In this chapter:

- *File System Design*
- *Registry Interaction*
- *Data Structures*
- *Dispatch Routine: Driver Entry*
- *Dispatch Routine: Create*
- *Dispatch Routine: Read*
- *Dispatch Routine: Write*

9

Writing a File System Driver I

Most of you reading this book will never design a file system implementation; as a matter of fact, the number of people who do design and implement complete commercially available file systems are truly very few. However, a lot of you probably have a strong desire or at the very least some amount of curiosity to learn about how file systems fit into the Windows NT operating system; many of you might even design some functionality that incorporates file-system-like features. For example, you may choose to design a source code management system as a pseudo-file system implementation;* or you may choose to design a filter driver that intercepts file system requests to examine or possibly modify them before passing them on to the file system driver. In either case, an understanding of the implementation of file systems in the Windows NT environment will be a good investment. Also, if you simply wish to learn a little bit more about what really happens when your I/O is received by the native NT file systems (e.g., FASTFAT, NTFS), the discussions on the various FSD dispatch routines should give you a fairly good overview.†

This chapter, as well as most of the remaining chapters in this book, focuses on the implementation of a file system in the Windows NT environment. The method of presentation is fairly simple: first, file system data structures are covered, followed by each of the dispatch routines that a file system would typically implement. To understand the implementation better, I first describe the functionality expected from the file system for the dispatch routine; this is typically accompa-

---

* A good example of this is the *ClearCase* source code control system from Atria Systems, Inc.

† Unfortunately, I cannot discuss the myriad details that the native FSD implementations have to take care of and which are dependent on the specifics on the on-disk layout used by the FSD implementation. Documenting all of that (if indeed such information were ever made public) would occupy a whole book by itself. You can, however, purchase the IPS kit from Microsoft, which seems to contain some modified source to at least two file system implementations (FASTFAT and CDFS).

nied by code or pseudo-code (if required) that illustrates the concepts followed by an explanation of the code (pseudocode) fragment provided.

This chapter starts off with some of the very basic functionality expected from a file system; file system driver initialization, create, read, and write operations are covered here. Some of the more advanced concepts for the read and write dispatch routines (as well as other dispatch entry points) will also be covered in the next two chapters.

File System Design

No file system implementation can be successful without a sound design serving as its base. To construct a sound design, you should have a very good understanding of the goals that your file system is being designed to achieve. For instance, some file systems are deliberately developed to be simple and fast; their fundamental design goal is to provide a reliable, easily maintainable, and uncomplicated means of managing stored data. These file systems make no guarantees about ensuring data consistency in the presence of software or hardware failures; neither do they provide some of the advanced functionality, such as data security, and compression features that you might expect from more sophisticated file systems. A prime example of a relatively simple, yet very reliable file system is the FAT file system implemented in Windows NT.

Other file system implementations are considerably more sophisticated. For example, the NTFS file system implementation under Windows NT is a log-based file system. This file system design stresses fast recoverability from system failures, ensuring data consistency in the presence of hardware or software failures, providing security for user data, and providing other useful functionality, including flexible byte-range locking and data compression.

There are other file system designs that are even more sophisticated and distributed in nature. The Distributed File System (DPS)* from the Open Software Foundation is an example of a considerably more complex file system implementation. This file system comprises local disk-based file systems that provide features similar to NTFS, and also client-server components that provide consistent, global accessibility with the benefits of a single name space across geographically distributed locations. The long-awaited Object File System (OFS) implementation from Microsoft is also an example of a distributed and complex file system implementation, which should provide sophisticated functionality such as online logical volume replication and location independence.

---

* The predecessor to DPS is the famous Andrew File System (AFS) implementation from Carnegie Mellon University. Commercial versions of both DPS and AFS are now available from Transarc Corporation.

For more information about some of the file system implementations mentioned here, consult the references provided at the end of this book.

Sample File System Code

The design goals for the sample file system implementation code provided in this book are to acquaint you with the interactions between the Windows NT operating system components and a file system driver. Therefore, I focus only on these interactions and exclude coverage of other file-system-specific implementation details.

Every file system implementation must interact intimately with the rest of the operating system. After all, the file system does not exist in a vacuum, and the only generic way for a user to access data managed by the file system is by using some well-defined system services.

All file systems must also manage the on-disk data structures that allow them to store user data. Figure 9-1 illustrates how a file system design can be composed of multiple layers to address the various functional requirements expected of the implementation. The *veneer* serves as the upper-level interface between the file system implementation and the remainder of the operating system environment. This layer is the most operating-system-specific layer. The *core* can be designed to serve the requirements of the veneer by managing on-disk data structure accesses; this layer can be designed to be relatively free from any operating-system-specific constraints. The *driver interface layer*, once again, must deal with lower-level disk or network drivers in the operating system and therefore has to conform to the interface presented by such drivers.

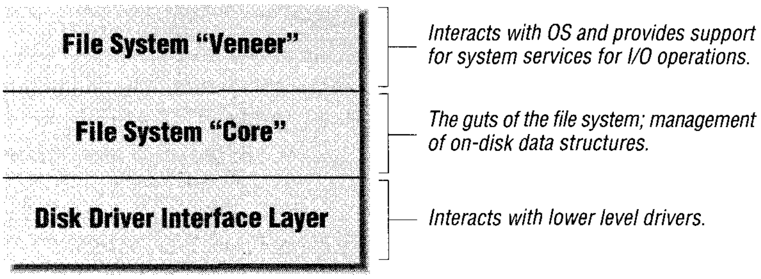
File System "Veneer"
Interacts with OS and provides support for system services for I/O operations.
File System "Core"
The guts of the file system; management of on-disk data structures.
Disk Driver Interface Layer
Interacts with lower level drivers.

Figure 9-1. File system components in a layeredfile system driver (FSD) design

Most file systems, including the native NT FSD implementations, conform to the high-level design illustrated in Figure 9-2. The FSD code samples in this book also use the same methodology. However the code samples ignore the two lower layers illustrated in Figure 9-1 and focus exclusively on the veneer. Therefore, you will not see any code that deals with the actual retrieval and manipulation of data to and from secondary storage; this book does not present any discussion of these topics either. Naturally, the code fragments provided in this book should only be used as a starting point for your development efforts. They do, however, illustrate the important aspects of interacting with the NT I/O Manager, the NT Cache Manager, and the NT Virtual Memory Manager. You can design the lower levels of your commercial file system driver and plug your implementation into the driver model presented here to get yourself a fully functional, "native" file system driver under Windows NT.

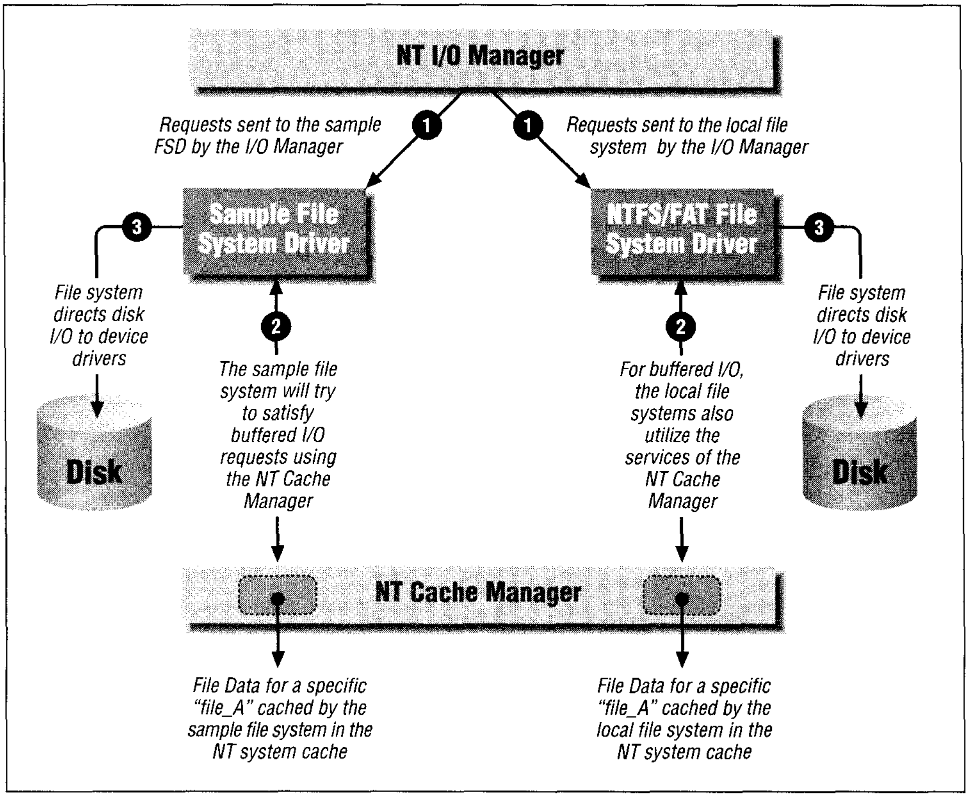
NT I/O Manager
1
Requests sent to the sample FSD by the I/O Manager
1
Requests sent to the local file system by the I/O Manager
Sample File System Driver
NTFS/FAT File System Driver
3
File system directs disk I/O to device drivers
3
File system directs disk I/O to device drivers
Disk
Disk
2
The sample file system will try to satisfy buffered I/O requests using the NT Cache Manager
2
For buffered I/O, the local file systems also utilize the services of the NT Cache Manager
NT Cache Manager
File Data for a specific "file_A" cached by the sample file system in the NT system cache
File Data for a specific "file_A" cached by the local file system in the NT system cache

Figure 9-2. High-level view ofa simplefile system implementation architecture

Logistics

The sample code will get you started understanding, designing, and developing a commercial NT file system or filter driver. Although not complete by any means, the code should serve as a framework, using which you can innovate and build in the functionality for your commercial file system driver. Keep the following points in mind as you read through the code samples and explanations accompanying the code:

Maintain focus on interactions between the NT operating system and the FSD.

For a commercial FSD implementation, there are a lot of conflicting design choices that must be made. Some of the more obvious ones include choosing between fast (*tuned*) implementations or cleaner, more abstract, though relatively slower designs. Or you might consider the tradeoffs involved in incorporating strict security requirements in your design as opposed to the inevitable resulting inconvenience caused to users. You may choose to increase concurrency at the expense of a complex design that is relatively more difficult to maintain and enhance; or you may choose a less parallel model, which might be quicker to design and implement and also be more easily maintainable.

The sample code in this book does not even attempt to enumerate the various choices available to the FSD designer, let alone provide solutions for such complex issues. Therefore, while reading through the sample FSD implementation, focus on learning about the interactions with the NT operating system, and consult the literature listed at the back of this book for additional information on the issues involved in developing file system drivers.

Note that all data structures and function names are prefixed by the letters SFsd.

This allows for easy identification of those function calls and data structures that are implemented by the sample FSD. NT driver conventions require that you should prefix your function and data structure names with an appropriate identifier unique to your driver.

Exception handling is built into the sample code (or code fragments).

The sample FSD implementation presented in this book utilizes structured exception handling. You may disregard some of the details pertaining to structured exception handling if you believe that providing this support is unnecessary from your perspective. However, I would strongly encourage you to consider incorporating SEH into your implementation at the onset of file system design, since the resulting benefits in terms of code robustness far outweigh the cost of the time commitments required for such support.

Comments are interspersed in the implementation.

Some people believe that comments included with source code are useful, while others do not. I have liberally interspersed comments in the sample code included with this book. Please read these comments, because much of the information they contain is not repeated in the accompanying text.

Look for alternative methods for implementation.

Designing and implementing kernel-mode drivers requires a certain amount of on-the-job experience and is often an iterative process. The sample implementations presented in this book should not be construed as the only way the desired functionality can be implemented. Alternatives typically abound, and you should always explore any such alternative methods if you can think of them. Use the implementation presented here as a general guideline, but always keep an eye out for alternative designs.

Be aware of memory allocation issues.

Kernel-mode drivers, including file system drivers or filter drivers, should be cognizant of their memory requirements. Your goal should always include efficient usage of system memory. If you require nonpaged memory (and you typically will), you should always carefully monitor your requirements and attempt to minimize your usage of these scarce resources.

Even if you are careful and separate your memory requirements into nonpaged memory required by your driver, as well as paged memory you can work with, remember that paging is not a cheap operation. Excessive page faults or TLB faults caused by your kernel-mode driver will lead to degraded performance by the entire system. Therefore, always be careful, to the point of being stingy, with your memory needs.

Having said all of that, you should note that the code you see in this book makes no attempt at efficient memory usage, except for an example in which zone structures are utilized.* That is something you must work at in your commercial implementations.

Be aware of synchronization issues.

Chapter 3, *Structured Driver Development,* explained the various objects that can be used to synchronize access to shared data structures in Windows NT. The sample code in this book uses one or more of these objects. For certain shared data structures, it may sometimes be possible to modify the synchronization methodology used in the code samples, such that performance is enhanced. Typically, this is done by carefully examining the various situations under which a particular shared data structure is accessed, and then possibly lowering the synchronization requirements associated with the shared data only if you have determined that data integrity will still be preserved. No such attempts at enhancing performance have been made in the sample code presented in the book. I have used a simpler, cleaner design that always uses synchronization primitives to monitor access to shared data structures. You can, however, analyze any drivers that you develop based on code samples provided in this book for obtaining performance gains using such methods.

---

* If your software only executes on Windows NT Version 4.0 and later, consider using lookaside lists instead of zones.

Remember, though, to always be conservative in your analysis; otherwise, you may inadvertently cause data corruption.

Registry Interaction

A typical file system implementation requires the creation of a number of keys and associated value-entries in the Windows NT Registry:

- HKEY_LOCAL_MACHINE\ SYSTEM\CurrentControlSet\ Services\SampleFSD

Table 9-1 shows the subkeys and value entries that should be created.

Table 9-1. Subkeys and Value Entries

Subkey/Value Entry	Type	Value	Description
ErrorControl	REG_DWORD	Oxl	Log an error and display a message box if driver fails to load, but continue initialization (if driver is being loaded automatically).
Group	REG_SZ	"File System"	This indicates the driver belongs to the group of file system drivers. If you develop a network redirector instead, replace the value with *Network Provider*.
ImagePath	REG_EXPAND_SZ	"%SystemRoot%\System32\drivers\sfsd.sys"	The complete path name of the driver image.
Start	REG_DWORD	0x2 or 0x3	A value of 0x2 specifies automatic start; 0x3 specifies manual start only.
Type	REG_DWORD	0x2	Indicates that this is a file system driver.
Parameters	–	–	This subkey contains driver required configurable parameters. For a list of parameters accepted by the sample FSD, see Table 9-2.

Table 9-2 lists the possible configurable parameters accepted by the sample FSD. In your driver, you can add other value entries under the *Parameters* subkey.

Table 9-2. Configurable Parameters Accepted by the Sample FSD

Value Entry	Type	Value	Description
`PreAllocated-NumStructures`	REG_DWORD	0 (Default)	Specifies the number of structures for which to preallocate memory (this illustrates how an FSD could use the Registry to obtain user-configurable information).

- HKEY_LOCAL_MACHINE\...\CurrentControlSet\Services\EventLog\System\ SampleFSD

  This key is added to allow any event log viewer application to decipher the messages logged by the sample FSD implementation to the NT Event Log.

  Table 9-3 shows the value entries that are created.

Table 9-3. Value Entries

Value Entry	**Type**	**Value**	**Description**
`EventMessageFile`	REG_EXPAND_SZ	%SystemRoot%\System32\sfsdevnt.dll	The complete pathname for the message catalog containing textual descriptions of events recorded by the sample FSD in the NT system event log.
`TypesSupported`	REG_DWORD	0x7	*Informational*, *Warning*, and *Error* messages will be logged.

- HKEY_LOCALJVIACHINE\SOFTWARE\SampleFSD

  Information contained below this key is not available at system load time. However, it is often useful to keep nonessential information about the driver itself, the manufacturer, or other information here.

The sample FSD implementation will create the following value entries and sub-keys as shown in Table 9-4.

Table 9-4. Value Entries and Subkeys

Value Entry	Type	Value	Description
`VendorName`	`REG_SZ`	`"Rajeev Nagar"`	Any string that uniquely identifies your organization. This field is simply informational in nature.
`CurrentVersion`	-	-	This subkey contains information pertaining to the current version of the driver.

The CurrentVersion subkey listed in Table 9-4 could contain the following value entries:

- `VersionMajor`
- `VersionMinor`
- `VersionBuild`
- `InstallDate`

The Windows NT Software Developers Kit and the Device Drivers Kit provide ample documentation and recommendations on these optional value entries.

Data Structures

At the core of any file system driver design are the data structures that together define the file system; these include the *on-disk* data structures that determine the management of the actual stored data, as well as the *in-memory* data structures that facilitate orderly access to such data. If you understand the data structures that might be required for a particular type of driver or kernel component, you have probably won half of the battle in your attempts at successfully designing such a component.

In an ideal world, the operating system should be completely independent of the on-disk data structures and layout maintained by a specific file system driver; the operating system should also be indifferent to the in-memory structures that a FSD implementation might implement, since these in-memory structures exist simply to help the implementation provide file-system-specific functionality. Windows NT is not an ideal operating system environment, and neither, for that matter, is any other commercially available operating system. However, Windows NT is relatively indifferent to the on-disk data structures maintained by a file system. Typically though, it would confuse a user of your file system tremendously if the FSD did not maintain expected information for files stored on physical media. For example, if your FSD did not maintain *last write time, last access time,* or a *file name* on disk, your file system could seem fairly strange to a Windows NT user, since such users have come to expect and depend on the existence of these attributes associated with file streams.

The native file systems supplied with Windows NT vary greatly in the features provided to users of the file system. The FASTFAT file system does not provide any security attributes for files (typically stored as Access Control Lists); it does not support multiple hard links to files, file compression, or fast recovery from system failures. The NTFS implementation does, however, support all of the features listed above, and therefore the on-disk data structures maintained by the NTFS implementation differ greatly from those maintained by the FAT file system implementation.

As mentioned earlier, the goals that you set for your file system will determine the on-disk data structures that you need to maintain. In the remainder of this book, we won't discuss on-disk structures any further, since they do not need to conform to any specific model for you to successfully implement a file system driver under Windows NT. However, as you design your file system, you should carefully study the various alternatives available to you in the format of the on-disk layout and associated structures for your file system.

The interesting structures from our perspective, therefore, are the in-memory data structures that your FSD should implement. Although Windows NT does not mandate that any specific structures be maintained, here are the two structures you should become familiar with:

- The File Control Block (FCB) structure
- The Context Control Block (CCB) structure

An FCB uniquely represents an open, on-disk object in system memory. Notice that I said that an FCB represents an on-disk object, not just an on-disk file. Directories, files, volume structures, and practically any other object that your FSD maintains and that can be opened by a user of your file system would be represented as an FCB.* If you have some background in UNIX implementations, you can easily draw an analogy between UNIX *vnode* structures, which are simply

---

* .Some fi/e system implementations, including the native file systems denote in-memory representations of directory structures as DCB objects (Directory Control Blocks). DCBs are not any different (functionally) from FCBs and the sample FSD (as well as the discussion provided throughout the course of the book) uses the FCB to represent both files and directories.

abstract representations of files in memory, and Windows NT file control block (FCB) structures. They both serve the same purpose of representing of the on-disk object in memory.

A CCB is simply a handle or the context created and maintained by the FSD to represent an open instance of an on-disk object. For example, when a user application performs an *open* operation on a file, it receives a handle from the operating system if the open request was successful. Corresponding to this handle, a Windows NT FSD creates a CCB structure, which is simply the kernel equivalent of the user handle. Is your FSD required to maintain a CCB for each open instance and an FCB to uniquely represent an open on-disk object? My answer to this is yes, it is. If you are not convinced of the necessity for maintaining these data structures, I would advise you to reserve judgment on this question until you have read through the next few chapters.

Representation of a File in Memory

You already know of the file object structure created by the I/O Manager to represent successful open operations on files and directories. To see how file objects, FCB structures, CCB structures, and VCB structures fit together, refer to Figure 9-3-

I would recommend that, to better understand the figure, you should start at the bottom of the illustration. Here is a description of its various components.

Physical device object

At the very bottom of the illustration, you see two device objects: the physical device object and the logical device object. The physical device object structure is typically a media-type object with a DeviceType of FILE_DEVICE_DISK, FILE_DEVICE_VIRTUAL_DISK, FILE_DEVICE_CD_ROM, or some such type.

This structure is created by a device driver, via the loCreateDevice () routine, to represent the physical or virtual disk object that it manages. At creation time, a VPB (Volume Parameter Block) structure is allocated and associated with media type objects by the NT I/O Manager. Initially, the VPB flags indicate that the physical media does not have any logical volume mounted on it, via the absence of the VPB_MOUNTED flag value. Later though, some file system implementation might verify the data structures on the physical media and decide to *mount* a logical volume on that physical device object.

This leads us to the next object depicted in the illustration, the logical volume device object.

