrIrpContext->Irp;
    ASSERT(Ptrlrp);

    // Now, check if the FSD was top level when the IRP was originally
```

```
II invoked and set the thread context (for the worker thread)
// appropriately.
if (PtrIrpContext->IrpContextFlags & SFSD_IRP_CONTEXT_NOT_TOP_LEVEL) {
    // The FSD is not top-level for the original request.
    // Set a constant value in the TLS to reflect this fact.
    IoSetTopLevelIrp( (PIRP) FSRTL_FSP_TOP_LEVEL_IRP) ;
}

// Since the FSD routine will now be invoked in the context of this
// worker thread, we should inform the FSD that it is perfectly OK to
// block in the context of this thread.
SFsdSetFlag ( Ptr IrpContext->IrpContextFlags ,
                SFSD_IRP_CONTEXT_CAN_BLOCK)    ;

FsRtlEnterFileSystem( ) ;

try {

    // Preprocessing has been completed; check the Major Function code
    // value either in the IrpContext (copied from the IRP) , or
    // directly from the IRP itself (we will need a pointer to the
    // stack location to do that) .
    // Then, switch, based on the value on the Major Function code.
    switch (PtrIrpContext->MajorFunction) {
    case IRP_MJ_CREATE :
        // Invoke the common create routine.
        (void) SFsdCommonCreate( PtrIrpContext, Ptrlrp) ;
        break;
    case IRP_MJ_READ:
        // Invoke the common read routine.
        (void) SFsdCommonRead( PtrIrpContext, Ptrlrp) ;
        break;
    // Continue with the remaining possible dispatch routines.
    default :
        // This is the case where we have an invalid major function.
        PtrIrp->IoStatus. Status = STATUS_INVALID_DEVICE_REQUEST;
        PtrIrp->IoStatus. Information = 0;

        loCompleteRequest (Ptrlrp, IO_NO_INCREMENT) ;
        break ;
    }
} except (SFsdExceptionFilter (PtrlrpContext,
            GetExceptionInformation() )) {

    RC = SFsdExceptionHandler (PtrlrpContext, Ptrlrp);

    SFsdLogEvent ( SFSD_ERROR_INTERNAL_ERROR, RC) ;
}

// Enable preemption
FsRtlExitFileSystemO;
```

```
    // Ensure that the top-level field is cleared.
    loSetTopLevelIrp(NULL);

    return;
}
```

The callback routine shown here performs some simple preprocessing before forwarding the request to the appropriate FSD dispatch routine, including indicating to the FSD (via a flag in the IrpContext structure) that it can now block in the context of the worker thread. Furthermore, if the FSD was not top-level when the IRP was originally dispatched to the driver, the worker thread callback routine indicates this by setting the FSRTL_FSP_TOP_LEVEL_IRP flag in the TLS for the system worker thread.

There is one additional, and extremely important, point you must be aware of when determining whether to post a request for asynchronous processing. Synchronous I/O requests for which the FSD is not top-level and recursive I/O requests should typically never be posted by the FSD, since attempting to acquire FSD resources when processing the request in a worker thread context could lead to a system deadlock (because resources would have been preacquired in the context of the original thread that initiated the request). If you do decide to handle such requests asynchronously, your FSD must be capable of some pretty sophisticated processing in the dispatch routines (e.g., SFsdCommonRead()) to determine if it is allowed to acquire resources or to skip such acquisition. Modified/mapped page writer requests can therefore never be posted, although they are asynchronous requests. However, you can be assured that your FSD will never block on file system resources for such requests, since the VMM preacquires these resources.

Here are a few specific dispatch entry points for which the FSD should be able to provide asynchronous processing capabilities:

- Read file stream
- Write file stream
- Query directory contents
- Notify when directory contents are changed
- Byte-range lock/unlock requests
- Device IOCTL requests
- File system IOCTL requests

All of the other possible FSD requests are *inherently synchronous.* Remember that the caller must use the API provided by the NT I/O Manager (or native NT I/O services like NtReadFile()) to obtain and modify file system data. The NT I/O Manager classifies all APIs, excluding the ones corresponding to the listed request

types, as synchronous APIs and will therefore perform a wait in the invoking thread's context, even if the caller has requested asynchronous processing. Therefore, the file system can process these other types of requests in the context of the invoking thread.

For synchronous I/O requests, the NT I/O Manager serializes the I/O. Therefore, if *thread-A* requests synchronous I/O using a file object opened for synchronous I/O, and concurrently, *thread-B* issues an I/O request using the same file object, the I/O request arriving later in the I/O Manager (say, *thread*-B's request) will be forced to wait by the I/O Manager until the first request has been completed.

Typically, a thread issuing asynchronous I/O requests can synchronize with the completion of the request using one of the three methods listed below:

Waiting for the file object handle itself
: The NT I/O Manager sets the file object handle to a not-signaled state when the I/O is requested and then signals the file object after `IoCompleteRequest()` has been invoked for the IRP representing the I/O request. This method is not error-proof, however, because if two asynchronous I/O requests are issued concurrently, the file handle is signaled when one of them finishes, and it is then not possible for the caller to determine which of the two requests actually completed.

Waiting for an event object supplied by the caller when requesting the I/O
: This method is mutually exclusive with waiting for the file object (i.e., if the caller supplies an event object when requesting the I/O, the NT I/O Manager will signal the event instead of signaling the file object). This method is more robust if multiple, concurrent, asynchronous I/O requests will be issued.

Specifying an APC to be invoked when the I/O is completed
: Each of the potentially asynchronous I/O APIs listed accepts the address of an optional APC routine that will be invoked by the I/O Manager after the IRP has been completed. This APC is invoked with the caller-supplied context and the address of the I/O status block containing the results of the I/O operation.

Now that you have a fairly good understanding of how to determine the top-level component for a request and how to asynchronously process FSD requests, we'll discuss other important FSD dispatch routine implementations. Let's start with the *set and query file information* requests.

Dispatch Routine: File Information

It is quite typical for a user to want to query and manipulate information about file streams such as the current file size, the date that the file stream was last

accessed, the date that the file stream was last modified, the number of links to the data associated with the filename entry, and other similar information.

Since a filename entry in a directory is considered an attribute associated with the file stream data, the Windows NT operating system allows the user to delete and add filename entries (links) to the file stream via the set file information routine. In fact, the only method allowed by the NT I/O Manager to delete filenames is to open the file stream, modify the file attributes by specifying that the filename entry be deleted using the set file information dispatch routine, and then closing the file handle.*

One of the peculiarities of the Windows NT I/O Manager and FSD interface is the method mandated by the I/O subsystem in processing user requests to rename file streams. A rename operation can be logically decomposed into the following two steps:

1. Remove the original filename entry from the source directory.
2. Add a new filename entry to the destination directory; this entry must refer to the same on-disk data stream that was pointed to by the original (source) filename.

As you can see, there are four objects that potentially need to be manipulated in a rename operation (if the source and target directories are the same, then you have only three objects to worry about):

- The filename being deleted
- The source directory that contains the filename being deleted
- The filename being added
- The target directory, which will contain the new filename entry

In the case where the source and target directories are different, the NT I/O Manager performs the following sequence of operations:

1. First, request the FSD to open the target directory and determine whether the target filename exists.

   This special create request sent to the FSD is recognizable by the presence of the SL_OPEN_TARGET_DIRECTORY flag in the Flags field of the current I/O stack location of the create IRP. In Chapter 9, you saw the response from the FSD in the create dispatch routine entry point.

' This sequence is performed transparently by Windows NT subsystems when a user application process requests that the file entry be deleted. The actual deletion of the file name entry is only performed by the FSD in the cleanup dispatch routine, which in turn is invoked after all of the user handles corresponding to a particular file object have been closed. You will see this in the description for the cleanup dispatch routine entry point provided later in this chapter.

The FSD is expected to respond by determining whether the target filename exists or not, and then by opening the parent directory of the target file. The FSD must also replace the name supplied in the create request (which is the complete path and filename leading to the target file) with only the name of the target file itself. For example, if the I/O Manager supplies a pathname *\directoryl\directory2\directory3\source_dir\foo,* the FSD should replace this name with the name *foo* in the FileName field of the file object structure created by the I/O Manager.

The following code fragment from the create dispatch routine entry point, describing the steps listed previously, was originally presented in Chapter 9 and is expanded upon and included here for completeness:

```
// Now we are down to the last component, check it out to see if it
// exists . . .
// Even for the "open target directory" case below, it is important
// to know whether the final component specified exists (or not) .

// If "open target directory" was specified:
if (OpenTargetDirectory) {
    if (NT_SUCCESS(RC) ) {
        // file exists, set this information in the Information
        // field.
        Returnedlnformation = FILE_EXISTS;
    } else {
        RC = STATUS_SUCCESS ;
        // Tell the I/O Manager that file does not exit.
        Returnedlnformation = FILE_DOES_NOT_EXIST;
    }

    // Now, do the following:
    // (a) Replace the string in the FileName field in the
    //     PtrNewFileObject to identify the target name
    //     only (i.e. the final component string without the path
    //     leading to the object) .

    {
        unsigned int Index =
            ( (AbsolutePathName. Length / sizeof (WCHAR) ) - 1) ;

        // Back up until we come to the last '\' .
        // But first, skip any trailing '\' characters.

        while (AbsolutePathName. Buffer[ Index] == L'\\') {
            ASSERT(Index >= sizeof (WCHAR) );
            Index -= sizeof (WCHAR) ;
            // Skip this length also.
            PtrNewFileObject->FileName. Length -= sizeof (WCHAR) ;
        }

        while (AbsolutePathName. Buffer [Index] != L'\\') {
            // Keep backing up until we hit one.
```

```
            ASSERT(index >= sizeof(WCHAR));
            Index -= sizeof(WCHAR);
        }

        //We must be at a 'V character.
        ASSERT(AbsolutePathName.Buffer[Index]  == L'\\');
        Index++;

        //We can now determine the new length of the filename
        // and copy the name over.
        PtrNewFileObject->FileName.Length -=
                (unsigned short)(Index*sizeof(WCHAR));
        RtlCopyMemory(&(PtrNewFileObject->FileName.Buffer[0]),
                    &(PtrNewFileObject->FileName.Buffer[Index]),
                    PtrNewFileObject->FileName.Length);
    }

    // (b) Return with the target's parent directory opened.
    // (c) Update the file object FsContext and KsContext2 fields
    //     to reflect the fact that the parent directory of the
    //     target has been opened.

    try_return(RC);
}
```

2. If the FSD returns FILE_EXISTS, and the original rename request does not request file replacement, the I/O Manager will return a STATUS_OBJECT_NAME_COLLISION error to the caller.
3. Now, the I/O Manager issues the IRP_MJ_SET_INFORMATION request to the FSD, passing in the target directory file object pointer, the full pathname of the source file, and the request to rename the file.

   A description of the processing performed by the FSD upon receiving the IRP_MJ_SET_INFORMATION request is described later in this chapter.

NOTE You may be wondering why the I/O Manager issues the "open target directory" request to the FSD prior to issuing the rename (and also link) IRPs. Remember that, in order to issue the IRP_MJ_SET_INFORMATION request, the I/O Manager requires an open file object pointer. Therefore, the logical choice for file stream to be opened (for which a file object will be created) is the target (parent) directory in which the rename (or link) -will be performed since this is the directory whose contents will definitely be modified as a result of processing the rename/link request.

You should note that the method used by the I/O Manager to create a new hard link for a file stream across directories is exactly the same as the rename operation previously described.

It is not required that an FSD use the same dispatch routine for both the IRP_MJ_QUERY_INFORMATION and the IRP_MJ_SET_INFORMATION major functions. However, that is the approach taken by the sample FSD driver. It can be easily changed, if you so desire, in your driver implementation.

Logical Steps Involved

The I/O stack location contains the following structures relevant to processing the query file information and the set file information requests issued to a FSD:

```
typedef struct _IO_STACK_LOCATION {

    // ....

    union {

        //....

        // System service parameters for:  NtQuerylnformationFile
        struct {
            ULONG Length;
            FILE_INFORMATION_CLASS FilelnformationClass;
        } QueryFile;

        // System service parameters for:  NtSetlnformationFile
        struct {
            ULONG Length;
            FILE_INFORMATION_CLASS FilelnformationClass;
            PFILE_OBJECT FileObject;
            union {
                struct {
                    BOOLEAN ReplacelfExists;
                    BOOLEAN AdvanceOnly;
                };
                ULONG ClusterCount;
                HANDLE DeleteHandle;
            };
        } SetFile;

        // ....
} Parameters;

// ....

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

The following logical steps are executed by the FSD upon receiving a query/set file information IRP. Note that to query or modify file attribute information, the caller must have previously opened the file stream. Therefore, the FSD is guaranteed to receive a pointer to a file object that was created during the open

operation, from which it can obtain pointers to the internal associated CCB and FCB data structures.

If your FSD does not support any of the query/set file information types described below, the FSD should return STATUS_INVALID_PARAMETER when asked to process the unsupported type.

IRP_MJ_QUERY_INFORMATION

The I/O Manager can request different types of information about the file stream. The Parameters.Query-Directory.FileInformationClass field in the current I/O stack location in the IRP contains the type of information requested by the caller. The information requested is one of the following:

FileBasicInformation (FILE_BASIC_INFORMATION)

```
typedef struct _FILE_BASIC_INFORMATION {
    LARGE_INTEGER    CreationTime;
    LARGE_INTEGER    LastAccessTime;
    LARGE_INTEGER    LastWriteTime;
    LARGE_INTEGER    ChangeTime;
    ULONG            FileAttributes;
} FILE_BASIC_INFORMATION, *PFILE_BASIC_INFORMATION;
```

The possible file attribute values that your FSD might return will be one or more of FILE_ATTRIBUTE_READONLY, FILE_ATTRIBUTE_HIDDEN, FILE_ATTRIBUTE_DIRECTORY (to indicate a directory type file stream), and other similar values defined in the DDK.

Fundamentally, the basic information requested includes the various time attributes associated with the file stream, as well as information on the type of the file. If your FSD does not support certain time values requested (e.g., your FSD may not support the concept of a separate CreationTime for the file stream), you should return a 0 value in the corresponding field.

