2

File System Driver Development

In this chapter:
- *What Are File System Drivers?*
- *What Are Filter Drivers?*
- *Common Driver Development Issues*
- *Windows NT Object Name Space*
- *Filename Handling for Network Redirectors*

The focus of this book is on kernel-mode file system driver and filter driver development for the Windows NT operating system. However, before beginning a discussion on how to design and implement a kernel-mode file system or filter driver, you need a good understanding of just what the file system and filter drivers do. Knowing what these drivers can and cannot do will help you decide whether it is worth all the trouble to design one.

In this chapter, I will briefly discuss the various types of file system drivers and filter drivers to give you some idea of the functionality that is traditionally expected from them. I will also discuss some common concepts used during the design and implementation of kernel-mode drivers in Windows NT. Topics discussed here include how to make portions of your kernel-mode driver pageable, how to allocate and free kernel memory required during execution, how to use some of the system-defined structures and functions to create linked lists, and how to troubleshoot and debug your driver. It may be best for you to skim through this material initially, then refer back to it once you have read some of the succeeding chapters and have begun the process of designing and developing your kernel-mode file system or filter driver.

One of the challenges I faced when trying to design a file system driver for Windows NT was understanding how user-specified filenames are treated. I will discuss this as part of a larger discussion on the name space, which is managed by the Windows NT Object Manager. I will also discuss the roles played by the Multiple Provider Router (MPR) component and the Multiple UNC Provider (MUP) in supporting network file system drivers, which must be integrated with the name space on the local node. Chapters following this one examine some of the topics presented here in considerable depth.

What Are File System Drivers?

A *file system driver* is a component of the storage management subsystem. It provides the means for users to store information to and retrieve it from nonvolatile media such as disks or tapes.

Functionality Provided by a File System Driver

A file system driver implementation typically provides the following functionality to the user:*

- Ability to create, modify, and delete *files*†
- Ability to share files and transfer information between them easily, though in a secure and controlled manner
- Ability to structure the contents of a file in a manner appropriate to the application
- Ability to identify stored files by their symbolic/logical names, instead of specifying the physical device name
- Ability to view the data logically, rather than dealing with a more detailed physical view

The above functionality is provided by all commercially available local (disk based) file system driver implementations. In addition to this functionality, remote file systems, both networked and distributed, provide the following functionality, to some degree or another, depending upon the sophistication of the file system used:

- Network transparency
- Location transparency
- Location independence
- User mobility
- File mobility

Not all of the functionality listed here provided by all remote file system implementations. However, as file system technology evolves, more and more sophisticated network file systems meet or exceed many of these goals.

---

* See the book *An Introduction To Operating Systems* by Harvey Deitel. Consult Appendix E, *Recommended Readings and References,* for more information.

† A *file* is a named collection of user data stored on secondary storage devices (e.g., disk drives).

Types of File System Drivers

There are different kinds of file system driver implementations that you can design, implement, and install. They include local file systems, network filesystems, and distributed file systems.

Disk (local) file system drivers

Local file systems manage data stored on disks connected directly to a host computer.

The file system driver receives requests to open, create, read, write, and close files stored on such disks. These requests typically originate in user processes and are dispatched to the file system via the I/O subsystem manager. Figure 2-1 illustrates how a local file system driver provides services to a user thread.