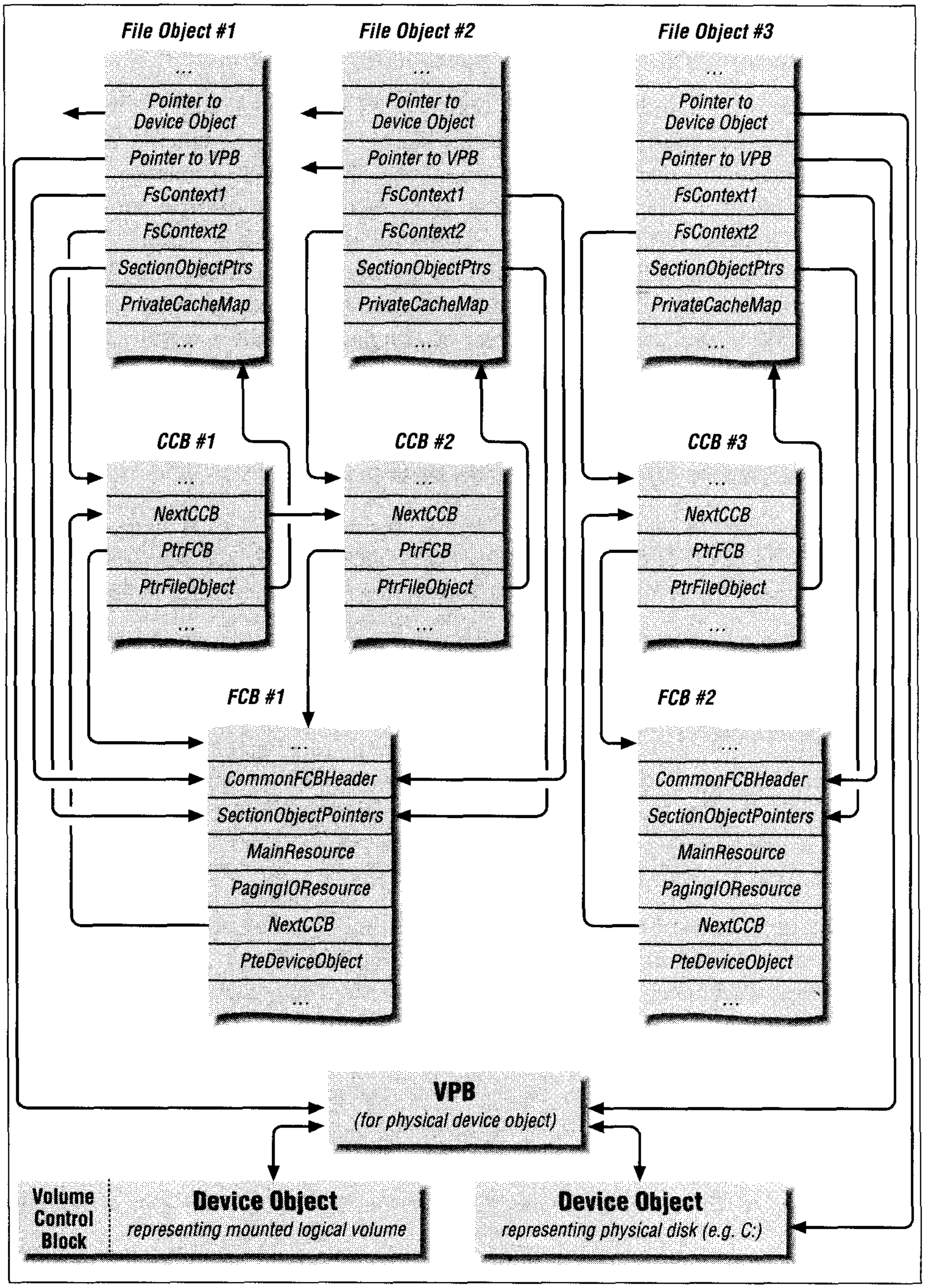

Figure 9-3. Representation of files in memory

Logical volume device object

The volume device object represents an instance of a mounted logical volume. This device object is created by a file system driver implementation; our sample FSD will also create volume device objects.

A *mount* operation is typically required on most operating systems before users are allowed to access file system data on secondary storage devices. This logical mount operation is performed by a file system driver implementation on that platform. The process of mounting a volume exists simply to allow a file system driver the opportunity to prepare the volume for subsequent access.

Most of the steps that a file system might undertake as part of mount operation are file-system-specific and the operating system does not interfere much during the process. Typically, a file system driver will first check the on-disk data structures to determine whether the medium to be mounted contains valid file system information. If these checks pass, the file system will then read in basic volume information such as volume size, root directory location, free block map, allocated cluster map, and so on, and then create the requisite in-memory structures that will be used to support access to the volume.

As part of mounting the logical volume, most operating systems (including Windows NT) do require that certain system-defined data structures be created and/or initialized, to establish linkage between the rest of the I/O Manager structures and the in-memory representation of the mounted logical volume. Therefore, file system designers must always understand the requirements that the operating system places upon the file system implementation and ensure that the correct data structures are initialized as required.

Under Windows NT, the I/O Manager requires that a device object representing the mounted logical volume be created, and for physical media, the VPB associated with the device object representing the physical media be correctly initialized.

Each volume device object is logically associated with a physical device object using the VPB structure belonging to the physical device object. This association occurs at volume mount time, which happens when the very first create/open request is received by the I/O Manager for an object residing on the physical device object.

The pseudocode fragment below illustrates how a logical volume device object structure is associated with the physical device object structure. This logical volume device object is important, because it is created by the FSD that mounts the logical volume and is used by the I/O Manager to determine the target driver/ device object for a create/open request.

The sequence follows:

```
I/O Manager receives an open/create request for an object residing on the
physical device object (e.g., an open for "C:\dirl");
I/O Manager obtains a pointer to the VPB structure associated with the
physical device object representing "C:";
I/O Manager checks the "Flags" field in the VPB to see if a mounted
logical volume exists for the physical device object;
if (no mounted volume exists, i.e., VPB_MOUNTED is not set) {
      I/O Manager requests each of the file systems to check whether
      they wish to perform a mount on the physical device object;
      One of the file system drivers performs a mount operation;
      As part of the "mount" process, the FSD will create a device object
      of type FILE_DEVICE_DISK_FILE_SYSTEM or the network equivalent;
      Now, the DeviceObject field in the VPB will point to the logical
      volume device object (this is done by the FSD that performs the
      mount);

When some file system has returned STATUS_SUCCESS for a mount request,
the I/O Manager will set the VPB_MOUNTED flag in the Flags field for
theVPB;
}
```

Now the I/O Manager can proceed with other instructions pertaining to the create/open request (described later in the book).

VPB

Immediately above the two device objects depicted in Figure 9-3 is the VPB structure. This structure performs the important task of creating a logical association between the physical disk device object and the logical volume device object. A VPB structure only exists for device objects that represent physical, virtual, and/or logical media that can be mounted. Therefore, if you design network redirectors and/or servers, you will not interact directly with the VPB structure.

Note that there is no physical association between the physical device object and the logical device object representing the mounted volume (i.e., there is no pointer leading from a logical volume device object directly to the physical device object or vice versa, as would typically happen when two device objects are connected via an *attach* operation); for example, later in this book, we'll discuss a routine called loAttachDevice() used by intermediate or filter drivers to create an association between their own device object and a target device object. Such attachments are performed with the intent of intercepting requests targeted to the original device object (the one being attached to).

For file system mount operations, however, the only association between the two device objects is the logical connection via the VPB structure. The I/O Manager is aware of this logical association between the two device objects, and always checks the VPB structure on receipt of a create/open request for an on-disk object to determine the file system volume device object to which the request should actually be directed.

Also note that each file object structure representing a successful open operation on an on-clisk object points to the VPB structure belonging to the physical device on which the opened on-disk object resides.

Volume control block (VCB)

As mentioned earlier, as part of the mount process, each file system implementation creates appropriate in-memory data structures that will enable the file system to permit orderly and correct access to data contained in the logical volume. One of the structures maintained by the sample FSD to assist in this process is the Volume Control Block structure.

The VCB structure contains such essential information as a pointer to the in-memory root directory structure (i.e., the FCB for the root directory), a count of the number of file streams that are currently open on the logical volume, some flags representing the state of the logical volume at any given instant, synchronization structures used to maintain the integrity of the VCB structure itself, and other similar information. However, just as with all other structures defined by the sample FSD, the VCB structure is slightly less comprehensive then it would be if we were developing a full-fledged physical-media-based file system driver implementation. In that case, the VCB would also typically contain pointers to structures containing information about available free clusters on disk, a list of allocated clusters, and any such on-disk structure management related information.

Note that the Windows NT I/O Manager does not require that a file system maintain a structure like the VCB structure. However, most file system implementations (including the native NT file system implementations) maintain some variant of such VCB data structures.

The sample FSD code allocates the VCB as part of the device object extension for the device object created to represent the mounted logical volume. This seems like a very logical manner in which to allocate the VCB structure, since this method of allocation creates the association between the mounted logical volume representation known to the rest of the system (i.e., the device object) and the file system internal representation (i.e., the VCB structure). This also implies that the VCB structure is allocated by the I/O Manager from nonpaged system memory (since a device extension is allocated by the I/O Manager on behalf of the caller when the device object is being created).

If you study the VCB structure, defined below by the sample FSD, you will see that it contains fields used in obtaining services of the NT Cache Manager. Many Windows NT FSD implementations use the Cache Manager to cache on-disk volume metadata structures. This is accomplished by creating a stream file object to represent the open on-disk volume information and then initiating caching for this file object. The data is mapped into the system cache using CcMapData ( ) , and it can be easily accessed, just as cached file stream data is typically accessed by user threads.

The VCB structure defined by the sample FSD is shown below:

```
typedef struct _SFsdVolumeControlBlock {
    SFsdldentifier                          Nodeldentifier;
    //a resource to protect the fields contained within the VCB
    ERESOURCE                               VCBResource;
    // each VCB is accessible on a global linked list
    LIST_ENTRY                              NextVCB;
    // each VCB points to a VPB structure created by the NT I/O Manager
    PVPB                                    PtrVPB;
    // a set of flags that might mean something useful
    uint32                                  VCBFlags;
    // A count of the number of open files/directories
    //As long as the count is != 0, the volume cannot
    // be dismounted or locked.
    uint32                                  VCBOpenCount;
    // we will maintain a global list of IRPs that are pending
    // because of a directory notify request.
    LIST_ENTRY                              NextNotifyIRP;
    // the above list is protected only by the mutex declared below
    KMUTEX                                  NotifylRPMutex;
    // for each mounted volume, we create a device object. Here then
    // is a back pointer to that device object
    PDEVICE_OBJECT                          VCBDeviceObject;
    //We also retain a pointer to the physical device object, which we
    // have mounted ourselves. The I/O Manager passes us a pointer to this
    // device object when requesting a mount operation.
    PDEVICE_OBJECT                          TargetDeviceObj ect;
    // the volume structure contains a pointer to the root directory FCB
    PtrSFsdFCB                              PtrRootDirectoryFCB;
    // For volume open operations, we do not create a FCB (we use the VCB
    //    directly instead). Therefore, all CCB structures for the volume
    //    open operation are linked directly to the VCB
    LIST_ENTRY                              VolumeOpenListHead;
    // Pointer to a stream file object created for the volume information
    // to be more easily read from secondary storage (with the support of
    // the NT Cache Manager).
    PFILE_OBJECT                            PtrStreamFileObj ect;
    // Required to use the Cache Manager.
    SECTION_OBJECT_POINTERS                 SectionObj ect;
    // File sizes required to use the Cache Manager.
    LARGE_INTEGER                           AllocationSize;
    LARGE_INTEGER                           FileSize;
    LARGE_INTEGER                           ValidDataLength;
} SFsdVCB, *PtrSFsdVCB;

// some valid flags for the VCB
```

```
#define          SFSD_VCB_FLAGS_VOLUME_MOUNTED          (0x00000001)
#define          SFSD_VCB_FLAGS_VOLUME_LOCKED           (0x00000002)
#define          SFSD_VCB_FLAGS_BEING_DISMOUNTED        (0x00000004)
#define          SFSD_VCB_FLAGS_SHUTDOWN                (0x00000008)
#define          SFSD_VCB_FLAGS_VOLUME_READ_ONLY        (0x00000010)
#define          SFSD_VCB_FLAGS_VCB_INITIALIZED         (0x00000020)
```

File control blocks (FCB)

Figure 9-3, shown earlier, depicts two FCB structures, each of which represents a file stream in memory.

Just as an application needs to maintain in-memory data structures to provide services to users, file systems must maintain some in-memory data structures. Traditional file systems typically manipulate two types of on-disk objects: *files* and *directories.* A file, as we understand it, simply represents a stream of bytes stored on disk. A directory is a file-system-defined structure that contains information about files; i.e., a directory is not useful in itself except as a means to locate files, which, in turn, contain the user's data. In database terms, the directory simply comprises an index for the actual data, where the data is defined as the individual file streams. Files and directories are examples of *persistent objects,* objects that persist across system reboots, since they are stored on nonvolatile secondary storage media.

Files are simply named objects contained within directories. An important concept is the logical separation between a *directory entry* (a named file object) and any data associated with the file object. For example, consider a file *foo,* contained in directory *dirl.* File *foo* is identified within directory *dirl* by the presence of a directory entry created within *dirl* representing file *foo.* Regardless of how much data is associated with the file, there will typically exist one directory entry of a fixed size within the directory *dirl* that names file *foo* as a valid object contained in the directory.

Further, if the file *foo* has 2 bytes associated with it, and now if you *truncate* the size to 0 bytes, the directory entry within the directory *dirl* will still exist, except that now it will be updated to reflect the new size. Truncating the file may have caused any on-disk storage assigned to the file to be released, but it does not free up the directory entry for the file. Deleting the file, however, will free the directory entry for the file and the storage allocated for the data will also (in this case) be freed.

Finally, if an FSD supports multiple linked files, the logical separation between a file name and the storage space allocated for file data becomes even more obvious. Now, you may have two separate directory entries referring to the same on-disk stored data. For example, file *foo* in directory *dirl* and file *bar* in directory *dir2* may simply be synonyms for the same on-disk data. Now, even if you delete a directory entry (say, the entry for *foo* in directory *dir1*), the storage allocated for the data will not be released, since the directory entry for *bar* in directory *dir2* still points to the allocated data. Space allocated for data will only be freed after all directory entries referring to the allocated storage space for data on secondary media are deleted.

When a user tries to access a file or a directory object, which exists on secondary storage, the file system must obtain information from secondary storage to satisfy the user request. Typically, this information will be the actual data stored on disk; sometimes, though, a user might want *control* information (also known as *meta-data),* such as the last time any process actually modified the contents of the file, or the last time a process tried to read the contents of the file. Therefore, all file systems should also be able to provide such information on request.

When a file system obtains data from secondary storage, it must keep this data in system memory somewhere to make it accessible to the user. If file data is stored somewhere in system memory, the file system is responsible for maintaining appropriate pointers to this data. Also, if multiple users try to access the same data, the file system should be able to consistently satisfy these concurrent requests. This requires that the file system assume some responsibility for providing appropriate synchronization when a thread tries to access on-disk data.

To provide the functionality just described, file systems create and maintain an abstract representation of open files and directories; i.e., each file system defines for itself the in-memory control data structures that it must maintain to satisfy user requests for access to persistent on-disk objects. Note that these in-memory representations of files and directories are not themselves persistent; they exist simply to facilitate access to on-disk data and can always be recreated from the data stored on secondary storage.

On most UNIX implementations, an in-memory abstraction of a file or directory is commonly called a *vnode.* On Windows NT systems, it is called a *File Control Block.* Regardless of the term used to identify this representation (we'll stick with the Windows NT terminology in the rest of the book), the important thing to note is that an on-disk object must always be represented by one, and only one, FCB. Therefore, even if your file system were to support multiple linked file streams, you should only create one FCB to represent this file stream in memory, regardless of the fact that two different processes may have used different path names or identifiers to open the same on-disk object.*

Here is the FCB defined by the sample FSD:

---

* Multiple hard links are simply alternate names for the same file stream. For example, a file *\directory1\foo* could also be identified by the path *\directory2\directory3\bar,* as long as both path names referred to the same on-disk byte stream; an FSD will represent the file by a single FCB structure.

```
typedef struct _SFsdNTRequiredFCB {
     // see Chapters 6-8 for an explanation of the fields here
    FSRTL_COMMON_FCB_HEADER              CoramonFCBHeader;
    SECTION_OBJECT_POINTERS              SectionObj ect;
    ERESOURCE                            MainResource;
    ERESOURCE                            PagingloResource;
} SFsdNTRequiredFCB, *PtrSFsdNTRequiredFCB;

typedef struct _SFsdDiskDependentFCB {
    // although the sample FSD does not maintain on-disk data structures,
    // this structure serves as a reminder of the logical separation that
    // your FSD can maintain between the disk-dependent and the disk-
    // independent portions of the FCB.
    uint16                               DummyField;         // placeholder
} SFsdDiskDependentFCB, *PtrSFsdDiskDependentFCB;

typedef struct _SFsdFileControlBlock {
    SFsdldentifier                       Nodeldentifier;
    // We will embed the "NT Required FCB" right here.
    // Note though that it is just as acceptable to simply allocate
    // memory separately for the other half of the FCB and store a
    // pointer to the "NT Required" portion here instead of embedding it
    // ...
    SFsdNTRequiredFCB                    NTRequiredFCB;
    // the disk-dependent portion of the FCB is embedded right here
    SFsdDiskDependentFCB                 DiskDependentFCB;
    // this FCB belongs to some mounted logical volume
    struct _SFsdLogicalVolume            *PtrVCB;
    // to be able to access all open file(s) for a volume, we will
    // link all FCB structures for a logical volume together
    LIST_ENTRY                           NextFCB;
    // some state information for the FCB is maintained using the
    // flags field
    uint32                               FCBFlags;
    // all CCBs for this particular FCB are linked off the following
    // list head.
    LIST_ENTRY                           NextCCB;
    // NT requires that a file system maintain and honor the various
    // SHARE_ACCESS modes ...
    SHARE_ACCESS                         FCBShareAccess;
    // to identify the lazy writer thread(s) we will grab and store
    // the thread id here when a request to acquire resource(s) arrives ...
    uint32                               LazyWriterThreadID;
    // whenever a file stream has a create/open operation performed,
    // the Reference count below is incremented AND the OpenHandle count
    // below is also incremented.
    // When an IRP_MJ_CLEANUP is received, the OpenHandle count below
    // is decremented.
    // When an IRP_MJ_CLOSE is received, the Reference count below is
    // decremented.
    // When the Reference count goes down to zero, the FCB can be
    // de-allocated.
    // Note that a zero Reference count implies a zero OpenHandle count.
    // This must always hold true ...
    uint32                                   ReferenceCount;
    uint32                                   OpenHandleCount;
    // if your FSD supports multiply linked files, you will have to
    // maintain a list of names associated with the FCB
    SFsdObjectName                           FCBName;
    // we will maintain some time information here to make our life easier
    LARGE_INTEGER                            CreationTime;
    LARGE_INTEGER                            LastAccessTime;
    LARGE_INTEGER                            LastWriteTime;
    // Byte-range file lock support (we roll our own).
    SFsdFileLockAnchor                       FCBByteRangeLock;
    // The OPLOCK support package requires the following structure.
    OPLOCK                                   FCBOplock;
} SFsdFCB, *PtrSFsdFCB;
```

Notice that the FCB in the sample FSD is divided into two main logical components:

- The SFsdFCB structure, which is operating system independent
- The SFsdNtRequiredFCB structure, which contains fields required to interact with the rest of the system

Additionally, the disk-dependent data structures can also be carved out into a separate data structure to provide increased portability and modularity, as is illustrated by the structures defined previously.

SFsdFCB. The first thing you should note about file control block structures is that the operating system does not determine the contents of this structure. Therefore, each file system implementation has complete flexibility over the fields contained in the FCB structure. The fields that exist in the SFsdFCB structure shown here are simply representative of the contents of typical FCB structures, defined by the various file system driver implementations under Windows NT. Usually, most file systems will maintain some information about the object names (hard links) associated with the FCB structure. Similarly, most FCB structures defined by file system drivers will maintain a field representing the current ReferenceCount for the FCB, and another field representing the current OpenHandleCount for the FCB. In this chapter and in the next two chapters, I will present code examples that manipulate the fields contained in the sample FCB structure. These code examples will assist you in understanding why such fields are helpful to file system drivers.

Another noteworthy aspect of the FCB defined in the sample FSD is the lack of any information about on-disk structures; e.g., there is no information about the actual on-disk clusters occupied by the file stream represented by the FCB. As mentioned earlier, we'll ignore those aspects of FSD implementations that require creating and maintaining on-disk data structures. A real file system, however, does not have this luxury and will contain far more information about on-disk file stream layout than shown in the sample FSD. If you wish to adapt the sample FSD for your own file system driver implementation, I would recommend that you isolate the on-disk format-related information into a separate data structure and then associate that separate structure with the FSD shown here, either via a pointer embedded in the FCB or by embedding the data structure itself into the FCB. This method will allow you to maintain a clean logical separation between the disk-independent and the disk-dependent parts of the FCB structure. For example, you can create a FCB structure as shown in Figure 9-4.

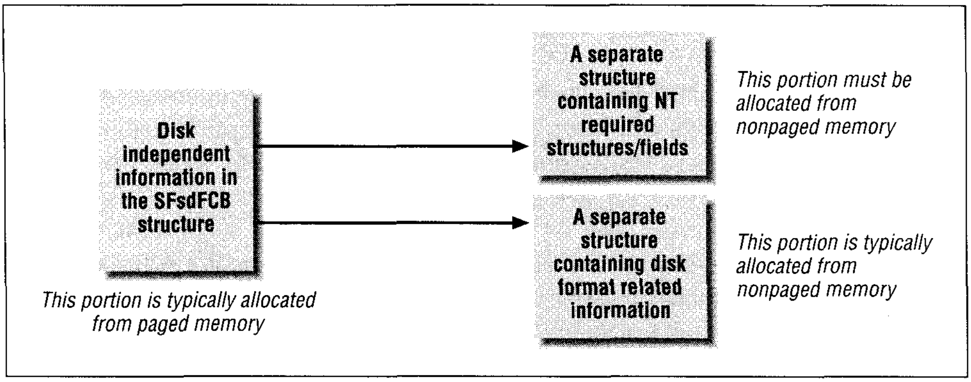
Disk independent information in the SFsdFCB structure
This portion is typically allocated from paged memory
A separate structure containing NT required structures/fields
This portion must be allocated from nonpaged memory
A separate structure containing disk format related information
This portion is typically allocated from nonpaged memory

Figure 9-4. A logical breakdown of the components of an FCB

This separation is illustrated by the presence of a DiskDependentFCB field of type SFsdDiskDependentFCB structure, embedded in the SFsdFCB structure shown above.

SFsdNtRequiredFCB. Although Windows NT allows an FSD considerable latitude in how it wishes to define its own FCB structures, successful integration with the Cache Manager and the Virtual Memory Manager requires that certain NT-defined structures also be associated with the FCB. Integration with the NT Cache Manager and the VMM is essential if an FSD needs to use the NT system cache and also support memory mapped files.

There are four fields that must be associated with each FCB as a prerequisite for successful integration:

- A single structure of type FSRTL_COMMON_FCB_HEADER (called the CommonFCBHeader field in the sample FCB above)
- A single structure of type SECTION_OBJECT_POINTERS (called the SectionObject field in the sample FCB)
- Two synchronization structures of type ERESOURCE (named as the MainResource and the PagingIoResource by the sample FSD)

Just as there must be only one FCB representing the file stream in memory, there must be only one instance of each of these NT-defined structures associated with a particular FCB. Typically, these fields do not need to be associated with FCB structures representing directory objects; however, you can still create them to maintain consistency.