The CreationTime is defined as the date and time that the file stream was created. The LastAccessTime specifies the date and time that the contents of the file stream were last accessed, the LastWriteTime specifies the date and time that the file stream was last written to, and the ChangeTime specifies the date and time that one or more attributes of the file stream were changed.

All time values are specified in the standard Windows NT system-time format, in which the absolute system time is the number of 100 nanosecond intervals since January 1, 1601.

The LastAccessTime value is initialized when a file stream is created. It is typically updated when the file data is read. For directories, this value is updated when query directory requests are received by the FSD.

The LastwriteTime is initialized when a file stream is created, superseded, or overwritten (during a create operation). For an ordinary file, it is typically updated when write requests are received by the FSD. For directories, the value is updated when a new file is created or superseded in a directory, or when a set file information request is received that affects the contents of the directory. These requests include the FileDispositionInformation, FileRenameInformation (affects the LastwriteTime for both the source and target directories), and the FileLinkInformation request.

The ChangeTime is initialized when a file stream is created. It is modified whenever the LastwriteTime for a file stream is modified. In addition, the ChangeTime should be updated when a set file information request of type FileAllocationInformation or FileEndOfFileInformation is received for an ordinary file. The FileDispositionInformation request (if successful) results in the ChangeTime being updated for the affected file as well as the directory containing the file; the FileRenameInformation type request results in the change time being modified for both the source and target directories, and the FileLinkInformation request results in the ChangeTime being modified for the file being linked to as well as the directory containing the file.

FileStandardInformation (FILE_STANDARD_INFORMATION)

```
typedef struct _FILE_STANDARD_INFORMATION {
    LARGE_INTEGER     AllocationSize;
    LARGE_INTEGER     EndOfFile;
    ULONG             NumberOfLinks;
    BOOLEAN           DeletePending;
    BOOLEAN           Directory;
} FILE_STANDARD_INFORMATION, *PFILE_STANDARD_INFORMATION;
```

The structure shown here is mostly self-explanatory. The NumberOfLinks field refers to the number of directory entries that point to the data for the file stream. This field has a value that is typically set to 1 but can have a value greater than 1 if your FSD supports multiply linked file streams. The DeletePending field is set to TRUE if some thread had previously invoked the set file information dispatch entry point, requesting that the file be marked for deletion. The AllocationSize and the EndOfFile size definitions were introduced in Chapter 6.

FileNetworkOpenInformation (FILE_NETWORK_OPEN_INFORMATION)

```
typedef struct _FILE_NETWORK_OPEN_INFORMATION {
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastwriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER AllocationSize;
    LARGE_INTEGER EndOfFile;
    ULONG         FileAttributes;
} FILE_NETWORK_OPEN_INFORMATION, *PFILE_NETWORK_OPEN_INFORMATION;
```

This particular form of file information request was added with Windows NT version 4.0 to speed-up network file information requests served by the LAN Manager Server. The Server Message Block (SMB) protocol, used by the LAN Manager Server and the LAN Manager Redirectors, contains a request to get *standard information* about a file stream. This standard information structure, as defined in the SMB protocol, consists of both the information obtained via FILE_BASIC_INFORMATION and the file size values normally obtained by issuing a second request for FILE_STANDARD_INFORMATTON. To avoid making two separate trips through the I/O Manager to the FSD, the Windows NT operating system designers decided to add this optimization of having a single call provide the necessary information in Version 4.0.

FileInternalInformation (FILE_INTERNAL_INFORMATION)

```
typedef struct _FILE_INTERNAL_INFORMATION {
    LARGE_INTEGER IndexNumber;
) FILE_INTERNAL_INFORMATION, *PFILE_INTERNAL_INFORMATION;
```

If your FSD can associate a unique numerical value with a particular file stream, you should return this value when *file internal information* is requested from you. The native FASTFAT implementation returns the cluster number index value in the logical volume for the on-disk FCB structure. The native NTFS implementation returns the on-disk index entry (record index) in the Master File Table (MFT) for the file stream.

The caller can subsequently supply the file identifier in a create/open request sent to your FSD, instead of a complete pathname leading to the file to be created. Your FSD should then be capable of identifying the file to be opened using the file identifier value. NTFS, for example, reads the particular MFT record identified by the file identifier into memory, and then continues processing the create/open request. Note that opening a file stream using the file identifier can be a lot quicker than doing so by supplying the entire pathname to be traversed.

FileEaInformation (FILE_EA_INFORMATION)

```
typedef struct _FILE_EA_INFORMATION {
    ULONG EaSize;
} FILE_EA_INFORMATION, *PFILE_EA_INFORMATION;
```

If your FSD supports extended attributes associated with a file stream, you should return the size of these extended attributes. Return 0 if no extended attributes are associated with the particular file stream.

FileNameInformation/FileAlternateNameInformation (FILE_NAME_INFORMATION)

```
typedef struct _FILE_NAME_INFORMATION {
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_NAME_INFORMATION, *PFILE_NAME_INFORMATION;
```

Your FSD must return the complete pathname for the open file stream (the name beginning with the root directory in the logical volume on which the file stream resides). If your FSD supports the DOS-style 8.3 names on-disk (in addition to the regular, long filename), and if the request is for FileAlternateNameInformation, you should return that name instead. However, it is not required that all FSD implementations support an alternate name for a file stream.

FileCompressionInformation (FILE_COMPRESSION_INFORMATION)

```
typedef struct _FILE_COMPRESSION_INFOKMATION {
    LARGE_INTEGER CompressedFileSize;
} FILE_COMPRESSION_INFORMATION, *PFILE_COMPRESSION_INFORMATION;
```

If your FSD supports compressed file streams, here is your opportunity to return the true on-disk size (compressed size) for the file.

FilePositionInformation (FILE_POSITION_INFORMATION)

```
typedef struct _FILE_POSITION_INFORMATION
    LARGE_INTEGER CurrentByteOffset
} FILE_POSITION_INFORMATION, *PFILE_POSITION_INFORMATION;
```

If the file object was opened for synchronous I/O, the I/O Manager does not even bother to call the FSD when file position information is requested, but instead fills in the information directly from the CurrentByteOffset field in the file object data structure. If, however, the file object was not opened for synchronous I/O, the I/O Manager invokes the FSD to satisfy the request. Unfortunately, though, the caller is not guaranteed, in this case, to have any valid information returned to it, unless the file position had been explicitly set at some prior time. The reason is that all of the current NT FSD implementations also appear to obtain the current file position from the file object structure; however, this structure is only guaranteed to be updated during synchronous I/O operations, and therefore contains a valid current file position value only if the file object had been opened for synchronous I/O.

FileAllInformation (FILE_ALL_INFORMATION)

```
typedef struct _FILE_ALL_INFORMATION {
    FILE_BASIC_INFORMATION          BasicInformation;
    FILE_STANDARD_INFORMATION       StandardInformation;
    FILE_INTERNAL_INFORMATION       InternalInformation;
    FILE_EA_INFORMATION             EaInformation;
    FILE_ACCESS_INFORMATION         AccessInformation;
    FILE_POSITION_INFORMATION       PositionInformation;
    FILE_MODE_INFORMATION           ModeInformation;
    FILE_ALIGNMENT_INFORMATION      AlignmentInformation;
    FILE_NAME_INFORMATION           NameInformation;
} FILE_ALL_INFORMATION, *PFILE_ALL_INFORMATION;
```

The FSD combines information that might otherwise be requested separately and returns it in this call. Take note of the fact that you do not need to worry about the AccessInformation, ModeInformation, and AlignmentIn-

formation requested. See the note below on how this information is filled into the user-supplied buffer.

`FileStreamlnformation (FILE_STREAM_INFORMATION)`

```
typedef struct _FILE_STREAM_INFORMATION {
    ULONG               NextEntryOffset;
    ULONG               StreamNameLength;
    LARGE_INTEGER       StreamSize;
    LARGE_INTEGER       StreamAllocations!ze;
    WCHAR               StreamName[l];
} FILE_STREAM_INFORMATION,  *PFILE_STREAM_INFORMATION;
```

This particular type of query file information call is supported only by NTFS, out of all the native file systems supported under Windows NT. The NTFS implementation supports multiple named/unnamed data streams for any on-disk file. This particular call can be used by the caller to obtain name and stream-length information for all the data streams for a named file object. The caller typically supplies a buffer that is of some appropriate size. NTFS determines all of the valid data streams for the file represented by the FCB and fills in information (using the structure defined previously) for each such stream into the caller-supplied buffer. If the buffer turns out to be too small to contain information on all streams, an appropriate error (STATUS_BUFFER_OVERFLOW) is returned to the caller. Each entry in the buffer contains information for a data stream, is quad-aligned (4 byte aligned), and contains the offset in the NextEntryOffset field for the next entry. The last entry contains a value of 0 in the NextEntryOffset field. Typically, a named data stream supported by NTFS has a name such as *Joes_Book:$DATA*, while an unnamed data stream will have a name such as *::$DATA*. If your FSD supports multiple byte streams, then you should also implement support for this query information call.

In addition to the information types previously described, a user may request FileAccessInformation (for information on the type of access to the file stream granted via the file object), FileModelnformation (information on whether the file object was opened with write-through specified, whether no intermediate buffering had been requested during the open, and so on), or FileAlignmentInformation (for the alignment mandated by the device object for the logical volume on which the file stream resided). This kind of requested information is considered FSD-independent by the I/O Manager, since it can be immediately obtained by the I/O Manager without having to invoke the FSD. Therefore, the NT I/O Manager fills in this information itself and returns control to the caller. In the case of FileAllInformation, the I/O Manager fills in the information contained in these FSD-independent categories and then forwards the request to the FSD.

When the FSD receives an IRP requesting information for the file stream, it performs the following simple logical steps to process the request:

- Obtain a pointer to the I/O-Manager-supplied system buffer
- Acquire the MainResource shared, to synchronize with any user requested changes
- Determine the type of information requested and copy it over into the supplied buffer

Note that the information requested is typically available immediately in memory from the file control block structure for the file stream. Most file system driver implementations update their FCB with the on-disk metadata associated with the file stream when it is first opened, and then subsequently keep the information updated in memory as long as the FCB is retained.

IRP_MJ_SET_INFORMATION

The following types of requests can be issued to modify file attributes:

`FileBasicInformation` (FILE_BASIC_INFORMATION)

This request type is used to modify file time and dates. Your FSD must determine whether to use caller-supplied values or values determined by your driver based upon any I/O performed by the caller.

`FileDispositionlnformation (FILE_DISPOSITION_INFORMATION)`

```
typedef struct _FILE_DISPOSITION_INFORMATION {
    BOOLEAN DeleteFile;
} FILE_DISPOSITION_INFOKMATION, *PFILE_DISPOSITION_INFORMATION;
```

This structure is used to mark a filename entry for deletion. Note that in the Windows NT I/O subsystem model, the caller must open a file stream using a link (name) associated with the file stream, mark the file name (link) for deletion using this set file information request, and then close the file handle. When the last IRP_MJ_CLEANUP request is received by the FSD (only after all user handles have been closed), the filename directory entry will actually be deleted. This means that any directory query operations issued in the interim will continue to see the filename entry.*

`FilePositionlnformation (FILE_POSITION_INFORMATION)`

This request is issued to set the byte offset field in the file object structure. The byte offset is used by the FSD to determine the position to read/write for file objects that are opened for synchronous access. Note that the FSD must

---

" Readers who have a UNIX file system background, or even those who have used UNIX file systems, will recognize that this is different from the method used there. In UNIX file systems, the filename directory entry is immediately removed when an `unlink()` operation is performed on the directory entry.

check for and deny any requests to set the byte offset to a value that is not aligned appropriately for the physical device object on which the logical volume resides, if the file object was opened with no intermediate buffering specified.

FileAllocationInformation (FILE_ALLOCATION_INFORMATION)

```
typedef struct _FILE_ALLOCATION_INFORMATION {
    LARGE_INTEGER AllocationSize;
} FILE_ALLOCATION_INFORMATION, *PFILE_ALLOCATION_INFORMATION;
```

This request is used by the caller to increase or decrease the allocation size of a file stream. Note that this request does not affect the end-of-file position for the file stream. Increasing the allocation size does not pose any problems; your FSD can do whatever it needs in order to reserve additional on-disk space for the file stream.

Decreasing the file stream allocation size requires a little bit more effort on the part of the FSD. The NT VMM does not allow file size decreases if any process has mapped the file stream into its virtual address space; this, however, does not apply to the mapping performed by the NT Cache Manager. Therefore, before attempting to decrease the allocation size for a file stream, the FSD must first request permission to proceed from the VMM. If the VMM agrees, then the file size modification can proceed; otherwise, the FSD is required to fail the request.

Whenever the allocation size is changed, the FSD must be careful to immediately inform the NT Cache Manager of any such changes.

FileEndOfFileInformation (FILE_END_OF_FILE_INFORMATION)

```
typedef struct _FILE_END_OF_FILE_INFORMATION {
    LARGE_INTEGER EndOfFile;
} FILE_END_OF_FILE_INFORMATION, *PFILE_END_OF_FILE_INFORMATION;
```

A change in the end-of-file position implies that the allocation size for the file stream could also be changed. Specifically, if the new end-of-file has a value that is larger than the current allocation size for the file stream, most FSD implementations will change the allocation size as well and reserve additional on-disk space at this time. It is not mandated by the NT I/O Manager that you do this; however, it would be prudent to avoid a nasty situation where your FSD successfully extends the current end-of-file, does not prereserve new space corresponding to the extended file stream, and later gets and returns a disk out-of-space error when the user actually attempts to write to the file.*

* In general, it is wise to report error conditions to users when they expect errors and are able to respond to them sensibly. In the scenario described above, a user of the FSD can possibly try to workaround the error condition if you return an out-of-disk-space error when the caller attempts to extend the file size. Not doing so at this time, but failing a subsequent write request because the space is simply not available on disk will lead to a very confused caller (since the caller probably deduced from the success of the file extend operation that sufficient disk space should be available).

The FSD must follow the rule described earlier when truncating the file stream. The FSD must first request permission from the NT VMM before proceeding; if such permission is denied because another process has the file stream mapped in its virtual address space, the NT VMM will deny the request.