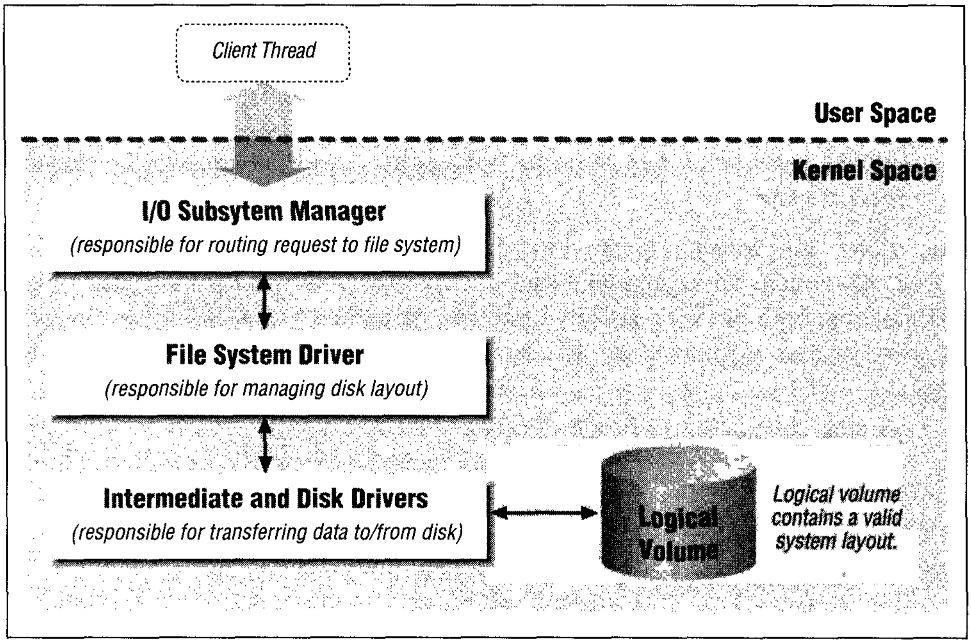
Client Thread
User Space
Kernel Space
I/O Subsytem Manager
(responsible for routing request to file system)
File System Driver
(responsible for managing disk layout)
Intermediate and Disk Drivers
(responsible for transferring data to/from disk)
Logical Volume
Logical volume contains a valid system layout.

Figure 2-1. Local file system

In the figure, the disk driver transfers data to and from a *logical disk* connected to the system. The logical disk is simply a storage abstraction; from the perspective of the file system, it is a linear sequence of fixed-size, randomly accessible blocks of storage. In reality, a logical disk could be a portion of a physical disk (commonly known as a *partition),* or it could be an entire physical disk, or it could even be some combination of partitions residing across multiple physical disks (known as a *logical volume).* Software modules called *logical volume* *managers* allow the file system driver to see a contiguous sequence of available disk space and hide all of the details of mapping logical blocks to the correct physical blocks.

Logical volume management software often provides features such as software mirroring of data, striping across multiple physical disks, as well as capabilities to resize logical volumes dynamically. Therefore, you will often see such software advertised as fault-tolerant software.

To be managed by a local file system driver, each logical volume must have a valid file system layout. The file system layout includes appropriate file system metadata information, specific to the type of file system driver used. For example, the FASTFAT file system driver requires a completely different on-disk layout than the NTFS file system driver. It uses structures very different from those used by NTFS to store user data.

On Windows NT systems, whenever you use the `format` utility on a logical volume, you are actually creating the file system metadata (management) structures that will later be used by the file system driver to provide functionality such as allocating space for user data storage, associating stored user data with the user-specified filename, and creating catalogs (directory structures) used in retrieving user files.

Before a user can begin accessing data stored on logical volumes, the logical volume must be mounted on the system. When a logical volume is mounted, a file system driver verifies the metadata and begins managing the volume, using the metadata stored on the volume and setting up appropriate in-memory data structures based on the metadata.

Local file systems provide a single name space for each mounted logical volume. Most commercially available, modern file system implementations provide a hierarchical, tree-structured layout. This tree structure consists of directories (container objects), and files (named user data objects) contained within directories. Each directory, as well as each file contained within a directory, has a unique filename associated with it. The valid character set that can be used to construct a filename is dependent upon the specific file system implementation. For example, the native NTFS file system allows some characters that the FASTFAT file system typically disallows. Most file systems and the I/O subsystem explicitly disallow certain characters. For example, the "\" character is used on Windows NT-based systems as a path separator and cannot be part of a valid filename.

