11

Writing a File System Driver III

In this chapter:

- *Handling Fast I/O*
- *Callback Example*
- *Dispatch Routine: Flush File Buffers*
- *Dispatch Routine: Volume Information*
- *Dispatch Routine: Byte-Range Locks*
- *Opportunistic Locking*
- *Dispatch Routine: File System and Device Control*
- *File System Recognizers*

Before continuing with some of the remaining file system dispatch routines, it would be useful to understand the fast I/O execution path defined by the NT I/O Manager. Because the FSD must provide support for callback routines that allow other NT components to preacquire FSD resources, an example of such a callback routine is provided and discussed in this chapter. Then, we'll discuss some of the remaining FSD dispatch routines that you should become familiar with before designing your file system, including the flush file entry point, get/set volume information support, support for byte-range locks on a file stream, and file system IOCTL support. We'll also see the NT LAN Manager opportunistic locking protocol, which you may wish to support in your FSD. I will conclude this chapter with a short overview of the file system driver load process, implemented by using a file system recognizer module.

Handling Fast I/O

The fast I/O execution path was apparently developed in response to a recognition by NT file system driver and I/O subsystem designers that the normal IRP dispatch mechanism did not meet some of the performance criteria they had set out to achieve. Although it was originally conceived to handle user read/write requests more efficiently, the fast I/O method has evolved to encompass the many different FSD requests that a user could issue, including requests to get or set file information, request byte-range locks, and request device lOCTLs. It has also become somewhat of a catch-all mechanism for issuing requests to pre-

acquire FSD resources, although this does not appear to be part of the original fast I/O design.

Chapter 7, *The NT Cache Manager II,* provides an introduction to the fast I/O method of data access. Refer to that chapter before proceeding with the following discussion.

Why Fast I/O?

Let's recall how a typical file system buffered I/O (read/write) request is handled:

1. First, the I/O Manager creates an IRP describing the request.
2. This IRP is dispatched to the appropriate FSD entry point, where the driver extracts the various parameters that define the I/O request (e.g., the buffer pointer supplied by the caller and the amount of data requested) and validates them.
3. The FSD acquires appropriate resources to provide synchronization across concurrent I/O requests and checks whether the request is for buffered or nonbuffered I/O.
4. Buffered I/O requests are sent by the FSD to the NT Cache Manager.
5. If required, the FSD initiates caching before dispatching the request to the NT Cache Manager.
6. The NT Cache Manager attempts to transfer data to/from the system cache.
7. If a page fault is incurred by the NT Cache Manager, the request will recurse back into the FSD read/write entry point as a paging I/O request.

   You should note, that in order to resolve a page fault, the NT VMM issues a paging I/O request to the I/O Manager, which creates a new IRP structure (marked for noncached, paging I/O) and dispatches it to the FSD. The original IRP is not used to perform the paging I/O.
8. The FSD receives the new IRP describing the paging I/O request and transfers the requested byte range to/from secondary storage.

   Lower-level disk drivers assist the FSD in this transfer.

There were two observations that NT designers made that will help explain the evolution of the fast I/O method:

- Most user I/O requests are synchronous and blocking (i.e., the caller does not mind waiting until the data transfer has been achieved).
- Most I/O requests to read/write data can be satisfied directly by transferring data from/to the system cache.

Once they had made the two observations listed, the NT I/O Manager developers decided that the sequence of operations used in a typical I/O request could be further streamlined to help achieve better performance. Certain operations appeared to be redundant and could probably be discarded in order to make user I/O processing more efficient. Specifically, the following steps seemed unnecessary:

Creating an IRP structure to describe the original user request, especially if the IRP was not required for reuse

Assuming that the request would typically be satisfied directly from the system cache, it is apparent that the original IRP structure, with its multiple stack locations and with all of the associated overhead in setting up the I/O request packet, is not really required or fully utilized. It seems to make more sense to dispense with this operation altogether and simply pass the I/O request parameters directly to the layer that would handle the request.

Invoking the FSD

This may seem a little strange to you but a legitimate observation made by the NT designers was that, for most synchronous cached requests, it seems to be redundant to get the FSD involved at all in processing the I/O transfer. After all, if all that an FSD did was route the request to the NT Cache Manager, it seemed to be more efficient to have the I/O Manager directly invoke the NT Cache Manager and bypass the FSD completely.

This can only be done if caching is initiated on the file stream, so that the Cache Manager is prepared to handle the buffered I/O request.

Becoming Efficient: the Fast I/O Solution

Presumably, after pondering the observations listed here, NT I/O designers decided that the new, more efficient sequence of steps in processing user I/O requests should be as follows:

1. The I/O Manager receives the user request and checks if the operation is synchronous.
2. If the user request is synchronous, the I/O Manager determines whether caching has been initiated for the file object being used to request the I/O operation.*

* The check made by the I/O Manager is simply whether the PrivateCacheMap field in the file object structure is nonnull. This field is set to a nonnull value by the Cache Manager as part of initializing caching for the particular file object structure.

For asynchronous operations, the I/O Manager follows the normal method of creating an IRP and invoking the driver dispatch routine to process the I/O request.

3. If caching has been initiated for the file object as determined in Step 2, the I/O Manager invokes the appropriate fast I/O entry point.

   The important point to note here is that the I/O Manager assumes that the fast I/O entry point must have been initialized if the FSD supports cached file streams. If you install a debug version of the operating system, you will actually see an assertion failure if the fast I/O function pointer is NULL.

   Note that a pointer to the fast I/O dispatch table is obtained by the I/O Manager from the FastIoDispatch field in the driver object data structure.

4. The I/O Manager checks the return code from the fast I/O routine previously invoked.

   A TRUE return code value from the fast I/O dispatch routine indicates to the I/O Manager that the request was successfully processed via the fast I/O path. Note that the return code value TRUE does not indicate whether the request succeeded or failed; all it does is indicate whether the request was processed or not. The I/O Manager must examine the IoStatus argument supplied to the fast I/O routine to find out if the request succeeded or failed.

   A return code of FALSE indicates that the request could not be processed via the fast I/O path. The I/O Manager accepts this return code value and, in response, simply reverts to the more traditional method of creating an IRP and dispatching it to the FSD.

   This point is very important for you to understand. The NT I/O subsystem designers did not wish to force an FSD to have to support the fast I/O method of obtaining data. Therefore, the I/O Manager allows the FSD to return FALSE from a fast I/O routine invocation and simply reissues the request using an IRP instead.

5. If the fast I/O routine returned success, the I/O Manager updates the CurrentByteOffset field in the file object structure (since this is a synchronous I/O operation) and returns the status code to the caller.

The advantage of using the new sequence of operations is that synchronous I/O requests can be processed without having to incur the overhead of either building an IRP structure (and the associated overhead of completion processing for the IRP), or routing the request via the FSD dispatch entry point.

Possible Problems in Bypassing the FSD

Not all file system implementations are alike; as a matter of fact, nearly all file systems have unique characteristics, requirements, and processing needs, specific to the particular implementation. Therefore, although bypassing the FSD completely and directly obtaining data from the Cache Manager appears, on the surface, to be a highly efficient method of data transfer, the following issues must be considered:

Acquiring FSD resources

It would be nice not to have to worry about FSD resources and simply obtain data from the Cache Manager. However, as you well know, the FSD tries to ensure data consistency usually by providing a shared (multiple) reader and single writer model to file system clients. To do this, the FSD typically acquires the MainResource either shared or exclusively, and in some cases (especially if the file size is to be modified), also synchronizes with paging I/O requests by acquiring the PagingloResource exclusively.

Even if the I/O Manager does bypass the FSD dispatch entry point when performing fast I/O, appropriate FSD resources should always be somehow acquired.

Presence of byte-range locks

This is a very obvious problem in the implementation and support of fast I/O routines that bypass the FSD dispatch entry points. In Chapter 9, *Writing a File System Driver /*, the code fragments presented for read/write operations noted that the FSD dispatch entry points always check to see whether the caller should be allowed to proceed with the I/O operation, or whether the operation should be denied, because some or all of the byte range being accessed/modified has a byte-range lock associated with it.

Since the typical Windows NT byte-range locking model implements mandatory byte-range locks, such checks should also be performed in the fast I/O case. The other alternative is to prevent fast I/O operations if the file stream has any byte-range locks associated with it.

Opportunistic locks

Opportunistic locking support is discussed in greater detail later in this chapter. However, just as in the case of byte-range locks, the FSD may wish to be careful about allowing fast I/O operations to proceed, depending on the state of the oplocks associated with the file stream.

Other FSD-specific issues

Consider a file system that must perform certain preprocessing before allowing file write operations to proceed on the file stream. For example, certain distributed file systems (e.g., DPS) may employ token-based or other

similar methods of ensuring data consistency across geographically dispersed nodes. For such complex file system implementations, the FSD may not allow fast I/O support without ensuring that the requisite preprocessing has been performed.

The first three concerns listed here can be placed into two categories:

- Ensuring acquisition of file system resources for the file stream being accessed
- Allowing the FSD to determine whether fast I/O should be allowed to proceed on a file stream or not

NT I/O subsystem designers seem to have thought through these issues and have provided support to FSD designers to address such problems. The solutions include providing generic fast I/O intermediate routines in the FSRTL package that always acquire appropriate FSD resources, and also allowing the FSD to specify, on a per-file-stream basis, whether fast I/O should be allowed for the file stream.

However, for more complex FSD implementations that always need to perform preprocessing before allowing any sort of I/O to proceed, the FSD designer must devise an FSD-specific method to also allow fast I/O access to file streams. There is no easy solution in such a situation.

Ensuring Correct FSD Resource Acquisition

You should either initialize the fast I/O dispatch routine function pointers in the fast I/O dispatch table to point to *intermediate* routines in your driver that perform appropriate FCB acquisition, or use the Windows NT FSRTL-provided generic routines instead.

In the sample FSD initialization code presented in Chapter 9, you will notice that I have initialized the fast I/O dispatch routine function pointers to sample FSD-provided routines (e.g., SFsdFastIoRead(), SFsdFastloWrite ( ), and so on). The theory is that these intermediate routines will not allow the I/O Manager to bypass the FSD completely, but instead will perform any required pre-processing, such as resource acquisition, before passing the request on to the Cache Manager (if appropriate).

You will find that this is still a lower overhead I/O operation (even though the FSD is not being completely bypassed) than the corresponding IRP-based I/O operation.

There are also some FSRTL-provided intermediate support routines that perform appropriate FSD resource acquisition and forward the fast I/O request to the NT Cache Manager. The two most widely used (and Microsoft recommended) are the FsRtlCopyRead() and FsRtlCopyWrite ( ) utility functions described later. If

you decide to use these functions, you should understand the assumptions made by them and the nature of the processing that they perform. Some file systems that have a lot of complex preprocessing required before they forward a request to the Cache Manager may wish to use a combination of their own fast I/O dispatch routines and the FSRTL-provided functions (one way of doing this is to have your FSD's fast I/O dispatch routine perform appropriate preprocessing, and *then* invoke the FSRTL routine).

Allowing Fast I/O on a File Stream

You must set the IsFastIoPossible field, in the CommonFCBHeader for the file stream, appropriately. You should also provide a callback function and initialize the FastIoCheckIfPossible function pointer field in the CommonFCBHeader to invoke this callback function when required.

One of the methods for an FSD to disable the fast I/O method for a specific file stream is by initializing the IsFastIoPossible field in the CommonFCBHeader to FastIoIsNotPossible. The other method is to set IsFastIoPossible to FastIoIsQuestionable, and then, after appropriate processing, return FALSE from the FastIoCheckIfPossible() function callback invocation.

Here are the three enumerated type values the IsFastIoPossible field can contain:

- FastIoIsPossible (enumerated type value = 0)
- FastIoIsNotPossible (enumerated type value = 1)
- FastIoIsQuestionable (enumerated type value = 2)

The FastIoIsNotPossible value results in fast I/O being disabled for the particular file stream until the contents of the IsFastIoPossible field are changed.

If the IsFastIoPossible field is initialized to FastIoIsPossible, the intermediate routine (whether your own or that provided by the FSRTL) proceeds with fast I/O processing for the request. If, however, the IsFastIoPossible field is initialized to FastIoIsQuestionable, then the FSRTL-provided intermediate routine issues a callback to the FSD to determine whether the fast I/O operation should be allowed to proceed or not (your internal intermediate routine can follow the same model). The callback function must be provided by the FSD and the callback function address must be initialized in the FastIoCheckIfPossible field of the fast I/O dispatch table (the sample FSD initializes this value to the SFsdFastIoCheckIfPossible() function address).

The FSD can determine, in the callback routine, whether the fast I/O operation should be allowed to proceed. If the FSD returns FALSE from the FastIoCheckIfPossible field, the FSRTL-provided intermediate routines (and also your own) will stop processing the fast I/O request and return FALSE to the NT I/O Manager; otherwise, the intermediate function will continue with processing the fast I/O request (since the FSD has essentially granted permission for the current fast I/O operation to proceed).

The following code fragment illustrates the implementation of a typical FastIoCheckIfPossible function callback implementation:

```
BOOLEAN SFsdFastIoCheckIfPossible(
IN PFILE_OBJECT                 FileObject,
IN PLARGE_INTEGER               FileOffset,
IN ULONG                        Length,
IN BOOLEAN                      Wait,
IN ULONG                        LockKey,
IN BOOLEAN                      CheckForReadOperation,
OUT PIO_STATUS_BLOCK            IoStatus,
IN PDEVICE_OBJECT               DeviceObject)
{
    BOOLEAN                     ReturnedStatus = FALSE;
    PtrSFsdFCB                  PtrFCB = NULL;
    PtrSFsdCCB                  PtrCCB = NULL;
    LARGE_INTEGER               IoLength;

    // Obtain a pointer to the FCB and CCB for the file stream.
    PtrCCB = (PtrSFsdCCB)(FileObject->FsContext2);
    ASSERT(PtrCCB);
    PtrFCB = PtrCCB->PtrFCB;
    ASSERT(PtrFCB);

    // Validate that this is a fast I/O request to a regular file.
    // The sample FSD, for example, will not allow fast I/O requests
    // to volume objects or to directories.
    if ((PtrFCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_VCB) / /
         (PtrFCB->FCBFlags & SFSD_FCB_DIRECTORY)) {
        // This is not allowed.
        return(ReturnedStatus);
    }

    IoLength = RtlConvertUlongToLargeInteger(Length);

    // Your FSD can determine the checks that it needs to perform.
    // Typically, an FSD will check whether there are any byte-range
    // locks that would prevent a fast I/O operation from proceeding.

    // ... (FSD specific checks go here).

    if (CheckForReadOperation) {
        // It would be nice to be able to use the FSRTL's services
        // for file lock operations. However, this chapter describes how
```

```
        II to design and implement your own file lock support routines.
        // Check here whether or not the read I/O can be allowed.
        ReturnedStatus = SFsdCheckLockReadAllowed(&(PtrFCB->
                                                    FCBByteRangeLock) ,
                                FileOffset, &IoLength, LockKey, FileObject,
                                PsGetCurrentProcess());

    } else {
        // This is a write request. Invoke the appropriate support routine
        // to see if the write should be allowed to proceed.
        ReturnedStatus =
            SFsdCheckLockWriteAllowed(&(PtrFCB->FCBByteRangeLock),
                                FileOffset, &IoLength, LockKey, FileObject,
                                PsGetCurrentProcess());
    }

    return(ReturnedStatus);
}
```

A legitimate question that you should have is, when should you modify/update the IsFastIoPossible field in the CommonFCBHeader?

The answer is — it depends. You should initialize the field when creating the FCB for the file stream, which is when the first open operation is performed on the file stream. Subsequent updates should always be made after acquiring the MainResource for the file stream exclusively. Typically, if byte-range locks have been granted on the file stream, or if opportunistic locks have been granted such that they would prevent fast I/O access, then you should set the IsFastIoPossible field value to either FastIoIsNotPossible or FastIoIsQuestionable.

A common method that sets the IsFastIoPossible field is shown in this pseudocode fragment:

```
if ((no opportunistic locks have been granted for the file stream) ||
     (if the caller has an exclusive opportunistic lock on the stream) ||
     (if my FSD-specific checks tell me that fast I/O is not a good
          idea)) {
    if ((there are any byte-range file locks) ||
         (if my FSD-specific checks tell me that fast I/O is
              questionable)) {
            // Force the FSD to be queried for permission before fast
            // I/O is allowed to proceed.
            IsFastIoPossible = FastIoIsQuestionable;
    } else {
        // Fast I/O seems safe at this time.
        IsFastIoPossible = FastIoIsPossible;
    }
} else {
    // Allowing fast I/O would not be a good idea. Force the IRP route
    // instead.
    IsFastIoPossible = FastIoIsNotPossible;
}
```

Note that there are no set rules that an FSD must follow in determining whether to allow fast I/O operations or not; the issue is highly FSD-specific. If, however, you do plan to use the methodology presented here, as opposed to simply refusing fast I/O outright, then there are a multitude of occasions during file system execution that you will have to execute the fragment and reevaluate if fast I/O should be allowed to proceed without question, allowed on a per-occasion basis, or never allowed.

The specific occasions on which you should reevaluate the status of fast I/O for a specific file stream include the following:

- At file stream open time
- Whenever read or write requests are dispatched to the file system
- Whenever byte-range lock/unlock requests are processed by the FSD
- Whenever file stream attributes are modified via a set file information request
- Whenever opportunistic locks are granted/broken
- At file stream cleanup
- For removable media, whenever a volume needs to be reverified due to media change

Of course, your FSD may have some very specific situations, in addition to those listed, when it may need to reevaluate the status of fast I/O vis-a-vis a specific file stream.

FSRTL Support for Fast I/O

The NT I/O subsystem designers recommend that FSD implementations use FSRTL-supplied routines to perform appropriate preprocessing (including acquiring FSD resources), before invoking the NT Cache Manager to complete a fast I/O read/write request. Specifically, the following generic support routines have been provided:*

- FsRtlCopyRead()
- FsRtlCopyWriteO

There are other fast I/O support routines that the NT FSRTL provides (e.g., FsRtlQueryBasicInformation(), FsRtlQueryStandardInformation(), and so on). The NT IPS kit lists all of the fast I/O support routines that

---

* The native NT file system implementations follow recommendations and use these FSRTL routines to perform fast I/O related preprocessing. Therefore, during file system initialization, they initialize the FastIoRead and FastIoWrite (function pointer) fields in the fast I/O dispatch table with FsRtlCopyRead() and FsRtlCopyWrite(), respectively.

your FSD can use. We will discuss the two I/O-related routines in greater detail here, because they encapsulate some of the most complex processing related to fast I/O support. The FSRTL routines provided for file-lock support are also discussed later in this chapter.

In the initialization code for the sample FSD implementation provided in Chapter 9, you will have noticed that the FastIoRead function pointer is initialized to SFsdFastIoRead(), and the FastIoWrite() function pointer is initialized to SFsdFastIoWrite(). This is not in keeping with the recommendation made by the NT I/O subsystem designers that these function pointers should be directly initialized to FsRtlCopyRead() and FsRtlCopyWrite(). The reason for not following these recommendations is simply to illustrate to the reader that it is possible for more complex file system implementations to perform any required pre-processing in their own routines (e.g., SFsdFastIoRead() for the sample FSD implementation), and then invoke the appropriate FSRTL routine directly from the FSD fast I/O function. This method is especially useful for more complex file system implementations such as distributed/networked file system drivers.