FileRenameInformation/FileLinkInformation (FILE_RENAME_INFORMATION/FILE_LINK_INFORMATION)

```
typedef struct _FILE_LINK_INFORMATION {
    BOOLEAN           ReplaceIfExists;
    HANDLE            RootDirectory;
    ULONG             FileNameLength;
    WCHAR             FileName[1];
} FILE_LINK_INFORMATION, *PFILE_LINK_INFORMATION;

typedef struct _FILE_RENAME_INFORMATION {
    BOOLEAN           ReplaceIfExists;
    HANDLE            RootDirectory;
    ULONG             FileNameLength;
    WCHAR             FileName[1];
} FILE_RENAME_INFORMATION, *PFILE_RENAME_INFORMATION;
```

Both the rename and the link operations are fairly complex to implement. The following steps must be taken to successfully process a rename or a link request:*

a. First, the source directory must be opened.

b. The I/O Manager supplies the file object pointer representing the opened target directory. The FSD should once again check whether the target filename already exists, and reject the request if it does and if the caller did not request replacement of the target filename.

c. The source filename directory entry must be deleted in the case of a rename operation (this is not required for link operations).

d. The new filename entry must then be added.

When the FSD receives an IRP requesting modifications to the file metadata, it performs the following logical steps to process the request:

- Obtain a pointer to the I/O-Manager-supplied system buffer containing the parameters defining the request.
- If the FSD supports opportunistic locking (described in the next chapter), check whether the caller can be allowed to proceed based upon the state of the oplocks for the file stream.

---

* The FASTFAT implementation supplied with the Windows NT operating system does not support multiple links to a file stream. NTFS does, however. Whether you have to worry about link requests depends upon the capabilities provided by your file system.

- Acquire the MainResource for the FCB exclusively, to synchronize with other threads.
- Determine whether any other resources need to be acquired for operations such as rename/link on the file stream (typically, your FSD will acquire the VCB resource exclusively and the PagingloResource for the FCB exclusively as well).
- Determine the nature of the request and invoke an appropriate routine to perform the requested functionality.

Code Fragment

```
NTSTATUS        SFsdCommonFileInfo(
PtrSFsdlrpContext                PtrlrpContext,
PIRP                             Ptrlrp)
{
    // Declarations go here ...

    try {
        // First, get a pointer to the current I/O stack location.
        PtrloStackLocation = IoGetCurrentIrpStackLocation(Ptrlrp);
        ASSERT(PtrloStackLocation);

        PtrFileObject = PtrIoStackLocation->FileObject;
        ASSERT(PtrFileObject);

        // Get the FCB and CCB pointers.
        PtrCCB = (PtrSFsdCCB)(PtrFileObject->FsContext2);
        ASSERT(PtrCCB);
        PtrFCB = PtrCCB->PtrFCB;
        ASSERT(PtrFCB);
        PtrReqdFCB = &(PtrFCB->NTRequiredFCB);

        CanWait = ((PtrIrpContext->IrpContextFlags
                    & SFSD_IRP_CONTEXT_CAN_BLOCK)
                ? TRUE : FALSE);

        // If the caller has opened a logical volume and is attempting to
        // query information for it as a file stream, return an error.
        if (PtrFCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_VCB) {
            // This is not allowed. Caller must use get/set volume
            // information instead.
            RC = STATUS_INVALID_PARAMETER;
            try_return(RC);
        }

        ASSERT(PtrFCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_FCB);

        // The NT I/O Manager always allocates and supplies a system
        // buffer for query and set file information calls.
        // Copying information to/from the user buffer and the system
```

```
        II buffer is performed by the I/O Manager and the FSD need not
        // worry about it.
        PtrSystemBuffer = PtrIrp->AssociatedIrp.SystemBuffer;

        if (PtrIoStackLocation->MajorFunction == IRP_MJ_QUERY_INFOKMATION)
        {
            // Now, obtain some parameters.
            BufferLength = PtrIoStackLocation->Parameters.QueryFile.Length;
         FunctionalityRequested =
            PtrIoStackLocation->Parameters.QueryFile.FilelnformationClass;

            // Acquire the MainResource shared (NOTE: for paging I/O on a
            // page file, we should avoid acquiring any resources and
            // simply trust the VMM to do the right thing, or else we
            // could possibly run into deadlocks).
            if (!(PtrFCB->FCBFlags & SFSD_FCB_PAGE_FILE)) {
                // Acquire the MainResource shared.
                if (!ExAcquireResourceSharedLite(&(PtrReqdFCB->
                                                   MainResource),
                                                   CanWait)) {
                    PostRequest = TRUE;
                    try_return(RC = STATUS_PENDING);
                }
                MainResourceAcquired = TRUE;
            }

            // Do whatever the caller asked us to do.
            switch (FunctionalityRequested) {
            case FileBasicInformation:
                RC = SFsdGetBasicInformation(PtrFCB,
                                 (PFILE_BASIC_INFORMATION)PtrSystemBuffer,
                                 &BufferLength);
                break;
            case FileStandardlnformation:
                // RC = SFsdGetStandardInformation(PtrFCB, PtrCCB, ...);
                break;
            // Similarly, implement all of the other query information
            // routines that your FSD can support.
#ifdef     _NT_VER_40_PLUS_
            case FileNetworkOpenlnformation:
                // RC = SFsdGetNetworkOpenlnformation(...);
                break;
#endif    // _NT_VER_40_PLUS_
            case Filelnternallnformation:
                // RC = SFsdGetlnternallnformation(...);
                break;
            case FileEalnformation:
                // RC = SFsdGetEalnformation(...);
                break;
            case FileNamelnformation:
                // RC = SFsdGetFullNamelnformation(...);
                break;
            case FileAlternateNamelnformation:
                // RC = SFsdGetAltNamelnformation(...);
```

```
            break;
        case FileCompressionInfomnation:
            // RC = SFsdGetCompressionInformation(...);
            break;
        case FilePositionInformation:
            // This is fairly simple. Copy over the information from
            // the file object.
            {
           PFILE_POSITION_INFORMATION        PtrFileInfoBuffer;

                PtrFileInfoBuffer =
                    (PFILE_POSITION_INFORMATION)PtrSystemBuffer;

                ASSERT(BufferLength >=
                    sizeof(FILE_POSITION_INFORMATION));
                PtrFileInfoBuffer->CurrentByteOffset =
                                        PtrFileObject->CurrentByteOffset;
                // Modify the local variable for BufferLength
                // appropriately.
                BufferLength -= sizeof(FILE_POSITION_INFORMATION);
            }
            break;
        case FileStreamInformation:
            // RC = SFsdGetFileStreamInformation(...);
            break;
        case FileAllInformation:
            // The I/O Manager supplies the Mode, Access, and Alignment
            // information. The rest is up to us to provide.
            // Therefore, decrement the BufferLength appropriately
            // (assuming that the above 3 types of information are
            // already in the buffer)
            {
           PFILE_POSITION_INFORMATION        PtrFileInfoBuffer;
           PFILE_ALL_INFORMATION             PtrAllInfo =
                                   (PFILE_ALL_INFORMATION)PtrSystemBuffer;

                BufferLength -= (sizeof(FILE_MODE_INFORMATION) +
                                 sizeof(FILE_ACCESS_INFORMATION) +
                         sizeof(FILE_ALIGNMENT_INFORMATION));

                // Fill in the position information.

                PtrFileInfoBuffer = (PFILE_POSITION_INFORMATION)
                            &(PtrAllInfo->PositionInformation);

                PtrFileInfoBuffer->CurrentByteOffset =
                                        PtrFileObject->CurrentByteOffset;

                // Modify the local variable for BufferLength
                // appropriately.
                ASSERT(BufferLength >=
                            sizeof(FILE_POSITION_INFORMATION));
                BufferLength -= sizeof(FILE_POSITION_INFORMATION);
```

```
                II Get the remaining stuff.
                if ( !NT_SUCCESS(RC =
                        SFsdGetBasicInformation(PtrFCB,
                            (PFILE_BASIC_INFORMATION)
                                &(PtrAllInfo->Basidnformation)   ,
                            &BufferLength) ) ) {
                    // Another method you may wish to use to avoid the
                    // multiple checks for success/failure is to have
                    // the called routine simply raise an exception
                    // instead.
                    try_return(RC) ;
                }
                // Similarly, get all of the others . . .
            }
            break ;
        default:
            RC = STATUS_INVALID_PARAMETER;
            try_return(RC) ;
        }

        // If we completed successfully, return the amount of
        // information transferred.
        if (NT_SUCCESS(RC) ) {
            PtrIrp->IoStatus . Information =
                PtrIoStackLocation->Parameters.QueryFile.Length
                                                    - BufferLength;
        } else {
            PtrIrp->IoStatus . Information = 0;
        }

    } else {
        ASSERT (PtrIoStackLocation->MajorFunction ==
                                        IRP_MJ_SET_INFORMATION) ;

        // Now, obtain some parameters.
     FunctionalityRequested =
           PtrIoStackLocation->Parameters . SetFile . Filelnformationclass ;

        // If your FSD supports opportunistic locking (described in
        // Chapter 11) , then you should check whether the oplock state
        // allows the caller to proceed.

        // Rename and link operations require creation of a directory
        // entry and possibly deletion of another directory entry.
        // Since, we acquire the VCB resource exclusively during
        // create operations, we should acquire it exclusively for
        // link and/or rename operations as well.
        // Similarly, marking a directory entry for deletion should
        // cause us to acquire the VCB exclusively as well.
        if ( (FunctionalityRequested == FileDispositionlnformation) | |
              (FunctionalityRequested == FileRenamelnformation) | |
              (FunctionalityRequested == FileLinklnformation) ) {
            if ( !ExAcquireResourceExclusiveLite(&(PtrVCB->
                                                    VCBResource)  ,
```

```
                                                    CanWait)) {
            PostRequest = TRUE;
            try_return(RC = STATUS_PENDING);
        }
        // We have the VCB acquired exclusively.
        VCBResourceAcquired = TRUE;
    }

    // Unless this is an operation on a page file, we should
    // acquire the FCB exclusively at this time. Note that we will
    // pretty much block out anything being done to the FCB from
    // this point on.
    if (!(PtrFCB->FCBFlags & SFSD_FCB_PAGE_FILE)) {
        // Acquire the MainResource exclusively.
        if (!ExAcquireResourceExclusiveLite(&(PtrReqdFCB->
                                                    MainResource),
                                                    CanWait)) {
            PostRequest = TRUE;
            try_return(RC = STATUS_PENDING);
        }
        MainResourceAcquired = TRUE;
    }

    // The only operations that could conceivably proceed from
    // this point on are paging I/O read/write operations. For
    // delete link (rename), set allocation size, and set EOF,
    // should also acquire the paging I/O resource, thereby
    // synchronizing with paging I/O requests. In your FSD, you
    // should ideally acquire the resource only when processing
    // such requests; here, however, I will block out all paging I/
    // O operations at this time (for convenience). However, be
    // careful when doing this, since if your callback for
    // NtCreateSection( ) (described in the next chapter), does not
    // also acquire the paging I/O resource appropriately, you
    // could cause a deadlock situation.*
    if (!ExAcquireResourceExclusiveLite(&(PtrReqdFCB->
                                            PagingloResource),
                                            CanWait)) {
        PostRequest = TRUE;
        try_return(RC = STATUS_PENDING);
    }

    //Do whatever the caller asked us to do
    switch (FunctionalityRequested) {
    case FileBasicInformation:
        RC = SFsdSetBasicInformation(PtrFCB, PtrCCB, PtrFileObject,
                        (PFILE_BASIC_INFORMATION) PtrSystemBuffer);
        break;
```