Figure 2-2 shows a hierarchical file system name space as presented by a local file system driver. Each object in this file system can be uniquely identified by a name, starting with the root of the file system. The important thing to note is that each mounted logical volume has its own hierarchical tree structure with a unique root directory serving as the top-level container object for that logical volume.

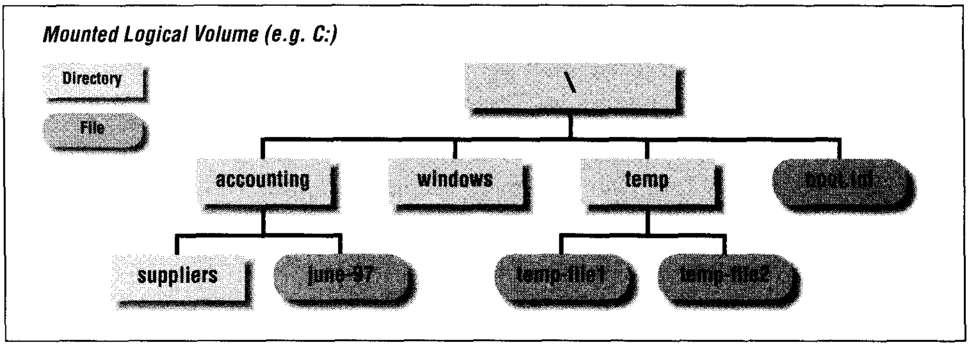

Figure 2-2. Hierarchical name space for directories and files

The user of a mounted logical volume is always aware of the particular mounted logical volume that she is accessing. If she wishes to access a file that does not reside on the currently mounted logical volume, she has to ensure that the logical volume on which the file resides is both accessible and mounted. Then she can specify the complete file pathname identifying the file, beginning at the root of the logical volume on which the file resides, to access the contents of the file.

Network file **systems**

As the name suggests, network file systems allow users to share locally connected disks with other users over a local or wide area network. For example, say you have a physical disk *C:* connected to your machine. Now you may want to allow me direct access to the files and directories stored under the *accounting* subdirectory on your *C:* local drive. To do this, both you and I would have to use the services of a network file system. This network file system would allow me to access the shared files on your disk, just as if I were accessing my own local disk.

There are two components to each network file system implementation:

The client-side redirector

There must be a software component, executing on my node, that will take my requests for accessing files stored in your *C:\accounting* directory and transfer them across the network to be processed on your machine. Furthermore, this software component must be capable of receiving data from your machine and handing it back to me.

The sewer on the node where the disk is being shared

Once the redirector on the client sends a request across the network, a software component on the server system must respond to this request.

> The server component then has two major tasks to perform; the first is to interface with the remote client using a well-defined protocol, and the second is to interface with the local file systems to obtain data on behalf of the client node.

Figure 2-3 shows the client and server components of the network file system implementation.