TIP You will also require these structures for FCBs representing directory objects if you intend to use the loCreateStreamFileObject() function to cache directory information.

File system drivers have to allocate memory for the NT-required fields; the ERESOURCE type objects can never be allocated from paged pool, neither should you allocate the CommonFCBHeader or the SectionObject fields from paged memory.

When are in-memory ***FCB structures created andfreed?*** A new FCB structure, composed of the disk-independent, disk-dependent, and NT-required parts, is allocated when a byte stream is being opened for the first time and no other FCB structure representing this byte stream currently exists in system memory. For example, if an application decides to open a file *foo* for the very first time since the system was booted up, the FSD managing the logical volume on which *foo* resides creates an FCB structure in response to the caller's open request (i.e., as part of processing an IRP_MJ_CREATE request). Subsequent requests to open the file *foo,* however, will not result in the creation of a new in-memory FCB structure, as long as the previously created FCB structure is still retained in memory by the FSD. The factor that determines whether an FCB is still retained by an FSD is the value of the ReferenceCount field (or its equivalent), maintained by the FSD in the FCB structure. See the discussion on FCB reference counts presented below for more information. If, however, the FCB for the file *foo* has already been discarded by the FSD when the new open request is received, the FSD will once again create a new FCB structure to represent file *foo* in memory.

This file control block structure serves as the single unique representation of the open byte stream in system memory. The FCB is retained as long as any NT component maintains a reference for it. The reference count for an FCB is contained in the ReferenceCount field in the sample FCB shown here; your file system driver is free to name an analogous field whatever it may choose. A ReferenceCount value of zero implies that the FCB can be safely deallocated, because no component in the system is actively accessing the byte stream associated with the byte stream represented by the FCB at that instant.

ReferenceCount and OpenHandleCountfields. Two fields in the SFsdFCB structure are the ReferenceCount field and the OpenHandleCount field. It is extremely important that you understand the significance of these two fields.

Both the *reference count* and the *open handle count* are internal counts maintained by the FSD in the FCB structure and are therefore not externally visible to any other kernel-mode or user-mode component. Both counts help to determine when a FSD can safely deallocate the FCB structure.

The reference count is simply a number that indicates the total number of outstanding references to this FCB structure, known to the file system driver. As long as the reference count is not zero, the FSD knows that some component is using the FCB and therefore, memory for the FCB structure cannot be deallocated. The reference count field is incremented by 1 whenever a successful create/open operation is processed by the FSD in response to an IRP_MJ_CREATE request. The contents of this field are decremented by 1 whenever a close operation is processed for the FCB in response to an IRP_MJ_CLOSE request. Note that the key concept here is that the open count is simply the number of references known to the FSD; if some external component stores away the pointer to a FCB without the FSD's knowledge, the open count associated with the FCB will not have been incremented, and therefore there is no guarantee made by the FSD that it will be retained once the reference count is 0.

You should also note that the NT I/O Manager expects the FSD to maintain a reference count that is incremented during a create request and decremented during a close request. Furthermore, the I/O Manager also expects that the FSD will free the memory for the FCB only after this count is equal to zero. This knowledge is used by the I/O Manager and other Windows NT components because although neither the I/O Manager nor other system components manipulate the reference count directly, they can and do indirectly manipulate when the counter is decremented, by controlling when a close IRP request (with major function IRP_MJ_CLOSE) is issued. Therefore, NT system components can be reasonably certain of the FSD's behavior and can manipulate how long a FCB structure is retained by the FSD.

The open handle count simply indicates the number of outstanding user open handles for the FCB. This field is also incremented as part of processing an IRP_MJ_CREATE request; it is, however, decremented only in response to an IRP_MJ_CLEAN request issued to the FSD. The IRP_MJ_CLEAN request is issued by the I/O Manager to the FSD whenever a user process closes a file handle for the last time (i.e., when the system open handle count for a file object representing all user open instances is equal to zero).

File object type structures, just like other Windows NT Executive-defined data structures, are maintained by the NT Object Manager. For file object structures, the Object Manager maintains two counts, a ProcessHandleCount for each process that has one or more open handles associated with the file object, and a SystemHandleCount that is the sum total of all ProcessHandleCount values associated with the file object. In addition to these two handle count values, the NT Object Manager maintains an ObjectReferenceCount for all objects.* This reference count is always incremented whenever either the ProcessHandleCount or the SystemHandleCount value is incremented. However, it is possible that the ObjectReferenceCount will be incremented even if neither the ProcessHandleCount nor the SystemHandleCount are incremented (e.g., a kernel-mode component references the object but does not request a new handle for the object).

When a process closes a open handle (i.e., when ZwClose() or NtClose() is invoked), the NT Object Manager decrements both the ProcessHandleCount and the SystemHandleCount for the object. It then invokes any *object-close method* associated with the object being closed; in the case of file objects, the close routine, called lopCloseFile (), is supplied by the Windows NT I/O Manager. The Object Manager supplies the ProcessHandleCount and the SystemHandleCount values to the lopCloseFile () function.

The lopCloseFile () function issues an IRP_MJ_CLEANUP request to the FSD if and only if all outstanding user handles for the file object have been closed and if the passed-in SystemHandleCount for the file object is equal to 1, meaning there was only one outstanding reference on the file object at the time the NtClose ( ) operation was invoked.

Once lopCloseFile () has completed processing (i.e., the FSD has completed processing the cleanup request, if invoked), the Object Manager decrements the ObjectReferenceCount for the file object structure. If this reference count value is 0, the Object Manager deletes the object and, prior to doing so, invokes the *delete method* (lopDeleteFile ()) associated with the file object. lopDeleteFile (), in turn, issues an IRP_MJ_CLOSE request to the FSD managing the file object structure.

Although an FSD receives a cleanup IRP whenever a user handle is closed, the FSD knows that it cannot free up the FCB until the last IRP_MJ_CLOSE request is received for that FCB.

---

* I have made up the symbolic names presented here since the field names for an object structure are not exposed by the Windows NT Object Manager. However, the actual symbolic names used by the Object Manager are relatively uninteresting from our perspective, as long as we understand the logic used by the Object Manager to determine how objects are retained in system memory and when they should be deleted.

For readers with a UNIX background, you can create the analogy where a IRP_MJ_CLEANUP request corresponds to a UNIX *vnode close* operation, and the *last* IRP_MJ_CLOSE request signifies that an *inactivate* operation should be performed on the *vnode* structure.

The reference count and the open handle count maintained internally by the FSD in the FCB structure together help the FSD determine the answer to the following two questions:

How many user handles are outstanding for the FCB?

> In other words, the FSD should have some idea about the total number of IRP_MJ_CREATE requests that were successful, and for which a corresponding IRP__MJ_CLEANUP has not yet been received. As long as this number is nonzero, the FSD knows that at least one thread has a valid open handle to the file stream represented by the FCB, and the FCB structure should be retained in memory. Note that this is in addition to the requirement that the FCB cannot be deleted as long as the ReferenceCount is not 0.
>
> You should also note that, although the OpenHandleCount in the FCB is incremented in response to the create operation (when a new file object is created by the I/O Manager), the count does not necessarily correspond to the system-wide handle count on the corresponding file object, which is maintained by the NT Object Manager. Each time a user file handle is duplicated (say between threads in the same process), or inherited (by a child of a parent process), or whenever some process requests a new handle from a file object pointer, the Object Manager increments the SystemHandleCount on the file object representing the open file instance. Since an FSD is not informed when such duplication or inheritance of file handles occurs, the OpenHandleCount maintained internally by the FSD does not get incremented at such occasions. However, this does not cause any problems for the FSD, since the NT I/O Manager will not invoke an IRP_MJ_CLEANUP request on the particular file object unless all user threads that had an open handle for that file object have also invoked a close operation on it. Therefore, the FSD will always see one cleanup operation corresponding to one create/open request and will decrement the open handle count in response to the cleanup request.

How many outstanding references exist for the FCB structure?

> The ReferenceCount field helps determine the total number of outstanding references for the FCB. It is entirely possible, and indeed very probable, that the ReferenceCount will be nonzero long after the OpenHandleCount has gone down to zero. This simply means that, although all user handles for the FCB have been closed, some kernel-mode component wishes to retain the FCB in memory.

> Typically, this situation arises when the NT Cache Manager and the NT VMM together conspire to keep file data cached in memory, even after a user application process has closed the file, indicating that it has finished processing the file stream. The reason that the Cache Manager and/or the VMM wish to retain the file data in memory (and remember that they cannot have file data retained in memory unless the FCB is also present) is to be able to provide relatively fast response if the user application needs to access the contents of the file stream once again. This may seem a bit silly to you but, quite often, application processes open and close the same file multiple times within the span of a few minutes, and retaining file contents in memory to help speed up the second and subsequent accesses to the same file stream's data enhances system throughput.

To sum up, how would the VMM or any component ensure that an FCB is retained by the FSD in memory? Here's the answer: if, for example, the VMM wishes to ensure that an FCB will stay around, it references some file object associated with the FCB. By referencing the file object, the VMM prevents the NT Object Manager from issuing a close request on that file object, even if all user handles for that particular file object are closed. Therefore, the FCB reference count is not decremented to 0 and the FCB is retained in memory.

Context control blocks (CCB)

A CCB structure is used by the FSD to store state information for a specific open operation performed on a file stream. As discussed earlier, each file stream is uniquely represented in memory by an FCB structure. The FCB structure, however, only contains information that assists in managing user accesses to the file stream as a whole; it does not contain any information about specific user open operations. The CCB structure is used instead for this purpose.

There is one CCB structure created by the FSD for each successful open operation on the file stream. Each CCB structure is typically associated in some way with the unique FCB structure representing the file stream; in the sample FSD, all CCB structures associated with a FCB structure are linked together and accessible from the FCB structure. Also, each CCB structure contains a pointer back to its associated FCB.

The CCB defined by the sample FSD is shown below:

```
typedef struct _SFsdContextControlBlock {
    SFsdldentifier                    Nodeldentifier;
    // Pointer to the associated FCB
    struct _SFsdFileControlBlock      *PtrFCB;
    // all CCB structures for a FCB are linked together
    LIST_ENTRY                        NextCCB;
    // each CCB is associated with a file object
    PFILE_OBJECT                             PtrFileObject;
    // flags (see below) associated with this CCB
    uint32                                   CCBFlags;
    // current byte offset is required sometimes
    LARGE_INTEGER                            CurrentByteOffset;
    // if this CCB represents a directory object open, we may
    // need to maintain a search pattern
    PSTRING                                  DirectorySearchPattern;
    // we must maintain user specified file time values
    uint32                                   UserSpecifiedTime;
} SFsdCCB, *PtrSFsdCCB;
```

Figure 9-3 shows three CCB structures, each created by an FSD in response to an IRP_MJ_CREATE request. Two of the CCB structures are for the same file stream *(file #T),* and they are therefore linked together on FCB #1. The other CCB structure represents an instance of a successful open operation on FCB *#2.*

Note carefully that there is a one-to-one mapping between a file object structure created by the I/O Manager in response to an open/create request and the CCB structure created by the FSD. Therefore, there may be only one FCB structure created for an open file stream, but there can potentially be many CCB structures created for the same file stream, each of which serves as the FSD's context for a successful open operation on the file stream.

Why would a file system wish to create a CCB structure representing each successful open operation? There are quite a few situations when the file system wishes to maintain some state that is not global to the entire file stream (i.e., a state that is not common to all open instances of the file). As an example, the CCB could be used to maintain information about byte-range locks requested by a thread using a particular file object; if the thread closes the file handle without unlocking all of the outstanding byte-range locks for that handle, the FSD can automatically perform the unlock operation upon receipt of an IRP_MJ_CLEANUP request on the file object by checking for the outstanding locks on the CCB associated it. Similarly, the CCB is often also used by an FSD to store information about the next offset from which to resume a directory search operation in response *to find-first and find-next requests* issued by an application.

As you can see, it is quite useful to have context maintained by the FSD for each outstanding open operation, thereby avoiding cluttering up the FCB with nonglobal state information.

The I/O Manager puts no requirements on the FSD about the contents of a CCB structure; the FSD is allowed complete control about whether it wishes to maintain such a structure in the first place. Also, if the FSD does maintain a CCB structure, the contents are completely opaque to the I/O Manager. All of the current NT file system implementations maintain one CCB structure per open operation.

File objects

Finally, Figure 9-3 also depicts three file object structures; two of these file objects represent open operations performed on *file #1* while the third file object represents an open operation performed on *file #2*. Each of these file object structures is allocated and maintained by the I/O Manager in response to an open request by a thread on a file stream. Chapter 4, *The NT I/O Manager,* describes the file object structure in considerable detail.

As was mentioned in Chapter 4, the file system driver is responsible for initializing the `FsContext` and the `FsContext2` fields in the file object structure. In Figure 9-3, you will observe that the FsContext2 field seems to be pointing to the CCB structure. As you read earlier, each instance of an open operation is represented by a CCB structure and, in order to be able to correctly associate a file object with the corresponding CCB, most file system implementations under Windows NT initialize the `FsContext2` field as part of processing an IRP_MJ_CREATE request to refer to the CCB structure that is newly allocated during the open operation. Note that this type of association is not mandated by the NT I/O Manager. If, however, you do develop a filter driver that attaches itself to a device object representing a mounted logical volume for one of the native NT file systems (e.g., FASTFAT, NTFS, or CDFS), you should expect that the `FsContext2` field will have been initialized by the file system implementation to refer to a CCB structure internal to the FSD.

Unfortunately, though, a file system driver does not have an equivalent amount of flexibility with respect to manipulating the `FsContext` field. Although, theoretically, this field also exists solely for driver use, the NT Cache Manager, I/O Manager, and the Virtual Memory Manager make certain assumptions about what this field points to; therefore, in order to integrate your FSD correctly with the rest of the system, your driver must initialize the `FsContext` field to point to the common FCB header structure associated with the FCB. This initialization is performed by the FSD as part of processing an IRP_MJ_CREATE request.

Other Data Structures

In addition to the data structures described above, a file system driver typically maintains other data structures that assist in providing standard file system functionality to the system. These data structures include the following:

Support for byte-range locking

Many file system implementations support byte-range locks. These locks can either be mandatory or advisory in nature. Mandatory locks are part of the specification to which NT FSD implementations must conform; i.e., if a thread acquires a lock on a certain byte range, the FSD implementation on the

Windows NT operating system should enforce the semantics associated with that lock for all other threads attempting to use the same byte range for the same file stream. On the other hand, advisory byte-range locks are a synchronization mechanism for different cooperating processes that need to coordinate concurrent access to the same byte range for a specific file stream. Advisory byte-range file locks are not really supported on the Windows NT platform, although your FSD does have the option of implementing advisory lock support instead of mandatory locks. Be careful if you do this, though, since most Windows-based applications expect locks to be mandatory.

If you design an FSD that supports byte range locking (as do all native NT file system driver implementations), you will undoubtedly maintain certain data structures associated with the FCB/CCB to support this functionality. The sample FSD code presented in this book implements some support for byte-range locking, a topic that is discussed in Chapter 11, *Writing a File System DriverIII.*

Supportfor a Dynamic Name Lookup Cache (DNLC) implementation

You may be familiar with the concept of a DNLC if you have studied file systems on the UNIX platform; if you are not, the DNLC is simply a per-directory cache of the files that were recently accessed within that directory. This list of recently accessed filenames with their on-disk metadata information, which is typically implemented as a hashed list, simply helps the FSD quickly look up a particular file within a specific directory. Normally, most FSD implementations use linear searching to look up specific file names within a directory; the DNLC helps speed things up by skipping the tedious linear search for the more recently accessed files.

Implementation of a DNLC is not mandatory; in fact, the NT system (or any operating system for that matter) does not care whether your FSD uses a DNLC or not. However, you should be aware of the term in case you happen to run into a FSD implementation that does implement this functionality.

Note that the sample FSD presented in this book does not implement any sort of DNLC functionality.

Support for file stream or directory quotas

Although the NT operating system does not yet support quotas associated with file streams, directories, or logical volumes, NT Version 5.0 is expected to provide such support. If you design an FSD that implements quota management, similar to the disk quota implementation on the BSD UNIX operating system, your FSD will have to maintain appropriate data structures to support this quota management functionality. These data structures will include both on-disk structures as well as in-memory representations of the data structures.

Quotas will not be discussed further in this book, though you should certainly be able to add support for this feature for the native FSD implementations on the current NT releases, using the filter driver information provided in Chapter 12, *Filter Drivers.*

Support for opportunistic locking (oplock) functionality in Windows NT

Opportunistic locks are a characteristic of the LAN Manager networking protocol implemented in the Windows family of operating system environments. Basically, *oplocks* are guarantees made by a server for a shared logical volume to its clients. These guarantees inform the client that the contents of a certain file stream will not be allowed to be changed by the server, or if some change is imminent, the client will be notified before the change is allowed to proceed.

The guarantees made by the server node are helpful in improving client response performance to requests accessing remote file streams, since the client can safely cache the file stream data, knowing that the data will not be changed behind its back, leading to data inconsistency across nodes.

Oplocks are not required for NT FSD implementations. However, if you expect logical volumes, managed by your FSD implementation, to be shared using the LAN Manager protocol shipped with the Windows NT operating system (typically, this does not apply if you are designing a network redirector), then you should get familiar with the requirements for successfully implementing oplock support. Otherwise, you may have to assuage unhappy clients who will complain that accessing shared logical volumes managed by your FSD over the LAN Manager network is slower that accessing (say) a shared NTFS logical volume.

Later in this book, we'll explore oplock support in greater detail.

Support for directory change notification

Directory change notification is another neat feature that was implemented in the Windows NT operating system by the native file system drivers and the I/O Manager. Basically, directory change notification functionality works somewhat like the following: a component (either user-mode or kernel-mode) needs to monitor changes to a specific directory or to a directory tree. This component can specify exactly which changes to monitor; e.g., it may be interested in being notified if new files get added to the directory, or it may only need to be notified if a specific file is accessed or modified. The I/O Manager receives a request from the component specifying the type of access the component wishes to monitor and the directory or directory tree that it wishes to monitor. In response to this request, the I/O Manager asks the FSD to asynchronously invoke a specific I/O Manager notification function when the to-be-monitored changes occur.

This feature is very powerful, since it allows a lot of applications to do away with the inefficient polling methodology used before in monitoring changes to particular directories, and to use this notification methodology instead. Supporting this functionality requires the active participation of the I/O Manager and the FSD. Supporting the directory change notification feature is not mandatory; however, all of the native NT FSD implementations support it and you should at least understand the requirements that your FSD would have to meet in order to provide this kind of support.

Directory change notification support is discussed further in the next chapter.

Support for data compression

NTFS provides data compression functionality. Your FSD might also implement online data compression. This would require that your FSD maintain on-disk and in-memory structures that indicate whether a file stream has been compressed or not, and if it has been compressed, store information such as the original file length and other such control information.

The NT I/O Manager provides support for FSD implementations that support data compression by providing system call interfaces allowing a user process to specify whether it expects to receive compressed or uncompressed data back (for read operations). Similarly, Version 4.0 of the operating system allows processes to request that compressed data be written out. In addition, the NT I/O Manager allows a user process to query control information, such as the compressed length, as well as the uncompressed length, of the file stream.

Support for encryption/decryption of data

Some sophisticated file system implementations could potentially provide support for dynamic encryption/decryption of stored data. If you design an FSD that provides such functionality, you will undoubtedly create appropriate data structures that help you manage the encrypted data and decrypt it when required.

It is also quite likely that you might choose to design a filter driver that layers itself on top of the native NT file system implementations and provides support for data encryption and decryption.

Support for logging for fast recovery

NTFS is an example of an FSD implementation that uses in-memory and on-disk logging to be able to provide quick recovery from unexpected system failures. A lot of research has been performed on the design and development of log-based and/or logging file system implementations. If you design such a log-based file system implementation, you will have to maintain appropriate on-disk and in-memory log file streams and also other supporting data structures that will allow you to provide the logging feature to users.

Maintain supportfor on-disk data structures

Typically, a disk-based file system, or a network redirector, will maintain support for the on-disk data structures, such as an on-disk file stream representation (i.e., on-disk FCB/vnode/inode), a directory entry structure (e.g., the `dirent` structure) used in obtaining the contents of a directory, on-disk bitmaps, volume information, and other similar structures. Your FSD may also need to provide appropriate translation routines that would convert the in-memory information to on-disk formats for storage on physical media or for transmitting across a network.

Dispatch Routine: Driver Entry

All kernel-mode drivers are required to have a driver entry routine. This routine is invoked by the NT I/O Manager in the context of a system worker thread at IRQL PASSIVE_LEVEL.

Functionality Provided

File system drivers typically perform the steps listed below in their driver entry routines. Note that these steps are not much different from those performed by other lower-level drivers in their initialization routines:

1. Allocate memory for global data structures and initialize these data structures.
2. Read Registry information if required.

   Although most file system implementations will not provide many configurable parameters, redirectors and servers (e.g., the LAN Manager software) do allow users to specify the values of many configurable parameters.
3. Create a device object to which requests targeted to the FSD itself (as opposed to requests targeted to logical volumes managed by the FSD) can be sent.

   This device object will be one of the following types of device objects:

   — `FILE_DEVICE_DISK_FILE_SYSTEM`

   This is used by disk-based FSD implementations such as NTFS (device object name is *\Ntfs)* and FAT (device object name is *\Fai).*

   — `FILE_DEVICE_NETWORK`

   This is used by network servers, e.g., the LAN Manager Server.

   — `FILE_DEVICE_TAPE_FILE_SYSTEM`

   — `FILE_DEVICE_NETWORK_FILE_SYSTEM`

This is used by the LAN Manager redirector, and other third-party NFS/DPS implementations.

4. Initialize the function pointers for the dispatch routines that will accept the different IRP requests.
5. Initialize the function pointers for the fast I/O path and the callback functions used for synchronization across modules.
6. Initialize any timer objects and associated DPC objects, that your FSD might require.

   Some FSD implementations use timer interrupts to perform asynchronous processing. This requires using timer objects.
7. Initiate asynchronous initialization, if required.

   For example, if your driver needs to create worker threads that perform some initialization asynchronously, the threads can be created either in the context of the DriverEntry ( ) routine or as an asynchronous operation.
8. Physical-media-based file system drivers will invoke loRegisterFileSystem ( ) to register the current loaded instance of the driver.