* It is unlikely that a deadlock would occur even in such a situation because the paging I/O resource is typically designated as an end-resource (by definition, your driver must not attempt to acquire any other resource object once an *end-resource* has been acquired) and should ideally not be held by any thread for a *long* period of time. However, it might be better to be prudent and understand the ramifications of this particular resource acquisition method shown in the code sample.

```
case FilePositionInformation:
    // Check    if no intermediate buffering has been
    // specified. If it was specified, do not allow nonaligned
    // set file position requests to succeed.
    {
   PFILE_POSITION_INFORMATION       PtrFilelnfoBuffer;

        PtrFilelnfoBuffer =
                    (PFILE_POSITION_INFOEMATION)PtrSystemBuffer;

        if (PtrFileObject->Flags &
              FO_NO_INTERMEDIATE_BUFFERING) {
            if (PtrFilelnfoBuffer->CurrentByteOffset.LowPart &
                PtrIoStackLocation->DeviceObject->
                                        AlignmentRequirement) {
                // Invalid alignment.
                try_return(RC = STATUS_INVALID_PARAMETER);
            }
        }

        PtrFileObject->CurrentByteOffset =
            PtrFilelnfoBuffer->CurrentByteOffset;
    }
    break;
case FileDispositionInformation:
    RC = SFsdSetDispositionInformation(PtrFCB, PtrCCB, PtrVCB,
              PtrFileObject, PtrlrpContext, Ptrlrp,
              (PFILE_DISPOSITION_INFORMATION) PtrSystemBuffer);
    break;
case FileRenameInformation:
case FileLinkInformation:
    // When you implement your rename/link routine, be careful
    // to check the following two arguments:
    // TargetFileObject =
    //PtrIoStackLocation->Parameters.SetFile.FileObject;
    // ReplaceExistingFile =
    //  PtrIoStackLocation->Parameters.SetFile.ReplaceIfExists;

    // The TargetFileObject argument is a pointer to the
    // "target directory" file object obtained during the
    // "create" routine invoked by the NT I/O Manager with the
    // SL_OPEN_TARGET_DIRECTORY flag specified. Remember that
    // it is quite possible that if the rename/link is
    // contained within a single directory, the target and
    // source directories will be the same. The
    // ReplaceExistingFile argument should be used by you to
    // determine if the caller wishes to replace the target
    // (if it currently exists) with the new link/renamed
    // file. If this value is FALSE, and if the target
    // directory entry (being renamed-to, or the target of the
    // link) exists, you shouldreturn a STATUS_OBJECT_NAME_
    // COLLISION error to the caller.

    // RC = SFsdRenameOrLinkFile(PtrFCB, PtrCCB, PtrFileObject,
```

```
                II      PtrlrpContext,
                //      Ptrlrp,  (PFILE_RENAME_INFORMATION)PtrSystemBuffer);

                // Once you have completed the rename/link operation, do
                // not forget to notify any "notify IRPs" about the
                // actions you have performed.
                //An example is if you renamed across directories, you
                // should report that a new entry was added with the
                // FILE_ACTION_ADDED action type. The actual modification
                // would then be reported as either
                // FILE_NOTIFY_CHANGE_FILE_NAME (if a file was renamed) or
                // FILE_NOTIFY_CHANGE_DIR_NAME (if a directory was
                // renamed).
                break;
            case FileAllocationInformation:
                RC = SFsdSetAllocationInformation(PtrFCB, PtrCCB, PtrVCB,
                            PtrFileObject,
                            PtrlrpContext, Ptrlrp, PtrSystemBuffer);
                break;
            case FileEndOfFileInformation:
                // RC = SFsdSetEOF(...);
                break;
            default:
                RC = STATUS_INVALID_PARAMETER;
                try_return(RC);
            }
        }

        try_exit:    NOTHING;

    } finally {

        if (PagingIoResourceAcquired) {
            SFSdReleaseResource(&(PtrReqdFCB->PagingIoResource));
            PagingIoResourceAcquired = FALSE;
        }

        if (MainResourceAcquired) {
            SFSdReleaseResource(&(PtrReqdFCB->MainResource));
            MainResourceAcquired = FALSE;
        }

        if (VCBResourceAcquired) {
            SFSdReleaseResource(&(PtrVCB->VCBResource));
            VCBResourceAcquired = FALSE;
        }

        // Post IRP if required
        if (PostRequest) {

            // Since, the I/O Manager gave us a system buffer, we do not
            // need to "lock" anything.

            // Perform the post operation which will mark the IRP pending
```

```
            II and will return STATUS_PENDING back to us
            RC = SFsdPostRequest(PtrIrpContext, PtrIrp);

        } else {

            // Can complete the IRP here if no exception was encountered
            if (!(PtrIrpContext->IrpContextFlags
                    & SFSD_IRP_CONTEXT_EXCEPTION)) {
                PtrIrp->IoStatus.Status = RC;

                // Free up the Irp Context
                SFsdReleaseIrpContext(PtrIrpContext);

                // complete the IRP
                IoCompleteRequest(PtrIrp, IO_DISK_INCREMENT);
            }
        } // can we complete the IRP ?
    } // end of "finally" processing

    return(RC);
}

NTSTATUS    SFsdGetBasicInformation(
PtrSFsdFCB                  PtrFCB,
PFILE_BASIC_INFORMATION     PtrBuffer,
long                        *PtrReturnedLength)
{
    NTSTATUS            RC = STATUS_SUCCESS;

    try {
        if (*PtrReturnedLength < sizeof(FILE_BASIC_INFORMATION)) {
            try_return(RC = STATUS_BUFFER_OVERFLOW);
        }

        // Zero-out the supplied buffer.
        RtlZeroMemory(PtrBuffer, sizeof(FILE_BASIC_INFORMATION));

        // Note: If your FSD needs to be even more precise about time
        // stamps, you may wish to consider the effects of fast I/O on the
        // file stream. Typically, the FSD/FSRTL package simply sets a flag
        // indicating that fast I/O read/write has occurred. Time stamps
        // are then updated when a cleanup is received for the file
        // stream. However, if the user performs fast I/O and subsequently
        // issues a request to query basic information, your FSD could
        // query the current system time using KeQuerySystemTime(), and
        // update the FCB time stamps before returning the values to the
        // caller. This gives the caller a slightly more accurate value.

        // Get information from the FCB.
        PtrBuffer->CreationTime = PtrFCB->CreationTime;
        PtrBuffer->LastAccessTime = PtrFCB->LastAccessTime;
        PtrBuffer->LastWriteTime = PtrFCB->LastWriteTime;
        // Assume that the sample FSD does not support a "change time."
```

```
        II Now fill in the attributes.
      PtrBuffer->FileAttributes = FILE_ATTRIBUTE_NORMAL;

        if (PtrFCB->FCBFlags & SFSD_FCB_DIRECTORY) {
            PtrBuffer->FileAttributes |= FILE_ATTRIBUTE_DIRECTORY;
        }

        // Similarly, fill in attributes indicating a hidden file, system
        // file, compressed file, temporary file, etc. if your FSD supports
        // such file attribute values.

        try_exit: NOTHING;
    } finally {
        if (NT_SUCCESS(RC) ) {
            // Return the amount of information filled in.
            *PtrReturnedLength -= sizeof (FILE_BASIC_INFORMATION) ;
        }
    }
    return(RC) ;
}

NTSTATUS     SFsdSetBasicInformation(
PtrSFsdFCB                    PtrFCB,
PtrSFsdCCB                    PtrCCB,
PFILE_OBJECT                  PtrFileObject,
PFILE_BASIC_INFORMATION       PtrBuffer)
{
    NTSTATUS           RC = STATUS_SUCCESS;
    BOOLEAN            CreationTimeChanged = FALSE;
    BOOLEAN            AttributesChanged = FALSE;

    try {

        // Obtain a pointer to the directory entry associated with
        // the FCB being modified. The directory entry is
        // part of the data associated with the parent directory that
        // contains this particular file stream.
        // Note that no other modifications
        // are currently allowed to the directory entry, because we have
        // the VCB resource exclusively acquired (as a matter of fact,
        // NO directory on the logical volume can be currently modified).
        // PtrDirectoryEntry = SFsdGetDirectoryEntryPtr(...);

        if (RtlLargelntegerNotEqualTozero(PtrBuffer->CreationTime)) {
            // Modify the directory entry time stamp.
            // ...

            // Also note that fact that the time stamp has changed
            // so that any directory notifications can be performed.
            CreationTimeChanged = TRUE;

            // The interesting thing here is that the user has set certain
            // time fields. However, before doing this, the user may have
            // performed I/O, which in turn could have caused your FSD to
```

```
            II mark the fact that write/access time should be modified at
            // cleanup (this is especially true for fast I/O read/write
            // operations) . You might wish to mark the fact that such
            // updates are no longer required since the user has
            // explicitly specified the values to be associated with the
            // file stream.
            SFsdSetFlag(PtrCCB->CCBFlags, SFSD_CCB_CREATE_TIME_SET);
        }

        // Similarly, check for all the time stamp values that your
        // FSD cares about. Ignore the ones that you do not support.
        // ...


        // Now come the attributes.
        if (PtrBuffer->FileAttributes) {
            // We have a nonzero attribute value.
            // The presence of a particular attribute indicates that the
            // user wishes to set the attribute value. The absence
            // indicates the user wishes to clear the particular attribute.

            // Before we start examining attribute values, you may wish
            //to clear any unsupported attribute flags to reduce
            // confusion.

            SFsdClearFlag(PtrBuffer->FileAttributes,
                                ~FILE_ATTRIBUTE_VALID_SET_FLAGS)    ;
            SFsdClearFlag(PtrBuffer->FileAttributes,
                                FILE_ATTRIBUTE_NOKMAL);

            // Similarly, you should pick out other invalid flag values.
            // SFsdClearFlag(PtrBuffer->FileAttributes,
            //  FILE_ATTRIBUTE_DIRECTORY I FILE_ATTRIBUTE_ATOMIC_WRITE . . . );


            if (PtrBuffer->FileAttributes & FILE_ATTRIBUTE_TEMPORARY) {
                SFsdSetFlag(PtrFileObject->Flags, FO_TEMPORARY_FILE);
            } else {
                SFsdClearFlag(PtrFileObject->Flags, FO_TEMPORARY_FILE);
            }

            // If your FSD supports file compression, you may wish to
            // note the user's preferences for compressing/not compressing
            // the file at this time. If the user requests that the file
            //be compressed and the file is currently not compressed,
            // your FSD will probably have to initiate a fairly complex
            // execution sequence at this time.
        }

        try_exit: NOTHING;
    } finally {
        ;
    }
    return(RC);
}
```

```
NTSTATUS   SFsdSetDispositionInformation(
PtrSFsdFCB                      PtrFCB,
PtrSFsdCCB                      PtrCCB,
PtrSFsdVCB                      PtrVCB,
PFILE_OBJECT                    PtrFileObject,
PtrSFsdIrpContext               PtrIrpContext,
PIRP                            PtrIrp,
PFILE_DISPOSITION_INFORMATION   PtrBuffer)
{
    NTSTATUS            RC = STATUS_SUCCESS;

    try {
        if (!PtrBuffer->DeleteFile) {
            // "un-delete" the file.
            SFsdClearFlag(PtrFCB->FCBFlags, SFSD_FCB_DELETE_ON_CLOSE);
            PtrFileObject->DeletePending = FALSE;
            try_return(RC);
        }

        // The easy part is over. Now, we know that the user wishes to
        // delete the corresponding directory entry (if this
        // is the only link to the file stream, any on-disk storage space
        // associated with the file stream will also be released when the
        // only link is deleted.)

        //Do some checking to see if the file can even be deleted.

        if (PtrFCB->FCBFlags & SFSD_FCB_DELETE_ON_CLOSE) {
            // All done.
            try_return(RC);
        }

        if (PtrFCB->FCBFlags & SFSD_FCB_READ_ONLY) {
            try_return(RC = STATUS_CANNOT_DELETE);
        }

        if (PtrVCB->VCBFlags & SFSD_VCB_FLAGS_VOLUME_READ_ONLY) {
            try_return(RC = STATUS_CANNOT_DELETE);
        }

        // An important step is to check if the file stream has been
        // mapped by any process. The delete cannot be allowed to proceed
        // in this case.
        if   ( !MmFlushImageSection(& (PtrFCB->NTRequiredFCB.SectionObject),
                        MmFlushForDelete)){
            try_return(RC = STATUS_CANNOT_DELETE);
        }

        // It would not be prudent to allow deletion of either a root
        // directory or a directory that is not empty.
        if (PtrFCB->FCBFlags & SFSD_FCB_ROOT_DIRECTORY) {
            try_return(RC = STATUS_CANNOT_DELETE);
        }
```

```
        if (PtrFCB->FCBFlags & SFSD_FCB_DIRECTORY) {
            // Perform your check to determine whether the directory
            // is empty or not.
            // if ( !SFsdlsDirectoryEmpty(PtrFCB, PtrCCB, PtrlrpContext) ) {
            //       try_return(RC = STATUS_DIRECTORY_NOT_EMPTY) ;
            // }
        }

        // Set a flag to indicate that this directory entry will become
        // history at cleanup.
        SFsdSetFlag(PtrFCB->FCBFlags, SFSD_FCB_DELETE_ON_CLOSE);
        PtrFileObject->DeletePending = TRUE;

        try_exit: NOTHING;
    } finally {
        ;
    }
    return(RC);
}

NTSTATUS    SFsdSetAllocationlnformation(
PtrSFsdFCB                      PtrFCB,
PtrSFsdCCB                      PtrCCB,
PtrSFsdVCB                      PtrVCB,
PFILE_OBJECT                    PtrFileObject,
PtrSFsdlrpContext               PtrlrpContext,
PIRP                            Ptrlrp,
PFILE_ALLOCATION_INFORMATION    PtrBuffer)
{
    NTSTATUS            RC = STATUS_SUCCESS;
    BOOLEAN             TruncatedFile = FALSE;
    BOOLEAN             ModifiedAllocSize = FALSE;

    try {
        // Increasing the allocation size associated with a file stream
        // is relatively easy. All you have to do is execute some FSD-
        // specific code to check whether you have enough space available
        // (and if your FSD supports user/volume quotas, whether the user
        //is not exceeding quota), and then increase the file size in the
        // corresponding on-disk and in-memory structures.
        // Then, all you should do is inform the Cache Manager about the
        // increased allocation size.

        // First, do whatever error checking is appropriate here (e.g.,
        // whether the caller is trying the change size for a directory,
        //etc.).

        // Are we increasing the allocation size?
        if (RtlLargeIntegerLessThan(
                PtrFCB- >NTRequiredFCB.CommonFCBHeader.AllocationSize,
                PtrBuffer->AllocationSize)) {

            // Yes. Do the FSD-specific stuff; i.e., increase reserved
            // space on disk.
```

```
        II RC = SFsdTruncateFileAllocationSize( . . . )

        ModifiedAllocSize = TRUE;

    } else if (RtlLargeIntegerGreaterThan(
                  PtrFCB->NTRequiredFCB.CoiranonFCBHeader.Allocations!ze,
                        PtrBuffer->AllocationSize) ) {
        // This is the painful part. See if the VMM will allow us to
        // proceed. The VMM will deny the request if:
        // (a) any image section exists OR
        // (b) a data section exists and the size of the user mapped
        //     view is greater than the new size
        // Otherwise, the VMM should allow the request to proceed.
        if ( !MmCanFileBeTruncated(&:(PtrFCB->
                                        NTRequiredFCB.SectionObject)     ,
                    &(PtrBuffer->AllocationSize) )) {
            // VMM said no way!
            try_return(RC = STATUS_USER_MAPPED_FILE) ;
        }

        // Perform your directory entry modifications. Release any on-
        // disk space you may need to in the process.
        // RC = SFsdTruncateFileAllocationSize( . . . );

        ModifiedAllocSize = TRUE;
        TruncatedFile = TRUE;
    }

    try_exit:

        // This is a good place to check if we have performed a
        // truncate operation. If we have performed a truncate
        // (whether we extended or reduced file size) , you should
        // update file time stamps.

        // Last, but not the least, you must inform the Cache Manager
        // of file size changes.
        if (ModifiedAllocSize && NT_SUCCESS(RC) ) {
            // Update the FCB Header with the new allocation size.
            PtrFCB->NTRequiredFCB.CommonFCBHeader.AllocationSize =
                PtrBuffer->AllocationSize;

            // If we decreased the allocation size to less than the
            // current file size, modify the file size value.
            // Similarly, if we decreased the value to less than the
            // current valid data length, modify that value as well.
            if (TruncatedFile) {
                if (RtlLargeIntegerLessThan(
                        PtrFCB->NTRequiredFCB.CommonFCBHeader .FileSize,
                        PtrBuffer->AllocationSize) ) {
                    // Decrease the file size value.
                    PtrFCB->NTRequiredFCB.CommonFCBHeader .FileSize =
                        PtrBuffer->AllocationSize;
                }
```

```
                    if (RtlLargeIntegerLessThan(
                          PtrFCB->
                             NTRequiredFCB.CommonFCBHeader.ValidDataLength,
                             PtrBuffer->AllocationSize)) {
                        // Decrease the valid data length value.
                        PtrFCB->
                            NTRequiredFCB.CommonFCBHeader.ValidDataLength =
                                        PtrBuffer->AllocationSize;
                    }
                }


                // If the FCB has not had caching initiated, it is still
                // valid for you to invoke the NT Cache Manager. It is
                // possible in such situations for the call to be no-op'ed
                // (unless some user has mapped in the file).

                // NOTE: The invocation to CcSetFileSizes() will quite
                // possibly result in a recursive call back into the file
                // system. This is because the NT Cache Manager will
                // typically perform a flush before telling the VMM to
                // purge pages, especially when caching has not been
                // initiated on the file stream, but the user has mapped
                // the file into the process's virtual address space.
                CcSetFileSizes(PtrFileObject,
                    (PCC_FILE_SIZES)
                        &(PtrFCB->
                            NTRequiredFCB.CommonFCBHeader.AllocationSize));

                // Inform any pending IRPs (notify change directory).
            }

    } finally {
        ;
    }
    return(RC);
}
```

Notes

Read the comments provided above for information on the approach taken to implement some of the set/query file information routines.