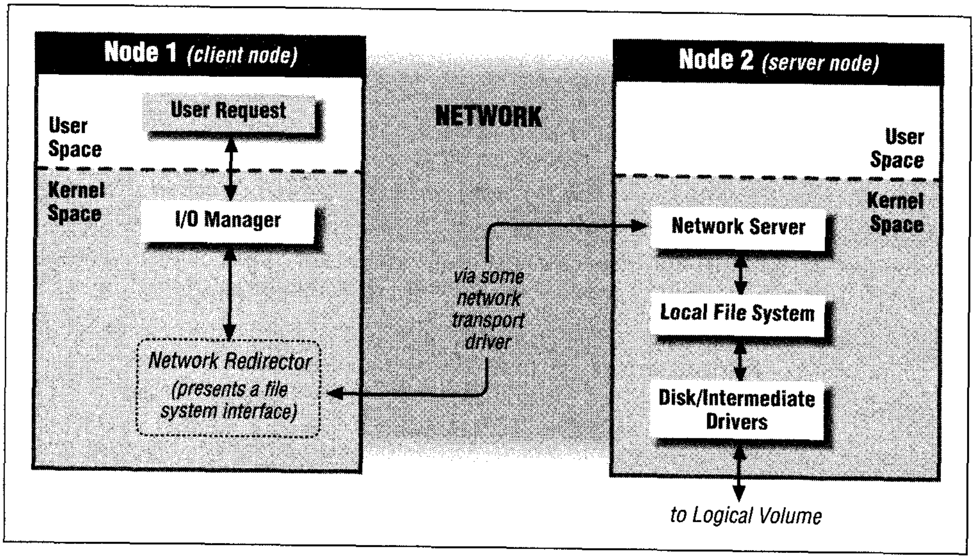

Figure 2-3. Remote (network)file system

The most common example of a remote file system to NT users is the LAN Manager Network, which supports the sharing of directories, logical volumes, printers, and other remote resources. The LAN Manager Network consists of the LAN Manager Redirector component executing in the kernel on client nodes and the LAN Manager Server software executing in the kernel on server nodes exporting local file systems or other resources such as printers and the 8MB (Server Message Block) network protocol used by the two components to transfer data across the network.

NOTE In 1996, Microsoft submitted a networking protocol specification called the Common Internet File System (CIFS) 1.0 to the Internet Engineering Task Force as an Internet-Draft document. Microsoft has since been working with other parties to get CIFS published as an Informational RFC. CIFS is the latest incarnation of the 8MB protocol specification and is expected to be a part of future updates to "Windows NT 4.x and Windows 95. Throughout this book, I use the term 8MB to refer to the networking protocol implementation used by the Microsoft LAN Manager Redirector and Server components; however, you can easily substitute the term CIFS for SMB.

Note that the redirector is the component that presents itself as a file system on the client node. This allows users to request access to remote data just as they would request data from her local file system. The redirector handles all of the mechanics of getting the data for users from across the network. Although networks are inherently unreliable (especially wide area networks), it is the responsibility of the redirector to try to reestablish lost connections transparently, or to return appropriate errors so that the application can retry the request if required.

The server does not need to present a file-system-like interface, because clients on the server node can use the services of the local file system directly to access data stored on the disk drives local to the server.

Both the redirector and the server use a transport protocol to transfer data and commands across the network. There are many transport protocols, such as the TCP/IP protocol, the UDP/IP, and Microsoft-specific protocols such as NetBIOS. The transport protocols may be connection-oriented (e.g., TCP/IP, NetBIOS), so that they provide a virtual circuit to the redirector and server software, or connectionless (e.g., UDP/IP).

Figure 2-4 illustrates how a server node can share a particular directory with clients across the network. To the client node, the shared directory forms the root of a distinct logical volume. Requests from the client node to the networked volume are handled by the redirector, which is responsible for transmitting the request across the network to the server node. The network server software on the server node processes the request, utilizing the local file system on the server node to access and manipulate the shared volume. Finally, the server returns the results of the operation to the remote client.

In the case of network file systems, the client is aware of the fact that the user is accessing data residing on the server node. Therefore, although all of the mechanics of data transfer are hidden from the user of the file system, the user is