Of course, for your FSD implementation, you may choose to initialize the function pointers with the appropriate FSRTL routines directly.

The FsRtlCopyRead() function is defined as follows:

```
BOOLEAN
FsRtlCopyRead (
IN PFILE_OBJECT             FileObject,
IN PLARGE_INTEGER           FileOffset,
IN ULONG                    Length,
IN BOOLEAN                  Wait,
IN ULONG                    LockKey,
OUT PVOID                   Buffer,
OUT PIO_STATUS_BLOCK        loStatus,
IN PDEVICE_OBJECT           DeviceObject
);
```

The arguments accepted by the FsRtlCopyRead() function match those required in the function type definition for a fast I/O read function defined in the NT DDK. Notice that all of the relevant parameters supplied by the user thread when invoking the NtReadFileO system service routine are passed directly to the fast I/O (FSRTL) read routine instead of being inserted into an IRP structure.*

Functionality Provided:

The FsRtlCopyRead() routine executes the following steps:

* Although I did not talk about the LockKey user-supplied argument in Chapter 9 when discussing read/write dispatch entry point implementations, note for now that it is possible for a user to read/write a locked byte range if the locker had associated a key with the byte-range lock, and if the reader/writer knows the key value. Byte-range locks are discussed in greater detail later in this chapter.

- It attempts to acquire the MainResource for the file stream shared.

  In case you are wondering how the FSRTL can get to the FCB MainResource pointer, remember that the FsContext field in the file object structure is always initialized to point to a common FCB header structure of type FSRTL_COMMON_FCB_HEADER. This structure contains the Resource field, which is initialized by the FSD to the address of the MainResource (ERESOURCE type) structure.

  If the caller is not prepared to block (i.e., the Wait argument has been set to FALSE), and if the MainResource cannot be acquired immediately without blocking, the FSRTL routine will simply return FALSE. The I/O Manager will then reissue the read request to the FSD via the traditional IRP method.

- If the IsFastIoPossible field in the CommonFCBHeader is set to FastIoIsNotPossible, the FsRtlCopyRead() routine returns FALSE to the I/O Manager.

- If the IsFastIoPossible field in the CommonFCBHeader is set to FastIoIsQuestionable, the FsRtlCopyRead() routine queries the FSD (as described earlier in this chapter) whether it should proceed with fast I/O or return FALSE to the caller.

- Once the FsRtlCopyRead() has determined that it is safe to proceed, it invokes the `CcCopyRead()/CcFastCopyRead()` function to transfer data to/from the system cache.

  The FSRTL is careful about setting itself as the top-level component for the request. It sets the TopLevelIrp field in the TLS to the FSRTL_FAST_IO_TOP_LEVEL_IRP constant value. Once the read operation has completed, the FsRtlCopyRead() function sets the FO_FILE_FAST_IO_READ flag in the file object structure.

- The FsRtlCopyRead() function releases the MainResource for the file stream and returns TRUE to the I/O Manager.

  The I/O Manager performs appropriate postprocessing (described earlier in this chapter) and returns control to the caller.

The FsRtlCopyWrite() function is defined as follows:

```
BOOLEAN
FsRtlCopyWrite (
IN PFILE_OBJECT          FileObject,
IN PLARGE_INTEGER        FileOffset,
IN ULONG                 Length,
IN BOOLEAN               Wait,
IN ULONG                 LockKey,
IN PVOID                 Buffer,
OUT PIO_STATUS_BLOCK     IoStatus,
IN PDEVICE_OBJECT        DeviceObject
);
```

Functionality Provided:

The FsRtlCopyWrite ( ) routine executes the following steps:

- If the file object has been opened with write-through specified, or if the Cache Manager CcCanlWrite () function call returns FALSE, this routine returns FALSE immediately.

  The I/O Manager will reissue the write request via the normal IRP method.

- The FsRtlCopyWrite ( ) routine acquires the FCB MainResource either shared or exclusive.

  This routine acquires the MainResource shared, unless the caller wishes to append to the file stream, or if the write will extend the valid data length for the file stream. If the FsRtlCopyWrite () routine cannot acquire the MainResource immediately and if Wait is set to FALSE, this routine returns FALSE to the NT I/O Manager.

- A check is made to determine whether fast I/O write should even be attempted.

  Just as in the case of the FsRtlCopyRead() routine, this function invokes the FSD to make the final determination on whether fast I/O should be attempted if IsFastloPossible is set to FastloIsQuestionable.

- The FsRtlCopyWrite () routine also returns FALSE immediately to the I/O Manager if the file is being extended such that the new file size would exceed the current allocation size for the file stream, or if the new file size results in a wrap-around of the allocation size for the file stream from a 32-bit value to a 64-bit value.*

- If the file size is being extended, the FsRtlCopyWrite () routine will acquire the PagingloResource exclusively, modify the file size in the CommonFCBHeader, and release the paging I/O resource.

- A CcZeroData ( ) is performed, if required (i.e., if the current write operation results in a hole between the current valid data length before the new write operation was attempted and the starting offset of the new write request).

* There are valid reasons for these checks. First, allowing a write to proceed without having adequate disk space preallocated could result in an unexpected out-of-disk-space error code being returned during a subsequent lazy-write/modified page write operation; this could even happen well after a user process had closed the file handle and exited, expecting that all of the data had made it (or would.) to secondary storage. Second, some file systems (e.g., FASTFAT) do not currently support 64-bit file sizes, while others (e.g., NTFS) do; therefore, the FSRTL package is unsure whether to allow such file I/O operations to proceed or not.

- The FsRtlCopyWrite() request issues a CcCopyWrite ()/CcFastCopy-Write ( ) request to actually transfer the data to the system cache.

  Just as in the case of the fast I/O read operation, the FsRtlCopyWrite () routine is careful to mark itself as the top-level component for the write request.

- Once the write operation has completed, the FsRtlCopyWrite ( ) routine marks the fact that a fast I/O write operation was performed by setting the FO_FILE_MODIFIED flag in the file object structure.

  If the file was extended, or if valid data length was changed, the routine also sets the FO_FILE_SIZE_CHANGED flag in the file object structure.

- The FsRtlCopyWrite () function releases the MainResource for the file stream and returns TRUE to the I/O Manager.

  The I/O Manager performs appropriate postprocessing (described earlier in this chapter) and returns control to the caller.

Rolling Your Own Fast I/O Routine

Now that you understand the methodology used by the FSRTL in providing generic fast I/O read/write support routines, you should be able to easily replace them with your own if required, and also supplement them with appropriate routines to support the other fast I/O entry points.

There are a couple of issues you should keep in mind when developing your own fast I/O support routines:

- It would be prudent for your driver to provide appropriate exception handling in your fast I/O routines.

  The FsRtlCopyRead() and the FsRtlCopyWrite ( ) functions do provide exception handlers since it is quite possible for a malicious user thread (or even a carelessly written application) to send in an invalid buffer, or to deallocate the buffer while the I/O is in progress using another thread, or to change the buffer permissions in such a way so as to cause an access violation error condition when the data transfer is attempted by the Cache Manager. Failure on your part to provide an exception handler could cause the system to crash.

- Your routine should encapsulate the fast I/O support within FsRtlEnter-FileSystemO and FsRtlExitFileSystem() calls.

  This is simply a reminder to you that, just as in the case of the regular IRP dispatch routines, your FSD should not allow kernel-mode APCs to be delivered while executing file system code. This will prevent nasty priority inversion situations, which could lead to a system deadlock.

NOTE The `FsRtlEnterFileSystem()` macro is simply defined to KeEnterCriticalRegion(), while the FsRtlExitFileSystem() macro is defined to KeLeaveCriticalRegion ().

Also remember that the fast I/O path started off as a more efficient method to transfer data; if you find that certain situations would result in your fast I/O routine having to perform an inordinate amount of extraneous processing simply to support this method of data transfer, it could be more efficient to just return FALSE from the fast I/O routine, since the I/O Manager will then issue a regular IRP-based request back to your driver.

The Pseudo Fast I/O Routines

You may have rightly noticed that the fast I/O dispatch table contains entries such as FastloQueryBasicInfo, FastloQueryStandardlnfo, and others that do not quite follow the original fast I/O model of bypassing the FSD and obtaining data from the NT Cache Manager. As explained earlier, there were two goals that the fast I/O method was designed to accomplish: avoiding the overhead associated with the creation and completion of an IRP structure and attempting to obtain data directly from the best source for the data, the NT Cache Manager.

The basic design goal for the fast I/O method is to achieve faster (better) performance. To achieve this goal, NT I/O subsystem designers seem to be providing fast entry points for some of the most frequently used FSD entry points. This is the reason behind the inclusion of most of the (non-I/O) fast I/O entries, including those previously listed.

There are also certain callbacks that have been lumped together with the regular fast I/O entry points in the fast I/O dispatch table, simply because the table seemed like a good, extensible container for these callback routines. Here are the specific callbacks:

- `AcquireFileForNtCreateSection` and `ReleaseFileForNtCreateSection`
- `FastloDetachDevice`
- `AccjuireForModWrite` and `ReleaseForModWrite`
- `AccfuireForCcFlush` and `ReleaseForCcFlush`

Only the first pair of callbacks, acquire/release for create section, existed in Windows NT Version 3.51. The others have been added with Version 4.0.

I presume it's harder for the NT designers to justify the inclusion of these callbacks in the fast I/O dispatch table. The only rational explanation for including them where they currently reside is that these seem to be last-minute solutions to synchronization/deadlock-related problems encountered during late testing, and the only extensible place where such callbacks could possibly reside, without breaking existing file system drivers, seemed to be the fast I/O dispatch table.

NOTE Recall from earlier chapters that the fast I/O dispatch table contains a field called SizeOfFastIoDispatch, which is initialized by an FSD to the size of the structure it knows about (when the driver was implemented). Since new fast I/O entry points are always added at the end of the dispatch table (thereby increasing its size), it is relatively easy for the caller of a fast I/O routine to check whether the underlying FSD knows about the new entries, by comparing the size of the dispatch table with the new entries in it to the size value initialized by the FSD. If the FSD specifies a size that -would include the particular fast I/O entry, the caller can proceed with the fast I/O operation; otherwise, the caller can assume that it is dealing with an older driver and simply skip the particular fast I/O call.

Unfortunately, this isn't a method your driver can use to skip fast I/O support altogether, since a basic assumption made by the NT I/O Manager is that your driver at least knows the initial fast I/O table, introduced with Version 3.51 of the operating system.

AcquireFileForNtCreateSection/ReleaseFileForNtCreateSection

To map a file stream into its virtual address space, a process must first create a section object for the file by invoking the NtCreateSection() system call.* This call is provided by the NT VMM, which performs all of the required processing to create the appropriate image/data section object for the caller.

The process requesting the create section operation specifies the length (in bytes) of the section object to be created. As part of processing the request, the VMM must query the FSD for the current file size associated with the file stream, and modify the file size as well, if the requested length is greater than the current end-of-file position. There are other operations that the VMM must perform, which could also cause the VMM to issue I/O requests to the underlying FSD managing the mounted logical volume on which the file stream resides.

When issuing file system get/set file size requests, the MmExtendSection () internal routine in the VMM acquires certain VMM resources, in order to synchro-

---

* This routine (actually the kernel-equivalent, ZwCreateSection()) is explained in detail in Chapter 5, *The NT Virtual Memory Manager.*

nize with other threads trying to perform another create section operation concurrently.

Unfortunately, though, it is still quite possible for another user thread to concurrently issue a cached read request for which the file system initiates caching, which, in turn, results in the CcInitializeCacheMap() routine in the Cache Manager possibly invoking the MmExtendSection() internal support routine provided by the VMM.

Similarly, other user threads trying to change the file size concurrently could invoke the set file information dispatch routine in the FSD; the FSD, in turn, would issue a CcSetFileSizes() request, and the Cache Manager would possibly invoke the MmExtendSection() routine internally.

Here, the stage is being set for a classic deadlock situation. For the thread performing the create section request, the VMM has acquired some global internal resources preventing other concurrent operations that could possibly result in any modifications to the section object. Then, the VMM invokes the FSD get/set file information entry point. As part of processing this request, the FSD attempts to acquire the MainResource exclusively, and later, tries to acquire the PagingloResource. However, the FSD can be forced to block when attempting the acquisition of the MainResource if some other thread either performing cached I/O or changing the file length acquired it first.

The thread performing a cached I/O or file size modification operation would, in turn, be blocked in the VMM on the same resource that the MmExtendSection() routine acquired to prevent concurrent modifications to the file stream size.

The result is deadlock; the reason is simply because the VMM broke the resource acquisition hierarchy of acquiring FSD resources for the file object *first,* before acquiring its internal resources.

After the NT I/O subsystem designers encountered this problem, they added the two callbacks to the fast I/O dispatch table. Now the VMM invokes the FSD AcquireForNtCreateSection() callback before acquiring its internal resources when processing a create section request. After all of the processing requiring interaction with the FSD has been completed, the VMM invokes the ReleaseForNtCreateSection() callback, to request the FSD to release FCB resources.

Here is the code fragment illustrating the implementation of the AcquireForNtCreateSection and ReleaseForNtCreateSection in the sample FSD:

```
void SFsdFastIoAcqCreateSec(
IN PFILE_OBJECT             FileObject)
{
```

```
    PtrSFsdFCB                PtrFCB = NULL;
    PtrSFsdCCB                PtrCCB = NULL;
    PtrSFsdNTRequiredFCB      PtrReqdFCB = NULL;

    // Obtain a pointer to the FCB and CCB for the file stream.
    PtrCCB = (PtrSFsdCCB)(FileObject->FsContext2);
    ASSERT(PtrCCB) ;
    PtrFCB = PtrCCB->PtrFCB;
    ASSERT(PtrFCB);
    PtrReqdFCB = &(PtrFCB->NTRequiredFCB);

    // Acquire the MainResource exclusively for the file stream
    ExAcquireResourceExclusiveLite(&(PtrReqdFCB->MainResource), TRUE);

    // Although this is typically not required, the sample FSD will
    // also acquire the PagingloResource exclusively at this time
    // to conform with the resource acquisition described in the set
    // file information routine.
    ExAcquireResourceExclusiveLite(&(PtrReqdFCB->PagingIoResource),   TRUE);;

    return;
}

void  SFsdFastloRelCreateSec(
IN PFILE_OBJECT            FileObject)
{

    PtrSFsdFCB                PtrFCB = NULL;
    PtrSFsdCCB                PtrCCB = NULL;
    PtrSFsdNTRequiredFCB      PtrReqdFCB = NULL;

    // Obtain a pointer to the FCB and CCB for the file stream.
    PtrCCB = (PtrSFsdCCB)(FileObject->FsContext2);
    ASSERT(PtrCCB) ;
    PtrFCB = PtrCCB->PtrFCB;
    ASSERT(PtrFCB);
    PtrReqdFCB = &(PtrFCB->NTRequiredFCB);

    // Release the PagingloResource for the file stream
    SFsdReleaseResource(&(PtrReqdFCB->PagingIoResource));

    // Release the MainResource for the file stream
    SFsdReleaseResource(&(PtrReqdFCB->MainResource));

    return;
}
```

The FastloDetachDevice callback will be covered in the next chapter when we discuss filter driver design and implementation.

AcquireForModWrite/ReleaseForModWrite

Before NT Version 4.0 was released, this callback did not exist. As discussed in detail in earlier chapters, it is extremely important for the NT VMM, the NT Cache

Manager, and the FSD implementations to ensure that resources are acquired in the correct order. This callback exists precisely to ensure that the resource acquisition hierarchy is maintained.

In Chapter 5, we discussed the design and philosophy of the modified/mapped page writer threads used by the NT VMM to asynchronously flush dirty pages, allowing the VMM to reuse these pages for other applications. When an asynchronous I/O request is issued to the FSD, the file system implementation may need to acquire the MainResource and/or the PagingloResource. To pre-acquire the appropriate resources and maintain the locking hierarchy across modules, the NT VMM issues a call to the FSRTL FsRtlAcquireFileForModWrite() support routine.

In Windows NT Version 3-51, the FSRTL routine simply acquired the file resources directly. In order to determine which resource to acquire (MainResource or PagingloResource) and if the resource needed to be acquired shared or exclusively, the FSRTL package depended on the following flag values set by the FSD in the CommonFCBHeader associated with the file stream:

```
#define FSRTL_FLAG_ACQUIRE_MAIN_RSRC_EX (0x08)
#define FSRTL_FLAG_ACQUIRE_MAIN_RSRC_SH (0x10)
```

If the flag FSRTL_FLAG_ACQUIRE_MAIN_RSRC_EX is set by an FSD in the CommonFCBHeader for the file stream, the FsRtlAcquireFileForModWrite() routine acquires the MainResource exclusively; a flag value of FSRTL_FLAG_ACQUIRE_MAIN_RSRC_SH results in the routine acquiring the MainResource shared. If neither flag is set, the routine acquires the PagingloResource shared if the a resource is present. Finally, in the most degenerate case of no flag having been set and the PagingloResource pointer in the CommonFCBHeader being NULL, the routine does not acquire any resource at all.

The fundamental rule that an FSD is supposed to follow in setting appropriate flag values is that the flag value cannot be changed unless the FSD acquired both resources before attempting the change; or in other words, if the FSRTL package managed to acquire either of the two resources, it is guaranteed that the flag value would stay constant.

You may wish to note that the FASTFAT file system does not appear to set any flag values at all in Version 3.51, (preferring to rely on the default behavior instead), and the only native FSD implementation that seems to care about these flag values and actively modify them is the NTFS file system. Furthermore, it should not surprise you to know the FsRtlAcquireFileForModWrite() jumps through a lot of hoops to acquire the right resource. It initially examines the flag values in an unsafe fashion and attempts to acquire the designated resource (without waiting). Once a resource is acquired, it reexamines the flag

values—since they could have changed between the time they were examined in an unsafe fashion and when the resource was actually acquired—and retries the resource acquisition after releasing the original resource, if the flag values have changed. All of this is done within a `while` (TRUE) { . . . } loop construct.

There were other problems with this implementation as well. It was sometimes possible for the VMM to want to acquire the FSD resource for write operations that would extend the valid data length. Unfortunately, if the FSD indicated that the MainResource should be acquired shared, following the FSD's instructions possibly leads to a deadlock situation when the write request is actually dispatched to the FSD. Therefore, the `FsRtlAcquirefFileForModWrite()` routine checks for the condition where the ending offset (starting-offset + write-length -1) exceeds the current valid data length, and internally ignores the FSD's instructions, preferring instead to acquire the MainResource exclusively.

It appears as though with Version 4.0 of the operating system, the I/O subsystem designers have realized just how messy, and FSD-dependent, the preceding implementation is.* Therefore, they implemented the AcquireForModWrite () callback, invoked by the `FsRtlAcquireFileForModWrite()` routine. Your FSD should acquire the appropriate resources in response to the callback and also return a pointer to the resource acquired in the ResourceToRelease argument passed in to your callback. The ReleaseForModWrite () callback will be invoked later by the VMM and your FSD can use the ResourceToRelease argument to determine which resource should be released.t

AcquireForCcFlush/ReleaseForCcFlush

This callback was added with Windows NT Version 4.0. It supports invocations to CcFlushCache () for a file stream by a component other the FSD. As described in Chapter 8, *The NT Cache Manager III,* the CcFlushCache () routine can be invoked (by an FSD) with driver resources either acquired exclusively, or left unowned. However, if the routine is invoked by a component other than an FSD, the potential for deadlock exists if FSD resources are not acquired before Cache Manager or VMM resources.