   Note that the physical-media-based FSDs supported by the NT I/O Manager are disk-, virtual disk-, CDROM-, and tape-based FSD implementations. By registering the FSD with the I/O Manager, your FSD will ensure that it is on the list of file system drivers asked by the I/O Manager to examine and potentially perform a mount operation on media accessed for the first time in a boot cycle.
9. Network file system implementations that support Universal Naming Convention (UNC) names will invoke FsRtlRegisterUncProvider () to register themselves with the MUP component.
10. Network redirectors and servers will also register a shutdown notification function using the IoRegisterShutdownNotification() routine, ensuring that the FSD has an opportunity to flush modified data before the system goes down, as well as perform any other necessary processing.

Code Fragment

The following DriverEntry () code sample performs the previously listed steps. The code fragment contains conditionally compiled code for disk-based file system drivers as well as for network redirectors:

```
NTSTATUS DriverEntry(
PDRIVER_OBJECT        DriverObject,         // created by the I/O subsystem
PUNICODE_STRING       RegistryPath)         // path to the Registry key
{
    NTSTATUS            RC = STATUS_SUCCESS;
    UNICODE_STRING      DriverDeviceName;
    BOOLEAN             RegisteredShutdown = FALSE;
    try {
        try {
            // initialize the global data structure
            RtlZeroMemory(ScSFsdGlobalData, sizeof(SFsdGlobalData));

            // initialize some required fields
            SFSdGlobalData.Nodeldentifier.NodeType =
                                            SFSD_NODE_TYPE_GLOBAL_DATA;
            SFsdGlobalData.Nodeldentifier.NodeSize =
                                            sizeof(SFsdGlobalData);

            // initialize the global data resource and remember the fact
            // that the resource has been initialized
            RC =
            ExInitializeResourceLite(&(SFsdGlobalData.GlobalDataResource));
            ASSERT(NT_SUCCESS(RC));
            SFsdSetFlag(SFsdGlobalData.SFsdFlags,
                SFSD_DATA_FLAGS_RESOURCE_INITIALIZED);

            // store a pointer to the driver object sent to us by the I/O
            // Mgr.
            SFsdGlobalData.SFsdDriverObject = DriverObject;

            // initialize the mounted logical volume list head
            InitializeListHead(&(SFsdGlobalData.NextVCB));

            // before we proceed with any more initialization, read in
            // user supplied configurable values ...
            if (!NT_SUCCESS(RC = SFsdObtainRegistryValues(RegistryPath))) {
                // in your commercial driver implementation, it would be
                // advisable for your driver to print an appropriate error
                // message to the system error log before leaving
                try_return(RC);
            }

            // we have the Registry data, allocate zone memory
            // This is an example of when FSD implementations
            // try to preallocate some fixed amount of memory to avoid
            // internal fragmentation and/or waiting later during runtime
            // ...
            if (!NT_SUCCESS(RC = SFsdlnitializeZones())) {
                // we failed, print a message and leave ...
                try_return(RC);
            }

            // initialize the IRP major function table, and the fast I/O
            // table
            SFsdlnitializeFunctionPointers(DriverObject);

            // create a device object representing the driver itself
            // so that requests can be targeted to the driver ...
            II e.g., for a disk-based FSD, "mount" requests will be sent to
            // this device object by the I/O Manager.
            // For a redirector/server, you may have applications
            // send "special" lOCTLs using this device object ...
            RtlInitUnicodeString(&DriverDeviceName, SFSD_FS_NAME);
            if (!NT_SUCCESS(RC = loCreateDevice(
                    DriverObject,        // our driver object
                    0,          // don't need an extension for this object
                    &DriverDeviceName,
               // name - can be used to "open" the driver
               // see the book for alternate choices
                    FILE_DEVICE_DISK_FILE_SYSTEM,
                    0,                         //no special characteristics
               // do not want this as an exclusive device, though you might
                    FALSE,
                    &(SFsdGlobalData.SFsdDeviceObject)))) {
                // failed to create a device object, leave ...
                try_return(RC);
            }

#ifdef   _THIS_IS_A_NETWORK_REDIR_OR_SERVER_

            // since network redirectors/servers do not register
            // themselves as "file systems," the I/O Manager does not
            // ordinarily request the FSD to flush logical volumes at
            // shutdown. To get some notification at shutdown, use the
            // IoRegisterShutdownNotification() instead ...
            if (!NT_SUCCESS(RC =
                    loRegisterShutdownNotification(
                        SFsdGlobalData.SFsdDeviceObject))){
                // failed to register shutdown notification ...
                try_return(RC);
            }
            RegisteredShutdown = TRUE;

            // Register the network FSD with the MUP component.
            if (!NT_SUCCESS(RC = FsRtlRegisterUncProvider(
                                                &(SFsdGlobalData.MupHandle),
                                                &DriverDeviceName,
                                                FALSE))) {
                try_return(RC);
            }

#else          // This is a disk-based FSD

            // register the driver with the I/O Manager, pretend as if
            // this is a physical-disk-based FSD (or in other words, this
            // FSD manages logical volumes residing on physical disk
            // drives)
            loRegisterFileSystern(SFsdGlobalData.SFsdDeviceObject);

#endif      // _THIS_IS_A_NETWORK_REDIR_OR_SERVER_

        } except (EXCEPTION_EXECUTE_HANDLER) {
            II we encountered an exception somewhere
            RC = GetExceptionCode();
        }

        try_exit:    NOTHING;
    } finally {
        // start unwinding if we were unsuccessful
        if ( !NT_SUCCESS(RC) ) {

#ifdef      _THIS_IS_A_NETWORK_REDIR_OR_SERVER_
      if (RegisteredShutdown) {
         IoUnregisterShutdownNotification (SFsdGlobalData. SFsdDeviceObject)  ;
      }
#endif     // _THIS_IS_A_NETWORK_REDIR_OR_SERVER_

            // Now, delete any device objects, etc. we may have created
            if (SFsdGlobalData. SFsdDeviceObject) {
                IoDeleteDevice( SFsdGlobalData. SFsdDeviceObject) ;
            SFsdGlobalData. SFsdDeviceObject = NULL;
            }

            // free up any memory we might have reserved for zones/
            // lookaside lists
            if ( SFsdGlobalData. SFsdFlags
                   & SFSD_DATA_FLAGS_ZONES_INITIALIZED)  {
            SFsdDestroyZones ( ) ;
            }

            // delete the resource we may have initialized
            if (SFsdGlobalData. SFsdFlags
                   & SFSD_DATA_FLAGS_RESOURCE__INITIALIZED) {
                // uninitialize this resource
                ExDeleteResourceLite(& (SFsdGlobalData. GlobalDataResource) )  ;
                SFsdClearFlag ( SFsdGlobalData . SFsdFlags,
                    SFSD_DATA_FLAGS_RESOURCE_INITIALIZED)   ;
            }
        }
    }

    return (RC) ;
}

void SFsdlnitializeFunctionPointers (
PDRIVER_OBJECT        DriverObject )        // created by the I/O subsystem
{
   PFAST_IO_DISPATCH    PtrFastloDispatch = NULL;

    // initialize the function pointers for the IRP major
    // functions that this FSD is prepared to handle ...
    // NT Version 4.0 has 28 possible functions that a
    // kernel mode driver can handle.
    // NT Version 3.51 (and earlier) has only 22 such functions,
    //of which 18 are typically interesting to most FSDs.
```

```
II The only interesting new functions that a FSD might (currently)
// want to respond to beginning with are the
// IRP_MJ_QUERY_QUOTA and the IRP_MJ_SET_QUOTA requests.

// The code below does not handle quota manipulation; neither
// does the NT Version 4.0 operating system (or I/O Manager).
// However, you should be on the lookout for any such new
// functionality that your FSD might have to implement in
// the near future.

// The functions that your FSD might wish to consider implementing
// (and are not covered below) are:

// Note that the "IRP_MJ_CREATE_NAMED_PIPE", and the "IRP_MJ_CREATE_
// MAILSLOT" requests won't be directed toward any FSD you would
// develop.
DriverObject->MajorFunction[IRP_MJ_.CREATE]                 = SFsdCreate;
DriverObject-•>MajorFunction[IRP_MJ__CLOSE]                 = SFsdClose;
DriverObject->MajorFunction[IRP_MJ_READ]                    = SFsdRead;
DriverObject-•>MajorFunction [IRP_MJ_.WRITE]                = SFsdWrite;
DriverObject -•>MajorFunction[IRP_MJ.QUERY_INFORMATION}     = SFsdFilelnfo;
DriverObj ect-•>Maj orFunction[IRP_MJ..SET_INFORMATION]     = SFsdFilelnfo;
DriverObj ect-•>Maj orFunction [IRP_MJ__FLUSH_BUFFERS]      = SFsdFlush;
DriverObj ect-•>Maj orFunction[IRP_MJ..QUERY_VOLUME_INFORMATION]
                                                            = SFsdVolInfo;
DriverObj ect->Maj orFunction[IRP_MJ__SET_VOLUME_INFORMATION]
                                                            = SFsdVolInfo;
DriverObj ect->Maj orFunction[IRP_MJ_.DIRECTORY_CONTROL]
                                                          = SFsdDirControl;
DriverObj ect->MajorFunction[IRP_MJ__FILE_SYSTEM_CONTROL]
                                                           = SFsdFSControl;
DriverObject->MajorFunction[IRP_MJ_.DEVICE_CONTROL]
                                                       = SFsdDeviceControl;
DriverObj ect-->Maj orFunction[IRP_MJ.SHUTDOWN]             = SFsdShutdown;
DriverObj ect-->MajorFunction[IRP_MJ_LOCK_CONTROL]
                                                          = SFsdLockControl;
DriverObject-->MajorFunction[IRP_MJ_.CLEANUP]             = SFsdCleanup;
DriverObj ect-->Maj orFunction[IRP_MJ_.QUERY_SECURITY]    = SFsdSecurity;
DriverObj ect- ->Maj orFunction[IRP_MJ__SET_SECURITY]
                                                          = SFsdSecurity;
DriverObject-->Maj orFunction[IRP_MJ_.QUERY_EA]
                                                       = SFsdExtendedAttr;
DriverObj ect- ->Maj orFunction[IRP_MJ_.SETJEA]
                                                       = SFsdExtendedAttr;

// Now, it is time to initialize the fast I/O stuff ...
PtrFastloDispatch = DriverObject->FastIoDispatch
                           = &(SFsdGlobalData.SFsdFastIoDispatch);

// initialize the global fast I/O structure
// NOTE: The fast I/O structure has undergone a substantial revision
// in Windows NT Version 4.0. The structure has been extensively
// expanded.
```

```
    II Therefore, if your driver needs to work on both V3.51 and V4.0+,
    // you will have to be able to distinguish between the two versions
    // at compile time.
    PtrFastIoDispatch->SizeOfFastIoDispatch    = sizeof (FAST_IO_DISPATCH) ;
    PtrFastIoDispatch->FastIoCheckIf Possible
                                                = SFsdFastloChecklfPossible;
    PtrFastIoDispatch->FastIoRead              = SFsdFastloRead;
    PtrFastIoDispatch->FastIoWrite             = SFsdFastloWrite;
    PtrFastIoDispatch->FastIoQueryBasicInfo    = SFsdFastloQueryBasidnfo;
    PtrFastIoDispatch->FastIoQueryStandardInfo = SFsdFastloQueryStdlnfo;
    PtrFastIoDispatch->FastIoLock              = SFsdFastloLock;
    PtrFastIoDispatch->FastIoUnlockSingle      = SFsdFastloUnlockSingle;
    PtrFastIoDispatch->FastIoUnlockAll         = SFsdFastloUnlockAll;
    PtrFastIoDispatch->FastIoUnlockAllByKey    = SFsdFastloUnlockAllByKey;
    PtrFastIoDispatch->AcquireFileForNtCreateSection
                                                = SFsdFastloAcqCreateSec;
    PtrFastIoDispatch->ReleaseFileForNtCreateSection
                                                = SFsdFastloRelCreateSec;

    // the remaining are only valid under NT Version 4.0 and later
#if (_WIN32_WINNT >= 0x0400)
    PtrFastIoDispatch->FastIoQueryNetworkOpenInfo = SFsdFastloQueryNetlnfo;
    PtrFastIoDispatch->AcquireForModWrite         = SFsdFastloAcqModWrite;
    PtrFastIoDispatch->ReleaseForModWrite         = SFsdFastloRelModWrite;
    PtrFastIoDispatch->AcquireForCcFlush          = SFsdFastloAcqCcFlush;
    PtrFastIoDispatch->ReleaseForCcFlush          = SFsdFastloRelCcFlush;

    // MDL functionality
    PtrFastIoDispatch->MdlRead               = SFsdFastloMdlRead;
    PtrFastIoDispatch->MdlReadComplete       = SFsdFastloMdlReadComplete;
    PtrFastIoDispatch->PrepareMdlWrite       = SFsdFastloPrepareMdlWrite;
    PtrFastIoDispatch->MdlWriteComplete      = SFsdFastloMdlWriteComplete;

    // although this FSD does not support compressed read/write
    // functionality, NTFS does, and if you design a FSD that can provide
    // such functionality,
    // you should consider initializing the fast I/O entry points for
    // reading and/or writing compressed data . . .
#endif    // (_WIN32_WINNT >= 0x0400)

    // last but not least, initialize the Cache Manager callback functions
    // which are used in CcInitializeCacheMap ( )
    SFsdGlobalData.CacheMgrCallBacks . AcquireForLazyWrite
                                                = SFsdAcqLazyWrite;
    SFsdGlobalData.CacheMgrCallBacks.ReleaseFromLazyWrite
                                                = SFsdRelLazyWrite;
    SFsdGlobalData.CacheMgrCallBacks.AcquireForReadAhead
                                                = SFsdAcqReadAhead;
    SFsdGlobalData.CacheMgrCallBacks .ReleaseFromReadAhead
                                                = SFsdRelReadAhead;

    return;
}
```

Notes

The two routines listed comprise the bulk of the driver entry code for the sample FSD driver. The code fragment doesn't initiate any asynchronous initialization; neither does it initialize any timer or DPC objects. However, your FSD can certainly perform such functions in its driver entry routine. Otherwise, the code pretty much follows the logical steps listed earlier that most FSD implementations perform in the initialization routine.

For additional details on some of the supporting functions invoked by the driver entry routine, consult the disk that accompanies this book.

Dispatch Routine: Create

As a file systems designer, you will probably count the "create" routine as one of the more difficult routines to design and implement. This routine forms the very core of your FSD, since in order to perform any operation on on-disk objects, the object must first be created and/or opened. Therefore, not only are create routines required to be robust, but the design and implementation of the create routine can also contribute significantly to overall system performance, because badly designed or implemented "create" routines can become a bottleneck very easily during frequent (high stress) file system manipulation operations.