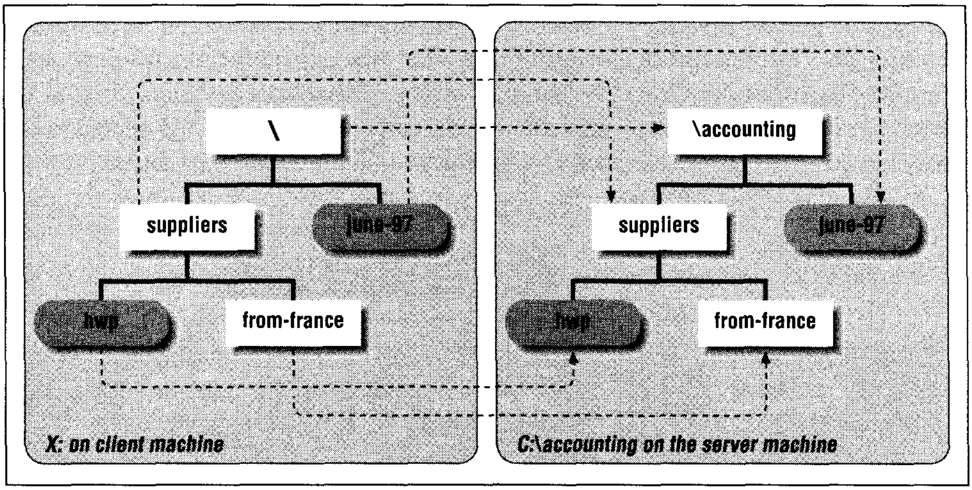

Figure 2-4. Sharing a directory across the network

always aware of which data is stored locally and which is obtained from a remote server node.

Finally, you should note that applications on the server node use local file system services to access file data residing on the shared logical volumes. In certain cases, this may lead to data consistency problems if file data from shared logical volumes is also cached on client nodes. Local (disk-based) file system drivers are often expected to cooperate with network server software to help avoid such data consistency problems whenever possible.

Distributed file systems

Distributed file systems have evolved from standard network file systems. They present a single name space to the user and completely hide the actual physical location of the data from the user of the file system.

This means that a user supplies a single pathname to identify the required file, regardless of the physical location of the file. Therefore, a user can access resources residing on a remote server machine without even realizing it.

Architecturally, distributed file systems look very much like network file systems, since they also have client software executing on client nodes and server software executing on remote nodes to make their resources available across the network. The primary difference, however, is the single name space provided by distributed file systems over and above what is offered by simpler network file systems. Note that both client and server software could be concurrently executing on any node that participates in the implementation of the distributed file system.

Figure 2-5 illustrates how a distributed file system presents a single name space to the user of the file system. A client of the file system on *node 1* can access all of the files and directories that constitute the file system without regard for where they physically reside. There is a single (virtual) global root directory for the file system tree. Although not illustrated in the figure, any point in the global name space could in actuality be a *mountpoint* for a remotely exported subtree.

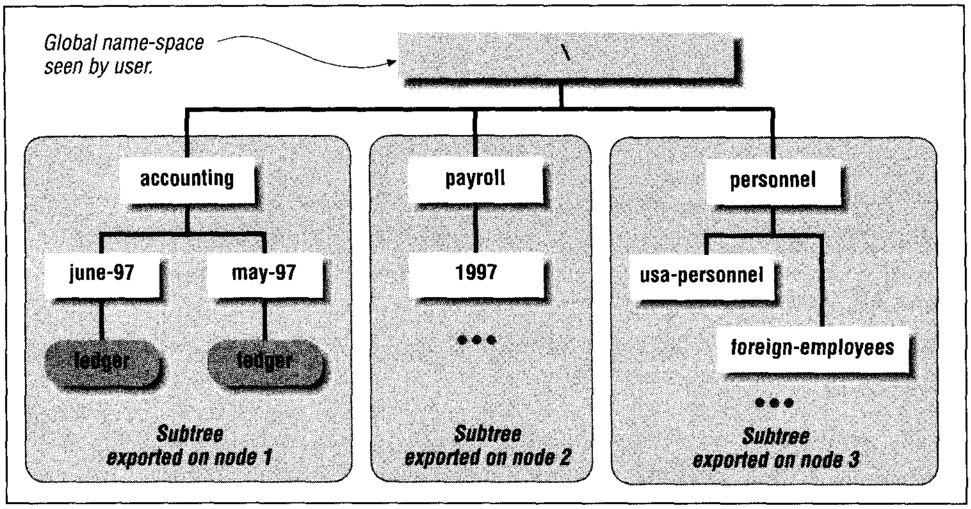
Global name-space seen by user.
\
accounting
june-97
may-97
ledger
ledger
Subtree exported on node 1
payroll
1997
•••
Subtree exported on node 2
personnel
usa-personnel
foreign-employees
•••
Subtree exported on node 3