An interesting point, not illustrated in the code example, pertains to the FileEndOfFileInformation request. In earlier chapters, we saw that the NT Cache Manager issues this call if the ValidDataLength for the file stream has been extended (due to a user writing beyond the current valid data length). This call can be distinguished by the fact that the AdvanceOnly field will be set to TRUE. When your FSD receives this special set end-of-file file information call, it should change the directory entry (on-disk) valid data length for the file stream

only if the new valid data length value is greater than the current value. This type of request is only utilized by the NT Cache Manager.

You should also be aware that both the query and the set file information calls can and do originate in the NT VMM when a process tries to create a section object for a file stream to prepare to map-in views for the file. Before issuing these calls, though, the NT VMM will invoke the FSD callback routine to acquire FSD resources for the FCB. An example of providing support for such a callback routine is given later in the next chapter.

Dispatch Routine: Directory Control

There are two kinds of directory control requests that are issued to a file system driver:

- Requests to obtain the contents of a directory
- Requests to inform the caller when specified changes occur to the files/directories contained within a directory (and in all directories recursively below the target directory)

The first type of request is by far the most common operation for which an FSD provides support. Users of file system routinely ask for a listing of the contents of a target directory. The type of information that a caller might be interested in is quite varied though; some callers may wish to find out all metadata information for all of the files and directories contained in the target directory, while other callers may be looking for a specific directory entry, and /or may wish to get some specific information only for objects that they search for in a directory.

The other type of directory control operation is the *notify change directory* request. This request is relatively uncommon and provides a transparent method for callers (both in kernel and user mode) to monitor a directory tree for specific actions that they might be interested in. As an example, consider the Windows Explorer utility provided with Windows NT. This application attempts to always list the most updated contents of a particular directory that might be actively accessed. If any changes occur (e.g., file additions, deletions, rename operations, and so on) while a user is browsing the contents of a directory, the application automatically updates the information presented to the user. In order to avoid having to constantly poll the file system to determine whether a directory tree has been modified, the Windows NT operating system instead provides the notification method where the caller can simply request that a specific callback be issued when the interesting changes occur.

Providing support for the notify change directory control request is not mandatory for a file system; and it is quite possible for a file system to return a STATUS_

NOT_IMPLEMENTED error upon receiving the notify change directory control request; however, it is a rather nice feature to support from the user's perspective.

Both the query directory contents and the notify change directory control requests are issued to the same dispatch routine servicing the IRP_MJ_DIRECTORY_CONTROL I/O request packet. However, the specific functionality requested can be determined by the minor function code that is supplied in the IRP. The IRP_MN_QUERY_DIRECTORY minor function value clearly indicates that the caller wishes to obtain some information on entries contained within the target directory, whereas the IRP_MN_NOTTFY_CHANGE_DIRECTORY minor function indicates that the caller is interested in monitoring events that affect the contents of the directory.

Logical Steps Involved

The I/O stack location contains the following structures relevant to processing the directory control request issued to an FSD:

```
typedef struct _IO_STACK_LOCATION {

    // ...

    union {

        //...

        // System service parameters for: NtQueryDirectoryFile
        struct {
            ULONG Length;
            PSTRING FileName;
            FILE_INFORMATION_CLASS FilelnformationClass;
            ULONG Filelndex;
        } QueryDirectory;

        // System service parameters for:  NtNotifyChangeDirectoryFile
        struct {
            ULONG Length;
            ULONG CompletionFilter;
        } NotifyDirectory;

        // ...
} Parameters;

// ...

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

The following logical steps are executed by the FSD upon receiving a directory control IRP. The caller must supply a valid file object pointer to a directory that was previously opened.

IRP_MN_QUERY_DIRECTORY

Conceptually, this routine is extremely simple to understand. The caller supplies a pointer to a file object for an open target directory, a search pattern that could be used when listing the contents of a target directory, and a specification on the type of information requested. The FSD is expected to simply perform a search of the directory for all entries that match the caller-supplied search pattern, and return information on one or more matching entries in the caller's buffer.

The following types of information can be requested by the caller:*

FileDirectorylnformation (FILE_DIRECTORY_INFORMATION)t

```
typedef struct _FILE_DIRECTORY_INFORMATION {
    ULONG                   NextEntryOffset;
    ULONG                   Filelndex;
    LARGE_INTEGER           CreationTime;
    LARGE_INTEGER           LastAccessTime;
    LARGE_INTEGER           LastWriteTime;
    LARGE_INTEGER           ChangeTime;
    LARGE_INTEGER           EndOfFile;
    LARGE_INTEGER           AllocationSize;
    ULONG                   FileAttributes;
    ULONG                   FileNameLength;
    WCHAR                   FileName[1];
} FILE_DIRECTORY_INFORMATION, *PFILE_DIRECTORY_INFORMATION;
```

For each of the directory entries that match the user-supplied search pattern, the FSD is expected to return all of the information defined by the FILE_DIRECTORY_INFORMATTON structure. You may notice that the information requested is a combination of the FILE_BASIC_INFORMATION and the FILE_STANDARD_INFORMATION query file information structures. The FileName field should contain the name of the entry contained in the target directory.

When information on multiple directory entries is returned by the FSD in the caller-supplied buffer, the FSD is expected to align each returned entry on a 8-byte (quadword-aligned) boundary. The NextEntryOffset field should contain either 0, indicating that there are no more entries in the buffer, or the byte offset to the next FILE_DIRECTORY_INFORMATION entry in the caller-supplied buffer.

---

* For eaeh type of information requested, the caller may request information on a single entry in the directory (that matches the optional search pattern supplied) or on multiple entries, limited by the size of the buffer supplied and the size of the directory itself.

t If you develop a distributed/networked file system, it would be advisable if your distributed protocol supported bulk *slat* features, for obtaining detailed information on multiple directory entries. The alternative of individually querying properties for each directory entry could become quite time consuming.

The FileIndex field should contain the index of the entry within the directory. Note that the FileNameLength field should contain the length of the filename in bytes; the FileName field expects a name in the UNICODE character set. The FileName should simply be appended to the end of the FILE_DIRECTORY_INFORMATTON structure and the length of the filename appropriately filled in.

NOTE The FileIndex is simply an FSD-specific value that your FSD can subsequently use (in the next request to get directory contents) to determine the offset from which to begin scanning the target directory. As an example, you could return the byte offset of the next entry in the directory and use this byte offset to begin searching the directory when the next *query directory* request is received.

FileFullDirectoryInformation (FILE_FULL_DIR_INFORMATION)

```
typedef struct _FILE_FULL_DIR_INFORMATION {
    ULONG              NextEntryOffset;
    ULONG              FileIndex;
    LARGE_INTEGER      CreationTime;
    LARGE_INTEGER      LastAccessTime;
    LARGE_INTEGER      LastWriteTime;
    LARGE_INTEGER      ChangeTime;
    LARGE_INTEGER      EndOfFile;
    LARGE_INTEGER      AllocationSize;
    ULONG              FileAttributes;
    ULONG              FileNameLength;
    ULONG              EaSize;
    WCHAR              FileName[1];
} FILE_FULL_DIR_INFORMATION, *PFILE_FULL_DIR_INFORMATION;
```

This request is similar to the FILE_DIRECTORY_INFOKMATION request; the only additional information requested is the total length of the extended attributes associated with the file stream (if any). For most third-party file systems, this request will be returned with the EaSize field set to 0.

FileBothDirectoryInformation (FILE_BOTH_DIR_INFORMATION)

```
typedef struct _FILE_BOTH_DIR_INFORMATION {
    ULONG              NextEntryOffset;
    ULONG              FileIndex;
    LARGE__INTEGER     CreationTime;
    LARGE_INTEGER      LastAccessTime;
    LARGE_INTEGER      LastWriteTime;
    LARGE_INTEGER      ChangeTime;
    LARGE_INTEGER      EndOfFile;
    LARGE_INTEGER      AllocationSize;
    ULONG              FileAttributes;
    ULONG              FileNameLength;
    ULONG              EaSize;
    CCHAR              ShortNameLength;
```

```
    WCHAR               ShortName[12];
    WCHAR               FileName[1];
} FILE_BOTH_DIR_INFORMATION, *PFILE_BOTH_DIR_INFORMATION;
```

This request is a superset of the FileFullDirectoryInformation request. Note that although native Windows NT applications support long filenames (255 characters or less), the older DOS-based applications often have difficulty with filenames that do not fit into the 8.3 format* mandated by that operating environment. The Windows NT I/O Manager attempts to support such legacy applications by working in tandem with file systems sensitive to their needs, which are prepared to maintain an abbreviated, unique alternate name that fits into the 8.3 format and can therefore be used by the older applications. The native NT file system implementations do support these alternate names and will provide such an alternate name in the FILE_BOTH_DIR_INFORMATION structure, in the ShortName field.

Note that it is not required that a file system support alternate names for directory entries.

FileNamesInformation (FILE_NAMES_INFORMATION)

```
typedef struct _FILE_NAMES_INFORMATION {
    ULONG               NextEntryOffset;
    ULONG               FileIndex;
    ULONG               FileNameLength;
    WCHAR               FileName[1];
} FILE_NAMES_INFORMATION, *PFILE_NAMES_INFORMATION;
```

This request type requires the least amount of information for each directory entry. The FSD simply has to supply the FileIndex for each entry in the directory, and the name for that entry. This is typically invoked in response to a DOS *dir/w* command.

All disk-based file systems, and for that matter, even networked file systems, have some internal representation of a directory entry structure (i.e., a structure that describes the on-disk or network-protocol-defined format representing a directory entry). When a directory control request is received by the FSD, your file system should obtain the contents of the directory, either by reading them from secondary storage, or by obtaining them from a server node across the network. Then it becomes a relatively simple matter of searching (typically sequentially) through all entries in the directory, looking for a match with the specified search pattern. Information on the matching entry can then be provided to the caller by

---

* For those readers that have somehow (luckily) managed to avoid using the DOS system, you may be amused to note that it could only support filenames that had a maximum name length of 8 characters, followed by an optional period, followed by an optional suffix that had a maximum length of 3 characters. This peculiar filename format has become well-known as the DOS 8.3 filename format.

converting the internal directory entry representation to one of the NT-defined structures described previously.*

NOTE Many current Windows NT file system implementations use the NT Cache Manager to cache directory contents just as file data is normally cached. To achieve this, they use the loCreateStreamFileObject() routine. This routine accepts two arguments (you need to supply just one of the two and the other can be NULL): a pointer to a file object structure and a pointer to a device object. Note that the pointer to the device object is ignored (and a device object pointer is obtained from the file object supplied) if a file object pointer is provided.

The implementation of the loCreateStreamFileObject() routine is quite simple: it creates and initializes a new file object structure, just as would have been created had an open operation been performed on the directory. As a side effect, it increments the ReferenceCount in the VPB structure associated with the device object to ensure that the logical volume cannot be dismounted as long as the stream file object is kept open.

Once a stream file object has been created representing the directory, the native NT file systems initialize it with appropriate pointers to the CCB and FCB structures for the directory. Now, caching can be initiated on this file object; typically, this is done by specifying PinAccess set to TRUE. This allows the FSD to read the directory contents directly into the system cache and access them using a virtual address pointer and appropriate offsets into the byte stream based on the internal directory entry representation. Furthermore, since the data in pinned into the system cache, the FSD is guaranteed that the information •will always be accessible and will not be discarded.

The stream file object can be closed by simply performing an ObDereferenceObject() operation on the file object structure. The FSD will receive a close request at this time on the file object that was dereferenced.

---

* There are some file systems that I have worked with, and that readers may he aware of, that do not store file attributes such as the file date and time stamps in the directory entry along with the file name. In these file system layouts, the FSD must obtain the address of the on-disk sector from the directory entry that contains it and read this information into memory to fill in the caller-supplied buffer. There is a large performance penalty due to the extra read operation for each directory entry for which information has to be returned. Such file systems cache a lot of information to avoid being discarded because of such poor on-disk file system layouts.

1RP_MN_NOTIFY_CHANGE_DIRECTORY

As described earlier, this functionality is not really mandated for an NT FSD. However, for users of the file system, this is a rather nice feature that file systems may be able to support.* This functionality allows file system users to specify that they be told when certain events occur to change the contents of a directory in some specified manner. For example, the caller may wish to be notified if file *foo* in directory *dirl* is deleted. Or, the user may wish to be notified if any file in directory *dirl* is deleted, or even if any file in the directory *dirl* or any directory under it is deleted or modified in some manner. Therefore, as you can see, this functionality can be a very powerful tool for file system clients who wish to monitor the file system.

Although it may seem a little difficult to implement, the Windows NT File System Runtime Library (FSRTL) does a rather nice job of providing supporting routines that your FSD can use.

WARNING The FSRTL routines providing support for the directory change notify support have not been officially exported by Microsoft. The function prototypes described here can be changed by Microsoft at will. Therefore, you could decide to use the routines described below or you can examine the description of the routines listed below to determine how to develop your own supporting routines that provide similar functionality.

The native NT FSD implementations have the support of the FSRTL package, however, and utilize the routines described below.

Basically, your FSD is responsible for executing the following steps to support this feature:

- When your FSD receives such a request, and after it has validated the user's request, it must invoke the FsRtlNotifyFullChangeDirectory() function to queue the user request. The FSD should then return STATUS_PENDING to the caller, which indicates that the IRP has been queued and will be completed when the desired event occurs.

---

* For distributed or networked file systems, it becomes practically impossible to support this feature, unless the networked/distributed protocol supports something that can be adapted to provide such functionality. The reason is simple: users can make directory changes from any node in a distributed file system. If the clients and servers do not have some means of being notified when specific changes occur, redirectors on the Windows NT client systems cannot possibly accurately support the notify change directory functionality (without resorting to extremely inefficient polling methods).

The implication is that notify change directory IRPs are held by the FSD (or the FSRTL package) until an event occurs that causes the FSD to report a monitored change to the caller.

- When changes occur to any directory, the FSD should invoke the FsRtlNotifyFullReportChange() FSRTL support routine to inform the library about the changes. The library routine is then responsible for scanning through all the notify requests that have been queued up and performing appropriate processing for those waiting for the occurrence of the particular event.
- Whenever a cleanup is performed on a particular file object (indicating that all user handles corresponding to the file object have been closed), the FSD should notify the FSRTL using the FsRtlNotifyCleanup() routine. The library routine will then dequeue and complete any IRPs that were using the particular file object.

  One of the peculiarities of the notify change directory request type is that it is a one-shot kind of request. Therefore, if an application wants to continuously monitor changes to a directory tree, it must keep reissuing the request whenever the request is satisfied because a watched-for modification occurs. However, it is possible for changes to occur to a directory in the period between the time when a notify change directory IRP is completed and the next notify change directory request arrives. In order not to lose information about such changes, the FSD (or the FSRTL package, if you use it) is responsible for keeping information about changes to the directory (or directory tree) during the period between completion of a notify change directory request and the arrival of the next such IRP.

  To achieve this objective, either the FSD or the FSRTL package allocates an internal buffer and associates it with the file object structure (using appropriate internal structures) to keep information about any changes that may occur. This buffer is only released (and monitoring consequently terminated) after the cleanup operation is received for the file object, indicating that all user handles have been closed.

The run-time library needs a list anchor from which it can queue all of the pending IRPs. The expectation is that the FSD will use one such list head for each mounted logical volume on which notify change directory requests could be issued. Furthermore, for synchronization, the library requires that the FSD allocate and initialize one MUTEX structure associated with the list head on which the notify requests can be queued.