Logical Steps Involved

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

```
typedef struct _IO_STACK_LOCATION {

    // …

    union {

        //…

        // System service parameters for:  NtCreateFile
        struct {
            PIO_SECURITY_CONTEXT SecurityContext;
            ULONG Options;
            USHORT FileAttributes;
            USHORT ShareAccess;
            ULONG EaLength;
        } Create;

        // …
} Parameters;

// …

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

Create routines are conceptually not very difficult. Unfortunately, however, the details involved in processing a create request sometimes become overwhelming. Logically, you need to perform the following operations when processing a create or an open request on an object stored on secondary storage:*

1. First, you would have to obtain the caller-supplied path that leads you to the object of interest.

   Note that most of the common, commercially available operating systems are equipped only to handle inverted-tree-based file system organizations. In this kind of file system arrangement, there is a *container object,* such as a directory structure, which in turn leads you to the actual *named data objects,* also known as file streams. Container objects typically also contain other container objects in addition to file streams, thereby leading to the inverted-tree structured file system layout.

   On some operating system platforms such as most commercial UNIX implementations, the FSD is not supplied with the entire path leading to a specific named file stream. Instead, the operating system performs the task of parsing the path leading to the target file stream, and only supplies the FSD with a handle to the container object and the name of an object to open within the container. See Figure 9-5 for an illustration of a file system layout. For example, if a thread wished to open an object *\dirl\dir2\...\foo* on such a platform, the operating system would first request the FSD to open \ (or the *root* of the tree), then request the FSD to open *dirl* given a handle to \ (received from the just concluded open request), then move on to *dir2* given a handle to *\dirl* and so on, until the operating system finally requests the FSD to open the object *foo*—the actual target of the create/open request

   On Windows NT platforms, however, the I/O Manager (which happens to be the component that invokes the FSD *create* dispatch entry point) does not perform such name parsing. Instead, the I/O Manager supplies the FSD with the entire caller-supplied path and then expects the FSD to perform any required processing that might be required, including parsing the pathname supplied; i.e., the I/O Manager would give the FSD the complete name, *\dirl\dir2\...\foo* in the example discussed earlier, for processing in the

---

* Note that a request to create a new object always implies that the caller wishes to open the object as well. Therefore, when I use the term *create in* this section, I do so generically, implying that this is either some form of *open* request, or a *create and open* request.

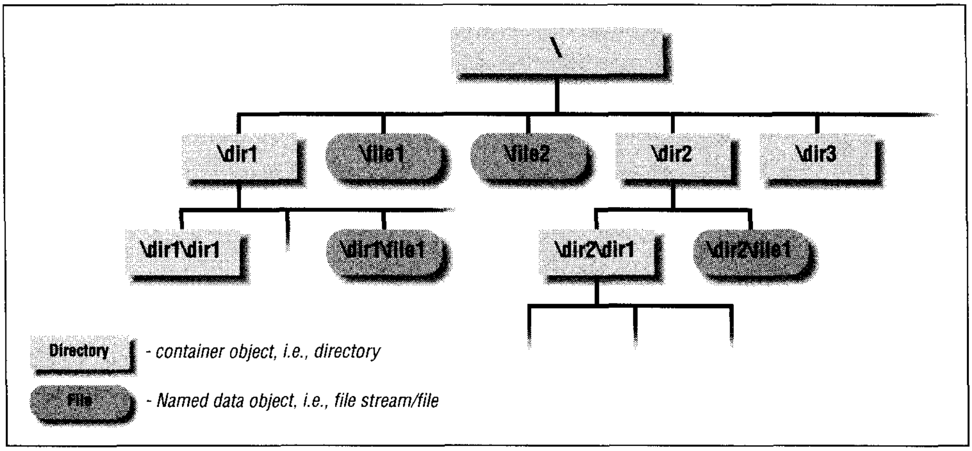

Figure 9-5. Simple representation of an inverted-tree file system layout

create dispatch routine. This is both a blessing and a curse: it affords considerable latitude to the FSD on how it can structure on-disk names and paths leading to on-disk objects, but the complete responsibility for parsing the caller-supplied name is the responsibility of the FSD implementation.

One final point should be mentioned here is that the NT I/O Manager supports the concept of relative open operations, where the pathname supplied by the caller is assumed to be relative to a container (directory) object opened earlier, instead of being a path beginning at the root of the file system tree. Your FSD must be able to distinguish between these two kinds of create/open operations and be able to deal with each of them correctly.

2. Now, your FSD can obtain other arguments supplied by the caller that will determine how you process the create/open request. These arguments include information on whether the caller has specified that the target must be created (and the FSD should return an error if the target already exists), whether the target must be opened only (and the FSD should return an error if the target object does not exist), or whether the target should be created conditionally (opened if it exists but created if it does not). Note that the caller may also request the creation of a hard link to an existing object. However, in the Windows NT operating system environment, this does not happen via the create entry point so we will discuss it later.*

   Other caller-supplied arguments include any specifications on the type of object being created or opened (e.g., whether the caller wishes to create or

---

* The method used to create a hard link in Windows NT is via the IRP_MJ_SET_INFORMATION request, which is described in the next chapter.

open a container object or a file-stream object). Furthermore, the caller can specify attributes to associate with the object if a new one is being created; e.g., a caller might specify that the *delete-on-close* attribute be associated with the object, to make it a temporary object that is automatically deleted by the FSD once the object has been closed for the last time. Other attributes include the sharing mode that a caller might wish to enforce with the object being opened.

3. Once your FSD has obtained all of the caller-supplied information, all it needs to do is to try to locate the target object, given the user-supplied pathname leading to this object.

   Your driver may or may not be successful in locating the target object, given the path to that object. If the object is found and the caller has requested that a new object must be created (`CreateDisposition` in sample code below is FILE_CREATE), your driver must return an *object exists* error code. If however the object is not found—i.e., it does not exist, but the caller has specified that the object must only be opened and not created (`CreateDisposition` is FILE_OPEN)—your driver must return an *object not found* error. Finally, the *create-if-not-found* or *open-if-found* option (`CreateDisposition` is FILE_OPEN_IF or FILE_OVERWRITE_IF) is the most flexible since it allows your driver to return success more often than not. Regardless, you will either decide to return an error at this point or move on to the next step.

4. At this time, your driver can check whether the caller can be allowed to create/open the object, given the caller's identity, the operation requested, and the sharing mode requested. The sharing mode assumes greater importance if another thread already has the target object open, in which case the new request must not conflict with the sharing mode allowed by the previous open operations.

   If your FSD provides access-checking functionality, you can use the caller's subject context to validate whether the caller has appropriate privileges to perform the desired operation on the target file/directory object. You may also be required to perform traverse access checking while parsing the pathname leading up to the target object.

5. If your FSD is successful in creating and/or opening the target object, it will have to create appropriate in-memory data structures that can be used later in accesses to the object. For example, if no FCB describing this object currently exists in memory, the FSD must create one. More likely than not, the FSD will also create a CCB structure at this time to maintain some context for this particular instance of a create request.

   Both of these structures, as well as any supporting structures, will have to be initialized appropriately by the FSD.

6. The Windows NT I/O Manager expects that, for a successful create/open operation, the FSD will initialize certain fields in the file object structure as described earlier. Your FSD should do so at this point if the create/open operation does succeed.
7. Lastly, your FSD must convey the results of the create/open request to the I/O Manager, which in turn will forward the results to the caller.

As long as your driver understands the steps, described here, that it must perform in response to a create request and implements them systematically, you should be able to handle create requests correctly.

Synchronization Issues

There are other, more advanced considerations that file systems often have to deal with in designing the create/open dispatch entry point, as well as other dispatch entry points.

One of the issues that you should understand well is the synchronization that your file system driver must perform to process create requests correctly and efficiently. For example, your FSD must be cognizant of the fact that multiple concurrent create/open operations could potentially be happening in the same file system and possibly in the same directory. All of these concurrent operations should be handled consistently. For example, consider the situation when two threads request that the target file *\dirl\foo* be created if it does not exist and opened if it does exist. Assume that the file does not exist. Unless your FSD is careful about synchronizing both requests correctly, it might actually allocate a new directory entry for the file twice in directory *dirl* and assign the same name *foo* to both the directory entries. The inconsistent result can be avoided, however, by serializing both requests with respect to each other.

Another consideration that your FSD must take into account is that delete and/or file close operations (leading to the possible destruction of a FCB) could also be happening concurrently with other create/open requests. In this case, your FSD must be careful to correctly synchronize concurrent operations, such that a consistent view of file system data structures is always maintained. For example, two threads may both be manipulating an object *\dirl\foo* concurrently. One thread might request that the object be deleted, while the other thread may wish to create/open the object. Your FSD must again be very careful to always maintain internal consistency. Note that it is not important to guarantee which thread is allowed access first; i.e., should the thread trying to delete be allowed to go first or should the thread performing the create/open be allowed to go first? No file system guarantees any such ordering to its clients. Regardless of which operation happens first, however (and yes, it might be possible for one of the user requests to get an error code returned, depending on the sequence in which the threads are allowed to proceed), it is important that both not be allowed to proceed together or file system internal data structures will surely become corrupted.

At other times, your FSD will have to synchronize between multiple threads performing I/O to the same file control block concurrently. For example, a thread could be performing a read operation at the same time as another thread is attempting a write.* In such cases, it is the FSD implementation's responsibility to ensure that the read and write requests are serialized with respect to each other. It is not the responsibility of the FSD to ensure that the read and write occur in some specific sequence; the caller can ensure such order using other methods, such as byte-range locking on the file stream. As long as the read and write routines do not overlap within the FSD (i.e., once the FSD has begun processing the read in `SFsdRead()`, the write is blocked in `SFsdWrite()` until the thread executing `SFsdRead()` completes processing, or vice versa), the FSD will have succeeded in maintaining its responsibilities.

You will also need to synchronize multiple threads attempting other directory manipulation operations concurrently. For example, one thread might be querying the contents of a directory while another might be in the process of deleting an object within the directory.

Regardless of which operation is being performed on a particular FCB, your file system driver will surely have to utilize some sort of synchronization mechanism to ensure orderly access. If you are developing a significantly more complex FSD, such as one that is inherently networked (e.g., NFS) and/or distributed in nature (e.g., DPS), your task is now made even more complicated by the fact that your create/open operations must now be made consistent across nodes as well.

Note that, in most situations, synchronization attempts described here and practiced throughout the sample code are fairly fine-grained; the FSD attempts to synchronize at the level of each FCB. If two concurrent operations proceed independently on two different FCB structures (on two different file streams or directories), the FSD will not serialize the two operations. At this point, you may be wondering why you should not simply disallow concurrent operations to the same logical volume? Although such synchronization might be a little bit coarser than that performed at the level of an FCB, it seems as if that would make life a

---

* Often, you may encounter the case where a delayed-write from the NT Cache Manager on an FCB proceeds concurrently with a user read request on the same FCB. At other times, a user write may be received at the same time as a Cache Manager read-ahead operation. In most such cases (unless the user has requested direct disk I/O), such operations will be allowed to proceed concurrently. This is in contrast to the situations where two concurrent user write requests on an FCB will be serialized with respect to each other or a write and a read operation proceeding concurrently and independently will also be serialized with respect to each other. This is discussed in detail later in the book as well.

lot simpler. Unfortunately, although serializing all accesses to a logical volume may make an FSD developer's life easier, generally speaking, for any file system, preventing concurrent access to different files within the same logical volume is simply not a viable option. Imagine how upset you might become if a file system made you wait two hours to respond to a dir request simply because it serialized all accesses to files within a logical volume! As a matter of fact, in order to enhance response time and throughput, you should always be looking for ways in which your FSD can safely increase parallelism and concurrency. For example, you should always allow multiple threads to concurrently read data for the same file stream, even though for write operations you would have to serialize across different threads.

Note that a file system driver typically never keeps any resources acquired across invocations of a particular dispatch routine. For example, if the create dispatch routine is invoked and the thread acquires some resources as part of the processing performed in the create, these resources must be released before the thread exits file system code (from the create entry point). Not doing this will lead either to a system crash or to a hang/deadlock. Resources are only acquired while processing some file system code within a particular dispatch routine.

Here is a set of general synchronization rules followed by the sample FSD to ensure consistent access to an FCB. Your driver is free to determine appropriate synchronization rules specific to your situation:

- The sample FSD will use the MainResource and the PagingloResource as the two synchronization resources for each file control block structure representing a file stream. These are read/write locks and will be acquired either shared or exclusively, depending upon the specific situation.

  Note once again that your FSD is free to use other synchronization primitives in addition to the MainResource and the PagingloResource. You can choose from any of the mutex/executive mutex primitives available or use counting semaphores.

- Since two resources will be used by the FSD to synchronize access to an FCB, a locking hierarchy must be maintained for these two resources. The locking hierarchy I will follow is that, if both resources need to be acquired, the MainResource will always be acquired before the PagingloResource is acquired.

  Whenever multiple synchronization primitives are used to synchronize access to an object, you must determine a hierarchy between such primitives to avoid deadlock scenarios. For example, consider the case where no hierarchy was defined and two threads tried to concurrently manipulate the same FCB. For some reason, both threads determine that they must acquire both the resources to completely shut out all other operations on the FCB. One of the threads may now acquire the MainResource and then try to acquire the PagingIoResource. The other thread, in the meantime, might have already acquired the PagingIoResource and could now be attempting to acquire the MainResource. Both threads will now block on each other, and neither can subsequently make any headway. This situation will not occur if the locking hierarchy described above is followed, since only one thread will be able to acquire the MainResource and that thread can continue on to acquire the PagingIoResource.

- Typically, the sample FSD will acquire the MainResource only to perform synchronization between multiple user threads accessing the same FCB structure. For example, if two user threads perform a *query directory* operation on an FCB representing a directory (container object), the SFsdDirControl() dispatch routine implementation will first attempt to acquire the MainResource exclusively for the target FCB on behalf of each thread. This will ensure that accesses will be serialized across both threads within the specific dispatch routine.

  Similarly, as described earlier, if multiple user threads try to perform I/O on the same file stream (FCB) concurrently, each of threads will use the MainResource to synchronize access to the FCB. All threads attempting a read I/O operation will acquire the MainResource shared, allowing multiple reads to proceed concurrently on the file stream. All threads attempting a write operation, however, will do so having acquired at least the MainResource exclusively, thereby preventing any other read or write operation to proceed concurrently.

- The sample FSD will typically acquire the PagingIoResource only when performing read or write operations that have the IRP_PAGING_IO flag set in the IRP structure.* This will simply allow the FSD to synchronize across multiple concurrent paging I/O operations. However, user read/write operations and paging I/O operations will typically not be serialized with respect to each other (since user requests use the MainResource while the NT VMM-initiated requests use the PagingIoResource) and will therefore proceed concurrently.

- On some occasions, the sample FSD will acquire both the MainResource and the PagingIoResource on behalf of the same thread, while ensuring

---

* The presence of this flag indicates that this request comes to the FSD via the NT VMM. The FSD must be extremely careful in handling paging I/O requests, since page faults are not allowed at that time. Also, most consistency checks will be bypassed by the FSD when a paging I/O request is directed to page files. The FSD implementation will simply trust the VMM to submit a valid request and pass on the request to the underlying lower-level device drivers for completion (by reading/writing the requested byte range from/to secondary storage devices).

that the resource acquisition hierarchy is always maintained (i.e., the MainResource is always acquired before the attempt to acquire the PagingIoResource). This will be done for specific situations only; e.g., both resources will be acquired when the size of file stream changes (a file is truncated/extended) .

TIP It is difficult to present any sort of cookbook on how and when resources should be acquired by all FSDs since each FSD has unique requirements that dictate the synchronization methodology adopted by it. However, in general, keep in mind that many critical in-memory fields contained in the FCB data structure are synchronized using the PagingIoResource. That said, however, file create operations and all user-initiated operations are usually synchronized using the MainResource. Finally, as mentioned, some operations (e.g., an IRP_MJ_CREATE that also specifies a certain file allocation size) will require both resources to be acquired.

As noted, it is certainly not easy and will require considerable thought on your part to determine the correct synchronization methodology that -will ensure data consistency for your FSD.

Consider the create dispatch entry point. When processing the create/open request, the FSD has to examine the contents of each directory that comprises the pathname leading up to the target object. When examining the contents of a directory object in the path, the FSD must ensure that the contents of the directory do not unexpectedly change. To do this, the FSD will block changes by acquiring the MainResource for the directory object's FCB. Your FSD can determine whether the MainResource needs to be acquired exclusively or shared. Acquiring the resource exclusively ensures that no other create or lookup operation can proceed concurrently in the same directory, while acquiring it shared does allow other create or lookup operations to proceed concurrently in the same directory.

One final note: often FSD implementations are fairly coarse-grained in the synchronization employed while processing create/open requests. For example, it appears that the Windows NT CDFS implementation simply acquires a resource associated with the volume control block exclusively for the logical volume on which the create operation is being performed. This ensures that no other create/open can proceed while the current request is being processed. There are, however, some FSD implementations that do not lock out all other create/open operations when processing any one of them. You can determine the appropriate locking for your FSD.

Simple Algorithm to Process a Create/Open Request

Before you examine the code/pseudocode provided here, understand the following simple algorithm used to process a create/open request of an on-disk file object. You know that the NT I/O Manager depends upon the FSD to parse a complete pathname in the create dispatch routine.* Assume that the I/O Manager has given a pathname *\dirl\dir2\dir3\foobar* to be processed. The following simple terms and rules are important:

- The *pathname* supplied is composed of individual components.
- Each component is identified by a string composed of characters, e.g., *dir2*.

  Your FSD can determine the range of characters and symbols that would be acceptable to it.
- Components are demarcated by the presence of a valid separator character; the convention on Windows platforms is to use the \ character.
- The last component in the pathname string is the actual *target* of the create/open operation.
- Each component in the pathname string preceding the last component must be a valid subdirectory.

The algorithm used to parse the pathname can be implemented as follows:

```
determine the starting directory from which to begin processing;
get current component and next component by parsing pathname
    (using the \ as the path separator);
current component = starting directory (= \ in example given);
open current component;
next component = string obtained from the pathname (= dirl)
while (TRUE) {
    perform traverse access checks if required, i.e., check whether
        caller has appropriate privileges to read contents of the
        directory identified by the current component;

    if (entire path has been parsed) {
        // we are down to the last token/component.
        //   In our example, we are at the point where current
        // component = dir3 and next component = foobar
        final component = next component;
        break;
    }
    lookup† next component in current component
```

---

* If you are familiar with UNIX file system structures and implementations, this is simply a `namei()` (name-to-inode) type of routine that a file system must implement.

† A lookup operation simply means checking whether the named object exists within the specified directory. To perform the lookup, most FSD implementations read in the list of entries that comprise the directory and perform a string comparison of the name being looked up with all of the entries in the directory in a linear fashion. If a match is found, the object exists in the directory; otherwise, the FSD determines that the object does not exist.