Figure 2-5- Global name space presented by distributed file systems

---

NOTE A *mountpoint* is simply a named directory in the file system name space to which a remotely exported subtree can be grafted. In Figure 2-5 above, you can see that the *accounting, payroll,* and *personnel* directories are mount points for the distributed file system. The *accounting* directory has a subtree from *node 1* grafted on, the *payroll* directory allows access to data stored on *node 2,* while the *personnel* directory allows access to data stored on *node 3-* Any user of this file system can now transparently access a file or a directory without regard for where the data actually resides. The user simply sees a single name space for the entire distributed file system.

When a user tries to access anything below a mount point, the client software on the node must forward the request to the remote server that is actually exporting the contents below the accessed mount point, allowing the server to process the user request.

Many distributed file systems use another approach to access data stored remotely. The client software often transfers data from the remote server on behalf of the requesting process and caches it locally. This obviates the need to contact a remote server every time a user asks for previously requested data stored there. However, sophisticated client-server cache consistency processes are required to maintain data coherency across the entire network.

Sometimes, distributed file systems provide global data consistency guarantees exceeding those provided by the network file system implementations. For example, a distributed file system could guarantee that all users of the file system would always see the same view of a file's contents even if they were concurrently accessing and modifying the file on multiple (geographically distributed) client nodes.

Special (pseudo)file systems

Often, you will encounter kernel-mode software that presents a file-system-like interface to the user but actually does something completely different when the interface calls are exercised. For example, the */proc* file system on UNIX systems actually allows a user to access and potentially modify the address space of a running process.

Basically, any kernel-mode driver that presents a file-system-like interface but performs special functionality (different from the traditional task of managing data stored on physical devices) can be considered a special file system implementation.

Other examples of special file system implementations include kernel-mode drivers that provide hierarchical storage management (HSM) functionality, or drivers that present virtual file systems (e.g., some commercially available source code control systems).

Windows NT and File System Drivers

File system drivers are a component of the I/O subsystem on the Windows NT platform and therefore must conform to the interface defined by the NT I/O Manager.

The Windows NT I/O Manager has defined a standard interface to which all kernel-mode drivers must conform. This interface applies equally to local file system drivers, network and distributed file system redirector software, intermediate drivers, filter drivers, and device drivers. File system drivers can be loaded dynamically under Windows NT and can theoretically also be unloaded dynamically.*

The Windows NT/ I/O Manager provides a comprehensive set of support routines for file system driver designers to use. These routines allow the new file system to utilize common services and behave consistently (just as the native file systems

---

* In practice, it is very difficult to implement a file system that can be dynamically unloaded. It is possible, though, with a lot of foresight and care in the design and implementation of the file system driver. Most people, however, do not find the result worth all of the effort required.

do) on Windows NT machines. Furthermore, there is a well-defined, although poorly documented,* set of interfaces that the file system driver designer must conform to, in order to interact successfully with the Windows NT Virtual Memory Manager and the Windows NT Cache Manager.

Using a File System

There are two ways in which a user can take advantage of the services provided by a file system driver:

Invoke standard system service calls

> This is by far the most commonly used method of requesting access to files and directories. The user process simply invokes standard system service calls to request operations such as opening or creating a file, reading or writing file data, and closing the file.

Use I/O control requests sent to afile system driver

> Sometimes, applications need to request specific services that cannot be requested using one of the canned system service calls. In these situations, as long as a file system can do the desired operation, a user can send the request and data directly to the file system driver via the *File System Control* CFSCTL) interface.