---

* The older method implicitly places a lot of faith in the FSRTL's judgment of what is the correct action to take under the different scenarios in which the routine can he invoked. This is not a particularly extensible policy, especially with the development of third-party file system implementations whose requirements could he very different from what the FSRTL expects. Therefore, letting the FSD determine what to do in response to the VMM request to preacquire resources is a step in the right direction.

t There is an additional benefit to having a callback into your FSD. You can now safely determine the thread ID of the modified/mapped page writer thread when the AcquireForModWrite () callback is issued and store it in the FCB, if you need such information.

Your FSD should ensure that appropriate resources have been acquired to support a subsequent paging I/O, synchronous write operation that will presumable soon follow.

Callback Example

In addition to the fast I/O dispatch routines and the fast I/O callbacks to preacquire FSD resources, the FSD also provides callbacks specifically for the use of the NT Cache Manager read-ahead thread and the lazy-writer thread. A pointer to an initialized CACHE_MANAGER_CALLBACKS structure is passed in by the FSD when invoking the CcInitializeCacheMap() routine (described earlier in Chapter 8). The callback's structure is defined as follows:

```
typedef struct _CACHE_MANAGER_CALLBACKS {
    PACQUIRE_FOR_LAZY_WRITE     AcquireForLazyWrite;
    PRELEASE_FROM_LAZY_WRITE    ReleaseFromLazyWrite;
    PACQUIRE_FOR_READ_AHEAD     AcquireForReadAhead;
    PRELEASE_FROM_READ_AHEAD    ReleaseFromReadAhead;
} CACHE_MANAGER_CALLBACKS, *PCACHE_MANAGER_CALLBACKS;
```

where:

```
typedef
BOOLEAN (*PACQUIRE_FOR_LAZY_WRITE) (
    IN PVOID        Context,
    IN BOOLEAN      Wait
);

typedef
VOID (*PRELEASE_FROM_LAZY_WRITE) (
    IN PVOID        Context
);

typedef
BOOLEAN (*PACQUIRE_FOR_READ_AHEAD) (
    IN PVOID        Context,
    IN BOOLEAN      Wait
);

typedef
VOID (*PRELEASE_FROM_READ_AHEAD) (
    IN PVOID        Context
);
```

The AcquireForLazyWrite and ReleaseFromLazyWrite callbacks are invoked by the NT Cache Manager lazy-writer thread to maintain resource acquisition hierarchy across the Cache Manager and the FSD modules. Similarly, the AcquireForReadAhead and ReleaseFromReadAhead callbacks are invoked by the read-ahead component of the NT Cache Manager.

By now, you should have a very good understanding of the motivating forces behind the design and implementation of these callback functions (i.e., to avoid deadlock situations due to the incorrect sequence of resource acquisitions). Here are examples of the AcquireForLazyWrite and ReleaseFromLazyWrite callback functions for the sample FSD:

```
BOOLEAN SFsdAcqLazyWrite(
IN PVOID                            Context,
IN BOOLEAN                          Wait)
{
    BOOLEAN                ReturnedStatus = TRUE;

    PtrSFsdFCB              PtrFCB = NULL;
    PtrSFsdCCB              PtrCCB = NULL;
    PtrSFsdNTRequiredFCB    PtrReqdFCB = NULL;

    // The context is whatever we passed to the Cache Manager when invoking
    // the CcInitializeCacheMaps( ) function. In the case of the sample FSD
    // implementation, this context is a pointer to the CCB structure.

    ASSERT(Context);
    PtrCCB = (PtrSFsdCCB) (Context);
    ASSERT(PtrCCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_CCB);

    PtrFCB = PtrCCB->PtrFCB;
    ASSERT(PtrFCB);
    PtrReqdFCB = &(PtrFCB->NTRequiredFCB);

    // Acquire the MainResource in the FCB exclusively. Then, set the
    // lazy-writer thread id in the FCB structure for identification when
    // an actual write request is received by the FSD.
    // Note: The lazy-writer typically always sets WAIT to TRUE.
    if (!ExAcquireResourceExclusiveLite(&(PtrReqdFCB->MainResource),
                                                    Wait)) {
        ReturnedStatus = FALSE;
    } else {
        // Now, set the lazy-writer thread id.
        ASSERT(!(PtrFCB->LazyWriterThreadID));
        PtrFCB->LazyWriterThreadID = (unsigned int)(PsGetCurrentThread( ));
    }

    // If your FSD needs to perform some special preparations in
    // anticipation of receiving a lazy-writer request, do so now.

    return(ReturnedStatus);
}

void SFsdRelLazyWrite(
IN PVOID                            Context)
{
    BOOLEAN                ReturnedStatus = TRUE;

    PtrSFsdFCB             PtrFCB = NULL;
```

```
    PtrSFsdCCB              PtrCCB = NULL;
    PtrSFsdNTRequiredFCB    PtrReqdFCB = NULL;

    // The context is whatever we passed to the Cache Manager when invoking
    // the CcInitializeCacheMaps() function. In the case of the sample FSD
    // implementation, this context is a pointer to the CCB structure.

    ASSERT(Context);
    PtrCCB = (PtrSFsdCCB)(Context);
    ASSERT(PtrCCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_CCB);

    PtrFCB = PtrCCB->PtrFCB;
    ASSERT(PtrFCB);
    PtrReqdFCB = &(PtrFCB->NTRequiredFCB);

    // Remove the current thread id from the FCB and release the
    // MainResource.
    ASSERT((PtrFCB->LazyWriterThreadID) ==
                          (unsigned int)PsGetCurrentThread());
    PtrFCB->LazyWriterThreadID = 0;

    // Release the acquired resource.
    SFsdReleaseResource(&(PtrReqdFCB->MainResource));

    // Your FSD should undo whatever else seems appropriate at this time.

    return;
}
```

Typically, the Cache Manager lazy-writer and read-ahead threads always set Wait to TRUE before invoking the FSD callback routines.

Dispatch Routine: Flush File Buffers

The flush file buffers dispatch routine is invoked by a user process to try to ensure that all of the cached information for a file stream or for a group of files has been either written out to secondary storage or flushed across the network to the server node.

Logical Steps Involved

The following logical steps are executed by a file system upon receiving a flush file buffers request:

1. The file system driver must obtain pointers to internal data structures for the object on which the flush file buffers operation has been requested.

   The flush file buffers invocation can be made for three types of objects:

— An open file stream (ordinary file)

— An open directory

— An open volume object representing the mounted logical volume

The FSD typically has different responses for a flush request on each of these object types.

2. If the flush buffers request is for an open file stream, the FSD should typically acquire the FCB exclusively and request that the Cache Manager flush the system cache for the file stream synchronously.

3. If the flush buffers request is on an open directory object, most FSD implementations simply return success without really doing anything.

   The exception to this is if the flush request is made for the root directory of the mounted logical volume. In this case, an FSD should treat the request as if it were a flush request for all open files on the mounted volume. The next step outlines the FSD's response in this situation.

4. If the flush buffers request is made for an open volume object, the FSD should try to flush all open file streams on the mounted logical volume to secondary storage devices.

   Typically, the caller would like to ensure that cached information for modified files residing on the logical volume being flushed is written out to secondary storage before this routine returns control. This is the behavior implemented by the native NT file system drivers as well. Note that a flush buffers request on the root directory is always treated in the same manner as a flush buffers request on the volume object representing a mounted logical volume.

5. Finally, it would be prudent for the FSD to pass the flush file buffers request on to the lower-level disk/network drivers, ensuring that any requests queued there would be processed immediately.

The following pseudocode illustrates how your FSD could implement the flush file buffers dispatch routine. Note that the code assumes the data structures to be those defined by the sample FSD. You can, however, substitute your own data structures (and associated fields) quite easily instead:

```
get pointer to FCB/VCB from file object;
if (VCB) {
    flush the volume;
    (this involves flushing all open file streams (see below for example),
     updating the directory entries, updating timestamp values,
     flushing directories, flushing log files, flushing bitmaps,
     and any other in-memory information that you may wish to write
     to disk)
} else {
```

```
    if (PtrFCB->FCBFlags & SFSD_FCB_ROOT_DIRECTORY) {
        // Treat this exactly the same as a flush volume request.
        flush the volume;
    } else if ( ! (PtrFCB->FCBFlags & SFSD_FCB_DIRECTORY) ) {
        // Flush the file stream from the system cache.
        // Note that the following operation is inherently synchronous;
        // therefore, if the caller did not wish to block, you should
        // have posted the request earlier.
        PtrReqdFCB = &(PtrFCB->NTRequiredFCB) ;
        CcFlushCache(&(PtrReqdFCB->SectionObject) , NULL, 0,
                            &(PtrIrp->IoStatus)) ;
        // Results of the operation are returned by the Cache Manager
        //in the loStatus structure.
        RC = PtrIrp->IoStatus . Status;
        // All done as far as the Cache Manager is concerned.
        // Now, you may wish to update the associated directory
        // entry for the file stream (e.g., with the latest file
        // size, timestamp values, etc.) and flush that to disk.
    }
    // We ignore flush requests for normal directories (just as the
    // native FSD implementations do).
}

// Now that the FSD has completed performing its processing, you
// should forward the flush request to lower-level drivers.
// CAUTION: Some drivers will return STATUS_INVALID_DEVICE_REQUEST
// to you. You should "eat-up" that error and simply return the actual
// status from your flush attempts to the caller. To do this you will
// also have to set a completion routine before invoking the lower-level
// driver.
```

Dispatch Routine: Volume Information

There are two kinds of volume information requests that your FSD should handle:

- Requests to get (query) volume information
- Requests to set (modify) volume information

Let us examine the logical steps involved in processing each of these two types of volume information requests.

Logical Steps Involved

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

```
typedef struct _IO_STACK_LOCATION {

    // ...
```

```
    union {

        //...

        // System service parameters for:  NtQueryVolumelnformationFile
        struct {
            ULONG Length;
            FS_INFORMATION_CLASS FslnformationClass;
        } QueryVolume;

        // System service parameters for:  NtSetVolumelnformationFile
        struct {
            ULONG Length;
            FS_INFORMATION_CLASS FslnformationClass;
        } SetVolume;

        // ...
} Parameters;

// ...

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

The type of volume information request dispatched to an FSD can be determined by examining the major function code contained in the request packet. The two major function codes of interest are IRP_MJ_QUERY_VOLUME_INFORMATION and IRP_MJ_SET_VOLUME_INFORMATION. Of course, your FSD could have separate dispatch routines to handle each kind of volume information request, unlike the sample FSD presented in this book, in which case the appropriate request type would be dispatched to the correct file system driver function.

IRP_MJ_QUERY_VOL UME_INFORMATION

The I/O Manager identifies the kind of information requested in the FS_ INFORMATION_CLASS enumerated type value, supplied in the current stack location of the query volume information IRP. Note that the Windows NT I/O subsystem allows any caller to obtain logical volume information. Furthermore, the caller can supply a handle to any open object associated with the logical volume (i.e., a file object representing an open instance of the logical volume itself, a file object representing an open instance of a file or directory contained in the logical volume, or a file object representing an open instance of the target device on which the logical volume has been mounted).

The following volume information request types should be supported by your **FSD:**

FileFsVolumeInformation (enumerated type value = 1)

The caller expects information about the volume to be returned in the FILE_ FS _VOLUME_ INFORMATION structure:

```
typedef struct _FILE_FS_VOLUME_INFORMATION {
    LARGE_INTEGER           VolumeCreationTime;
    ULONG                   VolumeSerialNumber;
    ULONG                   VolumeLabelLength;
    BOOLEAN                 SupportsObjects;
    WCHAR                   VolumeLabel[1];
} FILE_FS_VOLUME_INFORMATION, *PFILE_FS_VOLUME_INFORMATION;
```

The fields are quite self-explanatory. The serial number is expected to be a unique integer value identifying the mounted logical volume. The volume label can be any string identifier associated with the logical volume. Note that it is possible that the buffer supplied by the caller may not be large enough to contain the entire volume label, in which case your FSD should copy over as much of the label as it can and return a status code of STATUS_BUFFER_OVERFLOW, indicating to the caller that not all of the information could be returned.

FileFsSizeInformation (enumerated type value = 3)

The caller expects information about the volume to be returned in the FILE_FS_SIZE_INFORMATION structure:

```
typedef struct _FILE_FS_SIZE_INFORMATION {
    LARGE_INTEGER       TotalAllocationUnits;
    LARGE_INTEGER       AvailableAllocationUnits;
    ULONG               SectorsPerAllocationUnit;
    ULONG               BytesPerSector;
} FILE_FS_SIZE_INFORMATION, *PFILE_FS_SIZE_INFORMATION;
```

As you can see, the kind of information expected by the caller is fairly generic and your FSD should be able to return some kind of sensible values that can translate into a valid total volume size.*

FileFsDeviceInformation (enumerated type value = 4)

The caller expects to receive information about the type of physical or logical device on which the logical volume has been mounted:

```
typedef struct _FILE_FS_DEVICE_INFORMATION {
    DEVICE_TYPE      DeviceType;
    ULONG            Characteristics;
} FILE_FS_DEVICE_INFORMATION, *PFILE_FS_DEVICE_INFORMATION;
```

The DeviceType field value can be set by your FSD to an appropriate device type. For example, CDFS specifies the DeviceType value as FILE_DEVICE_CD_ROM, while FASTFAT and NTFS use FILE_DEVICE_DISK instead. For a network redirector, the DeviceType field can be set to an appropriate value depending upon the type of connection made. If, for example, the query volume information request is issued using a file object

* For read-only volumes (e.g., for those managed by CDFS), the AvailableAllocationUnits value is set to 0.

representing an open instance of the network redirector itself, the value could well be set to FILE_DEVICE_NETWORK_FILE_SYSTEM.

The Characteristics field should be set to an appropriate value from the following (one or more flag values can be set):

```
// Volume mounted on removable media.
#define FILE_REMOVABLE_MEDIA            0x00000001
#define FILE_READ_ONLY_DEVICE           0x00000002
#define FILE_FLOPPY_DISKETTE            0x00000004
#define FILE_WRITE_ONCE_MEDIA           0x00000008
#define FILE_REMOTE_DEVICE              0x00000010
#define FILE_DEVICE_IS_MOUNTED          0x00000020
#define FILE_VIRTUAL_VOLUME             0x00000040
```

Note that if you have designed a network redirector and if you set the FILE_REMOTE_DEVICE flag in the Characteristics field, the logical volume cannot be reshared across the LAN Manager Network.

FileFsAttributeInformation (enumerated type value = 5)

The structure used to request file system attribute information is defined as follows:

```
typedef struct _FILE_FS_ATTRIBUTE_INFORMATION {
    ULONG               FileSystemAttributes;
    LONG                MaximumComponentNameLength;
    ULONG               FileSystemNameLength;
    WCHAR               FileSystemName[l];
} FILE_FS_ATTRIBUTE_INFORMATION, *PFILE_FS_ATTRIBUTE_INFORMATION;
```

The file system attributes can be one or more of the following (note that additions to these values are likely with different versions of the operating system that add additional functionality):

```
#define FILE_CASE_SENSITIVE_SEARCH      0x00000001
#define FILE_CASE_PRESERVED_NAMES       0x00000002
#define FILE_UNICODE_ON_DISK            0x00000004
#define FILE_PERSISTENT_ACLS            0x00000008
#define FILE_FILE_COMPRESSION           0x00000010
#define FILE_VOLUME_IS_COMPRESSED       0x00008000
```

NTFS, for example, sets all of these attribute values except for the FILE_VOLUME_IS_COMPRESSED.

The MaximumComponentNameLength field is typically set to 255 characters by most native FSD implementations. Your FSD can set this field to any appropriate value. The FileSystemName field simply identifies the current FSD processing the request. NTFS, for example, will set the contents of the buffer to NTFS.

If the buffer supplied by the caller is too small to contain all of the information your FSD wishes to return, your driver should return the STATUS_

BUFFER_OVERFLOW return code and copy in as many bytes of information as it possibly can.

There are other volume information request types that have not yet been fully implemented by the I/O Manager, and they are not yet completely supported, even by the native FSD implementations. For example, there are query volume information types such as FileFsQuotaQuerylnformation (enumerated type value = 6 in Version 3.51 and value = 7 in Version 4.0) and a corresponding FileFsQuotaSetlnformation (enumerated type value = 7 in Version 3-51 and value = 8 in Version 4.0), which will become part of the set volume information request. Your FSD should currently return STATUS_INVALID_PARAMETER for all query volume information request types other those previously defined.

The sequence of steps followed in processing a query volume information request is extremely simple:

1. Obtain a pointer to the volume control block for which the request operation has been dispatched.
2. Acquire the VCB shared.
3. Find out the type of information requested and get a pointer to the caller-supplied buffer from the current stack location.

   The following fields give you this information:

   — The Parameters.QueryVolume.FsInformationClass field from the current stack location will tell you the type of information requested.

   — The I/O Manager always supplies a system virtual address for a buffer allocated by the I/O Manager.

   A pointer to this buffer can be obtained from the Associatedlrp. SystemBuffer field in the IRP. The length of this buffer is given by the Parameters . QueryVolume. Length field in the current stack location.
4. Ensure that the length of the buffer supplied is at least equal to the size of the associated structure (appropriate for the type of volume information request).

   If the amount of information your FSD returns exceeds the length of the supplied buffer, then return STATUS_BUFFER_OVERFLOW *after* filling in as much information as the supplied buffer can contain.
5. Complete the IRP after releasing any resources that were acquired.

IRP_MJ_SET_VOL UMEJNFORMATION

A user can also request that volume attributes should be modified. Currently, the only set volume information type request that your FSD should consider supporting is a request to set the label for the logical volume. This label is a string

identifier, supplied by the user so as to be able to identify the volume more easily. Although other set volume information request types have been defined (e.g., FileFsQuotasetInformation), they have not been well-defined yet, and they are not supported by the native FSD implementations.

The sequence of steps executed in response to a set volume information request closely mirrors those followed by the query volume information described previously.

1. The FSD obtains a pointer to the VCB from the file object supplied with the request.
2. The VCB should be acquired exclusively.
3. The type of request and a pointer to the caller supplied buffer can be obtained from the IRP.

   The request type for a set volume information request can be determined from the Parameters.SetVolume.FsInformationClass field in the current I/O stack location. Currently, the only legitimate request type is FileFsLabelInformation (enumerated type value = 2). The type of structure passed in by the caller for this request type is defined as follows:

```
typedef struct _FILE_FS_LABEL_INFOKMATION {
    ULONG    VolumeLabelLength;
    WCHAR    VolumeLabel[1];
} FILE_FS_LABEL_INFORMATION, *PFILE_FS_LABEL_INFORMATION;
```

   A pointer to the system buffer allocated by the I/O Manager can be obtained from the Associatedlrp. SystemBuffer field in the IRP. The length of the system-allocated buffer can be obtained from the Parameters. SetVolume . Length field.

4. After validating that the length of the caller-supplied buffer is correct, the FSD should perform appropriate operations to update the label (string) associated with the logical volume.
5. If the request type is anything other than what is supported by the FSD, an error code of STATUS_INVALID_PARAMETER should be returned to the caller.
6. The IRP can now be completed after releasing the VCB resource.

The actual code implementing a query/set volume information request is very similar to that shown in Chapter 10, *Writing A File System Driver II,* for handling query/set file information requests. Study that code example for details on how the FSD should structure the query/set volume information dispatch entry routine to execute the logical steps previously detailed.

Dispatch Routine: Byte-Range Locks

Windows NT supports mandatory byte-range file locks. The term *mandatory* implies that it is the responsibility of the FSD to ensure that access to a byte range by a thread during I/O operations is validated against any byte-range locks that have been granted for the file stream. Therefore, two or more threads do not have to actively cooperate in order to synchronize access to the file stream; as long as one of the threads is careful about obtaining the appropriate byte-range locks on the file, it can be ensured that data access (read or write) by any other thread belonging to other processes will be closely monitored. If such access is not allowed by the nature of the lock granted (and depending on the type of access requested), the FSD will deny the I/O operation with an error code of STATUS_ FILE_LOCK_CONFLICT.

The native NT FSD implementations do not appear to check for byte-range lock conflicts encountered during paging I/O operations. However, if your FSD is even stricter about checking for locked byte ranges and returns the STATUS_FILE_ LOCK_CONFLICT error code to the VMM, the VMM, in turn, will either raise an exception, informing the caller about the error, if this happened to be synchronous paging I/O request; or will pop up an error message box, indicating loss of write-behind data in the case of an asynchronous I/O write operation.

Byte-range locks in general are associated with processes and *not* with individual threads within a process. Therefore, if a single thread in the process acquires a specific byte-range lock, this will not prevent other threads within the same process from continuing to access the locked byte range even if the type of access performed conflicts with the nature of the granted byte-range lock. The byte lock obtained will prevent conflicting accesses by threads belonging to processes other than the one that obtained the lock.

NOTE It is possible for threads -within a process to obtain thread-specific byte-range locks by specifying a Key value when performing the lock operation. The Key argument is described later in this section. However, you should note that this method is often employed by a thread to ensure that the byte-range can be accessed only in a very selective manner by other threads.

The Windows NT I/O subsystem defines the following kinds of byte-range locks:

Read locks obtained for a specific byte range

Multiple processes can potentially obtain a read lock concurrently for the same byte range or for an overlapping byte range on the same file stream. The read lock simply guarantees the caller that no write/modify operations

are allowed on the file stream as long as the read lock is maintained by the process.

Write (exclusive) lock obtained for a byte range

Write locks are exclusive locks (i.e., once a process acquires a write lock for a specific byte range, no other process is allowed either to read or write in that byte range). By definition, granted write locks are non-overlapping.

Different processes can concurrently lock different byte ranges in the same file stream. It is also quite possible (and not at all unusual) for a thread to obtain a byte-range lock starting and/or extending well beyond the current end-of-file. This is simply a means whereby the process can ensure that appending the file stream can be performed in some sort of synchronized fashion.

Note that byte-range locking can possibly allow a process to synchronize access to the byte stream even across multiple nodes, as long as the network protocol providing remote file system access supports the byte-range locking protocol. For example, the LAN Manager redirector client and server support the byte-range locking protocol. The NFS (Network File System) protocol supports only advisory byte-range locks, whereas the DPS (Distributed File System) protocol can be used to obtain mandatory file locks.

It may be obvious to you by now that supporting byte-range file locks is not really an FSD-specific operation. In fact, it can be implemented in a fairly generic fashion, allowing multiple, installable file systems to take advantage of common code. The Windows NT I/O subsystem designers recognized this and have actually implemented file-lock-supporting code in the FSRTL. These routines are used by the native NT FSD implementations. Unfortunately, for reasons that seem incomprehensible, the developers do not want to encourage third-party FSD designers to take advantage of such support provided in the FSRTL. This may (I hope) change in the future.

In this section, we saw how to provide support for file lock operations if you have to implement such support yourself. Obviously, if any FSD-independent code is provided by Microsoft for the support of byte-range lock requests, you should utilize that code instead.

Type of File Lock Requests Received by an FSD

Broadly speaking, the FSD will receive two types of requests related to byte-range lock operations:

Requests to obtain a byte-range lock for a file stream

The request could specify either a read or a write lock. Furthermore, the caller could specify either a blocking or nonblocking lock request. If the

caller agrees to block, the IRP describing this request is not completed until the lock is granted or the IRP is canceled (which could be due to the caller closing the file handle). If the caller does not wish to wait for the lock to be granted and if some other thread has already acquired a conflicting lock that would prevent the current request for a byte-range lock from being completed successfully, an error code of STATUS_LOCK_NOT_GRANTED is returned to the caller.

Requests to unlock one or all byte-range locks acquired by the process for a specific file object

The caller can request that a specific, uniquely identifiable locked byte range be unlocked, or the caller can request that all byte-range locks on the file stream acquired using a specific file object and owned by the calling process be unlocked.

When a process closes all open handles associated with a file object for a file stream, if the process had ever acquired any byte range locks using that file object on the file stream, the I/O Manager will issue an unlock-all type of byte-range unlock request on the file stream, on behalf of the process closing the handle to the file stream. Similarly, whenever an FSD receives a cleanup request on a file stream for a specific file object, the FSD is expected to automatically unlock all byte-range locks that may have been acquired by the calling process using the file object for which the cleanup is being received.*

Lock requests

The lock request is dispatched to the FSD dispatch routine serving as the IRP_MJ_LOCK_CONTROL major function entry point. The lock request is distinguished by a minor function code of IRP_MN_LOCK. The arguments supplied to the FSD as part of the lock request are as follows:

Pointer to the file object

The FSD can easily obtain the file object pointer from the IRP for the open file stream on which the lock operation has been requested. Note that most FSD implementations will reject a byte-range file lock request if the object on which the lock has been requested is not an open, ordinary file. Therefore, directories, open logical volumes, and other such open objects typically cannot be locked with byte-range locks.

---

* There is a subtle point here that you must be aware of: the FSD must not unlock all byte-range locks owned by the process on the file stream associated with the file object on which the cleanup request has been received. Rather, only those byte-range locks must be unlocked for which the file object and the process ID both match.

`ByteOffset`

The starting offset for the lock request. This is contained in the Parameters.LockControl.Length field in the current stack location for the I/O request packet. As noted earlier, this offset could be well beyond the current end-of-file.

Length

The number of bytes that should be locked for the file stream. Once again, note that the ByteOffset value plus the Length value could extend well beyond the current end-of-file. This is a legitimate situation for lock requests.

Key

This is an unsigned long value that the requesting thread can associate with the lock to be granted. If the lock is granted, subsequent accesses to the byte range will only be allowed if the process ID and the key value match. You may recall from the discussion on read/write requests, presented in Chapter 9, *Writing a File System Driver* 7, that the requesting thread can supply a Key argument with the I/O request. That argument is subsequently used when checking whether or not the I/O request will be allowed to proceed.

This is a method where a thread in a process can potentially exclude even other threads in the same process from accessing the locked byte range.

Process ID

Although not explicitly supplied as part of the IRP sent to the FSD, the FSD can easily determine the current process ID for the process requesting the lock operation, by using the loGetRequestorProcess() I/O Manager service routine. This routine accepts a pointer to the IRP as an argument and returns a pointer to the process structure (of type PEPROCESS).

`Fai1Immediately`

This BOOLEAN value can be obtained by checking for the presence of the SL_FAIL_IMMEDIATELY flag in the Flags field of the current stack location in the IRP. The presence of the flag indicates that FailImmediately should be set to TRUE, which in turn means that the caller would not like to wait if the lock cannot be immediately granted.

The absence of the flag indicates that the caller does not mind waiting for the lock request to be granted at some later time. In this case, set the value of Faillmmediately to FALSE.

`WriteLockRequested`

The presence of the SL_EXCLUSIVE_LOCK flag in the Flags field of the current I/O stack location indicates that the caller wishes an exclusive (write) lock for the byte range specified. In this case, set the value of WriteLockRequested to TRUE.

The absence of the SL_EXCLUSIVE_LOCK flag indicates that the caller wishes to obtain a read (shared) byte-range lock only, and therefore the value of WriteLockRequested should be set to FALSE.

Unlock requests

The unlock request is distinguished by any one of the following minor function code values:

`IRP_MN_UNLOCK_SINGLE`

The FSD must unlock only one byte-range lock. The lock that would be unlocked (if found) is the single matching lock for which all of the following passed-in parameters match:

— Process ID associated with the lock, identifying the owner of the byte-range lock

— File object

— Starting offset

— Length in bytes of the locked range

— Key value

If any of the parameters listed here do not match, then no unlock operation will be performed.

`IRP_MN_UNLOCK_ALL`

This is the brute-force approach employed by a process to unlock all of the byte-range locks acquired by any thread associated with the process using the target file object. In response to this request, the FSD will unlock all byte-range locks for which the following match:

— Process ID associated with the lock, identifying the owner of the byte-range lock

— File object

This request is typically sent by the I/O Manager to an FSD when a process closes all open handles associated with a file object, but there are other open handles associated with the same file object belonging to other processes. If all handles for a file object have been closed, the I/O Manager skips sending the unlock-all request, since the expectation is that the FSD will generate this request internally in response to a cleanup request received by the file system driver.

IRP_MN_UNLOCK_ALL_BY_KEY

A thread or a process can unlock all byte-range locks for a file object owned by all threads belonging to the process, as long as the supplied key value and the key stored with the byte-range lock match. Typically, this method is

slightly less brute-force than the previous one, since this can also be used by a thread to close a specific set of byte-range locks all identified by the same key value.

In response to this request, the FSD will unlock byte-range locks for which all of the following match:

— Process ID associated with the lock, identifying the owner of the byte-range lock

— File object

— Key value

In order to determine the parameters supplied with the unlock IRP, use exactly the same fields (and methods) as described earlier for the lock request operations.

Structures Requiredfor File Lock Support

To implement byte-range lock support in your FSD, you will typically require some variation of the following structures:

```
typedef struct SFsdFileLockAnchor {
    LIST_ENTRY            GrantedFileLockList;
    LIST_ENTRY            PendingFileLockList;
} SFsdFileLockAnchor, *PtrSFsdFileLockAnchor;

typedef struct SFsdFileLocklnfo {
    SFsdldentifier                          Nodeldentifier;
    uint32                                  FileLockFlags;
    PVOID                                   OwningProcess;
    LARGE_INTEGER                           StartingOffset;
    LARGE_INTEGER                           Length;
    LARGE_INTEGER                           EndingOffset;
    ULONG                                   Key;
    BOOLEAN                                 ExclusiveLock;
    PIRP                                    PendingIRP;
    LIST_ENTRY                              NextFileLockEntry;
} SFsdFileLocklnfo,  *PtrSFsdFileLockInfo;

#define        SFSD_BYTE_LOCK_NOT_FROM_ZONE                  (0x80000000)
•define        SFSD_BYTE_LOCK_IS_PENDING                     (0x00000001)
```

Typically, you should embed an SFsdFileLockAnchor structure into the FCB for the file stream. This structure serves as a list anchor for the following two linked lists:

- A list containing SFsdFileLocklnfo structures, each of which represents a granted lock for the file stream
- A list containing SFsdFileLocklnfo structures, each of which represents a pending lock for the file stream

The SFsdFileLocklnfo structure represents an instance of a granted or pending byte-range lock request. An instance of this request is allocated whenever a byte-range lock request is received. The structure is freed only when the lock is failed immediately, the IRP is canceled (or the file handle closed) while the lock request is still queued, or an unlock operation is eventually received for a granted file lock. The OwningProcess, StartingOffset, Length, Key, and ExclusiveLock fields are initialized based upon information supplied in the byte-range lock request as described earlier.

The PendinglRP field is only valid when the request has been queued, awaiting an unlock operation. This field then points to the IRP received containing the byte-range lock request for which STATUS_PENDING was returned. The Ending-Offset field contains a value that is computed and stored for convenience.

The NextFileLockEntry field is used to queue the SFsdFileLocklnfo structure to either the GrantedFileLockList or the PendingFileLockList in the SFsdFileLockAnchor structure contained in the FCB for the file stream on which the lock operation has been requested.

The FileLockFlags field is used internally to determine where the structure has been allocated from and also to mark a pending lock request for easy identification.

Logical Steps Involved

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

```
typedef struct _IO_STACK_LOCATION {

    // ...

    union {

        //...

        // System service parameters for:  NtLockFile/NtUnlockFile
        struct {
            PLARGE_INTEGER Length;
            ULONG Key;
            LARGE_INTEGER ByteOffset;
        } LockControl;

        // ...
} Parameters;

// ...

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

Processing a file lock or unlock request is quite a simple operation to implement. The following steps outline the processing required for file lock operations:

1. Obtain the parameters described earlier that are supplied with a typical byte-range lock request.
2. Obtain FSD-specific pointers to the FCB and CCB structures for the file stream.
3. Acquire the FCB MainResource exclusively.
4. Allocate and initialize a new SFsdFileLocklnfo structure to contain the caller-supplied parameters.
5. Check if any conflicting locks have been previously granted.

   For an exclusive lock request, the FSD must check if any portion of the requested byte range overlaps with a byte range on which a file lock had been previously granted. To check this, the FSD can simply scan through all of the granted file locks identified by the SFsdFileLocklnfo structures linked to the GrantedFileLockList in the FCB.

   For a shared lock request, the FCB should ensure that no portion of the requested byte range overlaps with a previously granted exclusively locked range. Overlaps with previously granted shared byte-range locks are acceptable if the current request also wants to obtain a lock for shared (read) access.
6. If no conflict has been found, queue the SFsdFileLocklnfo structure to the GrantedFileLockList to indicate that a new file lock has been granted and complete the IRP with STATUS_SUCCESS returned to the caller.
7. If a conflict is detected, check whether the caller is prepared to wait to obtain the file lock.

   If the caller is not prepared to wait (i.e., if FailImmediately is set to TRUE), then complete the IRP with a status code of STATUS_LOCK_NOT_GRANTED. Otherwise, queue the IRP to the PendingFileLockList list anchor, contained within the SFsdFileLockAnchor structure in the FCB.

   To properly queue the request, initialize the PendingIRP field in the SFsdFileLocklnfo structure to point to the IRP sent to the FSD by the I/O Manager for the file lock request. Also set the SFSD_BYTE_LOCK_IS_PENDING flag value in the FileLockFlags field. Mark the IRP itself as pending, set a cancellation routine for the IRP, and return a status code of STATUS_PENDING to the I/O Manager.

   The expectation is that for queued lock requests, the FSD will complete the request whenever the lock is granted (i.e., whenever the conflicting conditions have been removed).

8. If any file locks have been granted, be sure to update the IsFastIoPossible field value to FastIoIsNotPossible in the CommonFCBHeader for the file stream.
9. Release the FCB MainResource and return control back to the I/O Manager.

To process a byte-range unlock request, the FSD typically performs the following logical steps:

1. Obtain required parameters, depending upon the type of unlock request.

   For example, for a IRP_MN_UNLOCK_SINGLE request, the FSD must get all of the information, described earlier, that is required to uniquely identify the single byte-range lock for which the unlock request has been received. However, for the case of IRP_MN_UNLOCK_ALL, the FSD simply needs to identify the process requesting the unlock operation and the file object for which the unlock operation has been requested.
2. Obtain FSD-specific pointers to the FCB and CCB structures for the file stream.
3. Acquire the FCB MainResource exclusively.
4. Scan through all of the SFsdFileLockInfo structures linked to the GrantedFileLockList list head in the FCB.

   The intent here is simple. If any matching file-lock structures are encountered, the unlock operation is processed for the structure. Processing the unlock operation is simple since it only involves unlinking the structure from the GrantedFileLockList and freeing up the allocated structure.

WARNING Whenever the unlock-all request is issued to the FSD, your driver must perform one additional step. It must scan through the PendingFileLockList, searching for any pending, matching file-lock requests. If such requests are found, your driver must complete the pending IRP (waiting for the byte-range lock) after removing any cancellation routine that may have been set, and then the FSD should unlink the SFsdFileLockInfo structure from the PendingFileLockList linked list and free it.

---

5. Go through all of the entries in the PendingFileLockList to see if any locks can now be granted.

   Since some unlock operations may have been performed in the preceding step, the FSD should now scan through the list of pending lock requests to see if any of them can be granted. If any such request can be granted, the pending IRP associated with the request should be completed with STATUS_SUCCESS after any cancellation routine that may have been specified is unset. The SFsdFileLockInfo structure for the pending request (that has

now been granted) should also be moved from the PendingFileLockList to the GrantedFileLockList (and the SFSD_BYTE_LOCK_IS_PENDING flag should be cleared).

6. If all granted file locks have been removed, be sure to update the IsFastIo-Possible field value to FastIoIsPossible or FastIolscruestion-able in the CommonFCBHeader for the file stream.

   Note that the actual value will depend on the state of the opportunistic locks associated with the FCB.

7. Release the FCB MainResource and return control back to the I/O Manager.

If your FSD follows this simple sequence of steps for lock control operations, you should be able to successfully implement byte-range lock support in your file system driver implementation.

Opportunistic Locking

Opportunistic locks (oplocks, for short), simply stated, are guarantees made by a network LAN Manager server node to one or more LAN Manager client nodes about the types of file stream accesses that will be allowed on a specific file stream. They are currently valid only within the LAN Manager network environment and allow a client to perform some type of local node caching, knowing that it will be protected from returning stale data to the user because of the presence of these guarantees.

For example, consider a situation where a server on node *sen>er_nodel* shares a local drive letter *X:*. Furthermore, imagine that a user on node *client_nodel* connects to this shared drive letter using the LAN Manager network and opens a regular file *foo* for both read and write access. In the absence of any server guarantees on the file stream *foo,* every read operation made by the user thread on the client node would result in the LAN Manager redirector having to issue a network read request to obtain the latest data from the server node. The LAN Manager server software, in turn, would have to request the file data from the local file system driver managing the shared logical volume corresponding to the drive letter *X:*. As you could imagine, this would lead to extremely slow access (and therefore a small throughput value) for the user on the client node.

Similarly, every write operation performed by the user on the client node would result in the LAN Manager redirector having to send the updated data to the server node across the network. The LAN Manager server software, in turn, would have to issue the write to the local file system managing the shared logical volume corresponding to the drive letter *X:* on which the file stream *foo* resides.

You can also imagine what this kind of data transfer would do in terms of saturating your network.

Needless to say, the LAN Manager redirector code on the client node could not hope to use the services of the NT Cache Manager at all, since data could never be cached locally.

To avoid this sort of constant data transfer to and from the client and server nodes participating in a LAN Manager network, the network protocol designers invented a crude form of cache support built into the protocol called opportunistic locking. This caching protocol allows the LAN Manager server to make one of three kinds of guarantees to the network redirector software on one or more client nodes:

- If an exclusive oplock is granted to a client node for a file stream, the client node is assured that no other thread, either executing locally on the server or on any other client node, will be allowed to access (or even open) the file stream for which the exclusive oplock has been obtained.

  Consider a client node that requests an open operation of file *foo* on shared drive letter *X:* served (say) by the sample FSD on the server. In response to the client node's open request made by the LAN Manager server locally on the server node (issued on behalf of the thread of the client that has actually requested the open), the sample FSD will create FCB and CCB structures, and also initialize the file object structure passed in by the I/O Manager. Note that this is no different from any other regular open operation except that the request originates on the server node in the LAN Manager server software (which executes in kernel mode) on behalf of a network client.

  Now, also imagine that after the open operation completes, the LAN Manager server asks the sample FSD to issue an exclusive oplock for the file stream *foo.* Imagine also that the sample FSD participates in the oplock protocol implementation, and therefore agrees to the request. Now, it is the responsibility of the sample FSD to notify the LAN Manager server whenever any thread requests an open for the file stream *foo* for either read or write access.* The reason for this is as follows: when the local FSD (in our case, the sample FSD) grants an oplock to the LAN Manager server on the server node, the server software, in turn, grants the oplock to the network redirector client. For the exclusive oplock, this assures the client that no other thread is actively reading or writing the same file stream. Now, the client software on the remote client node can cache file stream data on the remote node without having to worry about data consistency issues.