```
        (may involve I/O to obtain directory contents) ;
    if (not found) {
        return(STATUS_OBJECT_PATH_NOT_FOUND);
    }
    if (next component looked-up != directory) {
        return(STATUS_OBJECT_PATH_NOT_FOUND);
    }
    close current component;
    open next component;
    current component = next component;
    next component = get next string identifier from pathname;
}
lookup final component;
if (not found) {
    if (create requested) {
        perform create ...;
     } else {
        return(STATUS_OBJECT_NAME_NOT_FOUND);
     }
} else {
     if (create only request) {
         return (STATUS_OBJECT_NAME_COLLISION);
     }
}
open final component;
initialize various internal (FCB/CCB) and external (file object)
    data structures;
return (results);
```

This algorithm is the general methodology used by file systems in response to a create/open request. Basically, the algorithm starts parsing the given pathname, beginning at the user-supplied starting point. This could either be the root of the file system, or in the case of relative file opens, it would begin at the directory identified by the RelatedFileObject field, initialized by the NT I/O Manager in the newly created file object for the target of the open.

Once the starting point has been determined, the file system driver simply iterates through all of the components that comprise the pathname leading to the target file/directory object. If the FSD is security conscious, it will always check whether the caller has appropriate privileges to traverse and/or read the directories in the pathname.

After the FSD has successfully traversed the user-supplied pathname, the FSD will look for the target file/directory object. Depending on whether the object already exists or not, and also on the type of create/open operation requested by the user, the FSD will either complete the request or return an appropriate error code to the caller. The code fragments later provide you with some more information about how to process create/open requests on the Windows NT platform.

More About the Name Supplied to the FSD

Before you examine the following code fragment, you may still be wondering about the composition of a pathname when it is sent to you by the I/O Manager. You might also wonder how the I/O Manager determines that a create/open request should be sent to your FSD.

Let me briefly summarize the answer to the latter question first. Consider physical-disk-based FSD implementations. Remember from Chapter 4, that the I/O Manager assigns drive letters to each physical disk in the system. When a thread decides to create/open an object, it must supply a path leading to the object. Typically this path will look like *D:\dir1\dir2....\target_file.* Now, this create/open request gets directed to the I/O Manager,* which determines that the target device object for this request is the physical disk represented by the letter *D:*. (Note that *D:* and other such symbolic identifiers are simply symbolic links that point to a particular physical device object; in this case, to a SCSI disk drive.)

The I/O Manager then checks the volume parameter block (VPB) associated with the device object for the physical disk to see whether any logical volume has been mounted on the physical disk identified by the device object. If no mount operation has been performed, the I/O Manager will attempt a new mount sequence, which is explained in further detail in the next chapter. If a logical mount had been previously performed or after a successful mount operation is completed, the I/O Manager will send the complete pathname excluding the portion that has already been parsed, i.e., excluding *D:,* to the FSD as an argument to the create/open dispatch routine. The logic here is simple: the NT Object Manager and the I/O Manager have already parsed (processed) some portion of the user-supplied pathname to determine the target FSD for the request. The FSD should not need that portion of the string. The remainder of the user-supplied string, however, has not yet been parsed/processed, and it is sent in its entirety to the FSD.

For network redirectors, the situation is not very different conceptually. Requests are directed to specific network redirectors, identified by the symbolic name associated with a device object created by the redirector. For example, a hypothetical redirector might create a symbolic link *F:* that points to a device object that handles all I/O-related requests for a remote, shared network drive. Once again, the actual pathname sent by the I/O Manager to the redirector device object dispatch routine will be the portion that has not been parsed by the I/O Manager or the Object Manager (i.e., everything excluding the string used to identify the target device object and therefore, everything excluding *F:).*

---

* In the next chapter, I will explain the mount process in a little more detail. At that time, I will also mention how the create/open request gets directed to the NT I/O Manager in the first place.

Code Fragment

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

    ASSERT(PtrlrpContext);
    ASSERT(Ptrlrp);

    try {

        AbsolutePathName. Buffer = NULL;
        AbsolutePathName. Length = AbsolutePathName. MaximumLength = 0;

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

        // If the caller cannot block, post the request to be handled
        // asynchronous ly
        if (! (PtrIrpContext->IrpContextFlags
                 & SFSD_IRP_CONTEXT_CAN_BLOCK) ) {
                // We must defer processing this request, since we could
                // block anytime while performing the create/open ...
            RC = SFsdPostRequest (PtrlrpContext, Ptrlrp);
         DeferredProcessing = TRUE;
            try_return(RC) ;
        }

        // Now, we can obtain the parameters specified by the user.
        // Note that the file object is the new object created by the
        // I/O Manager, in anticipation that this create/open request
        // will succeed.
        PtrNewFileObjec t    = PtrIoStackLocation->FileObject;
        TargetObjectNam e    = PtrNewFileObject->FileName;
        PtrRelatedFileObject = PtrNewFileObject->RelatedFileObject;

        // If a related file object is present, get the pointers
        //to the CCB and the FCB for the related file object
        if (PtrRelatedFileObject) {
            PtrRelatedCCB = (PtrSFsdCCB) (PtrRelatedFileObject->FsContext2) ;
            ASSERT (PtrRelatedCCB) ;
            ASSERT (PtrRelatedCCB->NodeIdentifier.NodeType
                       == SFSD_NODE_TYPE_CCB) ;
            // each CCB in turn points to a FCB
            PtrRelatedFCB = PtrRelatedCCB->PtrFCB;
            ASSERT (PtrRelatedFCB) ;
            ASSERT ( (PtrRelatedFCB->NodeIdentif ier . NodeType
                        == SFSD_NODE_TYPE_FCB)
                      II
                        (PtrRelatedFCB->NodeIdentif ier .NodeType
                            ==SFSD_NODE_TYPE_VCB) ) ;
    RelatedObjectName = PtrRelatedFileObject->FileName;
}

// Allocation size is only used if a new file is created
// or a file is superseded.

AllocationSize = Ptrlrp->0verlay. AllocationSize. LowPart;

// Note: Some FSD implementations support file sizes > 2GB.
// The following check is only valid if your FSD does not support
//a large file size. With NT version 5.0, 64-bit support will
// become available and your FSD ideally should support large files
if (PtrIrp->Overlay. AllocationSize. HighPart) {
    RC = STATUS_INVALID_PARAMETER;
    try_return(RC) ;
}

// Get a pointer to the supplied security context
PtrSecurityContext =
    PtrIoStackLocation->Parameters . Create . SecurityContext ;

// The desired access can be obtained from the SecurityContext
DesiredAccess = PtrSecurityContext->DesiredAccess;

// Two values are supplied in the Create .Options field:
// (a) the actual user-supplied options
// (b) the create disposition
RequestedOptions =
    (PtrIoStackLocation->Parameters .Create. Options &
                        FILE_VALID_OPTION_FLAGS) ;

// The file disposition is packed with the user options ...
// Disposition includes FILE_SUPERSEDE, FILE_OPEN_IF, etc.
RequestedDisposition =
    ( (PtrIoStackLocation->Parameters. Create. Options » 24)
         && OxFF) ;

FileAttributes    =
    (uint8) (PtrIoStackLocation->Parameters .Create .FileAttributes
                    & FILE_ATTRIBUTE_VALID_FLAGS) ;
ShareAccess   = PtrIoStackLocation->Parameters . Create . ShareAccess ;

// If your FSD does not support EA manipulation, you might return
// invalid parameter if the following are supplied.
// EA arguments are only used if a new file is created or a file is
// superseded
PtrExtAttrBuf fer   = PtrIrp->AssociatedIrp. SystemBuf fer;
ExtAttrLength = PtrIoStackLocation->Parameters .Create. EaLength;

// Get the options supplied by the user

// User specifies that returned object MUST be a directory.
// Lack of presence of this flag does not mean it *cannot* be a
// directory *unless* FileOnlyRequested is set (see below)
```

```
II Presence of the flag however, does require that the returned
// object be a directory (container) object.
DirectoryOnlyRequested =
    ((RequestedOptions & FILE_DIRECTORY_FILE) ? TRUE : FALSE);

// User specifies that returned object MUST NOT be a directory.
// Lack of presence of the flag below does not mean it cannot be a
// file unless DirectoryOnlyRequested is set (see above).

// Presence of the flag, however, does require that the returned
// object be a simple file (noncontainer) object.
FileOnlyRequested =
    ((RequestedOptions & FILE_NON_DIRECTORY_FILE) ? TRUE : FALSE);

// We cannot cache the file if the following flag is set.
// However, things do get a little bit interesting if caching
// has been already initiated due to a previous open ...
// (maintaining consistency then becomes a little bit more
// of a headache - see read/write file descriptions)
NoBufferingSpecified =
    ((RequestedOptions & FILE_NO_INTERMEDIATE_BUFFERING)
         ? TRUE : FALSE);

// Write-through simply means that the FSD must not return from
// a user write request until the data has been flushed to
// secondary storage (either to disks directly connected to the
// node, or across the network in the case of a redirector)
WriteThroughRequested =
    ((RequestedOptions & FILE_WRITE_THROUGH) ? TRUE : FALSE);

// Not all of the Windows NT file system implementations support
// the delete-on-close option. The presence of this flag implies
// that after the last close on the FCB has been performed, your
// FSD should delete the file. Specifying this flag saves the
// caller from issuing a separate delete request. Also, some FSD
// implementations might choose to implement a Windows NT
// idiosyncratic behavior where you could create such "delete-on-
// close"-marked files under directories marked for deletion.
// Ordinarily, an FSD will not allow you to createa new file under
// a directory that has been marked for deletion.
DeleteOnCloseSpecified =
    ((RequestedOptions & FILE_DELETE_ON_CLOSE) ? TRUE : FALSE);

NoExtAttrKnowledge =
    ((RequestedOptions & FILE_NO_EA_KNOWLEDGE) ? TRUE : FALSE);

// The following flag is only used by the LAN Manager redirector
// to initiate a "new mapping" to a remote share.
// Third-party FSD implementations will not see this flag.
CreateTreeConnection =
    ((RequestedOptions & FILE_CREATE_TREE_CONNECTION)
        ? TRUE : FALSE);
```

```
II The NTFS file system, for example, supports the OpenByFileld
// option. Your FSD may also be able to associate a unique
// numerical ID with an on-disk object. Any thread can then obtain
// this ID via a "query file information" call to your FSD.
// Later, the caller might decide to reopen the object; this time,
// though, it may supply your FSD with the file identifier instead
// of a file/pathname.
OpenByFileld =
    ((RequestedOptions & FILE_OPEN_BY_FILE_ID) ? TRUE : FALSE);

// Are we dealing with a page file? Page files are not very
// different from any other kind of on-disk file stream though you
// should allocate the FCB, CCB, and other structures for a page
// file from nonpaged pool.
PageFileManipulation =
    ((PtrIoStackLocation->Flags & SL_OPEN_PAGING_FILE)
         ? TRUE : FALSE);

// The open target directory flag is used as part of the sequence
// of operations performed by the I/O Manager is response to a
// file/dir rename operation. See the explanation in the book for
// details.
OpenTargetDirectory =
    ((PtrIoStackLocation->Flags & SL_OPEN_TARGET_DIRECTORY) ?
                            TRUE : FALSE);

// If your FSD supports case-sensitive file name checks, you may
// choose to honor the following flag. It is not mandatory for your
// FSD to support case-sensitive name matching (e.g., FAT/CDFS do
// not support case-sensitive name comparisons.
IgnoreCaseWhenChecking =
    ((PtrIoStackLocation->Flags & SL_CASE_SENSITIVE)
         ? TRUE : FALSE);

// Ensure that the operation has been directed to a valid VCB ...
PtrVCB = (PtrSFsdVCB)(PtrIrpContext->TargetDeviceObject->
                                            DeviceExtension);
ASSERT(PtrVCB);
ASSERT(PtrVCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_VCB);

// Use coarse-grained locking and acquire the VCB exclusively. This
// will lock out all other concurrent create/open requests.
ExAcquireResourceExclusiveLite(&(PtrVCB->VCBResource), TRUE);
AcquiredVCB = TRUE;

// Disk-based file systems might decide to verify the logical
// volume (if required and only if removable media are supported)
//at this time.

// Implement your own volume verification routine ...
// Read the DDK for more information on when a FSD must verify a
// volume (this is typically done when a lower-level disk driver
// for removable drives reports that the media in the drive might
// possibly have been changed; i.e., user ejected and inserted some
II media) . Chapter 11 also describes the volume verification
// process in considerable detail.

// If the volume has been locked, fail the request. Users may have
// locked the volume to issue a dismount request
if (PtrVCB->VCBFlags & SFSD_VCB_FLAGS_VOLUME_LOCKED) {
    RC = STATUS_ACCESS_DENIED;
    try_return(RC) ;
}

// If a "volume open" is requested, satisfy it now.
if ( (PtrNewFileObject->FileName. Length == 0) &&
      ( (PtrRelatedFileObject == NULL) ||
       (PtrRelatedFCB->NodeIdentifier .NodeType
           == SFSD_NODE_TYPE_VCB) ) ) {
    //If the supplied file name is NULL and either there exists
    // no related file object or a related file object was supplied
    // but it refers to a previously opened instance of a logical
    // volume, this open must be for a logical volume.

    // Note: your FSD might decide to do special things (whatever
    // they might be) in response to an open request for the
    // logical volume.

    // Logical volume open requests are done primarily to get/set
    // volume information, lock the volume, dismount the volume
    // (using the IOCTL FSCTL_DISMOUNT_VOLUME) , etc.

    // If a volume open is requested, perform checks to ensure that
    // invalid options have not also been specified . . .
    if ( (OpenTargetDirectory) || (PtrExtAttrBuffer) ) {
        RC = STATUS_INVALID_PARAMETER;
        try_return(RC) ;
    }

    if (DirectoryOnlyRequested) {
        //a volume is not a directory
        RC = STATUS_NOT_A_DIRECTORY;
        try_return(RC) ;
    }

    if { (RequestedDisposition != FILE_OPEN) &&
          (RequestedDisposition != FILE_OPEN_IF) ) {
        // cannot create a new volume, I'm afraid ...
        RC = STATUS_ACCESS_DENIED;
        try_return(RC)  ;
    }

    RC = SFsdOpenVolume(PtrVCB, PtrlrpContext, Ptrlrp,
                ShareAccess, PtrSecurityContext, PtrNewFileObject);
    Returnedlnformation = PtrIrp->IoStatus . Information;

    try_return(RC)  ;
}
```

```
II Your FSD might implement the open-by-id option. The "id"
// is an FSD-defined unique numerical representation of the on-
// disk object. The caller can subsequently give you this file id
// and your FSD should be completely capable of opening the object.
if (OpenByFileld) {
    // perform the open . . .
    // RC = SFsdOpenByFileId(PtrIrpContext, Ptrlrp ....);
    // try_return(RC);
}

// Now determine the starting point from which to begin the parsing
if (PtrRelatedFileObject) {
    //We have a user-supplied related file object.
    // This implies a relative open; i.e., relative to the
    // directory represented by the related file object ...

    // Note: The only purpose FSD implementations ever have for
    // the related file object is to determine whether this
    // is a relative open or not. At all other times (including
    // during I/O operations), this field is meaningless from
    // the FSD's perspective.
    if ( ! (PtrRelatedFCB->FCBFlags & SFSD_FCB_DIRECTORY) ) {
        // we must have a directory as the "related" object
        RC = STATUS_INVALID_PARAMETER;
        try_return(RC);
    }

    // So we have a directory, ensure that the name begins with
    // a \ (i.e., begins at the root and does *not* begin with a
    // \\) .
    // NOTE: This is just an example of the kind of pathname string
    // validation that an FSD must do. Although the remainder of
    // the code may not include such checks, any commercial
    // FSD *must* include such checking (no one else, including
    // the I/O Manager will perform checks on your FSD's behalf).
    if ( (RelatedObjectName.Length ==0) ||
          (RelatedObjectName.Buffer[0] !=L'\\')) {
        RC = STATUS_INVALID_PARAMETER;
        try_return(RC) ;
    }

    // Similarly, if the target file name starts with a \, it
    // is wrong, since the target file name can no longer be
    // absolute if a related file object is present.
    if ( (TargetObjectName.Length != 0) &&
          (TargetObjectName.Buffer[0] == L'\\')) {
        RC = STATUS_INVALID_PARAMETER;
        try_return(RC) ;
    }

    // Create an absolute pathname. You could potentially use
    // the absolute pathname if you cache previously opened
    // file/directory object names.
    {
```

```
                AbsolutePathName.MaximumLength = TargetObjectName.Length +
                        RelatedObjectName.Length + sizeof(WCHAR);
                if (!(AbsolutePathName.Buffer =
                            ExAllocatePool(PagedPool,
                                AbsolutePathName.MaximumLength))) {
                    RC = STATUS_INSUFFICIENT_RESOURCES;
                    try_return(RC);
                }

                RtlzeroMemory(AbsolutePathName.Buffer,
                                    AbsolutePathName.MaximumLength);

                RtlCopyMemory((void *)(AbsolutePathName.Buffer),
                                    (void *)(RelatedObjectName.Buffer),
                                    RelatedObjectName.Length);
                AbsolutePathName.Length = RelatedObjectName.Length;
                RtlAppendunicodeTostring(&AbsolutePathName, L"\\");
                RtlAppendunicodeTostring(&AbsolutePathName,
                        TargetObjectName.Buffer);
            }

        } else {

            // The supplied pathname must be an absolute pathname.
            if (TargetObjectName.Buffer[0] != L'\\') {
                RC = STATUS_INvALID_PARAMETER;
                try_return(RC);
            }

            {
                AbsolutePathName.MaximumLength = TargetObjectName.Length;
                if (!(AbsolutePathName.Buffer =
                            ExAllocatePool(PagedPool,
                                AbsolutePathName.MaximumLength))) {
                    RC = STATUS_INSUFFICIENT_RESOURCES;
                    try_return(RC);
                }

                RtlzeroMemory(AbsolutePathName.Buffer,
                                    AbsolutePathName.MaximumLength);

                RtlCopyMemory((void *)(AbsolutePathName.Buffer),
                                    (void *)(TargetObjectName.Buffer),
                                    TargetObjectName.Length);
                AbsolutePathName.Length = TargetObjectName.Length;
            }
        }

        // Go into a loop parsing the supplied name
        // Use the algorithm supplied in the book to implement this loop.

        // Note that you may have to open intermediate directory objects
        // while traversing the path. You should try to reuse existing code
        // whenever possible; therefore, you should consider using a common
```

```
// open routine regardless of whether the open is on behalf of the
// caller or an intermediate (internal) open performed by your
// driver.

// But first, check if the caller simply wishes to open the root
//of the file system tree.
if (AbsolutePathName.Length == 2) {
    // This is an open of the root directory, ensure that
    // the caller has not requested a file only
    if (FileOnlyRequested ||
         (RequestedDisposition == FILE_SUPERSEDE)
        || (RequestedDisposition == FILE_OVERWRITE) ||
         (RequestedDisposition == FILE_OVERWRITE_IF)) {
        RC = STATUS_FILE_IS_A_DIRECTORY;
        try_return(RC);
    }

    // Insert code to open root directory here.
    // Include creation of a new CCB structure.

    try_return(RC);
}

if (PtrRelatedFileObject) {
    // Insert code such that your "start directory" is
    // the one identified by the related file object
} else {
    // Insert code to start at the root of the file system
}

// NOTE: If your FSD does not support access checking (i.e.,
// your FSD does not check traversal privileges), you could
// easily maintain a prefix cache containing pathnames and
// open FCB pointers. Then, if the requested pathname is already
// present in the cache, you can avoid the tedious traversal
// of the entire pathname performed below and described in the
// book.

// If you do not maintain such a prefix table cache of previously
// opened object names, or if you do not find the name to be opened
// in the cache, then get the next component in the name to be
// parsed. Note that obtaining the next string component is
// similar to the strtok library routine where the separator is a
// \.

// Your FSD should also always check the validity of the token
//to ensure that only valid characters comprise the path/file
//name.

// Insert code to open the starting directory here.

while (TRUE) {
    // Insert code to perform the following tasks here:
```

```
    II    (a)  acquire the parent directory FCB MainResource
    //         exclusively.
    //    (b)  ensure that the parent directory in which you will
    //         perform a lookup operation is indeed a directory.
    //    (c)  if there are no more components left after this one
    //         in the pathname supplied by the user, break.
    //    (d)  attempt to lookup the subdirectory in the parent.
    // (e) if not found, return STATUS_OB JECT_PATH_NOT_FOUND .
    //    (f)  otherwise, open the new subdirectory and make it
    //         the new parent .
    //    (g)  close the current parent directory (after releasing
    //         resources that were acquired in step (a) above.
    //    (h)  go back and repeat the loop for the next component in
    //         the path.

    //    NOTE: If your FSD supports it, you should always check
    // that the caller has appropriate privileges to traverse
    // the directories being searched.
}

// Now we are down to the last component, check it out to see if it
// exists …
// Even for the "open target directory" case below, it is important
// to know whether the final component specified exists.

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

    // Now, do the following:
    // (a) Replace the string in the FileName field in the
    //     PtrNewFileObject to identify the target name
    //     only (i.e., the final component string without the path
    //     leading to the object) .
    // (b) Return with the target's parent directory opened.
    // (c) Update the file object FsContext and FsContext2 fields
    //     to reflect the fact that the parent directory of the
    //     target has been opened.

    try_return(RC) ;
}

// We make the check here to see if the file stream already exists.
// Assume that RC will contain the status (success/failure) for our
// check.
```

```
if ( !NT_SUCCESS(RC) ) {
    // Object was not found, create if requested
    if ( (RequestedDisposition == FILE_CREATE) |
         (RequestedDisposition == FILE_OPEN_IF) ||
         (RequestedDisposition == FILE_OVERWRITE_IF) ) {
        // Create a new file/directory here.

        // Open the newly created object.

        // Note that a FCB structure will be allocated at this time
        // and so will a CCB structure. Assume that these are
        // called PtrNewFCB and PtrNewCCB respectively.
        // Further, note that since the file is being created, no
        // other thread can have the file stream open at this time.

        // Set the allocation size for the object is specified.

        // Set extended attributes for the file.

        // Set the Share Access for the file stream.
        // The FCBShareAccess field will be set by the I/O Manager.
        IoSetShareAccess(DesiredAccess, ShareAccess,
                         PtrNewFileObject,
                         &(PtrNewFCB->FCBShareAccess));

        RC = STATUS_SUCCESS;
        ReturnedInformation = FILE_CREATED;
    }

    try_return(RC);

} else {

    // File stream does exist. Now we must perform some additional
    // error checking.

    if (RequestedDisposition == FILE_CREATE) {
        ReturnedInformation = FILE_EXISTS;
        RC = STATUS_OBJECT_NAME_COLLISION;
        try_return(RC);
    }

    // Insert code to open the target here, return if failed.

    // The FSD will allocate a new FCB structure if no such
    // structure currently exists in memory for the file stream.
    // A new CCB will always be allocated.
    // Assume that these structures are named PtrNewFCB and
    // PtrNewCCB respectively.
    // Further, you should obtain the FCB MainResource exclusively
    //at this time.

    // Once you have opened the file stream and created an FCB,
    // you should perform some additional checks to verify whether
```

```
II the user open request should be succeeded.

// Check if caller wanted a directory only and target object
// not a directory, or caller wanted a file only and target
// object not a file.
if (FileOnlyRequested && (PtrNewFCB->FCBFlags
                            & SFSD_FCB_DIRECTORY)) {
    // Close the new FCB and leave
    // SFsdCloseCCB(PtrNewCCB);
    RC = STATUS_FILE_IS_A_DIRECTORY;
    try_return(RC);
}

// Check whether caller-specified flags are incompatible
// with the type of object being returned.
if ( (PtrNewFCB->FCBFlags & SFSD_FCB_DIRECTORY ) &&
     ( (RequestedDisposition == FILE_SUPERSEDE) ||
       (ReguestedDisposition == FILE_OVERWRITE) j j
       (ReguestedDisposition == FILE_OVERWRITE_IF) ) ) {
    // SFsdCloseCCB(PtrNewCCB) ;
    RC = STATUS_FILE_IS_A_DIRECTORY;
    try_return(RC) ;
}

if (DirectoryOnlyRequested &&
        ! (PtrNewFCB->FCBFlags & SFSD_FCB_DIRECTORY) ) {
    // Close the new FCB and leave
    // SFsdCloseCCB(PtrNewCCB) ;
    RC = STATUS_NOT_A_DIRECTORY;
    try_return(RC) ;
}

// Check share access and fail if the share conflicts with an
// existing open.
if (PtrNewFCB->OpenHandleCount > 0) {
    // The FCB is currently in use by some thread.
    // We must check whether the requested access/share access
    // conflicts with the existing open operations.

    if (!NT_SUCCESS(RC = locheckshareAccess(DesiredAccess,
                                ShareAccess,
                                PtrNewFileObj ect,
                                &(PtrNewFCB->FCBShareAccess),
                                TRUE))) {
        // SFsdCloseCCB(PtrNewCCB);
        try_return(RC);
    }
} else {
        // Store the fact that an open is being satisfied with
        // the specified share access.
        losetshareAccess(DesiredAccess, ShareAccess,
                               PtrNewFileObj ect,
                               &(PtrNewFCB->FCBShareAccess)) ;
}
```

```
            ReturnedInformation = FILE_OPENED;

            // If a supersede or overwrite was requested, do it now.
            // Your FSD may need to determine whether any byte-range
            // locks exist on the file stream. For overwrite requests (as
            // opposed to requests to supersede the file stream), your FSD
            // may wish to deny the request if a conflicting byte-range
            // lock has been obtained by another process.
            if (RequestedDisposition == FILE_SUPERSEDE) {
                // Attempt the operation here ...
                // RC = SFsdSupersede(...);
                if (NT_SUCCESS(RC)) {
                    ReturnedInformation = FILE_SUPERSEDED;
                }
            } else if ((RequestedDisposition == FILE_OVERWRITE) ||
                           (RequestedDisposition == FILE_OVERWRITE_IF)){
                // Attempt the operation here ...
                // RC = SFsdOverwrite(...);
                if (NT_SUCCESS(RC)) {
                    ReturnedInformation = FILE_OVERWRITTEN;
                }
            }
        }

        try_exit:    NOTHING;

    } finally {
        // Complete the request unless we are here as part of unwinding
        // when an exception condition was encountered, OR
        // if the request has been deferred (i.e., posted for later
        // handling)
        if {RC != STATUS_PENDING) {
            // If we acquired any FCB resources, release them now.

            // If any intermediate (directory) open operations were
            // performed, implement the corresponding close (do not
            // however close the target you have opened on behalf of the
            // caller) .

            if (NT_SUCCESS(RC) ) {
                // Update the file object such that:
                // (a) the FsContext field points to the NTRequiredFCB
                //       field in the FCB
                // (b) the FsContext2 field points to the CCB created as a
                //       result of the open operation

                //If write-through was requested, then mark the file
                // object appropriately.
                if (WriteThroughRequested) {
               PtrNewFileObject->Flags |= FO_WRITE_THROUGH;
                }

                // Release the PtrNewFCB MainResource at this time.
            } else {
                II Perform failure-related postprocessing now.
            }

            //As long as this unwinding is not being performed as a
            // result of an exception condition, complete the IRP.
            if (!(PtrIrpContext->IrpContextFlags
                     & SFSD_IRP_CONTEXT_EXCEPTION)) {
                PtrIrp->IoStatus.Status = RC;
                PtrIrp->IoStatus.Information = Returnedlnformation;

                // Free up the IRP Context.
                SFsdReleaselrpContext(PtrlrpContext);

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

        if (AcquiredVCB) {
            ASSERT(PtrVCB);
         SFsdReleaseResource(&(PtrVCB->VCBResource));
            AcquiredVCB = FALSE;
        }

        if (AbsolutePathName.Buffer != NULL) {
            ExFreePool(AbsolutePathName.Buffer);
        }
    }

    return(RC);
}
```

Notes

The FSD implementation can receive a create/open request for one of the following objects:

The FSD device object itself

This open will typically be received by the FSD if a process sends an IOCTL to the FSD to affect the behavior of the driver. This request can be implemented by your driver in any manner appropriate to your driver implementation. The sample FSD simply succeeds such a request immediately (see the accompanying diskette).

A mounted logical volume

If a process wants to request that a volume be dismounted, the process must first open the logical volume itself, as opposed to opening an object contained on the mounted logical volume. Similarly, processes may wish to perform query and/or set label operations on the logical volume, in which case they might request an open of the logical volume. Finally, some threads might wish to read the volume information directly off media for which they would need to open the volume device object directly.

A file or directory object on the logical volume

These are the more commonly received create/open requests. A user process may wish to open either a file object or a directory object, both of which are supported by all FSD implementations. Open operations on directories are performed to query the contents of the directory. Normal file open operations are performed to be able to access/modify file stream data or control information.

The preceding code fragment above is mostly self-explanatory. Read the comments to understand the code better. The first thing you will notice is that a lot of the routines have been commented out. These are placeholders for you to replace with appropriate functionality suitable for your FSD implementation.

Basically, the implementation follows the logical steps, listed earlier, that an FSD should perform upon receiving a create/open request. The objective is to try to find the target object, given a path leading to that object. If the object exists and the user wishes to open it, the FSD will create a file control block (if none currently exists), create a context control block, and initialize the I/O Manager file object appropriately. If the object does not exist and the user wants to create it, the FSD will first create the object on secondary storage (on the logical volume) and then open the object for the caller, creating an FCB, a CCB, and initializing the file object structure. If a new object is created, the FSD may also set an allocation size for the created object if the caller has specified such a size.

The code fragment describes a special type of request from the NT I/O Manager indicated by the presence of the SL_OPEN_TARGET_DIRECTORY flag in the current I/O stack location. This flag is slightly unusual and quite specific to the Windows NT environment. Basically, when the I/O Manager receives a request to move or rename a file or directory, it sends this request to the FSD, supplying the target name in the rename/move request. For example, if a user requests that file *\dirl\dir2\dir3\foo* be renamed/moved to *\dirl\dir4\bar,* the I/O Manager will send the latter string to the FSD with the SL_OPEN_TARGET_DIRECTORY flag set. The FSD must respond as follows when this flag is set:

- The FSD must first check to see if the target object (i.e., *bar)* exists on the path leading to *bar.*

  The presence or absence of the target should be conveyed back to the I/O Manager.

- The FSD must replace the pathname in the file object with the last component; i.e., replace *\dirl\dir4\barin* our example with the string *bar.*

This replace operation is required due to the manner in which the I/O Manager subsequently invokes the FSD SFsdFileInfo() dispatch entry point. In Chapter 10, *Writing A File System Driver II,* this routine is described in further detail.

- Instead of opening/creating the actual target *(bar* in our example), the FSD must open the parent directory (in our example, the FSD must return with *dir4* having been opened for the caller).

Note that although an FCB structure is initialized as part of the processing performed for a successful create/open operation, the FSD does not invoke CcInitializeCacheMap() at this time for the file object and the FCB representing the open file. The reason for this is fairly simple; often commonly used applications open file objects simply to perform a query-file-information operation on them and subsequently close the file stream without ever attempting any I/O. Therefore, requesting the Cache Manager to perform any initialization in anticipation of buffered I/O would simply degrade performance in such situations. Therefore, it is recommended in Windows NT that the FSD implementation defer any Cache Manager-related initialization until the time when a read or write operation is actually attempted for the first time.

Although it is not discussed in the code fragment, it is possible for an FSD to replace the name supplied with the file object created by the I/O Manager and return STATUS_REPARSE to the I/O Manager. Be careful, though, to free the memory allocated by the I/O Manager for the original file name buffer and allocate new memory from paged pool. Also, if applicable, you may wish to set the RelatedFileObject pointer to NULL in this situation.

If your FSD does not support page file create requests, you can return an error when such a request is received by your driver. Page file create requests will only be initiated by the NT VMM internal routine called (NtCreatePagingFile(), which invokes the I/O Manager internal routine (not exported) called IoCreateFileO with an attribute that specifies that the file to be created is a page file. Page files are not really different from any other ordinary file created on a logical volume. However, most NT FSD implementations return STATUS_ACCESS_DENIED or STATUS_SHARING_VIOLATION if a thread tries to open an already-opened page file. Also, you should ensure that all in-memory representations of a page file are allocated from nonpaged pool; this includes the FCB and the CCB structure for the file. The rationale is simply to allow your FSD to safely access these in-memory structures without incurring a page fault when a paging I/O is received by your driver, because page faults at that time will crash the system.

Finally, you will have noticed that the code fragment checks the access requested and whether the caller is allowed to open the file for the desired access. An I/O

Manager routine IoCheckShareAccess ( ) is used to determine whether the desired access and the specified share access conflict with any previous open for the file stream. If no such conflict is present, the I/O Manager updates the FCBShareAccess field (this is a result of the last argument, called UpdateShareAccess, to the routine being set to TRUE). Of course, if this is the first open operation on the file stream (or if all previous open handles have been closed), then the FSD directly invokes the `IoSetShareAccess` () function to set the share access in the FCBShareAccess field in the FCB structure. In the next chapter, you will see that the share access stored in the FCB will be removed when the last file handle corresponding to the file object is closed (i.e., when the IRP_MJ_CLEANUP request is received by the FSD).

Dispatch Routine: Read

The read dispatch entry point is invoked in response to user requests to access file data. Most FSD implementations allow users to access data for ordinary file streams only, and any attempt to directly access directory contents will typically be rejected with a STATUS_ACCESS_DENIED error.