A typical example of using standard system services to request access to a file is when a process must read the contents of file *C:\payroll\june-97.* The sequence of operations executed by a typical application process using the Win32 subsystem is as follows:

1. Open the file.

   The requesting process will typically invoke the Win32 `CreateFile()` service routines, specifying the name of the file to be opened, the access mode desired for the open file, and other related arguments. Internally, the Win32 subsystem invokes the `NtCreateFileO` system service call to request the open operation on behalf of the caller.t

   At this point, the CPU switches to kernel-mode privilege level. The code implementing the system call `NtCreateFile()` is implemented by the I/O Manager, which is a component of the Windows NT Executive, and the kernel-mode privilege level is required to run functions implemented by the I/O Manager. The open/create request meanders around the NT Executive,

---

* Until this book was written.

t Any user-space process can directly invoke the `NtCreateFile ()` system service routine. Unfortunately, these system service routines have not been well documented by Microsoft. Appendix A, *Windows NT System Services,* has a comprehensive list of the available system services.

dispatched first to the I/O Manager via the `NtCreateFile()` invocation, then to the NT Object Manager to parse the user-supplied name, and finally back into the I/O Manager to identify the file system driver managing the mounted logical volume *C:*. Once the file system driver has been identified, the I/O Manager invokes the file system driver create/open dispatch entry point to process the user request.

Finally, the file system driver performs appropriate processing and returns the results of the create/open operation to the I/O Manager, which in turn returns the results to the Win32 subsystem (the privilege level switches back to user-mode), and the Win32 subsystem eventually returns the results to the requesting process.

2. Read the file data.

   If the open operation succeeds, a handle is returned back to the requesting process. The requesting process now asks to read data in the file, specifying the starting offset and the number of bytes to be read. Typically, the `ReadFile()` function call provided by the Win32 subsystem invokes the `NtReadFile()` system service routine on behalf of the requesting process.

   The `NtReadFile()` routine is also implemented by the NT I/O Manager. Because the requesting process must supply a valid file handle, obtained from a previous successful create operation, to request a read, the I/O Manager can easily identify an internal data structure corresponding to the open operation performed earlier. This internal data structure, called a *file object,* will be comprehensively described later in this book. From the file object structure, the I/O Manager can determine the logical volume that contains the open file and will then forward the read request to the file system driver for further processing.

   The file system driver will return as much of the user-requested data as it can and will return the results of the operation back to the I/O Manager. Eventually, the results of the read request will be returned back to the requesting process via the Win32 subsystem.

3. Close the file.

   Once the requesting process has finished processing the contents of the file, it performs a close operation for the file handle received from the previously executed open request. The close handle operation informs the system that the process no longer needs to access the file data.

   The close file process invokes the Win32 `CloseHandle()` function to close the open file handle. The Win32 subsystem in turn invokes the `NtClose()` system service routine.

> The file system is notified by the I/O Manager that the user process has closed the file handle, and the file system is free to dispose of any state information it may have maintained for the open file.

There are many file operations that can be requested by a user in addition to the three described here. However, the basic methodology is the same: a process or thread opens or creates a file, performs some operations on the file, and finally closes the open file handle. Note that the NT system services are available to all threads executing on a Windows NT system, including user-mode threads and kernel-mode threads. Furthermore, the NT system services are available regardless of the subsystem (Win32, POSIX, OS/2) used by a requesting process.

NOTE The system service routines provided by the NT I/O Manager are generic and very comprehensive. They have to be generic because, as mentioned earlier, the services must be capable of supporting requests generated by a user from any one of the supported Windows NT subsystems, which are quite diverse in themselves.

As a matter of fact some of the most powerful functionality provided by the I/O Manager and the file system drivers is often not available (or provided) by the Windows NT subsystems and the *only* way to request the desired functionality is to invoke the system services directly. Therefore, it is more of a pity that Microsoft does not do a better job documenting the available Windows NT system service calls.

Support provided for file system control requests by file system drivers is described in detail later in this book.