---

* For an exclusive oplock, the FSD is allowed to let threads open the file stream without breaking the oplock if they only open the file for read attributes and/or write attribute access.

Read caching

The network redirector client obtains file stream data from the server node and then satisfies all read requests from the user thread locally. Data could even be returned directly from the system cache in this situation.

Write caching

The user thread could modify the data for the file stream and the network redirector client would simply cache the modified data in the system cache on the remote node, from which it would be asynchronously written out every once in awhile.

As you can see, having an exclusive oplock on a file stream can improve network throughput tremendously.

What happens when another thread, either from the same client node, from some other client node, or locally from the server node also tries to open file *foo* for read and/or write access? The local FSD that granted the oplock (in our case, the sample FSD) will have to break the oplock (i.e., inform the LAN Manager server that it should, in turn, inform the client that the client can no longer run amuck with the file data). Since this is an exclusive oplock, where the client may actually have modified data cached remotely, the local FSD must then wait for the client node to flush (and purge) all cached information to the server. The flush results in write requests being issued to the local FSD from the server software on behalf of the remote client. Eventually, all of the data is updated on the server node, and the local FSD on the server can allow the new open to proceed. The client is also now aware that it no longer has exclusive access to the file stream and will therefore not try to modify the data remotely and keep it cached.

Note that the local FSD on the server node makes the new open request wait until all of the data has been updated by the client to the server node. The exclusive oplock is considered completely broken only after the data transfer has been completed.

- There are also shared oplocks that can be granted by a local FSD to the server software, which will grant the oplock to one or more network redirector clients.

  Consider the situation where multiple threads, residing on one or more client nodes, including local threads on the server node itself, have file *foo* open for read and write access. Although the local FSD will no longer grant an exclusive oplock to the file stream *foo,* it will allow client nodes to request shared oplocks. Shared oplocks are the next best thing to exclusive oplocks because they assure the network redirector software on the client node that as long as

the oplock is granted, the client node can cache data remotely for read operations.

Whenever the local FSD on the server node receives a write request, it is expected to break all of the read oplocks that were granted to all of the client nodes concurrently accessing the file stream *foo.* The oplock breaks inform the network redirector software on the client nodes that the data they have cached may no longer be valid. The network redirector software on all the client nodes will, in response to the oplock break, purge the system cache of all cached data. The next read request issued by a thread on one of the remote nodes will cause the network redirector software on that remote node to request fresh data from the server.

- Finally, due to its DOS heritage, the LAN Manager protocol also provides for batch oplocks to be granted to client nodes.

  Consider the batch files (with extension *.bat)* that are simple scripts, which can be executed by the DOS shell on any Windows NT machine. The method used by the shell to execute the different statements in a batch file follows:

  — The shell opens the batch file.

  — It reads the next line to be executed.

  — It closes the batch file.

  This sequence is repeated in a loop until the entire batch file has been executed. Now consider the situation where the file opened by a remote client on the shared drive *X:* is called *foo.bat.* Furthermore, imagine that the shell on the remote client is busily going through the loop where it opens the file stream, reads a line, and closes the file stream. This would typically result in a whole lot of open/close requests flying across the network.

  Instead, the network redirector client typically requests a batch oplock from the server software, which in turn requests this oplock from the FSD on the server node. Once a batch oplock has been granted, the network redirector software on the client node will no longer close the file handle in response to a close performed by the user thread (the shell) on that remote node. Instead, the network redirector will continue to keep the file open, fully expecting the user thread to come back and rerequest an open operation, once the current line read from the file stream has been executed. Furthermore, the grant of a batch oplock has the same characteristics as an exclusive oplock, where the remote client is assured that it has full and exclusive access to the file stream.

NOTE Maintaining cache coherency across multiple nodes for shared file objects is a difficult problem to solve. A lot of research has been done on the subject, and you can consult some of the references provided at the end of the book for more information.

There are also commercially available file system implementations that do a much more sophisticated job of maintaining cache coherence across nodes. An example of this is the Andrew File System (AFS) implementation originally designed at Carnegie Mellon University and the OSF DPS (Distributed File System) implementation.

Although I believe that the method devised by the LAN Manager Network protocol is crude at best, it does work and supporting this feature could make remote accesses to shared logical volumes managed by your FSD much faster.

Some Points to Remember About Oplocks

When (and if) you decide to support the oplock protocol, keep the following points in mind:

- Oplocks are typically only requested by the LAN Manager server software on behalf of a remote client.

  There is nothing, however, to prevent some other component from requesting an oplock from the FSD.

- Oplocks are requested from the FSD on the server node that manages a shared logical volume.

  Although this may seem obvious, keep sight of the fact that as the FSD managing the shared logical volume on the server node, you have full control over whether or not to support the opportunistic locking protocol. Furthermore, under normal situations, oplock requests will only be issued to your FSD if the logical volume that your FSD is managing has been shared across the LAN Manager Network.

- Oplocks have funky semantics that, unfortunately, need to be maintained.

  As an example, consider the case when an exclusive oplock is being broken by the local FSD because another thread wishes to open file *foo* on the server for read and/or write access. Your FSD would typically expect to block the new open request until the client node that has the exclusive oplock completes the break by flushing all modified data back to the server.

  Typically, that is exactly what your FSD should do. However, the engineers who designed this messy protocol found that, because the LAN Manager server software on the server node has a fixed number of worker threads that

it uses to service remote requests, it is theoretically possible that all of these threads get blocked on servicing open requests for file streams that have opportunistic locks acquired by some remote clients. In such situations, neither the FSD nor the server software can truly determine when the open request would complete (with either a success or failure code), since this would depend on how quickly the client nodes could flush the data back to the server. It may even be possible for the server to encounter a deadlock if all threads are blocked because of the presence of exclusive oplocks and there are no threads available to service the client flush request required to complete the oplock break sequence.

In typical DOS-style Microsoft fashion, the designers decided to work around this problem by allowing the LAN Manager server to specify a special flag in the open request. The flag value of FILE_COMPLETE_IF__OPLOCKED is specified in the Parameters.Create.Options field in the create IRP. If such an option has been specified, the FSD is not supposed to block the current open, even though the oplock break has not yet been completed. Instead, the FSD must execute the open, returning the STATUS_OPLOCK_BREAK_IN_PROGRESS return code in the Status field (provided all other conditions would allow the open request to succeed). This code value is equivalent to STATUS_SUCCESS (i.e., the macro NT_SUCCESS (STATUS_OPLOCK_BREAK_IN_PROGRESS) will return TRUE).

The strange thing about the FILE_COMPLETE_IF_OPLOCKED flag is the semantics associated with this flag value. The FSD allows the open to succeed, knowing full well that there is now nothing to prevent the caller from violating the trust and performing read/write operations even before the oplock break has been completed. However, the expectation is that, since the caller could only be the LAN Manager server, it will do the right thing and not issue any I/O requests until the client that has the exclusive/batch lock on the file stream has flushed all its data, and the oplock break has been completed.

WARNING As an FSD designer, you can never trust any other component to do the right thing. Therefore, do not buy into this philosophy in general and always, always, validate before allowing a caller to proceed with a file system operation.

Unfortunately, when providing support for opportunistic locks, the FSD may have to conform to the model determined by the I/O subsystem designers, which requires some trust to be maintained. The only recourse available to you in this case is not to support oplocks (which could lead to degraded performance for users of your FSD).

- Your FSD does not have to support oplock requests.

  If you have begun to think that oplocks are too strange for your tastes, I would agree with you. Therefore, note that you do not have to support opportunistic locking in your FSD. However, if your FSD manages logical volumes that could potentially be shared, supporting oplocks (oddities and all) would be a nice feature to have.

- Even if your FSD does provide oplock support, a remote client that tries to map the file stream in memory will not be able to enjoy any data coherency guarantees.

  As described in Chapter 5 in the discussion on the NT VMM, it is currently not possible for an FSD (including a network redirector) to purge user-mapped pages from the system cache. Therefore, processes on remote clients that decide to map a file stream in memory are effectively shut out from any synchronization/cache coherency guarantees provided by the LAN Manager network protocol.

How Is an Oplock Granted and Broken?

The LAN Manager server issues an oplock request by utilizing the File System Control (FSCTL) interface (described later in this chapter). Basically, your FSD will receive FSCTL requests that indicate the server wishes to obtain an exclusive/shared/batch oplock on a particular file stream identified by the file object used in the file system control IRP.

If your FSD grants the oplock request, it must mark the IRP as pending and queue the IRP internally. A return code of STATUS_PENDING to the caller of the file system control request indicates that the oplock has been granted.*

So, once again, the rules are simple:

- When you receive an oplock request, either return a status code immediately of STATUS_OPLOCK_NOT_GRANTED, indicating that the request was denied, or return STATUS_PENDING, which the caller treats as success in obtaining the oplock.

- An oplock is broken by simply completing the IRP that was queued (and STATUS_PENDING returned) when the oplock had been previously granted.

  Typically, the LAN Manager server software specifies an IRP completion routine that is invoked whenever the oplock is broken (i.e., the IRP is simply completed by the FSD, and this is sufficient to indicate that the break has occurred). This completion routine initiates the break processing across the

* In the wonderfully twisted world that some designers at Microsoft live in, all of this makes perfect sense.

network, which could result in I/O flush operations from the remote client node to the FSD. Remember that the LAN Manager server software executes in kernel mode, is very tightly integrated with the rest of the I/O subsystem, creates and manages its own IRP structures (just as the I/O Manager does), and is therefore capable of using all sorts of methods directly without having to go through the NT I/O Manager.

There are a couple of other return values you should be aware of:

- The special return code status of STATUS_OPLOCK_BREAK_IN_PROGRESS returned in response to a create/open request, indicating that a break is underway, and the caller should wait until the break has been completed
- A value of FILE_OPBATCH_BREAK_UNDERWAY, returned sometimes in the Information field of the loStatus structure when a create/open request is received

  This value is only returned in the Information field if the create/open request is being denied due to a sharing violation, but your FSD wishes to inform the caller that a break operation is underway for the file stream. The intent here is to allow the caller to possibly modify the share access requested and resubmit the create/open request.
- A value of FILE_OPLOCK_BROKEN_TO_LEVEL_2 (with value = 0x00000007) returned in the Information field when an IRP is being completed to indicate an oplock break

  This is another one of the optimizations added by the oplock protocol designers. If an exclusive or a batch file oplock is being broken, the FSD has the option of offering a shared oplock to the thread whose exclusive/batch lock is being broken. The idea here is that even if the original requesting network redirector code on the remote node can no longer have the absolute power that an exclusive/batch oplock could provide, it can at least take advantage of the functionality (and guarantees) that come with owning a shared oplock.

  Therefore, when breaking an exclusive/shared oplock, the FSD could (but is not required to) return the FILE_OPLOCK_BROKEN_TO_LEVEL_2 value in the Information field. In turn, the network redirector software on the client node also has the option of either accepting this newly offered shared oplock, or not.
- A value of FILE_OPLOCK_BROKEN_TO_NONE (with value = 0x00000008) returned in the Information field when an IRP is being completed, to indicate an oplock break.

  This is the alternative Information field value returned by the FSD whenever an oplock is being broken, and it does not offer even a shared oplock in return.

Oplock Processing Sequence

The following sequence of operations is performed in granting an oplock request from the perspective of an FSD supporting the oplock functionality:

1. The LAN Manager server requests an opportunistic lock on an open file stream uniquely identified by a file object structure on behalf of a remote LAN Manager redirector client.

   The request is issued to the FSD in the form of a file system control (FSCTL) IRP. FSCTL requests are discussed in more detail later in this chapter. Note for now, however, that the major function code in the IRP is IRP_MJ_FILE_SYSTEM_CONTROL. The minor function is IRP_MN_USER_FS_REQUEST. The possible FSCTL code values to request an opportunistic lock are:

   - FSCTL_REQUEST_OPLOCK_LEVEL_1

     This is a request for a Level 1, or an exclusive oplock, on the file stream. Here is the code value:

     ```
     CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 0,
              METHOD_BUFFERED, FILE_ANY_ACCESS)
     ```

   - FSCTL_REQUEST_OPLOCK_LEVEL_2

     This is a request for a Level 2, or a shared oplock, on the file stream. Here is the code value:

     ```
     CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 1,
              METHOD_BUFFERED, FILE_ANY_ACCESS)
     ```

   - FSCTL_REQUEST_BATCH_OPLOCK

     This is a request for a batch oplock on the file stream. Here is the code value:

     ```
     CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 2,
              METHOD_BUFFERED, FILE_ANY_ACCESS)
     ```

2. The FSD decides either to grant or deny the request for an oplock.

   The rules defined to grant/deny the oplock request are as follows:

   - For an exclusive lock or a batch lock:

     If there is more than one open handle for the file stream (indicated by the OpenHandleCount field in the FCB in the case of the sample FSD), the oplock request is denied. All synchronous oplock requests are always denied since, by the method employed to grant the oplock (i.e., return STATUS_PENDING), it would be foolish to grant the oplock request.*

---

* The I/O Manager always blocks on behalf of the requesting thread for file objects opened for synchronous processing. Granting an oplock in such a situation would result in the invoking thread being blocked forever in the I/O Manager code.

If there is only one open handle for the file stream (representing the open operation performed by the thread requesting the exclusive/batch oplock), and if no exclusive/batch oplocks have currently been granted on the file stream, the request will succeed.

If there is only one Level 2 oplock previously granted to the same thread now requesting an exclusive oplock, the Level 2 oplock will be broken and the new exclusive/batch oplock granted.

In any other situation, the oplock request will be denied.

— For a shared oplock:

Just as in the case of the exclusive oplock request, all synchronous oplock requests are immediately denied. Otherwise, if there are no oplocks currently outstanding on the file stream or if the only type of oplocks that have been granted are shared oplocks, the request is allowed to succeed.

Note that even if an exclusive/batch oplock is currently being broken (break is underway), the request will be denied.

3. If a decision is made to grant the oplock, the IRP will be marked pending, a cancellation routine will typically be set for the IRP, the IRP will be queued by the FSD on some internal list associated with the FCB, and STATUS_PENDING will be returned to the caller.
4. If a decision is made to deny the oplock request, the IRP will be completed with a return code value of STATUS_OPLOCK_NOT_GRANTED.

Consider the situation when an oplock (shared/exclusive/batch) has been granted. The following events will lead to the oplock being broken:

- An exclusive/batch oplock had been granted, and another thread decides to open the file.

  The FSD knows that it must break the exclusive/batch oplock to continue processing the open request. The only determination to be made by the FSD at this time is whether to offer a shared oplock in return or to simply break the oplock completely. If the file stream is being superseded or overwritten, the FSD will break the oplock completely, and no shared oplock will be offered.

  However, if the file stream is not being overwritten or superseded, a shared oplock will be offered in lieu of the exclusive/batch oplock that is now being broken.

- A write request is received by the FSD, and shared oplocks had previously been granted.

  The FSD must break the shared oplocks completely.

- A lock/unlock request is received by the FSD, and shared oplocks had previously been granted.

  The FSD must break the shared oplocks completely.

- A read request is received by the FSD, and exclusive/batch oplocks had previously been granted.

  The FSD must break the exclusive/batch oplock and offer a shared oplock instead.

- A flush buffers request is received, and oplocks had previously been granted.

  The FSD must break the oplock and offer a shared oplock instead.

- The end-of-file mark or allocation size value is decreased.

  The FSD must break any oplocks granted completely.

- A cleanup request is received for the file object, indicating that all user handles have been closed for the file object.

  Any oplocks granted using the particular file object must be completely broken and outstanding IRPs completed.

- The remote client that requested the oplock no longer needs it.

  The remote network redirector client that requested the oplock can request a break of the oplock. This break notification is issued to the FSD via the LAN Manager server in the form of a FSCTL request. The code value of the FSCTL code is FSCTL_OPLOCK_BREAK_ACKNOWLEDGE which is defined as follows:

  ```
  CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 3, METHOD_BUFFERED, FILE_ANY_ACCESS)
  ```

  Note that this FSCTL is also used by a client to acknowledge an oplock break initiated by the FSD (described later). However, an asynchronous (spontaneous) FSCTL from a client node with this value indicates that the caller, itself, wants to break the oplock. When this request is received by the FSD, all it has to do is complete the IRP that was queued when the oplock was originally granted, clean up any oplock state maintained, and complete any pending IRPs that may have been received and blocked awaiting a break.

If the LAN Manager server has requested oplocks on a file stream using a particular file object on behalf of a remote network redirector client, and the client decided to perform I/O operations conforming with the state of the oplock that had been granted, the oplock cannot be broken by the FSD.

Whenever the FSD decides to break an oplock before allowing the current request to proceed, the current IRP is simply made to block until the oplock break has been completed.

Once the FSD has determined either to break or downgrade the oplock (from an exclusive/batch oplock to a shared oplock), the following sequence of events must be executed by the FSD (in each case, the thread that requested the oplock broken must acknowledge the break as described later).

Oplocks that are completely broken
: The FSD will complete the original IRP that was queued when granting the oplock. The Information field value in the IoStatus structure will be set to FILE_OPLOCK_BROKEN_TO_NONE.

Oplocks that are downgraded to shared oplocks
: The FSD will complete the original IRP that was queued when granting the oplock. The Information field value in the IoStatus structure will be set to FILE_OPLOCK_BROKEN_TO_LEVEL_2.

As previously mentioned, the FSD makes the current IRP (causing the oplock break to occur) block, awaiting acknowledgment of the oplock break notification.

To acknowledge an oplock break, the LAN Manager server issues a new FSCTL request to the FSD. The possible FSCTL code values are as follows:

FSCTL_OPLOCK_BREAK_ACKNOWLEDGE
: This FSCTL code value is used by the LAN Manager server on behalf of a remote network redirector client to acknowledge (or initiate) an oplock break notification.

  If this FSCTL code is received by the FSD after it broke or downgraded a Level 1 (exclusive) or batch oplock, the FSD is assured that the remote client has completed flushing all of the dirty data that may have been cached remotely back to the server node.

  If the FSD offered a shared oplock to the client in lieu of an exclusive or batch oplock that was being broken, receipt of this FSCTL code in the IRP indicates to the FSD that the client node has accepted the new shared oplock that was offered. The FSD would then perform the following steps:

  a. Update internal structures to reflect the fact that the original exclusive/batch oplock has been broken completely.

  b. Process the current FSCTL IRP as if it were a request to obtain a new shared oplock, mark the IRP pending, set a cancellation routine, and return STATUS_PENDING to the caller, indicating that a new Level 2 (shared) oplock has been granted.

  If the oplock being broken was originally a shared oplock, or if the FSD did not offer a shared oplock in lieu of the exclusive/batch oplock being broken, the FSD can simply update internal data structures to indicate that oplock break processing has been completed. The FSCTL IRP should be completed

with a STATUS_SUCCESS return code, and the Information field in the IoStatus structure of the FSCTL IRP should be set to FILE_OPLOCK_BROKEN_TO_NONE.

Any IRPs that were queued by the FSD awaiting the oplock break can be allowed to continue processing at this time.