All NT FSD implementations support two kinds of read I/O requests:

- Buffered read operations
- Nonbuffered read operations satisfied directly from secondary storage

By default, an FSD will attempt to satisfy the read request using buffered (cached) data. All of the native Windows NT FSD implementations use the services of the NT Cache Manager in caching file data in memory. You can, however, choose some other caching module with your FSD implementation, though the NT Cache Manager does a fairly good job and you should at least seriously consider using it instead.

In order for the caller to request that the read be satisfied directly from secondary storage, the file object used in the read operation should have been opened with FILE_NO_INTERMEDIATE_BUFFERING set. The only other read operations that are directly satisfied from secondary storage by the FSD are those marked as paging I/O. These operations come to the FSD from the NT VMM and cannot be satisfied by a recursive call back to the NT Cache Manager, but should be sent to the underlying disk driver for further processing.

Logical Steps Involved

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

```
typedef struct _IO_STACK_LOCATION {

    // ....

    union {

        //....

        // System service parameters for:  NtReadFile
        struct {
            ULONG Length;
            ULONG Key;
            LARGE_INTEGER ByteOffset;
        } Read;

        // ....
} Parameters;

// …

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

An FSD performs the following simple tasks upon receiving a read request on a file object:

1. Get a pointer to the CCB and FCB for the file stream.
2. Verify that the read operation is allowed.

   Typically, FSD implementations on any operating system platform will reject user read requests directed to directory objects.

3. Identify the type of read operation: a paging I/O operation, a normal non-cached operation, a non-MDL cached read operation, or an MDL read operation.

   This is probably the most important task that your FSD will perform in the read operation. You should be able to identify whether the read is a recursive operation, or whether the read request comes to your FSD directly from a user thread, from the VMM, or from the NT Cache Manager. Furthermore, your FSD should be able to identify whether the read is as a result of read-ahead being performed by the Cache Manager. For more discussion on this topic, read through the next chapter.

4. Obtain any resources that are appropriate to ensure consistency of data.

   Most FSD implementations, including the sample code provided here, acquire the MainResource shared if the read request has been received directly from a user thread. This prevents other write operations from proceeding concurrently, but does allow concurrent read operations. If, however, the read request is due to a page fault, the FSD will acquire the PagingIoResource shared instead.

5. Obtain the starting offset, length, and buffer pointer supplied by the caller.

   The starting offset and length uniquely determine the byte range requested by the caller. The caller provides a buffer for the FSD to return data. In Chapter 4, I explained the various buffering mechanisms that can be used by callers of kernel-mode drivers. Most NT FSD implementations and the sample code provided here choose METHOD_NEITHER as the buffering option for the device objects created to represent mounted logical volumes. The result is that the I/O Manager does not manipulate the caller buffer, but sends it down as-is to the FSD.

6. Lock the user's buffer if required, and also create a Memory Descriptor List (MDL) for requests that must be directed to lower-level drivers.

7. Check if the byte range requested by the caller has been locked; it is possible with Windows NT, however, for the caller to provide a key that would still allow the read request to proceed.

   If the byte range desired by the user has a byte-range lock that does not permit read access by other processes, the FSD will return an error to the caller.

8. Determine whether the byte range specified by the caller is valid, and if not, return an appropriate error code to the caller.

9. If this is a buffered I/O request and caching has not yet been initiated on the FCB, invoke `CcInitializeCacheMap()` to initiate caching at this time.

10. If this is a buffered non-MDL I/O request, forward the request on to the NT Cache Manager via an invocation to `CcCopyRead()`, or if this is a nonbuffered (direct I/O or paging I/O request), forward it on to the lower-level driver for further processing.

    If this is an MDL read request, use the `CcMdlRead()` function, provided by the Cache Manager, to return an MDL containing file data to the caller.

11. Once data has been obtained either from the Cache Manager or from lower-level drivers, release FCB resources acquired and return the results to the caller.

Code Fragment

```
NTSTATUS     SFsdCommonRead(
PtrSFsdIrpContext             PtrIrpContext,
PIRP                          PtrIrp)
{
    // Declarations go here ...

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

        // If this happens to be an MDL read complete request, then
        // there is not much processing that the FSD has to do.
        if (PtrIoStackLocation->MinorFunction & IRP_MN_COMPLETE) {
            // Caller wants to tell the Cache Manager that a previously
            // allocated MDL can be freed.
            SFsdMdlComplete(PtrlrpContext, Ptrlrp,
                                    PtrloStackLocation, TRUE);
            // The IRP has been completed.
            Completelrp = FALSE;
            try_return(RC = STATUS_SUCCESS);
        }

        // If this is a request at IRQL DISPATCH_LEVEL, then post
        // the request (your FSD may process it synchronously
        // if you implement the support correctly).
        if (PtrIoStackLocation->MinorFunction & IRP_MN_DPC) {
            Completelrp = FALSE;
            PostRequest = TRUE;
            try_return(RC = STATUS_PENDING);
        }

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

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

        // Get some of the parameters supplied to us.
        ByteOffset = PtrIoStackLocation->Parameters.Read.ByteOffset;
        ReadLength = PtrIoStackLocation->Parameters.Read.Length;

        CanWait =
            ((PtrIrpContext->IrpContextFlags & SFSD_IRP_CONTEXT_CAN_BLOCK)
                ? TRUE : FALSE);
        Paginglo = ((PtrIrp->Flags & IRP_PAGING_IO) ? TRUE : FALSE);
        NonBufferedlo = ((PtrIrp->Flags & IRP_NOCACHE) ? TRUE : FALSE);
        SynchronousIo =
            ((PtrFileObject->Flags & FO_SYNCHRONOUS_IO) ? TRUE : FALSE);

        // A 0 byte read can be immediately succeeded.
        if (ReadLength == 0) {
            try_return(RC);
        }

        // NOTE: if your FSD does not support file sizes > 2GB, you
        // could validate the start offset here and return end-of-file
        // if the offset begins beyond the maximum supported length.

        // Is this a read of the volume itself?
```

```
if (PtrFCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_VCB) {
    // Yup, we need to send this on to the disk driver after
    // validation of the offset and length.
    PtrVCB = (PtrSFsdVCB)(PtrFCB);

    // Acquire the volume resource shared.
    if (!ExAcguireResourceSharedLite(&(PtrVCB->VCBResource),
                                          CanWait)) {
        // Post the request to be processed in the context of a
        // worker thread.
        Completelrp = FALSE;
        PostRequest = TRUE;
        try_return(RC = STATUS_PENDING);
    }
    PtrResourceAcquired = &(PtrVCB->VCBResource);

    // Insert code to validate the caller-supplied offset here.

    // Lock the caller's buffer.
    if (!NT_SUCCESS(RC = SFsdLockCallersBuffer(Ptrlrp,
                                    TRUE, ReadLength))) {
        try_return(RC);
    }

    // Forward the request to the lower-level driver.

    // For synchronous I/O wait here, else return STATUS_PENDING.
    // For asynchronous I/O support, read the discussion in
    // Chapter 10.

    try_return(RC);
}

//If the read request is directed to a page file (if your FSD
// supports paging files), send the request directly to the disk
// driver. For requests directed to a page file, you have to trust
// that the offsets will be set correctly by the VMM. You should
// not attempt to acquire any FSD resources either.
if (PtrFCB->FCBFlags & SFSD_FCB_PAGE_FILE) {
    loMarklrpPending(Ptrlrp);
    // You will need to set a completion routine before invoking
    //a lower- level driver.
    // Forward request directly to disk driver.
    // SFsdPageFileIo(PtrIrpContext, Ptrlrp);

    Completelrp = FALSE;

    try_return(RC = STATUS_PENDING);
}

// If this read is directed to a directory, it is not allowed
// by the sample FSD. Note that you may choose to create a stream
// file for FSD (internal) directory read/write operations, in
// which case you should modify the check below to allow reading
```

```
II (directly from disk) directories as long as the read originated
// from within your FSD. Your driver will have to be smart enough
// to recognize that the read originated in your FSD (e.g., via
// the contents of the TopLevellrp field in TLS described in the
// next chapter) .
if ( PtrFCB->FCBFlags & SFSD_FCB_DIRECTORY) {
    RC = STATUS_INVALID_DEVICE_REQUEST;
    try_return(RC) ;
}

PtrRegdFCB = &(PtrFCB->NTRequiredFCB) ;

// This is a good place for oplock-related processing.
// Chapter 11 expands upon this topic in greater detail.

// Check whether the desired read can be allowed depending
// on any byte-range locks that might exist. Note that for
// paging I/O, no such checks should be performed.
if ( ! Paginglo) {
    // Insert code to perform the check here . . .
    //    if (! SFsdCheckForByteLock (PtrFCB, PtrCCB, Ptrlrp,
    //        PtrCurrentloStackLocation) ) {
    //    try_return(RC = STATUS_FILE_LOCK_CONFLICT) ;
    // }
}

// There are certain complications that arise when the same file
// stream has been opened for cached and noncached access. The FSD
//is then responsible for maintaining a consistent view of the
// data seen by the caller.
// Also, it is possible for file streams to be mapped in both as
// data files and as an executable. This could also lead to
// consistency problems since there now exist two separate
// sections (and pages) containing file information.
// Read Chapter 10 for more information on the issues involved in
// maintaining data consistency.
// Insert appropriate code here.

// Acquire the appropriate FCB resource shared.
if (Paginglo) {
    // Try to acquire the FCB PagingloResource shared.
    if (!ExAcquireResourceSharedLite(&(PtrReqdFCB->
                                        PagingloResource),
                    CanWait)) {
        Completelrp = FALSE;
        PostRequest = TRUE;
        try_return(RC = STATUS_PENDING);
    }
    // Remember the resource that was acquired.
 PtrResourceAcquired = &(PtrReqdFCB->PagingIoResource);
} else {
    // Try to acquire the FCB MainResource shared.
    if (!ExAcquireResourceSharedLite(&(PtrReqdFCB->MainResource),
                    CanWait)) {
```

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

    // Validate start offset and length supplied.
    // If start offset is > end-of-file, return an appropriate
    // error. Note that since an FCB resource has already been
    // acquired, and since all file size changes require acquisition
    //of both FCB resources (see Chapter 10), the contents of the FCB
    // and associated data structures can safely be examined.

    // Also note that I am using the file size in the Common FCB
    // Header to perform the check. However, your FSD might keep a
    // separate copy in the FCB (or some other representation of the
    // file associated with the FCB).
    if (RtlLargeIntegerGreaterThan(ByteOffset,
            PtrReqdFCB->CommonFCBHeader.FileSize)) {
        // Starting offset is > file size.
        try_return(RC = STATUS_END_OF_FILE);
    }

    // We can also truncate the read length here
    // such that it is contained within the file size.

    // This is a good place to set whether fast I/O can be performed
    //on this particular file or not. Your FSD must make its own
    // determination on whether or not to allow fast I/O operations.
    // Commonly, fast I/O is not allowed if any byte-range locks exist
    // on the file or if oplocks prevent fast I/O. Practically any
    // reason chosen by your FSD could result in your setting
    // FastloIsNotPossible
    //OR FastloIsQuestionable instead of FastloIsPossible.
    //
    // PtrReqdFCB->CommonFCBHeader.IsFastIoPossible = FastloIsPossible;


    //  Branch here for cached vs. noncached I/O.
    if (!NonBufferedIo) {

        // The caller wishes to perform cached I/O. Initiate caching if
        // this is the first cached I/O operation using this file
        // object.
        if (PtrFileObject->PrivateCacheMap == NULL) {
            // This is the first cached I/O operation. You must ensure
            // that the Common FCB Header contains valid sizes at this
            // time.
            CcInitializeCacheMap(PtrFileObject,
                (PCC_FILE_SIZES)(&(PtrReqdFCB->
                                   CommonFCBHeader.AllocationSize)),
                FALSE,      // We will not utilize pin access for
```

```
                              II this file
                &(SFsdGlobalData.CacheMgrcallBacks), // callbacks
                PtrCCB) ;       // The context used in callbacks
        }

        // Check and see if this request requires an MDL returned to
        // the caller.
        if (PtrIoStackLocation->MinorFunction & IRP_MN_MDL) {
            // Caller does want an MDL returned. Note that this mode
            // implies that the caller is prepared to block.
            CcMdlRead(PtrFileObject, &ByteOffset, TruncatedReadLength,
                            &(PtrIrp->MdlAddress)  ,
                            &(PtrIrp->IoStatus));
            NumberBytesRead = PtrIrp->IoStatus.Information;
            RC = PtrIrp->IoStatus.Status;

            try_return(RC);
        }

        // This is a regular run-of-the-mill cached I/O request. Let
        // the Cache Manager worry about it.
        // First though, we need a buffer pointer (address) that is
        // valid.
        PtrSystemBuffer = SFsdGetCallersBuffer(Ptrirp);
        if (!CcCopyRead(PtrFileObject, &(ByteOffset), ReadLength,
                      CanWait, PtrSystemBuffer, &(PtrIrp->IoStatus))) {
            // The caller was not prepared to block and data is not
            // immediately available in the system cache.
            Completelrp = FALSE;
            PostRequest = TRUE;
            // Mark IRP Pending . . .
            try_return(RC = STATUS_PENDING);
        }

        //We have the data
        RC = PtrIrp->IoStatus.Status;
        NumberBytesRead = PtrIrp->IoStatus.Information;

        try_return(RC);

    } else {

        // Send the request to lower-level drivers.

        // For paging I/O, the FSD has to trust the VMM to do the right
        // thing.

        // First, mark the IRP as pending, then invoke the lower-level
        // driver after setting a completion routine.
        // Meanwhile, this particular thread can immediately return a
        // STATUS_PENDING return code.
        // The completion routine is then responsible for completing
        // the IRP and unlocking appropriate resources.
```

```
            // Also, at this point, your FSD might use the
            // information contained in the ValidDataLength field to simply
            // return zeroes to the caller for reads extending beyond
            // current valid data length.

            IoMarkIrpPending(PtrIrp) ;

            // Invoke a routine to read disk information at this time.
            // You will need to set a completion routine before invoking
            // a lower-level driver.

            CompleteIrp = FALSE;

            try_return(RC = STATUS_PENDING) ;
        }

        try_exit :   NOTHING ;

    } finally {
        // Post IRP if required.
        if (PostRequest) {
            // Implement a routine that will queue-up the request to be
            // executed later (asynchronously) in the context of a system
            // worker thread. See Chapter 10 for details.

            if (PtrResourceAcquired) {
                SFsdReleaseResource(PtrResourceAcquired);
            }
        } else if (CompleteIrp && ! (RC == STATUS_PENDING ) ) {
            // For synchronous I/O, the FSD must maintain the current byte
            // offset.
            // Do not do this however, if I/O is marked as paging I/O.
            if (SynchronousIo && !PagingIo && NT_SUCCESS (RC) ) {
                PtrFileObject->CurrentByteOffset =
                    RtlLargeIntegerAdd(ByteOffset,
                    RtlConvertUlongToLargeInteger( (unsigned
                                                     long)NumberBytesRead) ;
            }

            // If the read completed successfully and this was not a
            // paging I/O operation,* you should modify the time stamp for
            // the file stream indicating that an access operation was
            // performed. You can do this in one of two ways:
            // (a) You could set a flag in the CCB indicating that the file
            //     stream was accessed and in the cleanup routine
            //     (described in the next chapter) , you would update the
            //     time value.
```

---

* Paging I/O requests are either asynchronous requests initiated by the VMM or the NT Cache Manager, or recursive requests from the FSD to the Cache Manager, back to the FSD. Therefore, most FSD implementations do not update the file access/modification time upon processing such requests. Paging I/O requests can also occur due to page faults on a user-mapped file stream. Unfortunately, by choosing not to update the access time for all paging I/O requests, your FSD will be unable to mark the fact that some user application accessed the file albeit via the memory-mapped file method.

```
            // (b) Or, you could simply get the current time and insert it
            //     into FCB structure now. Then at file cleanup time, you
            //     would update the directory entry for the file.
            if (NT_SUCCESS(RC) && !PagingIo) {
                // The following is method (a) above. If you wish to be
                // more accurate, then update the time in the FCB now.
                // Also remember in this case to remove the
                // FO_FILE_FAST_IO_READ flag from the the file object.
                SFsdSetFlag(PtrCCB->CCBFlags, SFSD_CCB_ACCESSED);
            }

            if (PtrResourceAcguired) {
                SFSdReleaseResource(PtrResourceAcquired);
            }

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

                // Free up the IRP Context.
                SFsdReleaselrpContext(PtrlrpContext);

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

    return(RC);
}

NTSTATUS SFsdLockCallersBuffer(
PIRP                    Ptrlrp,
BOOLEAN                 IsReadOperation,
uint32                  Length)
{
    NTSTATUS            RC = STATUS_SUCCESS;
    PMDL                PtrMdl = NULL;

    ASSERT(Ptrlrp);

    try {
        // Is an MDL already present in the IRP?
        if (!(PtrIrp->MdlAddress)) {
            // Allocate an MDL.
            if (!(PtrMdl = loAllocateMdl(PtrIrp->UserBuffer, Length, FALSE,
                        FALSE, Ptrlrp))) {
                RC = STATUS_INSUFFICIENT_RESOURCES;
                try_return(RC);
            }

            // Probe and lock the pages described by the MDL.
            II We could encounter an exception doing so, swallow the
            // exception.
            // NOTE: The exception could be due to an unexpected (from our
            // perspective), invalidation of the virtual addresses that
            // comprise the passed-in buffer.
            try {
                MmProbeAndLockPages(PtrMdl, PtrIrp->RequestorMode,
                        (IsReadOperation ? IoWriteAccess:IoReadAccess));
            } except(EXCEPTION_EXECUTE_HANDLER) {
                RC = STATUS_INVALID_USER_BUFFER;
            }
        }

        try_exit:    NOTHING;

    } finally {
        if ( !NT_SUCCESS(RC) && PtrMdl) {
            loFreeMdl(PtrMdl);
            // You must NULL the MdlAddress field in the IRP after freeing
            // the MDL, or else the I/O Manager will also attempt to free
            // the MDL pointed to by that field during I/O completion. The
            // pointer becomes invalid once you free the allocated MDL and
            // you will encounter a system crash during IRP completion.
            PtrIrp->MdlAddress = NULL;
        }
    }

    return(RC);
}

void *SFsdGetCallersBuffer(
PIRP                PtrIrp)
{
    void            *ReturnedBuffer = NULL;

    // If an MDL is supplied, use it.
    if (PtrIrp->MdlAddress) {
      ReturnedBuffer = MmGetSystemAddressForMdl(PtrIrp->MdlAddress);
    } else {
        ReturnedBuffer = PtrIrp->UserBuffer;
    }

    return(ReturnedBuffer);
}

void SFsdMdlComplete(
PtrSFsdlrpContext            PtrlrpContext,
PIRP                         Ptrlrp,
PIO_STACK_LOCATION           PtrloStackLocation,
BOOLEAN                      ReadCompletion)
{
    NTSTATUS                 RC = STATUS_SUCCESS;
    PFILE_OBJECT             PtrFileObject = NULL;
    PtrFileObject = PtrIoStackLocation->FileObject;
    ASSERT(PtrFileObject);

    // Not much to do here.
    if (ReadCompletion) {
        CcMdlReadComplete(PtrFileObject, PtrIrp->MdlAddress);
    } else {
        // The Cache Manager needs the byte offset in the I/O stack
        // location.
      CcMdlWriteComplete(PtrFileObject,
            &(PtrIoStackLocation->Parameters.Write.ByteOffset),
            PtrIrp->MdlAddress) ;
    }

    // Clear the MDL address field in the IRP so the loCompleteRequest ( )
    // does not try to play around with the MDL.
    PtrIrp->MdlAddress = NULL;

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

    // Complete the IRP.
    PtrIrp->IoStatus.Status = RC;
    PtrIrp->IoStatus.Information = 0;
    loCompleteRequest(PtrIrp, IO_NO_INCREMENT);

    return;
}
```

Notes

This code fragment follows the list of logical steps, described earlier, that a file system driver typically implements to satisfy a read request. In response to the request, the FSD must first obtain the parameters supplied by the caller. Validation of the starting offset and the read length is typically not required for requests issued by the VMM.

Notice that the sample FSD conditionally acquires either the volume control block resource, the file control block MainResource, or the FCB PagingIoResource, depending upon the nature of the request. The various ways in which the read routine can be invoked are discussed in greater detail in the next chapter.

The buffer supplied by the caller is passed directly to the FSD by the I/O Manager. The FSD, in turn, creates a memory descriptor list (MDL) and locks pages in memory before forwarding the request to a lower-level disk driver. This allows the disk driver to obtain the data in the context of any arbitrary thread, directly into the locked pages. The routine `SFsdLockCallersBuffer()` illustrates the method used in creating a memory descriptor list and locking pages in memory so that lower-level drivers can subsequently access the buffer in the context of any arbitrary thread, even at a high IRQL.

Before a read request is allowed to proceed, the FSD should also ensure that there are not any conflicting byte locks on the range requested by the caller. If conflicting locks do exist, the caller should get an appropriate error code returned to it. Note, however, that byte lock checks are not performed for requests marked as paging I/O, since these requests originate from the VMM in response to a page fault incurred by the thread trying to access the data, it is assumed that the checks must have been performed at some earlier point in time. Unfortunately, though, you will notice that the byte-range lock check will also be skipped for page faults incurred when accessing data mapped into the virtual address space of a process (memory-mapped file).

The caller of the read routine can request either cached or noncached I/O. When a request for buffered I/O is received by the FSD, the driver checks if caching had previously been initiated on the file stream using that particular file object. If this happens to be the first cached I/O request received for that particular file object, the FSD initiates caching by invoking the `CcInitializeCacheMap()` function call. Chapter 7, *The NT Cache Manager II*, describes this routine in greater detail. Once caching has been initiated, the FSD can simply forward the cached I/O request to the NT Cache Manager for further processing. Note that it is quite possible that invoking `CcCopyRead()` might result in a page fault incurred by the Cache Manager, which causes the FSD read routine to be recursed into, this time for paging I/O. The FSD will then handle the page fault by obtaining data directly from disk or from across the network (for redirectors) and complete the paging I/O request.