```
VOID
FsRtlNotifyFullChangeDirectory (
IN PNOTIFY_SYNC                 NotifySync,
```

```
IN PLIST_ENTRY                  NotifyList,
IN PVOID                        FsContext,
IN PSTRING                      FullDirectoryName,
IN BOOLEAN                      WatchTree,
IN BOOLEAN                      IgnoreBuffer,
IN ULONG                        CompletionFilter,
IN PIRP                         Notifylrp,
IN PCHECK_FOR_TRAVERSE_ACCESS   TraverseCallback OPTIONAL,
IN PSECURITY_SUBJECT_CONTEXT    SubjectContext OPTIONAL
);
```

where

```
typedef PVOID PNOTIFY_SYNC;

typedef
BOOLEAN (*PCHECK_FOR_TRAVERSE_ACCESS) (
IN PVOID        NotifyContext,
IN PVOID        TargetContext,
IN PSECURITY_SUBJECT_CONTEXT    SubjectContext
);
```

Resource Acquisition Constraints:

None. Typically, though, you should acquire the FCB MainResource shared (at least) before invoking this routine.

Parameters:

NotifySync

This should be a pointer to a FSD-allocated KMUTEX structure. The sample FSD volume control block (VCB) structure has a field called NotifylRP-Mutex that is used for this purpose. This mutex should be used to protect the list of queued notify requests for the logical volume. Do not be misled into thinking that this can be any other synchronization object because of the definition of PNOTIFY_SYNC.

NotifyList

This should be a pointer to the list head for queued notify IRP structures. The library expects that you maintain one such list for each mounted logical volume. The sample FSD uses the NextNotifyIRP field for this purpose.

FsContext

This is determined by the FSD and is used to uniquely identify the notify structure. You should use the CCB pointer as the argument for this particular field. This becomes particularly useful when a cleanup is received on the file object, and the FSD can supply the CCB pointer to the run-time library to notify it to complete all pending requests for the file object.

FullDirectoryName

Exactly as its name implies, this is a complete pathname for the directory the caller wishes to monitor. Do not deallocate the memory for this string until the pending request has been completed. The FSRTL routine accepts either a Unicode or ASCII name string.

WatchTree

To monitor all directories that are children of the directory being monitored, set this variable to TRUE.

The FSD can determine whether the caller wishes to monitor the directory tree by checking for the presence of the SL_WATCH_TREE flag in the IRP flags field.

IgnoreBuffar

When a user asks to be notified of specific changes to the contents of a directory, the caller can also supply a buffer to contain the specific changes that occurred. (For example, the user process may be monitoring for any file entry that is deleted; when such a deletion occurs, it would like to know which directory entry was deleted.) The other option for the caller is simply to request to be notified whenever some change occurs, without requiring the FSD to list the specific changes that caused the notification. The caller will subsequently reenumerate the directory contents.

Providing a list of changes is slower than simply telling the user to reenumerate the directory upon being notified. The FSRTL routine allows the FSD to decide whether it wishes to speed up operations by setting the IgnoreBuffer value to TRUE and forcing the user to reenumerate the directory.

CompletionFilter

The CompletionFilter is provided by the caller issuing the notify change directory request and is invoked when the monitored event occurs.

TraverseCallback

Remember that the caller has the option of specifying that all subdirectories within a directory also be monitored for changes. If your FSD is security conscious like NTFS, you would want to ensure that the caller has appropriate permissions to monitor changes in a specific subdirectory. Therefore, your FSD has the option of supplying a callback function that is invoked by the runtime library before notifying the caller. If your callback returns FALSE, the runtime library will not notify the user of the changes that have occurred.

SubjectContext

If your FSD supplies a TraverseCallback function pointer, you need to know what the calling process is in order to check whether it has appropriate privileges. The SubjectContext is one of the arguments passed in to your

callback routine and you can obtain it (when queuing the notify request) by using the SeCaptureSecurityContext () function, which takes a pointer to an FSD-allocated SECURITY_SUBJECT_CONTEXT structure.*

Functionality Provided:

This routine will enqueue the IRP in the list of pending notify structures, if no such notify request already exists (remember that notify requests are uniquely identified by the FsContext field). Here is a logical list of steps that this function goes through:

- The FsRtlNotifyFullChangeDirectory ( ) routine obtains the current stack location pointer from the IRP and also obtains a pointer to the file object structure used in the current request.
- It then waits to acquire the supplied KMUTEX object to ensure synchronization.
- If the file object has already undergone cleanup (while the runtime library was waiting), then it immediately completes the IRP with a STATUS_ NOTTFY_CLEANUP return code. The runtime library checks for the presence of the FO_CLEANUP_COMPLETE flag in the file object structure to determine whether the file object has undergone cleanup.
- If there is a notify pending, then it completes the IRP.

  As mentioned earlier, once the first notify change directory request has been received for a specific file object, it becomes the responsibility of the FSD (or the FSRTL package in the case when the FSD uses it) to maintain information about changes to the directory being monitored, even if there is no current notify change directory request pending. This is because the FSD expects that the caller will soon reissue the notify request and therefore does not want to lose any intermediate changes between the time when the last request was completed and a new request is received. Therefore, the FsRtlNotify-FullChangeDirectory ( ) maintains state about whether any changes had occurred and immediately completes the new notify change directory request if information about any intermediate changes is already present in its internal buffer.
- If the IRP has been canceled, then it completes the IRP.
- Otherwise, if no other notify structure exists in the queue, it queues up this request.

Note that the implication is that a thread can only have one pending notify IRP per file object.

---

* This .structure is defined in the DDK.

The FsRtlNotifyFullReportChange () routine is defined as follows:

```
VOID
FsRtlNotifyFullReportChange (
IN PNOTIFY_SYNC            NotifySync,
IN PLIST_ENTRY             NotifyList,
IN PSTRING                 FullTargetName,
IN USHORT                  TargetNameOffset,
IN PSTRING                 StreaniName OPTIONAL,
IN PSTRING                 NormalizedParentName OPTIONAL,
IN ULONG                   FilterMatch,
IN ULONG                   Action,
IN PVOID                    TargetContext
);
```

where Action is one of the following:

```
tdefine FILE_ACTION_ADDED                0x00000001
tdefine FILE_ACTION_REMOVED              0x00000002
tdefine FILE_ACTION_MODIFIED             0x00000003
tdefine FILE_ACTION_RENAMED_OLD_NAME     0x00000004
#define FILE_ACTION_RENAMED_NEW_NAME     0x00000005
#define FILE_ACTION_ADDED_STREAM         0x00000006
#define FILE_ACTION_REMOVED_STREAM       0x00000007
tdefine FILE_ACTION_MODIFIED_STREAM      0x00000008
```

and FilterMatch is one of the following:

```
ttdefine FILE_NOTIFY_CHANGE_FILE_NAME       0x00000001
tdefine FILE_NOTIFY_CHANGE_DIR_NAME         0x00000002
tdefine FILE_NOTIFY_CHANGE_NAME             0x00000003
tdefine FILE_NOTIFY_CHANGE_ATTRIBUTES       0x00000004
tdefine FILE_NOTIFY_CHANGE_SIZE             0x00000008
tdefine FILE_NOTIFY_CHANGE_LAST_WRITE       0x00000010
tdefine FILE_NOTIFY_CHANGE_LAST_ACCESS      0x00000020
tdefine FILE_NOTIFY_CHANGE_CREATION         0x00000040
tdefine FILE_NOTIFY_CHANGE_EA               0x00000080
tdefine FILE_NOTIFY_CHANGE_SECURITY         0x00000100
tdefine FILE_NOTIFY_CHANGE_STREAM_NAME      0x00000200
tdefine FILE_NOTIFY_CHANGE_STREAM_SIZE      0x00000400
tdefine FILE_NOTIFY_CHANGE_STREAM_WRITE     0x00000800
tdefine  FILE_NOTIFY_VALID_MASK            0x00000fff
```

Resource Acquisition Constraints:

None.

Parameters:

NotifySync

This should be a pointer to the FSD-allocated KMUTEX structure used in the preceding FsRtlNotifyFullChangeDirectory {) routine.

NotifyList
: This is a pointer to the list head for all pending notify IRPs for the mounted logical volume.

FullTargetName
: This is the name of the target file or directory that had its attributes modified.

TargetNameOffset
: This is the byte offset of the last component in the name supplied in the FullTargetName field. The notify change directory call returns only the relative target name (relative to the directory on which the notify change directory IRP is pending).

`StreamName`
: This optional argument can be used to supply a stream name in addition to the filename. This is used by FSDs that support multiple data streams for a named file object. If supplied, the FSRTL package appends the StreamName to the stored target name.

`NormalizedParentName`
: This is the name of the parent directory for the target file or directory (optional argument).

`FilterMatch`
: FilterMatch can have any one or more of the values listed above to indicate what directory actions have occurred. This field is compared with the CompletionFilter field in the pending notify IRP structures. If any of the bit positions match, then the caller for that pending IRP is notified.

Action
: If a user buffer was supplied with the pending IRP, this Action value will be stored there along with the relative file/directory name for the object modified.

`TargetContext`
: The second argument to be passed to the FSD in the traverse access check callback.

Functionality Provided:

This routine performs the following functionality. It walks through the list of pending notify IRP structures, searching for one that matches the FilterMatch argument supplied (i.e., one or more bit values are the same), and checking whether the found entry is an exact match or an ancestor of the target file/directory name.*

* Either the matching entry has a directory name that matches the parent directory name for the target file, or the matching entry has a directory name that is some ancestor of the target file.

The caller of a notify change directory request has two options:

- All pending notify IRP structures that match the above criteria are completed at this time.
- Supply a buffer in which the names of the modified objects and actions performed on them will be returned.

Not supply any buffer, in which case the notify change directory IRP will simply be completed with the STATUS_NOTTFY_ENUM_DIR status.*

The FsRtlNotifyFullReportChange ( ) routine simply completes a matching, pending IRP with the STATUS_NOTIFY_ENUM_DIR status immediately if no buffer was provided by the caller.

NOTE Note that since the FSD (or the FSRTL package) must maintain information about a directory, even if no pending IRPs exist (as long as one instance of a notify change directory request was received), the FSD/FSRTL package maintains state about whether the caller had supplied a buffer the first time the notify request is made for a file object. Even if subsequent requests do not supply a buffer, the FSD/FSRTL package will continue to maintain an internally allocated buffer with information on changes to objects in the directory tree being monitored.

If the caller has supplied a buffer, however, the caller expects to receive information about objects that have changed and the actual changes performed on the modified objects. Once again, though, if the changes are numerous and cannot fit into the buffer, the FSD/FSRTL package always has the option of returning STATUS_NOTIFY_ENUM_DIR to the caller.

If you do decide to develop your own notify change directory support routines, be extremely careful about handling user-supplied buffers correctly, you should have your FSD create a memory descriptor list to describe the user buffer and obtain a system address for the MDL before queuing the IRP and returning STATUS__PENDING to the caller. This will allow your FSD to copy information into the user's buffer in the context of any thread (typically the one performing the modifications leading to the completion of the pending notify change directory IRP).

The structure returned by the FSD to the caller of the notify change directory request is defined below:

---

* If you check the actual value of this symbolic name, you will see that the NT_SUCCESS ( ) macro will treat this value equivalent to STATUS_SUCCESS.