NOTE If the FSCTL request containing the FSCTL_OPLOCK_BREAK_ACKNOWLEDGE FSCTL code value is issued as a synchronous I/O request, the FSD cannot grant any shared oplock and will always complete the FSCTL IRP with status code set to STATUS_SUCCESS and the Information field value set to FILE_OPLOCK_BROKEN_TO_NONE.

`FSCTL_OPBATCH_ACK_CLOSE_PENDING`

An FSCTL IRP with this FSCTL code value is issued to the FSD by the LAN Manager server on behalf of a remote network redirector client in response to an oplock break notification request for either an exclusive or a batch oplock request.

This FSCTL is issued instead of the FSCTL_OPLOCK_BREAK_ACKNOWLEDGE FSCTL to indicate that the network redirector client does not want the shared oplock, offered by the FSD in lieu of the exclusive/batch oplock, being broken.

The FSD can simply clean up internal data structures to indicate that the oplock break has been completed and complete the FSCTL IRP with status code set to STATUS_SUCCESS.

Any IRPs that were awaiting the oplock break notification can be allowed to proceed once this FSCTL has been processed.

There is one additional FSCTL code that your driver should expect to receive: FSCTL_OPLOCK_BREAK_NOTTFY. The caller wants to be notified when a Level 1 oplock break operation has been completed. If no Level 1 oplock break operation is in progress when the request is received, even if there are oplocks (exclusive/shared/batch) currently granted for the file stream, the IRP should be immediately completed with STATUS_SUCCESS. However, if a Level 1 oplock break operation is underway when this request is received, the IRP should be queued and only completed when the oplock break operation has been completed.

FSRTL Support for Oplock Processing

The native Windows NT FSD implementations use common routines provided by the FSRTL to provide oplock support. Unfortunately, Microsoft has chosen not to encourage third-party developers to use the routines exported by the FSRTL package. However, the description of the functionality expected from your FSD should help you in designing and developing your own opportunistic locking support package.

Dispatch Routine: File System and Device Control

File system drivers receive file system control requests to perform processing that cannot otherwise be requested via the standard dispatch entry points. Device control requests are also directed by the I/O Manager to the file system driver that performs a mount operation on a target physical or virtual device.

Types of FSCTL Requests

Most file system driver implementations respond to one or more file system control requests. The I/O Manager dispatches a file system control request (FSCTL request) to the FSD via an IRP with a major function code value of IRP_MJ_FILE_SYSTEM_CONTROL. There are four types of distinguishing minor function codes that the FSD must check for whenever it receives a FSCTL request from the Windows NT I/O Manager:

`IRP_MN_USER_FS_REQUEST`

This minor function code is used in the most common case, when a thread opens a file system object (file/directory/volume/device) and issues a FSCTL to the file system driver. There is a set of standard system-defined FSCTL codes (defined by Microsoft) that can be used by user threads; these will be discussed later in this section.

In addition to the Microsoft-defined FSCTL codes, it is always possible for FSD designers to develop their own private FSCTL codes, used internally between helper applications/user threads and the FSD itself. These can be issued to the FSD either to request some required functionality or to transfer data to and from the driver and the user-space application processes.

For example, your FSD may provide some special information in response to specific FSCTL requests issued by a helper application from user space. Or, your helper application may issue a special FSCTL request to request the FSD to format a specific disk. To accomplish such functionality, you would typi-

cally define some private FSCTL codes, to be used only by your helper applications and the FSD.

Issuing FSCTL requests is the easiest, most private, and most convenient method of information transfer between a kernel-mode driver and a user space thread. To use this method of data transfer, simply define a new FSCTL code, using the guidelines extensively documented in the DDK, implement support for the specific FSCTL in the kernel-mode FSD, and have a user-space thread issue the FSCTL whenever required. That is all you have to do to accomplish the data transfer, or to have the FSD perform some specific operation requested by the user thread.

`IRP_MN_MOUNT_VOLUME`

This special request is issued only by the I/O Manager to request a mount operation, in response to a change in media reported by a lower-level driver (for removable media only), or more likely, when the first user open is received for a file/directory residing on a physical disk that has not had a mount operation performed on it.

Later in this chapter, you can read a detailed discussion on the mount process and the functionality provided by the FSD in response to a mount request identified by the IRP_MN_MOUNT_VOLUME minor function code.

IRP_MN_LOAD_FILE_SYSTEM

This request originates in the I/O Manager. It is only issued by the I/O Manager to special mini-FSD implementations, requesting them to perform a load of the full file system driver image. Later in this chapter is a discussion on how you can design and develop a file system recognizer for your FSD. This FSCTL code is discussed in detail at that time.

`IRP_MN_VERIFY_VOLUME`

This is also a special type of FSCTL issued by the I/O Manager to an FSD managing a mounted logical volume on removable media. This request is issued by the I/O Manager when a lower-level disk driver indicates that the media in the removable driver appears to have been removed or changed. We will discuss how an FSD can develop an appropriate response to be executed in response to this type of FSCTL request.

Methods of Data Transfer for FSCTL Requests

Each FSCTL code value (used with the IRP_MN_USER_FS_REQUEST minor function) uniquely determines the method used for data transfer for that particular FSCTL operation, if such data transfer is requested. The two least significant bits in the FSCTL code value are used to identify the method of data transfer for the particular FSCTL request.

WARNING When a file system driver creates a device object to represent the file system itself or to represent an instance of a mounted logical volume, it can specify whether it wishes to receive buffered I/O requests (DO_BUFFERED_IO flag set in the Flags field for the device object), direct I/O requests (DO_DIRECT__IO flag set), or the user-supplied buffer pointer (neither of the two flags should be set).

You must note, however, that those flags are not used to determine the method of data transfer for file system control or device control requests. The method used in such cases is specific to each FSCTL or IOCTL sent to the driver and is determined by the FSCTL or IOCTL *code value* as described below.

The following are the available options:

- If the FSCTL code is defined with METHOD_BUFFERED, the I/O Manager allocates a system buffer on behalf of the caller.

  This method of data transfer can be defined by setting a value of 0 in the two least significant bits of the FSCTL code.

  The caller can supply either an input buffer only (used to transfer information to the FSD), an output buffer only (used to receive information back from the FSD), or both (data transfer occurs in both directions). However, the I/O Manager only allocates a single system buffer for the data transfer.

  The FSD can obtain the address of this single system buffer allocated by the I/O Manager from the AssociatedIrp->SystemBuffer field in the IRP. The Flags field in the IRP is set with the IRP_BUFFERED_IO and the IRP_DEALLOCATE_BUFFER flag values (used internally by the I/O Manager).

  If an input buffer is supplied by the caller, the I/O Manager will copy data from the input buffer to the I/O Manager-allocated system buffer, before passing the request to the FSD. If an output buffer is supplied by the caller, the I/O Manager will set the IRP_INPUT_OPERATION flag value in the Flags field in the IRP. The I/O Manager will check for the existence of this flag upon IRP completion and will copy data from the system buffer to the user-supplied output buffer if this flag is set.

  Note that, since the I/O Manager supplies a single buffer for the use of the FSD, even in the case when a caller may have provided both an input and an output buffer, the size of the system buffer allocated by the I/O Manager will be the greater of the size of the input and output buffers provided by the requesting thread. The initial contents of the I/O Manager-allocated system buffer will be overwritten by the FSD when it returns information back to the I/O Manager.

- If the FSCTL code is defined with METHOD_NEITHER, the I/O Manager simply sends the user-supplied buffer pointers directly to the FSD.

  This method of data transfer can be defined by setting a value of 3 in the two least-significant bits of the FSCTL code.

  If the caller provides an input buffer (i.e., a buffer in which the caller has provided data for the FSD), the I/O Manager initializes the Parameters.DeviceIoControl.Type3InputBuffer field in the current stack location with the pointer to the caller-supplied buffer. Your FSD can obtain data provided by the caller directly from this buffer.

  If the caller also wants to receive data back from the FSD, it would provide an output buffer pointer when invoking the NT system service routine.* In this case, the I/O Manager initializes the UserBuffer field in the IRP with the address of the caller-supplied output buffer. The FSD can return data to the caller by using the address provided in this field to write to the caller-supplied buffer.

  Note that the user-supplied buffer pointer addresses are supplied as-is to the FSD by the I/O Manager. No checks are performed by the I/O Manager on the user-supplied virtual addresses for either the input or the output buffers provided by the caller. Therefore, it is FSD's responsibility to ensure that the virtual addresses are still valid when it tries to perform data transfer for such requests. If the request is posted for asynchronous processing, the FSD must lock the input and/or the output buffers itself and also obtain valid system virtual addresses for each buffer.

- If the FSCTL code is defined with either the METHOD_IN_DIRECT or the METHOD_OUT_DIRECT FSCTL codes, the I/O Manager allocates a system buffer for the caller-supplied input buffer and creates an MDL for the caller-supplied output buffer.

  The METHOD_IN_DIRECT method of data transfer can be defined by setting a value of 1 in the least two significant bits of the FSCTL code. The METHOD_OUT_DIRECT method of data transfer can be defined by setting a value of 2 in the least two significant bits of the FSCTL code.

  The caller can supply an input buffer and an output buffer for both types of data transfer methods. The I/O Manager allocates a system buffer corresponding to the input buffer provided by the caller and copies the caller-supplied data from the input buffer into the I/O Manager-allocated system buffer. The

---

* The system service routine provided by the Windows NT I/O Manager for FSCTL requests is called NtFsControlFile(). For more information on this system service, consult *Appendix A.* The Win32 subsystem also provides a method of issuing device IOCTL requests to kernel-mode drivers.

address of this I/O Manager-allocated system buffer can be obtained by the FSD from the AssociatedIrp.SystemBuffer field in the IRP.

If the caller supplies an output buffer when invoking the NtFsControlFile() system service routine, the I/O Manager creates an MDL for the output buffer and also locks the pages for the MDL. The only difference between the METHOD_IN_DIRECT and the METHOD_OUT_DIRECT methods is that the pages locked by the I/O Manager are locked with read access specified in the former case (for METHOD_IN_DIRECT) and write access specified in the latter (for METHOD_OUT_DIRECT).

Note that the I/O Manager does not copy any data back into the caller-supplied output buffer upon IRP completion; since the output buffer is directly accessible to the FSD (via the MDL created by the I/O Manager), no such copy operation is required.

Standard User File System Control Requests

The I/O stack location contains the following structure relevant to processing file system control requests issued to an FSD:

```
typedef struct _IO_STACK_LOCATION {

    // ...

    union {

        //...

    // System service parameters for:  NtFsControlFile
    // Note that the user's output buffer is stored in the UserBuffer field
    // and the user's input buffer is stored in the SystemBuffer field.
    struct {
        ULONG OutputBufferLength;
        ULONG InputBufferLength;
        ULONG FsControlCode;
        PVOID Type3InputBuffer;
    } FileSystemControl;

        // ...
} Parameters;

// ...

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

The following FSCTL code values are defined by the system and should be supported by a disk-based FSD. For each FSCTL code mentioned below, there is a brief description of the type of processing performed by the native FSD implemen-

tations. This should provide you with a fairly good idea of the functionality expected from your FSD.

Note that each of these standard FSCTL requests is dispatched to the FSD in an IRP containing a major function code of IRP_MJ_FILE_SYSTEM_CONTROL and a minor function code of IRP_MN_USER_FS_REQUEST.

`FSCTL_LOCK_VOLUME`

Most NT FSD implementations typically execute the following steps:

a. If the file object supplied with the request does not refer to an open instance of the logical volume object, deny the request with an error code of STATUS_INVALID_PARAMETER.

b. Acquire the resource associated with your volume control block exclusively.

c. If the VCB state indicates that it is already locked, or if there are any open/referenced file objects for the logical volume represented by the VCB, deny the request (complete the IRP after releasing the VCB resource) with a STATUS_ACCESS_DENIED error code.

d. Mark the VCB as locked, preventing any new file create/open operations.

e. Flush any cached, modified metadata information about the logical volume (e.g., bitmaps).

f. Release the VCB resource and complete the IRP with a return code of STATUS_SUCCESS.

For the native FSD implementations, utilities such as *chkdsk* always lock a logical volume before beginning processing for the volume. Some FSD implementations may actually interpret this request as the beginning of a dismount sequence for a logical volume and prepare themselves accordingly.

`FSCTL_UNLOCK_VOLUME`

This simply undoes the lock operation performed with the FSCTL_LOCK_VOLUME request and clears any flags set in the VCB indicating that the volume has been locked.

Just as in the case of the lock volume request described, the FSD will reject the request if the file object supplied does not refer to an open instance of the previously locked logical volume.

`FSCTL_DISMOUNT_VOLUME`

The FSD will perform checks to ensure that the VCB indicates the logical volume was previously locked after acquiring the VCB resource exclusively. The FSD should then tear down all structures allocated to support the mounted logical volume, including the VCB structure itself. This would also

include uninitializing any cache map for the stream file object created to cache logical volume metadata information (see the description later in this chapter about the volume mount sequence). Of course, your FSD should ensure that all modified information (including log files, bitmaps, etc.) for the logical volume have been flushed to secondary storage before discarding this information from memory.

Also, the FSD should somehow indicate in the volume parameter block structure for the physical/virtual device object on which the volume had been mounted that the volume is no longer mounted.

There are a variety of ways in which your FSD could accomplish this. One way is to set the DO_VERIFY_VOLUME flag in the Flags field of VPB structure for the device object representing the physical/virtual disk. Another method could be to simply clear the VPB_MOUNTED flag in the VPB structure for the physical/virtual device object. Finally, your FSD could take drastic measures and free up the VPB structure allocated for the physical/virtual device object and replace it with a newly allocated "clean" VPB structure (remember to allocate it from nonpaged pool).

If you wish to modify flags in the VPB structure for the real device object on which your FSD mounted the logical volume (e.g., you decide to clear the VPB_MOUNTED flag), you should consider acquiring a global I/O Manager spin lock to ensure synchronized access to the structure. Here is the routine you can invoke to acquire this global spin lock:

```
VOID
loAcquireVpbSpinLock(
    OUT PKIRQL Irql
);
```

This routine will simply acquire the same global spin lock that the I/O Manager acquires internally before examining any VPB (e.g., to check whether the VPB is mounted during a create/open operation). Remember to pass in a pointer to a KIRQL structure so that the I/O Manager can return the IRQL at which your code was executing before it acquired the Executive spin lock. You will need this value when you are finished modifying the VPB structure, and you invoke this corresponding release spin lock routine:

```
VOID
IoReleaseVpbSpinLock(
    IN KIRQL Irql
);
```

Your FSD should check that this routine was invoked by a caller with appropriate privileges before allowing the request to be processed. Furthermore, if the logical volume was mounted on removable media that you had locked

into the drive, do not forget to issue an IOCTL to the removable media disk driver unlocking the medium from the drive.

`FSCTL_MARK_VOLUME_DIRTY`

Your FSD should confirm that the file object passed in reflects a valid instance of an open operation on the logical volume itself. This is simply a request to ensure that your in-memory and on-disk data structures reflect that the system memory may contain information that needs to be flushed out to disk. If the system crashes before your FSD has a chance to perform a flush operation and clear the on-disk flag reflecting the fact that the volume is dirty, you may decide to perform the equivalent of a *chkdsk* operation during the next boot cycle before allowing a logical volume mount request to complete.

Remember to acquire the VCB exclusively before modifying any in-memory or on-disk structures indicating that the volume is dirty and needs to be flushed to disk.

`FSCTL_IS_VOLUME_MOUNTED`

Ensure as before that a valid file object has been sent to you for this request. Typically, your FSD would support this FSCTL if you support removable media. If your FSD supports removable media and has a volume mounted on some such removable medium, you should perform the equivalent of a *verify volume operation* (described later) to ensure that everything is all right with the volume. An appropriate status code containing the results of the verify operation should be returned as part of completing the IRP.

`FSCTL_IS_PATHNAME_VALID`

The `AssociatedIrp->SystemBuffer` field in the FSCTL IRP will contain a pointer to this structure:

```
typedef struct _PATHNAME_BUFFER {
    ULONG         PathNameLength;
    WCHAR         Name[1];
} PATHNAME_BUFFER, *PPATHNAME_BUFFER;
```

Your mission is to examine the characters contained in the pathname to see if they are supported by your FSD. Return a status code of either STATUS_OBJECT_NAME_INVALID or STATUS_SUCCESS.

`FSCTL_QUERY_RETRIEVAL_POINTERS`

This request will only be directed to your FSD if it manages a boot partition on which a paging file resides. Providing a bootable FSD requires support from Microsoft. Therefore, we will ignore this FSCTL-type request and return STATUS_INVALID_PARAMETER to the caller.

In addition to the FSCTL codes listed, your FSD may support many privately defined file system control codes. Furthermore, if you design a network redirector driver, you may wish to provide functionality such as:

- Starting the redirector on demand
- Binding to transports used by your redirector
- Returning statistics pertinent to your driver
- Enumerating all open connections
- Deleting specific connections to remote shared objects
- Stopping redirector activities
- Unbinding from specific transports

You must determine the sort of functionality your driver will provide and implement appropriate FSCTL support.

Verify Volume Support

If your FSD supports removable media, there may be occasions when a verify volume request is issued to your driver. Typically, this happens whenever a user injects media into the removable drive.

Disk driver's actions

Whenever the media status in the removable drive appears to have changed, the disk driver performs the following actions for I/O requests targeted to the device.

1. Check if the VPB indicates whether a logical volume had been previously mounted on the media in the removable drive.

   This can be easily determined by the disk driver by the presence/absence of the VPB_MOUNTED flag in the Flags field in the VPB structure. If the flag is not set, no logical mount operation has been performed, and the driver simply returns STATUS_VERIFY_REQUIRED for IRPs sent to the device.

2. If a logical volume had been mounted, indicate that the media needs to be verified.

   The disk driver will OR in the DO_VERIFY_VOLUME flag in the VPB structure. It will set the return code value for the IRP to STATUS_VERIFY_REQUIRED. It will then invoke the IoSetHardErrorOrVeriFyDevice () function, which will store the pointer to the supplied device object (one of the arguments to this well-documented function) in the Tail.Overlay . Thread->DeviceToVerify field of the IRP.

3. The disk driver will then complete the IRP.

FSD response

Note that most I/O operations to a disk drive with a mounted logical volume associated with the media in the drive originate in the FSD. Whenever an FSD gets a STATUS_VERIFY_REQUIRED error from an I/O request sent to the target device object for a logical volume, it performs the following actions:

1. Obtain a pointer to the target device object for the verify operation to be performed.

   The FSD should use the loGetDeviceToVerify() function call to get a pointer to this device object. It should then reset the DeviceToVerify field in the TLS to NULL by invoking loSetDeviceToVerify() function with the arguments: (PsGetCurrentThread(), NULL).

2. Initiate a verify operation.

   The FSD can simply invoke the loVerifyVolume() function to initiate a verify operation:

```
NTSTATUS
loVerifyVolume(
    IN PDEVICE_OBJECT        DeviceObject,
    IN BOOLEAN               AllowRawMount
);
```

   The FSD can pass in the pointer to the device object for the device to be verified (obtained earlier from the loGetDeviceToVerify() function call). The AllowRawMount is typically set to FALSE, unless the user was trying to perform a create/open operation on the physical device itself, and the FSD encountered the verify status code when processing this request.

   Note that the invocation to loVerifyVolume() will return either STATUS_SUCCESS or STATUS_WRONG_VOLUME.

I/O manager's response

When an FSD invokes the loVerifyVolume() function call as described, the I/O Manager will do the following:

1. If the logical volume had not been previously mounted (FALSE in the scenario described here), simply invoke a mount sequence.

   The mount sequence consists of going through the linked list of all registered, loaded instances of file system drivers and invoking each one of them, requesting a mount operation. Later in this chapter, you will read a detailed discussion on how a mount request is processed by the FSD.

   Note that a mount request is issued to the FSD via an FSCTL that has a minor function value of IRP_MN_MOUNT_VOLUME.

2. Since the logical volume had been previously mounted, issue a verify volume FSCTL request to the FSD.

   The I/O Manager will create a new IRP for the FSCTL request. The minor function code in the current I/O stack location will be set to IRP_MN_VERIFY_VOLUME. The I/O Manager will then issue the IRP to the FSD, since it manages the logical volume device object identified by the DeviceObject field in the VPB structure for the device object representing the removable drive to be verified.*

   The Parameters.VerifyVolume.Vpb field in the current stack location of the verify volume IRP dispatched to the FSD contains a pointer to the VPB, associated with the device object representing the removable drive containing the media to be verified. The Parameters.VerifyVolume.DeviceObject field contains a pointer to the logical volume device object created by the FSD.

3. If the verify volume FSCTL returns STATUS_SUCCESS, there is nothing more the I/O Manager needs to do.

4. If the verify volume FSCTL returns STATUS_WRONG_VOLUME, the I/O Manager will initiate a fresh mount sequence for the device.

   To initiate a new mount sequence, the I/O Manager will free the original VPB structure associated with the device object on which the mount operation will be attempted. It will allocate a new VPB structure and associate it with the device object. It will then begin the typical mount sequence.

FSD's response to the verify volume FSCTL request

The IRP issued by the I/O Manager to verify the volume can be easily identified by the FSD by the IRP_MN_VERIFY_VOLUME minor function code value in the current I/O stack location.

The I/O stack location contains the following structure relevant to processing the verify volume request issued to an FSD:

```
typedef struct _IO_STACK_LOCATION {

    // ...

    union {

        //...