The preceding code fragment doesn't elaborate on the steps taken by an FSD to forward an I/O request to the lower-level disk driver to get data from secondary storage. The methodology used by your FSD depends upon the specific requirements for your driver. However, some common steps are performed by all FSD implementations before forwarding a request to lower-level drivers:

- Your FSD will determine the logical block offset and number of logical blocks that need to be read.
- The FSD may be able to obtain data in a single I/O operation or, for discontiguous data, your FSD might need to make multiple requests.

  Your FSD may initiate multiple I/O requests concurrently to the disk driver to handle the discontiguous data case.
- If a single I/O request is being sent to the lower-level disk driver, the FSD will initialize the next IRP stack location in the IRP sent to it and will also set

a completion routine before forwarding the IRP down to the next driver in the hierarchy.

It is important for the FSD to set a completion routine so that the correct status can be returned to the caller and also to ensure that all resources acquired during the read operation are released before the request is returned to the caller. Furthermore, the FSD can respond to errors returned by the lower-level disk drivers by initiating appropriate processing from the completion routine.

- If multiple I/O requests are required to read all of the data from secondary storage, the FSD can initiate all I/O requests concurrently or sequentially.

  The FSD can initiate concurrent read operations by creating multiple associated IRP structures, initializing the IRPs appropriately, creating partial MDLs for each of the concurrent requests, setting a completion routine for each associated IRP, and sending the associated IRP requests down to the lower-level drivers.*

For read I/O requests, a caller can specify that an MDL be returned containing the file data. This request for an MDL-read operation can be identified by checking for the IRP_MN_MDL flag value in the MinorFunction field of the current I/O stack location. The code fragment above invokes the CcMdlRead() function, which results in an MDL being allocated by the Cache Manager. Once the caller has completed processing the data contained in the MDL, a second read request is issued to the FSD. This special read request is only issued to inform the Cache Manager that the MDL structure can now be freed (and pages reallocated, if required) and is identified by the IRP_MN_COMPLETE flag value in the MinorFunction field. The FSD must simply invoke the CcMdlReadComplete() Cache Manager function in response to this request as is illustrated in the SFsdMdlComplete() function.

Dispatch Routine: Write

The steps involved in processing a write request are very similar to those performed in processing read requests.

---

* Note that the IoMakeAssociatedIrp() routine can be used to request that the I/O Manager allocate an associated IRP structure for the FSD while the IoBuildPartialMdl() routine will create the partial MDL for the FSD. One side effect of creating associated IRP structures is that the I/O Manager will automatically complete the master IRP once all associated IRPs have been completed. To prevent this from happening, simply increment the AssociatedIrp count in the master IRP before sending requests to the lower-level driver. This trick will cause the I/O manager to believe that there is some associated IRP pending (even after the last one has been completed), and your FSD can subsequently complete the master IRP itself (remember, though, to decrement the AssociatedIrp count before completing the master IRP yourself).

Logical Steps Involved

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

```
typedef struct _IO_STACK_LOCATION {

    // ....

    union {

        //....

        // System service parameters for:  NtWriteFile
        struct {
            ULONG Length;
            ULONG   Key;
            LARGEJNTEGER ByteOffset;
        } Write;

        // ....
} Parameters;

// ....

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

An FSD performs the following simple tasks upon receiving a write request for a file object:

1. Get a pointer to the CCB and the FCB for the file stream.
2. Verify that the write operation is allowed.

   Most FSD implementations will reject user write requests directed to directory objects.
3. Identify the type of write operation: a paging I/O operation, a normal non-cached operation, or a cached write operation. Also determine if the write request was initiated by the lazy-writer thread or by the modified page/block writer thread.

   It is extremely important that your dispatch routine know the caller initiating the write request. In Windows NT, paging I/O asynchronous requests are not synchronized with user file size changes, and therefore, your FSD should be able to always determine whether a write operation should be allowed to proceed or should be disregarded.
4. Obtain any resources that are appropriate to ensure consistency of data.

   The sample FSD implementation provided here acquires the MainResource exclusively if the write request has been received directly from a user thread. This prevents all other user-initiated read or write requests from proceeding concurrently. If, however, the write request has been marked as paging I/O, the FSD will acquire the `PagingloResource` exclusively as well.*

5. Obtain the starting offset, length, and buffer pointer supplied by the caller.

   The starting offset and length uniquely determine the byte range requested by the caller. The caller provides a buffer, as well, for the FSD to transfer data from.

6. Lock the user's buffer, if required, and also create a memory descriptor list (MDL) for requests that must be directed to lower-level drivers.
7. Check that the byte range requested by the caller has been locked; it is possible with Windows NT, however, for the caller to provide a key that would still allow the write request to proceed.
8. Determine whether the byte range specified by the caller is valid.

   In the case of write requests, a starting offset beyond end-of-file for a user-initiated request implies that the user is extending the file size.

9. If this is a buffered I/O request and caching has not yet been initiated on the FCB, invoke `CcInitializeCacheMap()` to initiate caching at this time.
10. If this is a buffered I/O request, forward the request to the NT Cache Manager via an invocation to `CcCopyWrite()`; if this is a nonbuffered (direct I/O or paging I/O) request, forward it on to the lower-level driver for further processing.
11. Once data has been transferred either to the Cache Manager buffers, or to secondary storage using lower-level drivers, release FCB resources acquired and return the results to the caller.

Code Fragment

```
NTSTATUS    SFsdCommonWrite(
PtrSFsdlrpContext             PtrIrpContext,
PIRP                          Ptrlrp)
{
    // Declarations go here

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

---

* Actually, the native NT implementations appear to use a different philosophy when determining how to acquire the resources for the FCB. They tend to acquire the resource shared, unless the write operation extends beyond the current end-of-file. The reason for acquiring the paging I/O resource shared is based on the philosophy that the VMM will correctly serialize paging I/O write operations to any specific byte-range, and that it is better to provide greater concurrency in writing dirty pages quickly to secondary storage. This method of acquisition is slightly more difficult to implement.

```
II If this is an MDL write complete request, then
// there is not much processing that the FSD has to do.
if (PtrIoStackLocation->MinorFunction & IRP_MN_COMPLETE) {
    // Caller wants to tell the Cache Manager that a previously
    // allocated MDL can be freed. This may cause a recursive write
    // back into the FSD.
    SFsdMdlComplete(PtrIrpContext, Ptrlrp,
                           PtrloStackLocation, FALSE);
    // The IRP has been completed.
    Completelrp = FALSE;
    try_return(RC = STATUS_SUCCESS};
}
// If this is a request at IRQL DISPATCH_LEVEL, then post
// the request (your FSD may process it synchronously
// if you implement the support correctly) .
if (PtrIoStackLocation->MinorFunction & IRP_MN_DPC) {
    Completelrp = FALSE;
    PostRequest = TRUE;
    try_return(RC = STATUS_PENDING) ;
}

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

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

// Get some of the other parameters supplied to us.
ByteOffset = PtrIoStackLocation->Parameters .Write. ByteOff set;
WriteLength = PtrIoStackLocation->Parameters .Write. Length;

CanWait = ( (PtrIrpContext->IrpContextFlags
               & SFSD_IRP_CONTEXT_CAN_BLOCK)
        ? TRUE : FALSE) ;
Paginglo = ( (PtrIrp->Flags & IRP_PAGING_IO) ? TRUE : FALSE);
NonBuf feredlo = ( (PtrIrp->Flags & IRP_NOCACHE) ? TRUE : FALSE) ;
Synchronouslo = ( (PtrFileObject->Flags & FO_SYNCHRONOUS_IO ) ?
                           TRUE : FALSE) ;

// You might wish to check at this point whether the file object
// being used for write really did have write permission requested
// when the create/open operation was performed. Of course, for
// paging I/O write operations, the check is not valid, since
// paging I/O (via the VMM) could use any file object (likely the
// first one with which caching wasinitiated on the FCB) to
// perform the write operation.

//A 0-byte write can be immediately succeeded.
if (WriteLength ==0 ) {
    try_return(RC) ;
}
```

```
    II NOTE: if your FSD does not support file sizes > 2GB, you
    // could validate the start offset here and return end-of-file
    // if the offset begins beyond the maximum supported length.

    // Is this a write of the volume itself?
    if (PtrFCB->NodeIdentifier.NodeType == SFSD_NODE_TYPE_VCB) {
        // Yup, we need to send this on to the disk driver after
        // validation of the offset and length.
        PtrVCB = (PtrSFsdVCB) (PtrFCB) ;

        // Acquire the volume resource exclusively
        if ( !ExAcquireResourceExclusiveLite (& (PtrVCB->VCBResource) ,
                                                        CanWaitM {
            // Post the request to be processed in the context of a
            // worker thread .
            Completelrp = FALSE;
            PostRequest = TRUE;
            try_return(RC = STATUS_PENDING) ;
        }
        PtrResourceAcquired = & (PtrVCB->VCBResource) ;

        // Insert code to validate the caller-supplied offset here.

        // Lock the caller's buffer.
        if ( !NT_SUCCESS(RC = SFsdLockCallersBuffer (Ptrlrp,
                                                 TRUE, WriteLength) ) ) {
            try__return(RC);
        }

        // Forward the request to the lower-level driver.

        // For synchronous I/O wait here, else return STATUS_PENDING .
        // For asynchronous I/O support, read the discussion in
        // Chapter 10.

        try_return(RC) ;
    }

    // Your FSD should check whether it is
    // convenient to allow the write to proceed by utilizing the
    // CcCanlWrite ( ) function call. If it is not convenient to perform
    // the write at this time, you should defer the request for a
    // while. The check should not, however, be performed for
    // noncached write operations. To determine whether we are
    // retrying the operation or not, use the IrpContext structure we
    // have created (see the accompanying diskette to this book for a
    // definition of the structure) .
   IsThisADeferredWrite =
         ( (PtrIrpContext->IrpContextFlags
              & SFSD_IRP_CONTEXT_DEFERRED_WRITE) ? TRUE : FALSE) ;
    if (!NonBufferedIo) {
        if (! CcCanlWrite (PtrFileObject, WriteLength, CanWait,
                IsThisADeferredWrite) ) {
            // Cache Manager and/ or the VMM does not want us to perform
        II the write at this time. Post the request.
        SFsdSetFlag(PtrIrpContext->IrpContextFlags,
            SFSD_IRP_CONTEXT_DEFERRED_WRITE);
        CcDeferWrite(PtrFileObject, SFsdDeferredWriteCallBack,
            PtrlrpContext, Ptrlrp, WriteLength,
            IsThisADeferredWrite);
        Completelrp = FALSE;
        try_return(RC = STATUS_PENDING);
    }
}

// If the write request is directed to a page file (if your FSD
// supports paging files), send the request directly to the disk
// driver. For requests directed to a page file, you have to trust
// that the offsets will be set correctly by the VMM. You should
// not attempt to acquire any FSD resources either.
if (PtrFCB->FCBFlags & SFSD_FCB_PAGE_FILE) {
    loMarklrpPending(Ptrlrp);
    // You will need to set a completion routine before invoking
    // a lower-level driver
    // forward request directly to disk driver
    // SFsdPageFileIo(PtrIrpContext, Ptrlrp);

    Completelrp = FALSE;

    try_return(RC = STATUS_PENDING);
}

// We can continue. Check whether this write operation is targeted
// to a directory object, in which case the sample FSD will
// disallow the write request. Once again though, if you create a
// stream file object to represent a directory in memory, you
// could come to this point as a result of modifying the directory
// contents internally by the FSD itself. In that case, you should
// be able to differentiate the directory write as being an
// internal, noncached write operation and allow it to proceed.
if (PtrFCB->FCBFlags & SFSD_FCB_DIRECTORY) {
    RC = STATUS_INVALID_DEVICE_REQUEST;
    try_return(RC) ;
}

PtrReqdFCB = &(PtrFCB->NTRequiredFCB) ;

// There are certain complications that arise when the same file
// stream has been opened for cached and noncached access. The FSD
// is then responsible for maintaining a consistent view of the
// data seen by the caller.
// If this happens to be a nonbuffered I/O, you should try to
// flush the cached data (if some other file object has already
// initiated caching on the file stream) . You should also try to
// purge the cached information, though the purge will probably
// fail if the file has been mapped into some process's virtual
// address space.
// Read Chapter 10 for more information on the issues involved in
```

```
II maintaining data consistency.
// Insert appropriate code here . . .
//CcFlushCache(. . .
//CcPurgeCacheSection(. . .

// Acquire the appropriate FCB resource exclusively.
if (Paginglo) {
    // Try to acquire the FCB PagingloResource exclusively.
    if (!ExAcquireResourceExclusiveLite(&(PtrReqdFCB->
                                             PagingloResource)   ,
                                             CanWait)) {
        Completelrp = FALSE;
        PostRequest = TRUE;
        try_return(RC = STATUS_PENDING);
    }
    // Remember the resource that was acquired.
 PtrResourceAcquired = &(PtrReqdFCB->PagingIoResource);
} else {
    // Try to acquire the FCB MainResource exclusively.
    if (!ExAcquireResourceExclusiveLite(&(PtrReqdFCB->
                                             MainResource)   ,
                    CanWait)) {
        Completelrp = FALSE;
        PostRequest = TRUE;
        try_return(RC = STATUS_PENDING);
    }
    // Remember the resource that was acquired.
 PtrResourceAcquired = &(PtrReqdFCB->MainResource);
}

// Validate start offset and length supplied.
// Here is a special check that determines whether the caller
// wishes to begin the write at current end-of-file (whatever the
// value of that offset might be).
if ((ByteOffset.LowPart == FILE_WRITE_TO_END_OF_FILE) &&
      (ByteOffset.HighPart == OxFFFFFFFF)) {
 WritingAtEndOfFile = TRUE;
}

// Paging I/O write operations are special. If paging I/O write
// requests begin beyond end-of-file, the request should be no-
// op'ed (see the next two chapters for more information). If
// paging I/O requests extend beyond current end of file, they
// should be truncated to current end-of-file.
// Insert code to do this here.

// This is also a good place to set whether fast I/O can be
// performed on this particular file or not. Your FSD must make
// its own determination whether or not to allow fast I/O
// operations. Commonly, fast I/O is not allowed if any byte-range
// locks exist on the file or if oplocks prevent fast I/O. Many
// reasons could result in setting FastloIsNotPossible
//OR FastloIsQuestionable instead of FastloIsPossible.
//
```

```
PtrReqdFCB->CommonFCBHeader.IsFastloPossible = FastloIsPossible;

// This is also a good place for oplock-related processing.
// Chapter 11 expands upon this topic in greater detail.

// Check whether the desired write can be allowed, depending
//on any byte-range locks that might exist. Note that for
// paging I/O, no such checks should be performed.
if (!PagingIo) {
    // Insert code to perform the check here ...
    // if (!SFsdCheckForByteLock(PtrFCB, PtrCCB, Ptrlrp,
    // PtrCurrentloStackLocation)) {
    // try_return(RC = STATUS_FILE_LOCK_CONFLICT);
    // }
}

// Check whether the current request will extend the file size,
//or the valid data length (if your FSD supports the concept of a
// valid data length associated with the file stream) . In either
// case, inform the Cache Manager using CcSetFileSizes() about
// the new file length. Note that real FSD implementations will
// have to first allocate enough on-disk space before they
// inform the Cache Manager about the new size to ensure that the
// write will subsequently not fail due to lack of disk space.

// if ((WritingAtEndOfFile) ||
//     ((ByteOffset + TruncatedWriteLength) >
//        PtrReqdFCB->CommonFCBHeader.FileSize)) {
//     we are extending the file;
//     allocate space and inform the Cache Manager
// } else if (same test as above for valid data length) {
//     we are extending valid data length, inform Cache Manager;
// }


//   Branch here for cached vs. noncached I/O.
if (!NonBufferedIo) {

    // The caller wishes to perform cached I/O. Initiate caching if
    // this is the first cached I/O operation using this file
    // object.
    if (PtrFileObject->PrivateCacheMap == NULL) {
        // This is the first cached I/O operation. You must ensure
        // that the Common FCB Header contains valid sizes.
        CcInitializeCacheMap(PtrFileObject,
            (PCC_FILE_SIZES)(&(PtrReqdFCB->
                                CommonFCBHeader.AllocationSize)),
            FALSE,        // We will not utilize pin access for
                          // this file.
            St(SFsdGlobalData.CacheMgrCallBacks), // Callbacks.
            PtrCCB);        // The context used in callbacks.
    }
```

```
        // Check and see if this request requires an MDL returned to
        // the caller.
        if (PtrIoStackLocation->MinorFunction & IRP_MN_MDL) {
            // Caller does want an MDL returned. Note that this mode
            // implies that the caller is prepared to block.
            CcPrepareMdlWrite( PtrFileObj ect,ScByteOf f set,
                              TruncatedWriteLength,
                              &(PtrIrp->MdlAddress), &(PtrIrp->IoStatus));
            NumberBytesWritten = PtrIrp->IoStatus . Information;
            RC = PtrIrp->IoStatus. Status;

            try_return(RC)  ;
        }

        // This is a regular run-of-the-mill cached I/O request. Let
        // the Cache Manager worry about it.
        // First though, we need a valid buffer pointer (address) .
        // More on this in Chapter 10.

        // Also, if the request extends the ValidDataLength, use
        // CcZeroDataO first to zero out the gap (if any) between
        // current valid data length and the start of the request.
        PtrSystemBuffer = SFsdGetCallersBuf fer (Ptrlrp) ;
        ASSERT(PtrSystemBuffer);
        if ( !CcCopyWrite(PtrFileObject, &(ByteOf fset) ,
                                  TruncatedWriteLength,
                                  CanWait, PtrSystemBuffer)) {
            // The caller was not prepared to block and data is not
            // immediately available in the system cache.
            Completelrp = FALSE;
            PostRequest = TRUE;
            // Mark IRP Pending . . .
            try_return(RC = STATUS_PENDING) ;
        } else {
            // We have the data
            PtrIrp->IoStatus. Status = RC;
            PtrIrp->IoStatus . Information
                    = NumberBytesWritten = WriteLength;
        }

    } else {

        // If the request extends beyond valid data length, and if the
        // caller is not the lazy-writer, then utilize CcZeroDataO to
        // zero out any blocks between current ValidDataLength and the
        // start of the write operation. This method of zeroing data
        // is convenient since it avoids any unnecessary writes to
        // disk. Of course, if your FSD makes no guarantees about
        // reading uninitialized data (native NT FSD implementations
        // guarantee that read operations will receive zeroes if the
        // sectors were not written to, thereby ensuring that old data
        // cannot be reread unintentionally or maliciously) , you can
        // avoid performing the zeroing operation altogether. You
        // must, however, be careful about correctly determining the
```

```
            II top-level component for the IRP so as to be able to extend
            // valid data length only when appropriate and also avoid any
            // infinite, recursive loops.
            // See Chapter 10 for a discussion on this topic.

            // Send the request to lower-level drivers.
            // Here is a common method used by Windows NT file system
            // drivers that are in the process of sending a request to the
            // disk driver. First, mark the IRP as pending, then invoke
            // the lower-level driver after setting a completion routine.
            // Meanwhile, this particular thread can immediately return
            // / a STATUS_PENDING return code.
            // The completion routine is then responsible for completing
            // the IRP and unlocking appropriate resources.

            loMarklrpPending(Ptrlrp);

            // Invoke a routine to write information to disk at this time.
            // You will need to set a completion routine before invoking
            // a lower-level driver.

            Completelrp = FALSE;

            try_return(RC = STATUS_PENDING);
        }

        try_exit:    NOTHING;

        // If a synchronous I/O write request succeeded, and if the file
        // size has changed as a result, you may wish to update the file
        // size and the modification time for the file stream in the
        // directory entry for the link at this time.

    } finally {
        // Post IRP if required.
        if (PostRequest) {
            // Implement a routine that will queue-up the request to be
            // executed later (asynchronously) in the context of a system
            // worker thread. See Chapter 10 for details.

            if (PtrResourceAcquired) {
                SFsdReleaseResource(PtrResourceAcquired);
            }
        } else if (Completelrp && !(RC == STATUS_PENDING)) {
            // For synchronous I/O, the FSD must maintain the current byte
            // offset. Do not do this however, if I/O is marked as paging
            // /I/O.
            if (Synchronouslo && !Paginglo && NT_SUCCESS(RC)) {
                PtrFileObject->CurrentByteOffset =
                    RtlLargeIntegerAdd(ByteOffset,
                RtlConvertUlongToLargelnteger((unsigned
                                                   long)NumberBytesWritten));
            }
            //If the write completed successfully and this was not a
            // paging I/O operation, set a flag in the CCB that indicates
            // that a write was performed and that the file time should be
            // updated at cleanup. The other option would be to set the
            // access time in the FCB directly now.
            if (NT_SUCCESS(RC) && !PagingIo) {
                SFsdSetFlag(PtrCCB->CCBFlags, SFSD_CCB_MODIFIED);
            }

            // If the file size was changed, set a flag in the FCB
            // indicating that this occurred.

            // If the request failed, and we had done some nasty stuff like
            // extending the file size (including informing the Cache
            // Manager about the new file size), and allocating on-disk
            // space etc., undo it at this time.

            // Release resources.
            if (PtrResourceAcquired) {
                SFsdReleaseResource(PtrResourceAcquired);
            }

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

                // Free up the IRP Context.
                SFsdReleaselrpContext(PtrlrpContext);

                // Complete the IRP.
                loCompleteRequest(Ptrlrp, IO_DISK_INCREMENT);
            }
        } // Can we complete the IRP?
    } // End of "finally" processing.

    return(RC);
}

void SFsdDeferredWriteCallBack (
void                        *Contextl,            // Should be
PtrlrpContext
void                        *Context2)            // Should be Ptrlrp
{
    // You should typically simply post the request to your internal
    // queue of posted requests (just as you would if the original write
    // could not be completed because the caller could not block).
    // Once you post the request, return from this routine. The write
    // will then be retried in the context of a system worker thread.
}
```

Notes

This code fragment provides you with a sound framework that you should follow when implementing a dispatch routine to process file system write requests. Conceptually, write requests are not very different from read operations, and can be handled simply by forwarding the request to the NT Cache Manager, or by forwarding the request down to a disk or network driver to transfer information to secondary storage (either locally or across the network).

Some of the issues that you should be concerned about when implementing the write dispatch routine include correctly identifying the caller of the entry point, ensuring that data consistency is maintained if the same file stream is opened for both cached and noncached access, and keeping the Cache Manager informed about any changes to the file size. We will discuss some of these issues further in the next chapter.