```
typedef struct _FILE_NOTIFY_INFORMATION {
    ULONG          NextEntryOffset;
    ULONG          Action;
    ULONG          FileNameLength;
    WCHAR          FileName[1];
} FILE_NOTIFY_INFORMATION,  *PFILE_NOTIFY_INFORMATION;
```

The fields in the FILE_NOTIFY_INFORMATION structure shown here are fairly self-explanatory. Note that there are no special alignment restrictions on the entries returned in the user buffer (i.e., none of the returned entries require any padding bytes).

Some explanation is probably in order for two of the notify actions listed, namely, FILE_ACTION_RENAMED_OLD_NAME and FILE_ACTION_RENAMED_NEW_NAME. These two notify actions are reported by the file system driver when processing a rename operation. The rules used in reporting these events are as follows:

- The FSD reports two notification events as part of processing the rename operation:
  - — The first event is reported when the source directory entry is deleted for the object being renamed.
  - — The second event is reported when the target directory entry is added, i.e., the rename has been completed.
- When the first notification event has to be reported, the FSD has a choice of reporting either the FILE_ACTION_RENAMED_OLD_NAME action type or simply FILE_ACTION_REMOVED.

  If the rename operation will be performed within the same directory and if the target of the rename operation does not exist, the FSD should report FILE_ACTION_RENAMED_OLD_FILE, else the FSD should report FILE_ACTION_REMOVED.*
- When the second notification event has to be reported, the FSD has a choice betweenFILE_ACTION_RENAMED_NEW_NAME,FILE_ACTION_MODIFIED, and FILE_ACTION_ADDED.

  If the target file name existed before the rename operation and was replaced as a result of the rename, the FSD should report FILE_ACTION_MODIFIED for the target of the rename operation. Otherwise, if the rename operation was performed across directories, the FSD should report FILE_ACTION_

* I did not make the rules here but am simply reporting them! The reason for giving you this information is to simply assist you in reporting events in a manner similar to that employed by the existing Windows NT FSD implementations.

ADDED. Finally, if neither of the above conditions holds TRUE, the FSD should report FILE_ACTION_RENAMED_NEW_NAME.

The FsRtlNotifyCleanup ( ) routine is defined as follows:

```
VOID
FsRtlNotifyCleanup (
IN PNOTIFY_SYNC           NotifySync,
IN PLIST_ENTRY            NotifyList,
IN PVOID                  FsContext
);
```

Resource Acquisition Constraints:

None.

Parameters:

NotifySync

This should be a pointer to the FSD allocated KMUTEX structure used in the FsRtlNotifyFullChangeDirectory() routine.

NotifyList

This is a pointer to the list head for all pending notify IRPs for the mounted logical volume.

FsContext

This is the unique identifier used to locate all pending notify IRP structures. Typically, this is a pointer to the CCB structure.

Functionality Provided:

This routine simply walks the list of pending notify IRP structures, finds those that match the supplied FsContext value, and processes these IRPs. The processing consists of removing any cancel routine that the FSRTL package may have set, and completing the IRP with a status of STATUS_NOTIFY_CLEANUP.

Code sample

Here is a code fragment from the sample FSD that illustrates how an FSD processes a directory control request (notify change directory requests, illustrated later, use the routines exported by the FSRTL package):

```
NTSTATUS    SFsdCommonDirControl(
PtrSFsdlrpContext            PtrlrpContext,
PIRP                         Ptrlrp)
{
    // Declarations go here .. .

    // First, get a pointer to the current I/O stack location
    PtrloStackLocation = loGetCurrentlrpStackLocation(Ptrlrp);
    ASSERT(PtrloStackLocation);
```

```
    PtrFileObject = PtrIoStackLocation->FileObject;
    ASSERT(PtrFileObject);

    // Get the FCB and CCB pointers
    PtrCCB = (PtrSFsdCCB)(PtrFileObject->FsContext2);
    ASSERT(PtrCCB);
    PtrFCB = PtrCCB->PtrFCB;
    ASSERT(PtrFCB);

    // Get some of the parameters supplied to us
    switch (PtrIoStackLocation->MinorFunction) {
    case IRP_MN_QUERY_DIRECTORY:
        RC = SFSdQueryDirectory(Ptrlrpcontext, Ptrlrp,
                  PtrloStackLocation,
                    PtrFileObject, PtrFCB, PtrCCB);
        break;
    case IRP_MN_NOTIFY_CHANGE_DIRECTORY:
        RC = SFsdNotifyChangeDirectory(Ptrlrpcontext, Ptrlrp,
                    PtrloStackLocation,
                    PtrFileObject, PtrFCB, PtrCCB);
        break;
    default:
        // This should not happen.
        RC = STATUS_INVALID_DEVICE_REQUEST;
        PtrIrp->IoStatus.Status = RC;
        PtrIrp->IoStatus.Information = 0;

        // Free up the Irp Context
        SFsdReleaselrpContext(Ptrlrpcontext);

        // complete the IRP
        loCompleteRequest(Ptrlrp, IO_NO_INCREMENT);
        break;
    }

    return(RC);
}

NTSTATUS    SFSdQueryDirectory(
PtrSFsdlrpContext           Ptrlrpcontext,
PIRP                        Ptrlrp,
PIO_STACK_LOCATION          PtrloStackLocation,
PFILE_OBJECT                PtrFileObject,
PtrSFsdFCB                  PtrFCB,
PtrSFsdCCB                  PtrCCB)
{
    // Declarations go here ...

    try {

        // Validate the sent-in FCB
        if ((PtrFCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_VCB) ||
             !(PtrFCB->FCBFlags & SFSD_FCB_DIRECTORY)) {
            //We will only allow notify requests on directories.
```

```
        RC = STATUS_INVALID_PARAMETER;
    }

    PtrReqdFCB = &(PtrFCB->NTReguiredFCB) ;
    CanWait = ( (PtrIrpContext->IrpContextFlags
                    & SFSD_IRP_CONTEXT_CAN_BLOCK)
             ? TRUE : FALSE) ;
  PtrVCB = PtrFCB->PtrVCB;

    // If the caller does not wish to block, it would be easier to
    // simply post the request now.
    if (!CanWait) {
        PostRequest = TRUE;
        try_return(RC = STATUS_PENDING) ;
    }

    // Obtain the caller's parameters
    BufferLength =
        PtrIoStackLocation->Parameters . QueryDirectory .Length ;
    PtrSearchPattern =
        PtrIoStackLocation->Parameters.QueryDirectory.FileName;
    FilelnformationClass =
        PtrIoStackLocation->
                    Parameters.QueryDirectory.FilelnformationClass;
    Filelndex =
        PtrloStackLocation->Parameters.QueryDirectory.Filelndex;

    // Some additional arguments that affect the FSD behavior
    RestartScan      = (PtrIoStackLocation->Flags & SL_RESTART_SCAN) ;
    ReturnSingleEntry = (PtrIoStackLocation->Flags
                                    & SL_RETURN_SINGLE_ENTRY) ;
    IndexSpecified    = (PtrIoStackLocation->Flags
                                    & SL_INDEX_SPECIFIED) ;

    // I will acquire exclusive access to the FCB.
    // This is not mandatory, however, and your FSD could choose to
    // acquire the resource shared for increased parallelism.
    ExAcquireResourceExclusiveLite(&(PtrReqdFCB->MainResource), TRUE);
    AcquiredFCB = TRUE;

    // We must determine the buffer pointer to be used. Since this
    // routine could be invoked directly either in the context of the
    // calling thread or in the context of a worker thread, here is
    //a general way of determining what we should use.
    if (PtrIrp->MdlAddress) {
        Buffer = MmGetSystemAddressForMdl(PtrIrp->MdlAddress) ;
    } else {
        Buffer = PtrIrp->UserBuffer;
    }

    // The method of determining where to look from and what to look
    // for is unfortunately extremely confusing. However, here is a
    // methodology you can broadly adopt:
    // (a) You have to maintain a search buffer per CCB structure.
```

```
// (b) This search buffer is initialized the very first time
//     a query directory operation is performed using the file
//     object.
// (For the sample FSD, the search buffer is stored in the
// DirectorySearchPattern field)
// However, the caller still has the option of "overriding" this
// stored search pattern by supplying a new one in a query
// directory operation.
//
if (PtrSearchPattern == NULL) {
    // User has supplied a search pattern
    // Now validate that the search pattern is legitimate; this is
    // dependent upon the character set acceptable to your FSD.

    // Once you have validated the search pattern, you must
    // check whether you need to store this search pattern in
    // the CCB.
    if (PtrCCB->DirectorySearchPattern == NULL) {
        // This must be the very first query request.
    FirstTimeQuery = TRUE;

        // Now, allocate enough memory to contain the caller-
        // supplied search pattern and fill in the
        // DirectorySearchPattern field in the CCB
        // PtrCCB->DirectorySearchPattern = ExAllocatePool( . . . );
    } else {
        //We should ignore the search pattern in the CCB and
        // instead use the user-supplied pattern for this
        // particular query directory request.
    }

} else if (PtrCCB->DirectorySearchPattern == NULL) {
    // This MUST be the first directory query operation (else the
    // DirectorySearchPattern field would never be NULL. Also, the
    // caller has neglected to provide a pattern so we MUST invent
    // one. Use "*" (following NT conventions) as your search
    // pattern and store it in the PtrCCB->DirectorySearchPattern
    // field.

    PtrCCB->DirectorySearchPattern = ExAllocatePool(PagedPool,
                        sizeof(L"*"));
    ASSERT(PtrCCB->DirectorySearchPattern);

    FirstTimeQuery = TRUE;
} else {
    // The caller has not supplied any search pattern that we are
    // forced to use. However, the caller had previously supplied
    // a pattern (or we must have invented one) and we will use it.
    // This is definitely not the first query operation on this
    // directory using this particular file object.

    PtrSearchPattern = PtrCCB->DirectorySearchPattern;
}
```

```
        II There is one other piece of information that your FSD must store
        //in the CCB structure for query directory support. This is the
        // index value (i.e., the offset in your on-disk directory
        // structure) from which you should start searching.
        // However, the flags supplied with the IRP can make us override
        // this as well.

        if (Filelndex) {
            // Caller has told us where to begin.
            // You may need to round this to an appropriate directory
            // entry alignment value.
         StartinglndexForSearch = Filelndex;
        } else if (RestartScan) {
         StartinglndexForSearch = 0;
        } else {
            // Get the starting offset from the CCB.
            // Remember to update this value on your way out from this
            // function. But, do not update the CCB CurrentByteOffset
            // field if you reach the end of the directory (or get an
            // error reading the directory) while performing the search.
         StartinglndexForSearch = PtrCCB->CurrentByteOffset.LowPart;
        }

        // Now, your FSD must determine the best way to read the directory
        // contents from disk and search through them.

        // If ReturnSingleEntry is TRUE, please return information on only
        // one matching entry.

        // One final note though:
        // If you do not find a directory entry OR while searching you
        // reach the end of the directory, then the return code should be
        // set as follows:

        // (a) If any files have been returned (i.e., ReturnSingleEntry
        //    was FALSE and you did find at least one match), then return
        //     STATUS_SUCCESS
        // (b) If no entry is being returned then:
        //    (i) If this is the first query, i.e., FirstTimeQuery is TRUE
        //         then return STATUS_NO_SUCH_FILE
        //    (ii) Otherwise, return STATUS_NO_MORE_FILES

        try_exit:   NOTHING;

        // Remember to update the CurrentByteOffset field in the CCB if
        // required.

        // You should also set a flag in the FCB indicating that the
        // directory contents were accessed.

    } finally {
        if (PostRequest) {
            if (AcquiredFCB) {
                SFsdReleaseResource(&(PtrReqdFCB->MainResource) );
            }
```

```
            II Map the user's buffer and then post the request.
            RC = SFsdLockCallersBuffer(Ptrlrp, TRUE, BufferLength);
            ASSERT(NT_SUCCESS(RC));

            RC = SFsdPostRequest(PtrlrpContext, Ptrlrp);

        } else if ( ! (PtrIrpContext->IrpContextFlags &
                            SFSD_IRP_CONTEXT_EXCEPTION) ) {
            if (AcguiredFCB) {
                SFsdReleaseResource(&(PtrReqdFCB->MainResource));
            }

            // Complete the request.
            PtrIrp->IoStatus.Status = RC;
            PtrIrp->IoStatus.Information = BytesReturned;

            // Free up the Irp Context
            SFsdReleaselrpContext(PtrlrpContext);

            // complete the IRP
            loCompleteRequest(Ptrlrp, IO_DISK_INCREMENT);
        }
    }

    return(RC);
}

NTSTATUS    SFsdNotifyChangeDirectory(
PtrSFsdlrpContext           PtrlrpContext,
PIRP                        Ptrlrp,
PIO_STACK_LOCATION          PtrloStackLocation,
PFILE_OBJECT                PtrFileObject,
PtrSFsdFCB                  PtrFCB,
PtrSFsdCCB                  PtrCCB)
{
    // Declarations go here . . .

    try {

        // Validate the sent-in FCB
        if ((PtrFCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_VCB) ||
             !(PtrFCB->FCBFlags & SFSD_FCB_DIRECTORY)) {
            //We will only allow notify requests on directories.
            RC = STATUS_INVALID_PARAMETER;
            CompleteRequest = TRUE;
        }

        PtrReqdFCB = &(PtrFCB->NTRequiredFCB);
        CanWait = ((PtrIrpContext->IrpContextFlags &
                          SFSD_IRP_CONTEXT_CAN_BLOCK)
                             ? TRUE : FALSE);
      PtrVCB = PtrFCB->PtrVCB;

        // Acquire the FCB resource shared
```

```
        if ( !ExAcquireResourceSharedLite(&(PtrReqdFCB->MainResource),
                        CanWait)) {
            PostReguest = TRUE;
            try_return(RC = STATUS_PENDING) ;
        }
      AcquiredFCB = TRUE;

        // Obtain some parameters sent by the caller
        CompletionFilter =
           PtrIoStackLocation->Parameters.NotifyDirectory.CompletionFilter;
        WatchTree = (PtrIoStackLocation->Flags
                        & SL_WATCH_TREE ? TRUE : FALSE) ;

        // If you wish to capture the subject context, you can do so as
        // follows:
        // {
        //         PSECURITY_SUBJECT_CONTEXT SubjectContext;
        //          SubjectContext = ExAllocatePool (PagedPool,
        //                                  sizeof (SECURITY_SUBJECT_CONTEXT) ) ;
        //        SeCaptureSubjectContext (SubjectContext) ;
        //    }

        FsRtlNotifyFullChangeDirectory(&(PtrVCB->NotifyIRPMutex),
                            &(PtrVCB->NextNotifyIRP)    ,
                             (void *)PtrCCB,
                             (PSTRING)(PtrFCB->FCBName->ObjectName.Buffer),
                            WatchTree, FALSE, CompletionFilter, Ptrlrp,
                            NULL,       // SFsdTraverseAccessCheck( . . . ) ?
                            NULL) ;    // SubjectContext?