        // Parameters for VerifyVolume
        struct {
```

* Sorry if that sounds cryptic but it is true, I promise.

```
            PVPB Vpb;
            PDEVICE_OBJECT DeviceObject;
        } VerifyVolume;

        // ...
} Parameters;

// ...

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

The FSD performs the following sequence of actions in response to the request:

1. Post the request if required.

   Note that a verify request is inherently synchronous, and the I/O Manager will wait in the context of the thread performing the verify operation if STATUS_PENDING is returned. Your FSD would typically post the request if the `IoIsOperationSynchronous()` function call returns FALSE.

2. Get a pointer to the VCB structure for the logical volume device object.

3. Acquire the VCB resource exclusively to ensure synchronized access.

4. Check if the `RealDevice->Flags` field is no longer marked with DO_VERIFY_VOLUME.

   Since multiple IRP requests to the disk driver could fail with a verify volume status code, one of those requests could have already resulted in a volume verify operation being completed. There is nothing the FSD needs to do in this situation but return STATUS_SUCCESS.

5. Issue requests to the disk driver to obtain information from the physical media.

   The steps performed here are similar to those executed during a logical mount operation described below. Basically, the FSD must obtain whatever information is required from the physical media, including getting the drive geometry by issuing IOCTL requests to the disk driver and issuing read requests to obtain volume metadata information from disk.

   In order to ensure that the disk driver does not fail the I/O requests with a STATUS_VERIFY_VOLUME error code, the FSD must set the SL_OVERRIDE_VERIFY_VOLUME flag in the `Flags` field of the stack location it sets up for the next lower driver in the chain.

   If any of the I/O operations sent to the lower-level driver fail, the FSD typically decides to return STATUS_WRONG_VOLUME. Skip directly to the step described below detailing the preprocessing required from the FSD before returning this error code to the I/O Manager.

6. Check the information obtained from disk.

   Your FSD may perform any appropriate checks to decide if the on-disk structures indicate the same volume as the one you had previously mounted.

7. If it determines that the volume is the same, flush and purge all cached metadata structures for the logical volume and reinitialize all cached information.

   This is similar to performing a remount operation on the logical volume. Once it has reinitialized cached data, your FSD should clear the DO_VERIFY_VOLUME flag in the VPB. Then it should complete the IRP with STATUS_SUCCESS as the return code.

8. If it decides that the volume is not the same as the one previously mounted, throw away all cached metadata information for the volume.

   Effectively, you will perform a forced dismount of the volume at this time. You should also clear the DO_VERIFY_VOLUME flag in the VPB. Then your FSD should complete the IRP with STATUS_WRONG_VOLUME as the return code. Since you will return the STATUS_WRONG_VOLUME error code, the I/O Manager will attempt a remount operation for the media.

9. Complete the FSCTL IRP with the appropriate return code value.

Handling Device IOCTL Requests

The I/O stack location contains the following structure relevant to processing the device IOCTL request issued to an FSD:

```
typedef struct _IO_STACK_LOCATION {

    // ...

    union {

        //...

        // System service parameters for:  NtDeviceloControlFile
        // Note that the user's output buffer is stored in the UserBuffer
        // field and the user's input buffer is stored in the SystemBuffer
        // field.
        struct {
            ULONG OutputBufferLength;
            ULONG InputBufferLength;
            ULONG loControlCode;
            PVOID Type3InputBuffer;
        } DeviceloControl;

        // ...
} Parameters;

// ...
```

```
} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

Typically, the FSD should simply forward a device IOCTL request to the target device object for the mounted logical volume. Study the following code fragment to see how this can be done.

```
NTSTATUS    SFsdCommonDeviceControl(
PtrSFsdlrpContext            PtrlrpContext,
PIRP                         Ptrlrp)
{
    NTSTATUS                 RC = STATUS_SUCCESS;
    PIO_STACK_LOCATION       PtrloStackLocation = NULL;
    PIO_STACK_LOCATION       PtrNextloStackLocation = NULL;
    PFILE_OBJECT             PtrFileObject = NULL;
    PtrSFsdFCB               PtrFCB = NULL;
    PtrSFsdCCB               PtrCCB = NULL;
    PtrSFsdVCB               PtrVCB = NULL;
    BOOLEAN                  Completelrp = FALSE;
    ULONG                    loControlCode = 0;
    void                     *BufferPointer = NULL;

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

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

        PtrCCB = (PtrSFsdCCB) (PtrFileObject->FsContext2) ;
        ASSERT(PtrCCB);
        PtrFCB = PtrCCB->PtrFCB;
        ASSERT(PtrFCB);

        if (PtrFCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_VCB ) {
            PtrVCB = (PtrSFsdVCB) (PtrFCB) ;
        } else {
            PtrVCB = PtrFCB->PtrVCB;
        }

        // Get the loControlCode value
        loControlCode =
            PtrIoStackLocation->Parameters.DeviceloControl.loControlCode;

        // You may wish to allow only volume open operations.

        switch (loControlCode) {
#ifdef    __THIS_IS_A_NETWORK_REDIR_
        case IOCTL_REDIR_QUERY_PATH:
            // Only for network redirectors.
            BufferPointer = (void *)
                (PtrIoStackLocation->
                    Parameters.DeviceIoControl.Type3InputBuffer);
            // Invoke the handler for this IOCTL.
```

```
            RC = SFsdHandleQueryPath(BufferPointer);
            Completelrp = TRUE;
            try_return(RC);
            break;
#endif    // _THIS_IS_A_NETWORK_REDIR_
        default:
            // Invoke the lower-level driver in the chain.
            PtrNextloStackLocation = loGetNextlrpStackLocation(Ptrlrp);
            *PtrNextIoStackLocation = *PtrIoStackLocation;
            // Set a completion routine.
            loSetCompletionRoutine(Ptrlrp, SFsdDevIoctlCompletion,
                    NULL, TRUE, TRUE, TRUE);
            // Send the request.
            RC = loCallDriver(PtrVCB->TargetDeviceObject, Ptrlrp);
            break;
        }

        try_exit:    NOTHING;

    } finally {

        // Release the IRP context
        if (!(PtrIrpContext->IrpContextFlags
                                & SFSD_IRP_CONTEXT_EXCEPTION)) {
            // Free-up the Irp Context
            SFsdReleaselrpContext(PtrlrpContext);

            if (Completelrp) {
                PtrIrp->IoStatus.Status = RC;
                PtrIrp->IoStatus.Information = 0;

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

    return(RC);
}

NTSTATUS SFsdDevIoctlCompletion(
PDEVICE_OBJECT           PtrDeviceObject,
PIRP                      Ptrlrp,
void                      *Context)
{
    if (PtrIrp->PendingReturned) {
        loMarklrpPending(Ptrlrp);
    }

    return(STATUS_SUCCESS);
}
```

This code also illustrates how *z* network redirector can provide support for the IOCTL_REDIR_QUERY_PATH IOCTL issued by the MUP component (discussed

earlier in Chapter 2, *File System Driver Development).* See the following code fragment for a skeletal SFsdHandleQueryPath( ) routine example.

```
NTSTATUS SFsdHandleQueryPath(
void            *BufferPointer)
{
    NTSTATUS                RC = STATUS_SUCCESS;
    PQUERY_PATH_REQUEST     RequestBuffer = (PQUERY_PATH_
REQUEST) BufferPointer;
    PQUERY_PATH_RESPONSE    ReplyBuffer = (PQUERY_PATH_
RESPONSE) BufferPointer;
    ULONG                   LengthOfNameToBeMatched =
                             RequestBuffer->PathNameLength;
    ULONG                   LengthOfMatchedName = 0;
    WCHAR                   *NameToBeMatched = RequestBuffer->FilePathName;

    // So here we are. Simply check the name supplied.
    // You can use whatever algorithm you like to determine whether the
    // sent-in name is acceptable.
    // The first character in the name is always a "\"
    // If you like the name sent-in (probably, you will like a subset
    // of the name), set the matching length value in LengthOfMatchedName.

    //if (FoundMatch) {
    //       ReplyBuffer->LengthAccepted = LengthOfMatchedName;
    // } else {
    //       RC = STATUS_OBJECT_NAME__NOT_FOUND;
    // }

    return(RC);
}
```

The following definitions are required by the code fragment:

```
#define IOCTL_REDIR_QUERY_PATH     \
CTL_CODE(FILE_DEVICE_NETWORK_FILE_SYSTEM, 99,
            METHOD_NEITHER, FILE_ANY_ACCESS)

typedef struct _QUERY_PATH_REQUEST {
    ULONG                          PathNameLength;
    PIO_SECURITY_CONTEXT           SecurityContext;
    WCHAR                          FilePathName[1];
} QUERY_PATH_REQUEST, *PQUERY_PATH_REQUEST;

typedef struct _QUERY_PATH_RESPONSE {
    ULONG                          LengthAccepted;
} QUERY_PATH_RESPONSE, *PQUERY_PATH_RESPONSE;
```

File System Recognizers

Simply stated, a file system recognizer is a mini-FSD implementation that loads initially instead of the full FSD.

Functionality Provided by a File System Recognizer

The function of the file system recognizer driver is as follows:

- Help conserve system resources by loading the recognizer instead of the complete FSD.

  The mini-FSD is, by definition, a small driver providing almost no functionality (except what is discussed below) and so consuming very few system resources.

- If a valid logical volume needs to be mounted, load the full (original) FSD so that it can proceed with mounting the volume and servicing user requests.

  Once the full FSD has been successfully loaded into memory, the file system recognizer essentially becomes dormant and stays out of the way. Because of the low resource requirements for the mini-FSD, keeping it loaded in memory even after the full FSD has been loaded is a small price to pay compared to the benefits of using the mini-FSD in the first place.

Basically, the file system recognizer helps the Windows NT operating system conserve system resources by obviating the necessity of always loading the entire FSD even if no logical volumes belonging to the FSD are ever mounted (or used) by users of the system. For example, consider the CD-ROM drive that exists on your system. It is possible that you may not use the CD-ROM at all during the current boot cycle. Or, it is quite possible that you have formatted all of your hard disk partitions with the NTFS file system format and therefore, you never need to use the FASTFAT file system driver on your machine until you decide to use a diskette formatted with the FAT file system format.

In such situations, loading the entire FASTFAT and/or CDFS file system drivers into memory is an unnecessary operation that is costly in terms of the time required to boot-up the system as well as the memory consumption associated with the FSD that is inevitable even for a dormant, loaded FSD.

A mini-FSD is a cost-effective method of always being prepared for the possibility that a user may require the services of the associated FSD, while not incurring the performance and resource penalties of actually having a fully functional FSD loaded in memory until it becomes necessary to do so.

Steps Executed by the File System Recognizer

The mini-FSD (or the file system recognizer, as it is commonly known), executes the following logical steps once it is loaded into memory:

1. Create a device object representing the mini-FSD in lieu of the file-system-type device object that the full fledged FSD would create, had it been loaded.

   The file system recognizer creates a device object of type FILE_DEVICE_CD_ROM_FILE_SYSTEM (for a CD-ROM file system recognizer) or FILE_DEVICE_DISK_FILE_SYSTEM (for the more common, disk-based file system recognizer). As you may have noted from the initialization code presented in Chapter 9, this is similar to the operation generally executed by the full FSD implementation.

2. Register with the I/O Manager as a file system driver so that the mini-FSD gets invoked whenever an I/O request is received targeted to a physical device on which no mount operation has been performed.

   Just as in the case of any other fully functional disk-based FSD (as illustrated in Chapter 9), the mini-FSD also invokes `IoRegisterFileSystem()` to inform the I/O Manager that a fully functional FSD has been loaded into memory.

3. Upon receiving a mount request for a physical/virtual device, check the on-disk information on the device by performing I/O operations to determine whether the device contains a valid (recognizable) logical volume.

   Recall from earlier chapters the sequence of operations undertaken by the I/O Manager whenever it receives a create/open request for an object on a physical/virtual device. For example, consider the situation when a user decides to open file *X:\directoryl\foo.* The NT Object Manager receives the request and translates *X:* (which is simply a symbolic link) to the linked object name, e.g., *\Device\PhysicalDriveO**

   So the complete name of the request as determined by the NT Object Manager is now *\Device\PhysicalDriveO\directoryl\foo.* Since the *\Device\PhysicalDriveO* name typically corresponds to the device object for the first partition on hard disk 0 (an object belonging to the I/O subsystem), the Object Manager recognizes that the request should be forwarded to the device object managed by the I/O Manager and therefore sends the request on to the I/O Manager for further processing. The portion of the name sent to the I/O Manager is *\directoryl\foo,* with the target device object for the request being clearly identified by the NT Object Manager.

   The I/O Manager, in turn, examines the volume parameter block structure associated with the physical device object to see if any logical volume has been mounted on the device object. The presence of a mounted logical

---

* The *\??\...* names in Windows NT Version 4.0 are simply symbolic links themselves to the corresponding *\I)eince\...* entries.

volume associated with a physical/virtual device object can be detected by checking for the VPB_MOUNTED flag value in the Flags field of the VPB structure. If a logical mount operation had been successfully performed, the I/O Manager will send the create/open request to the FSD that manages the logical volume (represented by a logical volume device object associated with the physical device object) to actually process the request.

However, if the VPB indicates that no logical mount operation had been performed for the target physical/virtual device object, the I/O Manager sends an IRP with the IRP_MJ_FILE_SYSTEM_CONTROL major function code and the IRP_MN_MOUNT_VOLUME minor function code to each of the registered disk/CD-ROM file system drivers loaded in the system. The first FSD to successfully perform the mount operation causes the I/O Manager to stop issuing any further mount requests to the remaining FSDs.

Since the mini-FSD has registered itself as a fully functional, loaded FSD, it, too, receives such mount requests from the I/O Manager. Upon receiving such a mount request for the physical/virtual device (in our example, the device object identified by *\Device\PhysicalDriveO),* the mini-FSD obtains the disk geometry and device type by issuing IOCTL requests to the device driver managing the device, and it also reads in the metadata information from appropriate physical sectors on the media.

The mini-FSD then checks to see whether the information obtained from the disk matches the expected information that would indicate that a valid, supported logical volume resides on the physical media (or on the virtual device).

4. If no valid structures are found on the target physical/virtual device, return an error code of STATUS_UNRECOGNIZED_VOLUME to the I/O Manager, which will cause the I/O Manager to pass on the request to the next registered file system (or mini-FSD).

   Any file system recognizer supplied along with your FSD must be capable of detecting the presence/absence of valid metadata information on the storage medium, to determine whether or not the disk actually contains a valid logical volume. These checks need not be conclusive; i.e., as long as the mini-FSD believes that a valid logical volume exists/does not exist on the disk, it can choose a reasonable course of action to pursue.

5. If structures (metadata) on the target physical device indicate that a valid logical volume exists on the device, then return STATUS_FS_DRIVER_ REQUIRED to the I/O Manager.

   Returning STATUS_FS_DRIVER_REQUIRED to the I/O Manager results in the I/O Manager issuing another IRP_MJ_FILE_SYSTEM_CONTROL request

to the file system recognizer; this time though, the minor function code will be IRP_MN_LOAD_FILE_SYSTEM indicating that the mini-FSD should proceed with attempting to load the full FSD implementation into memory.

6. Upon receiving the FSCTL request with a minor function of IRP_MN_LOAD_FILE_SYSTEM, attempt to load the full FSD into memory.

   This can be accomplished by using the ZwLoadDriver() support routine. See the sample code fragment provided for an example of the usage of this routine.

7. Remember the result of the load operation and take appropriate steps.

   Typically, if the load request succeeds, the mini-FSD can render itself dormant by simply unregistering itself from the list of registered file system implementations maintained by the I/O Manager. This ensures that the I/O Manager will no longer send mount requests to the mini-FSD.

   If the load request fails, it is recommended by the NT I/O subsystem designers that the mini-FSD remember this failure in a device extension field and never again (during the current boot cycle) attempt to reload the FSD. Instead, upon receiving further mount requests, the mini-FSD should simply reject them immediately with the return code STATUS_UNRECOGNIZED_VOLUME. This will allow the I/O Manager to try some other loaded FSD instead.

   Note that it is not mandatory for your mini-FSD to remember a previous failure if it believes that the next time around there is a better chance of the load request succeeding.

You should note that file system recognizers typically exist only for disk-based (including CD-ROM-based) file system implementations. Network redirectors typically do not use a VCB structure, and they also do not typically have mini-FSD implementations.

The sample code fragment illustrates how you could develop your own file system recognizer:

Code Sample

```
NTSTATUS DriverEntry(
PDRIVER_OBJECT     DriverObject,
PUNICODE_STRING    RegistryPath)
{
    NTSTATUS                RC = STATUS_SUCCESS;
    UNICODE_STRING           DriverDeviceName;
    UNICODE_STRING           FileSystemName;
    OBJECT_ATTRIBUTES       ObjectAttributes;
    HANDLE                   FileSystemHandle = NULL;
```

```
 IO_STATUS_BLOCK           loStatus;
PtrSFsRecDeviceExtension   PtrExtension = NULL;

 try {
     try {
         // Initialize the IRP major function table
         DriverObject->MajorFunction[IRP_MJ_FILE_SYSTEM_CONTROL]=
                                                    SFsRecFsControl ;
         DriverObject->DriverUnload = SFsRecUnload;