        RC = STATUS_PENDING;

        try_exit:    NOTHING;

    } finally {

        if (PostRequest) {
            // Perform appropriate post-related processing here
            if (AcquiredFCB) {
                SFsdReleaseResource(&(PtrReqdFCB->MainResource)  )  ;
                AcquiredFCB = FALSE;
            }
            RC = SFsdPostRequest (PtrlrpContext, Ptrlrp);
        } else if (CompleteRequest) {
            PtrIrp->IoStatus .Status = RC;
            PtrIrp->IoStatus. Information = 0 ;

            // Free up the Irp Context
            SFsdReleaselrpContext(PtrlrpContext);

            // complete the IRP
            loCompleteRequest (Ptrlrp, IO_DISK_INCREMENT) ;
        } else {
            // Simply free up the IrpContext, since the IRP has been queued
```

```
            SFsdReleaselrpContext(PtrlrpContext);
        }

        // Release the FCB resources if acquired.
        if (AcquiredFCB) {
            SFsdReleaseResource(&(PtrReqdFCB->MainResource)                    );
            AcquiredFCB = FALSE;
        }

    }

    return(RC);
}
```

Dispatch Routine: Cleanup

The cleanup dispatch routine entry point is invoked for each file object created as part of a successful create/open request. Therefore, for each create/open operation that succeeds, your FSD will receive a corresponding cleanup request.

Invoking the FSD Cleanup Entry Point

The cleanup entry point is invoked by the NT I/O Manager. Threads executing on the Windows NT platform cannot really invoke the cleanup routine directly; all they can do is open or close handles to file streams, or reference/dereference file objects that are the I/O-Manager-created structures representing open file streams.

In Chapter 4, we discussed file object structures in detail. You may recall that file object structures are managed by the NT Object Manager. One question that may occur to you is how does that Object Manager know about I/O-Manager-defined structures, such as the file object structure?

The answer is: at system initialization time, the NT I/O Manager registers all the different I/O Manager objects (including the file object structure) with the NT Object Manager. The ObCreateObjectType ( ) Object Manager routine is used for this purpose. Although this routine is not exposed by the NT Executive, it serves to make the NT Object Manager aware of a new object type. When invoking this routine, the I/O Manager also supplies the functions that must be invoked by the Object Manager to manipulate the object being defined. For file object structures, the I/O Manager supplies an internal routine called lopClose-File() to be invoked whenever any handle associated with the file object has been closed.

The lopCloseFile () routine is fairly simple in its implementation. It performs the following logical steps whenever invoked (i.e., whenever a handle to the file object is closed).

- If the process closing the handle has other handles open for the file object, the I/O Manager does not do anything, but simply returns.*
- The I/O Manager checks whether it needs to issue a request to the FSD to unlock any byte-range locks obtained by the current process.

  It is possible that the process performing the close handle operation may have requested byte-range locks on the file stream associated with the file object structure. In the next chapter, we'll discuss byte-range locking in more detail, but note for now that the I/O Manager remembers when a process has requested byte-range locks and uses this information when the process is closing the last handle to the file object structure. The I/O Manager then creates a IRP_MJ_LOCK_CONTROL IRP, with a minor function of IRP_MN_UNLOCK_ALL, requesting the FSD to unlock any byte ranges locked by the particular process.t

  Note that the I/O Manager issues this request to unlock all byte ranges previously locked by the process only if other processes in the system still have open handles to the file object structure. If all handles to the file object have been closed, the I/O Manager does not explicitly issue an unlock request, but instead directly issues an IRP_MJ_CLEANUP request. The implicit expectation is that, as part of performing cleanup-related processing, the FSD will unlock any byte-range locks acquired by the current process.
- Now, if all handles to the file object have been closed (the system-wide handle count for the file object is 0), the I/O Manager creates an IRP with a major function of IRP_MJ_CLEANUP and invokes the FSD entry point.

  Note that the I/O Manager does not really care about the results of a cleanup request, except to perform a wait in the context of the thread closing the handle if the FSD returns STATUS_PENDING. Therefore, a cleanup request is an inherently synchronous request. Even if your FSD returns an error code from the cleanup routine, the I/O manager will ignore the error and proceed. Remember that cleanup operations must continue even in the face of errors; there are no second chances here!

---

* There is a handle count associated with a file object structure that is specific to each process that has an open handle to the file object. Also, there is a system-wide handle count, which is the sum of all process-specific handle counts. Here, the I/O Manager checks whether the process-specific handle count is equal to 0, or whether the process still has other open handles for the file object.

t Actually, the I/O Manager will first attempt to use the fast I/O path to issue this request and will use an IRP if the fast I/O method does not work.

Logical Steps Involved

Now that you know how the cleanup routine is invoked in your file system driver, here are the steps that most FSD implementations take in processing such a request:

- Synchronize cleanup requests with create requests by acquiring the volume control block resource exclusively during a cleanup operation.

  This also serializes all cleanup requests for the particular logical volume. If the VCB resource cannot be acquired immediately, the FSD can post the request for asynchronous processing (remember that the I/O Manager will still be waiting for the cleanup operation to complete in the context of the requesting thread).

- Your FSD should also acquire the MainResource for the file object exclusively to synchronize with other user-initiated I/O operations on the file stream (remember that other processes may still have open references/handles to the file stream).

  If you cannot acquire the MainResource for the FCB, you should post the request to be handled asynchronously.

- If the cleanup request is for a regular file and if your FSD supports opportunistic locking, you should invoke the FSRTL-supplied oplock package, informing it about the cleanup request, thereby allowing it to perform any cleanup that it needs to at this time.

- If the cleanup operation is for a directory, the cleanup routine now invokes the FsRtlNotifyCleanup ( ) routine, described earlier, to complete any pending notify IRPs for the file object.

- For cleanup operations on files, the FSD must now unlock any byte-range locks acquired by the process in whose context the cleanup routine was invoked.

- If the file was accessed or modified, your FSD should update appropriate time stamp values for the file stream (in the directory entry for the file stream) at this time. You may also need to update the file size value in the directory entry if it has changed and the directory entry does not reflect the current value for the file stream.

  Note that your FSD may use a different approach to updating time stamp values (e.g., your FSD might update time stamp values at the time when the access/modification actually occurred, in which case you would not need to do anything at cleanup time). If, however, you do change some time stamp values, you should invoke the FSRTL FsRtlNotifyFullReport-

Change ( ) routine once all the modifications have been done locally in the FCB/directory entry for the file stream.

Note that for fast I/O read/write operations, your FSD may not have had the opportunity to update the access/modify/change time stamp values. However, you can determine that such I/O occurred by the presence of the F0_ FILE_FAST_IO_READ flag in the file object structure (indicating that at least one fast I/O read operation was processed), or by the presence of FO_FILE_ MODIFIED (some thread performed a modification, implying that the modification time should be updated), and/or by the presence of the FO_FILE_ SIZE_CHANGED flag (indicating that the file size was changed in the FCB header, and the FSD might need to modify the directory entry for the file stream appropriately).

- If this is the last cleanup request that you expect for the file stream, you should check for any pending file stream truncate requests.

  Remember that with the NT I/O Manager—mandated model for file stream deletion (for directories as well as for regular files), the FSD actually deletes the directory entry for a file stream only when the last user handle has been closed. In this case, if the link count for the file stream is also 0 (if your FSD does not support multiply linked files, then any delete operation will always cause the link count to equal 0), the FSD must also free up all of the on-disk space for the file stream. Therefore, the FSD must do two things if the file has been marked for deletion:

  — Check if the link count is equal to 0. If so, then acquire the PagingloResource exclusively (blocking if you have to in the process), and set the file size and valid data length fields to 0.

    Now, the FSD can release the PagingloResource, and release the on-disk space reserved for the file stream. However, there is one important step that the FSD must perform before actually deallocating the on-disk space reserved for the file stream; the FSD must invoke MmFlushlmage-Section() to ensure that the VMM purges any pages containing mapped data for the file stream.

  — If the current link is being deleted, the FSD should remove the directory entry corresponding to the current link at this time.

    Remember to invoke the FsRtlNotifyFullReportChange ( ) routine to notify pending IRPs about the fact that an entry has been deleted.

- Decrement the OpenHandleCount in the FCB structure.
- Invoke CcUninitializeCacheMap() for all FCB structures, regardless of whether or not caching had been initiated using the file object on which the cleanup is being performed.

If the file stream was truncated, supply the new size for the file stream in the TruncateSize argument to the CcUninitializeCacheMap() function call. This will result in the Cache Manage purging the truncated pages from the system cache.

- Be sure to set the FO_CLEANUP_COMPLETE flag in the Flags field of the file object structure.
- Invoke the I/O Manager routine loRemoveShareAccess() to update the share access associated with the FCB.

  The reason for updating the share access at cleanup time instead of in a close operation, is because the close could theoretically take a very long time to be issued, and it would be unfriendly to prevent fresh user open operations simply because of some stale, conflicting share access value set in the FCB.

  Note that the loRemoveShareAccess() routine accepts two arguments: a pointer to the file object structure on which the cleanup is being performed and a pointer to the unique share access value stored in the FCB structure (the address of the FCBShareAccess field in the sample FSD).

The steps listed here are simply a checklist of items that the FSD is expected to perform as part of processing a cleanup request. Many sophisticated file systems may need to perform additional operations to ensure data consistency on disk. For example, some file systems may make certain guarantees about the validity of the on-disk data once a file handle has been closed, and therefore they will undoubtedly have file-system-specific operations to perform, including flushing file data and/or writing log files to disk at this time.

I have not provided a code fragment for the cleanup or close routine, since they are quite FSD-specific. However, as long as your FSD follows the steps listed here, you should be able to derive such a routine successfully.

Dispatch Routine: Close

Just as a cleanup request will always be issued for a file object structure representing an open file stream, a close request will always be issued for the file object some time after the cleanup has been processed.

The fundamental rule is that the file object is not really closed until the IRP_MJ_CLOSE is received. The IRP_MJ_CLEANUP means that all user file handles for the file object have been closed; however, if the file object has any references pending, then the IRP_MJ_CLOSE will be delayed until the reference count on the file object structure (maintained by the NT Object Manager) is equal to 0.

Invoking the FSD Close Entry Point

Just as is the case for the cleanup request, the close entry point is invoked by the NT I/O Manager. Threads executing on the Windows NT platform cannot directly issue a close request to the FSD; the best they can do is to dereference a file object structure that they might have previously referenced. Note that there are two methods exposed by the Object Manager to reference a file object, and correspondingly, there are two methods to dereference the file object. To reference a file object, a thread can create a file handle for the file object either by opening the file object, (NtCreateFile (), NtOpenFile (), ZwCreateFile ()), or by requesting a new handle from the file object pointer (ObReferenceObjectByHandle ()). Similarly, the thread can also cause the reference count on the file object to be incremented by invoking the ObReferenceObjectByPointer () routine (described in Chapter 5).

To dereference a previously referenced file object, the thread can either close a file handle using ZwClose () or NtClose (), or dereference the file object directly by invoking ObDereferenceObject () on the file object pointer.

Whenever a user closes a file handle, the NT Object Manager invokes the IopCloseFile() routine for the particular file object structure on which that close has been invoked. From the previous discussion, you know that the IopCloseFile() routine could lead to an IRP_MJ_CLEANUP being issued to the FSD. Once the I/O Manager returns control back to the Object Manager, it invokes ObDereferenceObject () internally. The ObDereferenceObject () routine decrements the reference count in the object header by 1, and invokes the delete routine associated with the object, if such a routine has been provided and if the reference count maintained by the Object Manager is equal to 0. In the case of file object structures, the I/O Manager supplies an internal (not exposed) delete routine called lopDeleteFile ().

The lopDeleteFile () routine performs the following operations:

- It creates an IRP with a major function of IRP_MJ_CLOSE and invokes the FSD close dispatch routine.
- Once the FSD returns control from the close dispatch routine, the I/O Manager frees any file name string allocated for the file object structure.
- The I/O Manager closes any completion ports associated with the file object.
- The I/O Manager decrements a reference count on the FSD driver object (the count is incremented by the I/O Manager as part of processing a create/open request, to ensure that the driver can't be unloaded while open file objects are present).

- If the driver reference count is equal to 0 and if the driver had an unload operation pending, the driver is unloaded at this time.

Just as in the case of a cleanup request, the I/O Manager doesn't expect an FSD close routine to return any errors. Also, it's important to note that the I/O Manager doesn't expect close requests to return STATUS_PENDING (i.e., the I/O Manager expects the close operation to be performed synchronously).*

Logical Steps Involved

Here are the steps most FSD implementations take in processing an IRP_MJ_ CLOSE request:

- Obtain a pointer to the FCB and CCB structures.
- Synchronize with other close/create/cleanup requests by acquiring the VCB resource exclusively.
- Delete the CCB structure and free any other associated memory objects.
- If this is the last close operation for the file, delete the in-memory FCB structure as well.

These steps are extremely simplified, although accurate. Close requests can often occur at some inconvenient moments, and it is quite probable that your FSD may not be able to acquire the required resources without blocking. However, blocking a close request will lead to some very unexpected deadlock conditions. Therefore, you should, as a rule, never make a close request block, waiting for resources to be acquired. The best thing to do in such situations is to simply obtain any necessary information from the file object structure, make a local copy of such information, and post a request to perform the close asynchronously. Your FSD can then immediately return success to the caller.

Note that if you do such asynchronous processing of a close request, you should be extremely careful when creating new FCB structures later to ensure that any subsequent create/open operations are well synchronized with the asynchronous delayed close operations. Similarly, if a user requests that a volume be dismounted and if the dismount is pending because of open file streams, your FSD should be able to perform appropriate processing when the last file object is closed for the last time to perform the dismount operation.

Note again that it is not a wise decision to post a close request and if your FSD expects to perform some sophisticated processing during a close operation, you should try to do it asynchronously.

---

* If you have to do something that can't he done synchronously, you will have to perform the operation asynchronously after obtaining whatever context is required from the file object structure; remember, though, that you simply must return STATUS_SUCCESS *to* the I/O Manager.