         // Before creating a device object, check whether the FSD has
         // been loaded already. You should know the name of the FSD
         // that this recognizer has been created for.
         RtllnitUnicodeString(&FileSystemName, L"\\SampleFSD");
         InitializeObjectAttributes(&ObjectAttributes, &FileSystemName,
                 OBJ_CASE_INSENSITIVE, NULL, NULL);
         // Try to open the file system now.
         RC = ZwCreateFile(&FileSystemHandle, SYNCHRONIZE,
                                  &ObjectAttributes,
                                  &IoStatus, NULL, 0,
                                  FILE_SHARE_READ | FILE_SHARE_WRITE,
                                  FILE_OPEN, 0, NULL, 0);
         if (RC != STATUS_OBJECT_NAME_NOT_FOUND) {
             // The FSD must have been already loaded.
             if (NT_SUCCESS(RC)) {
                 ZwClose(FileSystemHandle);
             }
             RC = STATUS_IMAGE_ALREADY_LOADED;
             try_return(RC) ;
         }

         // Create a device object representing the file system
         // recognizer. Mount requests are sent to this device object.
         RtllnitUnicodeString(ScDriverDeviceName,
                                 L"\\SampleFSDRecognizer");

         if ( !NT_SUCCESS(RC = loCreateDevice(
                 DriverObject,            // Driver object for the file
                                          // system rec.
                 sizeof(SFsRecDeviceExtension), // Did a load fail?
                 &DriverDeviceName,     // Name used above
                 FILE_DEVICE_DISK_FILE_SYSTEM,
                 0,                   //No special characteristics
                 FALSE,
                 &(PtrFSRecDeviceObject)))){
             try_return(RC) ;
         }

         PtrExtension =
             (PtrSFsRecDeviceExtension)(PtrFSRecDeviceObject->
                                                  DeviceExtension) ;

         PtrExtension->DidLoadFail = FALSE;
```

```
            II Register the device object with the I/O Manager.
          IoRegisterFileSystem(PtrFSRecDeviceObject)  ;

        } except (EXCEPTION_EXECUTE_HANDLER) {
            /* we encountered an exception somewhere, eat it up */
            RC = GetExceptionCode ( ) ;
        }

        try_exit:    NOTHING;
    } finally {
        /* start unwinding if we were unsuccessful */
        if ( !NT_SUCCESS(RC) && PtrFSRecDeviceObject) {
            IoDeleteDevice(PtrFSRecDeviceObject)    ;
            PtrFSRecDeviceObject = NULL;
        }
    }

    return (RC) ;
}

void SFsRecUnload(
PDRIVER_OBJECT        PtrFsRecDriverObject)
{
    // Simple. Unregister the device object, and delete it.
    if (PtrFSRecDeviceObject) {
        loUnregisterFileSys tern (PtrFSRecDeviceObject) ;
        IoDeleteDevice( PtrFSRecDeviceObject) ;

        PtrFSRecDeviceObject = NULL;
    }

    return;
}

NTSTATUS SFsRecFsControl(
PDEVICE_OBJECT        DeviceObject,
PIRP                  Irp)
{
    NTSTATUS                          RC = STATUS_UNRECOGNIZED_VOLUME;
    PIO_STACK_LOCATION                PtrloStackLocation = NULL;
    PtrSFsRecDeviceExtension          PtrExtension = NULL;
    PDEVICE_OBJECT                    PtrTargetDeviceObject = NULL;
    UNICODE_STRING                    DriverName;

    FsRtlEnterFileSystem();

    try {
        try {
            PtrloStackLocation = loGetCurrentlrpStackLocation(Irp) ;
            ASSERT(PtrloStackLocation);

            // Get a pointer to the device object extension.
            PtrExtension =
                 (PtrSFsRecDeviceExtension) (PtrFSRecDeviceObject->
                                                    DeviceExtension)        ;
```

```
        switch (PtrIoStackLocation->MinorFunction) {
        case IRP_MN_MOUNT_VOLUME:
            // Fail the request immediately if a previous load has
            // failed. You are not required to do this, however, in
            // your driver.
            if (PtrExtension->DidLoadFail) {
                try_return(RC);
            }

            // Get a pointer to the target physical/virtual device
            // object.
            PtrTargetDeviceObject =
                PtrIoStackLocation->
                          Parameters.MountVolume.DeviceObject;

            // The operations that you perform here are highly FSD
            // specific. Typically, you would invoke an internal
            // function that would
            // (a) Get the disk geometry by issuing an IOCTL
            // (b) Read the first few sectors (or appropriate sectors)
            //     to verify the on-disk metadata information.
            //To get the drive geometry, use the documented I/O
            // Manager routine called IoBuildDeviceIoControlRequest( )
            //to create an IRP. Supply an event with this request
            // that you will wait for in case the lower-level driver
            // returns STATUS_PENDING. Similarly, to actually read on-
            // disk sectors, create an IRP using the
            // loBuildSynchronousFsdRequest{) function call with a
            // major function of IRP_MJ_READ.

            // After you have obtained on-disk information, verify the
            // metadata. RC =
            // SFsRecGetDisklnfoAndVerify(PtrTargetDeviceObject);

            if (NT_SUCCESS(RC) ) {
                // Everything looks good. Prepare to load the driver.
                try_return(RC = STATUS_FS_DRIVER_REQUIRED);
            }
            break;
        case IRP_MN_LOAD_FILE_SYSTEM:
            // OK. So we processed a mount request and returned
            // STATUS_FS_DRIVER_REQUIRED to the I/O Manager.
            // This is the result. Talk about an ungrateful I/O
            // Manager making us do more work!
            RtlInitUnicodeString(&DriverName,
    L" \\Registry\\Machine\\System\\CurrentControlSet\\Services\\SFsd" ) ;
            RC = ZwLoadDriver(&DriverName);
            if ( ( !NT_SUCCESS(RC) ) && (RC !=
                                        STATUS_IMAGE_ALREADY_LOADED) ) {
                PtrExtension->DidLoadFail = TRUE;
            } else {
                // Load succeeded. Mission accomplished.
                loUnregisterFileSys tern (PtrFSRecDeviceObj ect) ;
            }
```

```
                break;
            default:
                RC = STATUS_INVALID_DEVICE_REQUEST;
                break;
            }

        } except (EXCEPTION_EXECUTE_HANDLER) {
            RC = GetExceptionCode();
        }

        try_exit:    NOTHING;

    } finally {
        // Complete the IRP.
        Irp->IoStatus.Status = RC;
        loCompleteRequest(Irp, IO_NO_INCREMENT);
    }

    FsRtlExitFileSystem();

    return(RC) ;
}
```

The following structure definitions are used by the code fragment:

```
typedef struct SFsRecDeviceExtension {
  BOOLEAN        DidLoadFail;
} SFsRecDeviceExtension, *PtrSFsRecDeviceExtension;

PDEVICE_OBJECT  PtrFSRecDeviceObject = NULL;
unsigned int    SFsRecDidLoadFail = 0;

extern NTSTATUS ZwLoadDriver(
IN PUNICODE_STRING       DriverName) ;
```

Notes

As you can see, developing a file system recognizer (mini-FSD) is not difficult at all. One point to note, in the event that you do provide a file system recognizer module with your FSD implementation, is how you should configure the Registry in order to load the recognizer automatically.

Chapter 9 lists the entries required in the Windows NT Registry to install a full FSD. You should make the following modifications:

- Modify *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Sample-FSD\Start* to have a value of 4.
- You should also add a new entry for the file system recognizer, e.g., *HKEY_LOCAL_ MACHINE\SYSTEM\CurrentControlSet\Services\SFsRec.*

This key should at least include the following value entries:

- `ErrorControl : REG_DWORD : 0`
- `Group : REG_SZ : Boot File System`
- `Start : REG_DWORD : Oxl`
- `Type : REG_DWORD : 0x8`

  This indicates that the type of kernel service is SERVICE_RECOGNIZER_ DRIVER (a file system recognizer).

What Happens After the FSD Is Loaded?

Once a file system has been successfully loaded, the mini-FSD returns STATUS_ SUCCESS to the Windows NT I/O Manager. The I/O Manager then queries all the loaded FSD instances once again, asking each one to mount the logical volume on the target physical/virtual device object.

This mount request will eventually reach the newly loaded (our sample) file system driver. The request is dispatched to the driver as a FSCTL request. The IRP contains a major function code of IRP_MJ_FILE_SYSTEM_CONTROL and a minor function code of IRP_MN_MOUNT_VOLUME. The **Flags** field in the current I/O stack location is a value of (IRP_MOUNT_COMPLETTON | IRP_ SYNCHRONOUS_PAG ING_1 0) .

The I/O stack location contains the following structure relevant to processing the mount request issued to an FSD:

```
typedef struct _IO_STACK_LOCATION {

    // ...

    union {

        //...

        // Parameters for MountVolume
        struct {
            PVPB Vpb;
            PDEVICE_OBJECT DeviceObj ect;
        } MountVolume;

        // ...
} Parameters;

// ...

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

The FSD must perform a logical mount operation upon receiving the mount request. The following sequence of logical steps are typically executed by the FSD when it receives the mount request:

1. The FSD will obtain the partition information, i.e., the driver geometry, by building a device IOCTL IRP and issuing it to the driver managing the target device object.

   The loBuildDeviceloControlRequest ( ) support routine, provided by the I/O Manager, can be used to create an IRP that is then sent to the lower-level driver. Typically, the IOCTL code used is IOCTL_DISK_GET_PARTITION_INFO for read/write media.

   Note that CDFS issues two separate IOCTL requests to the disk driver with IOCTL codes specified as IOCTL_CDROM_CHECK_VERIFY and IOCTL_CDROM_GET_DRIVE_GEOMETRY, respectively.

2. Once partition information has been successfully obtained, the FSD will typically create a device object representing the instance of the mounted volume.

   The device object created would have a specified type of either FILE_DEVICE_DISK_FILE_SYSTEM or FILE_DEVICE_CD_ROM_FILE_SYSTEM.

   Note that the sample FSD defines a volume control block structure representing an instance of a mounted logical volume. The sample FSD implementation allocates this VCB structure as the device extension for the device object created to represent the mounted logical volume. Your driver does not have to use the same methodology. However, this would be a good place for your driver to allocate a VCB structure (from nonpaged pool) and initialize it appropriately.

   To see the kind of initialization performed by the sample FSD, consult this code fragment:

```
void SFsdInitializeVCB(
PDEVICE_OBJECT           PtrVolumeDeviceObj ect,
PDEVICE_OBJECT           PtrTargetDeviceObj ect,
PVPB                      PtrVPB)
{
    NTSTATUS             RC = STATUS_SUCCESS ;
    PtrSFsdVCB           PtrVCB = NULL;
    BOOLEAN              VCBResourcelnitialized = FALSE;

    PtrVCB = (PtrSFsdVCB)(PtrVolumeDeviceObject->DeviceExtension);

    // Zero it out (typically this has already been done by the I/O
    // Manager but it does not hurt to do it again).
    RtlZeroMemory(PtrVCB, sizeof(SFsdVCB));

    // Initialize the signature fields
    PtrVCB->NodeIdentifier.NodeType = SFSD_NODE_TYPE_VCB;
    PtrVCB->NodeIdentifier.NodeSize = sizeof(SFsdVCB);
```

```
II Initialize the ERESOURCE object.
RC = ExInitializeResourceLite(&(PtrVCB->VCBResource));
ASSERT(NT_SUCCESS(RC));
VCBResourcelnitialized = TRUE;

// We know the target device object.
// Note that this is not necessarily a pointer to the actual
// physical/virtual device on which the logical volume should
// be mounted. This is a pointer to either the actual
// device or any device object that may have been
// attached to it. Any IRPs that we send should be sent to this
// device object. However, the "real" physical/virtual device
// object on which we perform our mount operation can be
// determined from the RealDevice field in the VPB sent to us.
PtrVCB->TargetDeviceObject = PtrTargetDeviceObject;

// We also have a pointer to the newly created device object
// representing this logical volume (remember that this VCB
// structure is simply an extension of the created device object).
PtrVCB->VCBDeviceObject = PtrVolumeDeviceObject;

//We also have the VPB pointer. This was obtained from the
// Parameters.MountVolume.Vpb field in the current I/O stack
// location for the mount IRP.
PtrVCB->PtrVPB = PtrVPB;

// Initialize the list-anchor (head) for some lists in this VCB.
InitializeListHead(&(PtrVCB->NextFCB));
InitializeListHead(&(PtrVCB->NextNotifyIRP));
InitializeListHead(&(PtrVCB->VolumeOpenListHead));

// Initialize the notify IRP list mutex
KeInitializeMutex(&(PtrVCB->NotifyIRPMutex), 0);


// Set the initial file size values appropriately. Note that your
// FSD may guess at the initial amount of information you would
// like to read from the disk until you have really determined
// that this a valid logical volume (on disk) that you wish to
// mount. PtrVCB->FileSize = PtrVCB->AllocationSize = ??

// You typically do not want to bother with valid data length
// callbacks from the Cache Manager for the file stream opened for
// volume metadata information
PtrVCB->ValidDataLength.LowPart = OxFFFFFFFF;
PtrVCB->ValidDataLength.HighPart = 0x7FFFFFFF;

// Create a stream file object for this volume.
PtrVCB->PtrStreamFileObject = loCreateStreamFileObject(NULL,
                                        PtrVCB->PtrVPB->RealDevice);
ASSERT(PtrVCB->PtrStreamFileObject);

// Initialize some important fields in the newly created file
// object.
```

```
    PtrVCB->PtrStreamFileObject->FsContext = (void *)PtrVCB;
    PtrVCB->PtrStreamFileObject->FsContext2 = NULL;
    PtrVCB->PtrStreamFileObject->SectionObjectPointer =
                    &(PtrVCB->SectionObject);

     PtrVCB->PtrStreamFileObject->Vpb = PtrVPB;

     // Link this chap onto the global linked list of all VCB
structures.

ExAcquireResourceExclusiveLite(&(SFsdGlobalData.GlobalDataResource),
                                                    TRUE);
     InsertTailList(&(SFsdGlobalData.NextVCB), &(PtrVCB->NextVCB));

     // Initialize caching for the stream file object.
     CcInitializeCacheMap(PtrVCB->PtrStreamFileObject,
                          (PCC_FILE_SIZES)(&(PtrVCB->AllocationSize)),
                               TRUE,        // We will use pinned
                                            // access.
                               &(SFsdGlobalData.CacheMgrCallBacks),
                               PtrVCB);


     SFsdReleaseResource(&(SFsdGlobalData.GlobalDataResource));

     // Mark the fact that this VCB structure is initialized.
    SFsdSetFlag(PtrVCB->VCBFlags, SFSD_VCB_FLAGS_VCB_INITIALIZED);
     return;
}
```

Remember to perform the following modifications to the device object created to represent the mounted logical volume instance:

— Check the alignment restriction enforced by the target physical/virtual device object.

  If the alignment requirement mandated by the target device object is greater than that specified by your FSD, modify the AlignmentRequirement field in the newly created device object to reflect that of the target device.

— Remember to clear the DO_DEVICE_INITIALIZING flag from the Flags field in the newly created device object.

  For device objects created during driver load time, the I/O Manager automatically performs this task for you. If, however, you forget to clear this flag value for device objects created by your FSD after the driver initialization has been completed, your FSD will not receive any IRPs, because the I/O Manager will fail any requests sent to the device immediately.

— Set the StackSize value in the newly created device object to be equal to (TargetDeviceObject->StackSize + 1).

3. Set the DeviceObject field in the PtrVPB structure, sent to the FSD as part of the mount request to point to the new device object.

   This is how a logical association is created between the VPB structure and the logical volume device object created by your FSD. This pointer value is used by the I/O Manager to determine the target device object whenever a create/open request is received for a mounted logical volume.

4. Clear the DO_VERIFY_VOLUME flag (if set) in the device object for the real (physical/virtual) device.

   If you do not clear this bit, any read operations issued by your FSD to the device will fail. Remember, though, if you clear this bit, you must reset it on your way out of the mount routine.

5. Read in some of the information required to verify that the logical volume can be mounted by your FSD.

   You can simply use CcMapData ( ) to map the sectors described by the on-disk volume structures. Remember that this routine returns a pointer to a buffer control block structure and also a buffer pointer to the mapped-in information, which is valid as long as the range is not unpinned.

6. Verify that the structures on disk are legitimate, performing additional read operations if required.

7. Create and initialize an FCB structure to represent the root directory.

   Typically, the FSD always maintains an internal reference on the root directory FCB, to keep it around in memory as long as the volume stays mounted. The PtrRootDirectoryFCB field in the VCB structure is initialized by the sample FSD to point to the newly created and opened root directory for the logical volume.

   Note that opening the root directory will involve reading the root directory contents from disk. Your FSD may use stream file objects created for internal directory I/O operations.

   By this time you should have a fairly good idea of the range that you need to perform map operations for and you should update the file size values in the VCB structure appropriately.

8. The native Windows NT FSD implementations appear to read the volume label off the disk to ensure that the volume was not previously mounted.

   If this happens to be a remount request for a previously mounted volume,* the FSD must remove the newly created VCB and stream file object structures

* Any user/kernel thread with the appropriate privileges may have issued a mount request. The I/O Manager will also issue a mount request if a previously issued verify volume operation (for removable media) to the FSD had a return code of other than STATUS_SUCCESS.

(ensuring that any pinned ranges have been unpinned) and reinitialize the old VCB structures appropriately. Reinitialization involves setting the following fields:

- The 01dVCB->PtrVPB->RealDevice must be updated to point to the PtrVPB->RealDevice field contents.
- The 01dVCB->TargetDeviceObject field must be initialized to refer to the new TargetDeviceObject, obtained from the current I/O stack location for the mount IRP.
- The PtrVPB->RealDevice->Vpb field must be reinitialized to 01dVCB->PtrVPB.
- The cache map for the stream file object associated with the OldVCB must be reinitialized.
- Any other FSD-specific operations should be performed here to ensure that any cached information from the previous mount has been discarded.

9. Now that the mount/remount operation is nearly finished, you should re-enable volume verification if you have cleared the DO_VERIFY_VOLUME flag from the real target device object.
10. Typically, native NT FSD implementations will issue an IOCTL request to the target device object to lock removable media in the drive (at least, NTFS appears to do this).
11. Set the appropriate flag value in your VCB structure to indicate that the mount/remount operation was successful.

    For example, the sample FSD will set the SFSD_VCB_FLAGS_VOLUME_MOUNTED flag in the VCBFlags field.
12. Unpin any byte ranges that were pinned due to an invocation of CcMapData ( ) and release any resources that may have been acquired.
13. Return STATUS_SUCCESS if the mount logical volume operation succeeded.

    If your FSD encounters an error during the mount process (e.g., I/O errors encountered when attempting to read on-disk information), it should return the appropriate error value after cleaning up any structures that may have been allocated in processing the mount request.*

    If your FSD returns STATUS_SUCCESS to the I/O Manager for the mount request, the I/O Manager will set the VPB_MOUNTED flag in the VPB structure.

---

* If a mount request issued by the I/O Manager fails for the physical cleviee objeet representing the system boot partition, the NT I/O Manager will bugcheck the system with the INACCESSIBLE_BOOT_DEVICE bugcheck code.

Once the logical volume has been mounted by a loaded FSD, the I/O Manager will send the original create/open request that resulted in all of this processing being performed. The create/open request will be sent to the newly created device object representing the mounted instance of the logical volume and referred to by the DeviceObject field in the VPB structure.

NOTE Mount operations performed on a logical volume are often very complex, especially for more sophisticated FSD implementations such as NTFS, which is a log-based file system driver. Therefore, you should use the steps listed previously as a general guideline to follow when designing the volume mount operation specific to your file system driver.

In the next chapter, we'll see how to design and develop filter drivers that could help provide unique value-added functionality for the Windows NT operating system.