The File System Driver Interface

A well-defined interface between the file system driver code and the rest of the operating system must exist, if the operating system is to support multiple file system drivers, including those developed by third-party companies. This interface should clearly document the various interactions between the components involved in satisfying a user request to access file data; the description must also provide for suitable abstractions so that the many varied types of file systems can be successfully integrated into the rest of the operating system.

The goal should be to create modularized components that can be easily substituted and extended without requiring extensive, complicated, and expensive redesign of the entire system. It seems as though the designers of the I/O subsystem started out trying to meet exactly these goals. Therefore, there are well-defined methods for a file system to install, load, and register itself with the rest of the operating system. The I/O Manager also sends very well defined I/O request packets describing user requests to a file system driver for further processing. Last, but not least, there is a fairly comprehensive list of supporting routines that a file system designer can use to make life easier and to better integrate the new file system with the rest of the system.

Unfortunately, things tend to become more than a little messy when you consider the different ways the file system and the operating system interact. Sometimes, as a result of these complex interactions, the abstractions that system designers try to maintain start to break down. The situation is made much worse when the operating system and the file system are jointly responsible to provide support for cached data, and also for supporting memory-mapped files. In Windows NT, for example, the Virtual Memory Manager depends on the file system to provide support for page files used to implement virtual memory support. However, the file system, in turn, depends upon the Virtual Memory Manager for allocation of memory required to process file system requests. This recursive relationship tends to make life even more complicated.

Although the designers at Microsoft who developed the Windows NT operating system seem to have made a strong effort to maintain a clean demarcation between the file system and the rest of the operating system, it seems as though, over time, the lines have gotten more than a little blurred and that more and more implicit behaviors and functionality have become ingrained in the system. This leads to more complicated design and code, and requires extensive documentation from Microsoft for third-party file system designers to develop a successful and robust file system driver.

The sort of documentation that third-party developers would like to have access to was not available when this book went to press. This book will help you understand the system better and give you a starting point to achieve your desired goals.

What Are Filter Drivers?

A filter driver is an intermediate driver that intercepts requests targeted to some existing software module (e.g., the file system or a disk driver). By intercepting the request before it reaches its intended target, the filter driver has the opportunity to either extend, or simply replace, the functionality provided by the original recipient of the request.

NOTE It isn't required that the filter driver always supplant the existing driver; that would simply become a case of unnecessarily reinventing the wheel. The filter driver can instead focus on providing whatever specialized functionality it needs to implement, while still allowing the existing code to perform what it does best, provide the original functionality.

For example, consider the existing file systems shipped with the Windows NT operating system. They consist of the FASTFAT (the legacy FAT file system support) file system, the NTFS (log-based) file system, the CDFS file system for CD-ROM media, the LAN Manager Redirector to access remote shared drives, and so on. None of the file systems, however, currently provides support for online encryption and decryption of stored data.

Now suppose that you are a security expert who knows how to design and implement an incredibly secure encryption algorithm. You wish to develop and sell software that would encrypt user data before it ever got stored on disk, and automatically decrypt it before giving it back to an authorized user. So how would you go about designing your software?

You certainly do not want to write a completely new file system driver, because that would be too time consuming, and it would not really provide any added value to the end user. What you really want to do is design a filter driver that intercepts requests in either of the following places:

Above the file system

To allow your code to intercept the user request before the file system driver ever gets the opportunity to see it.

Below the file system

To allow your driver to perform any required processing after the file system has finished its tasks. However, your driver can do whatever you need before the request is received by a disk driver, or by a network driver that is asked by the file system to obtain data from secondary storage devices or from across the network.

In this scenario, you can perform your magic somewhere along the way before the data either is written to the disk or returned to the user.

Figure 2-6 illustrates two different places where you can insert your filter driver software.

Once you have inserted your filter driver at an appropriate place in the driver hierarchy, you can intercept I/O requests from the user, perform your magic, and then forward the request to the existing module (either the file system or the disk

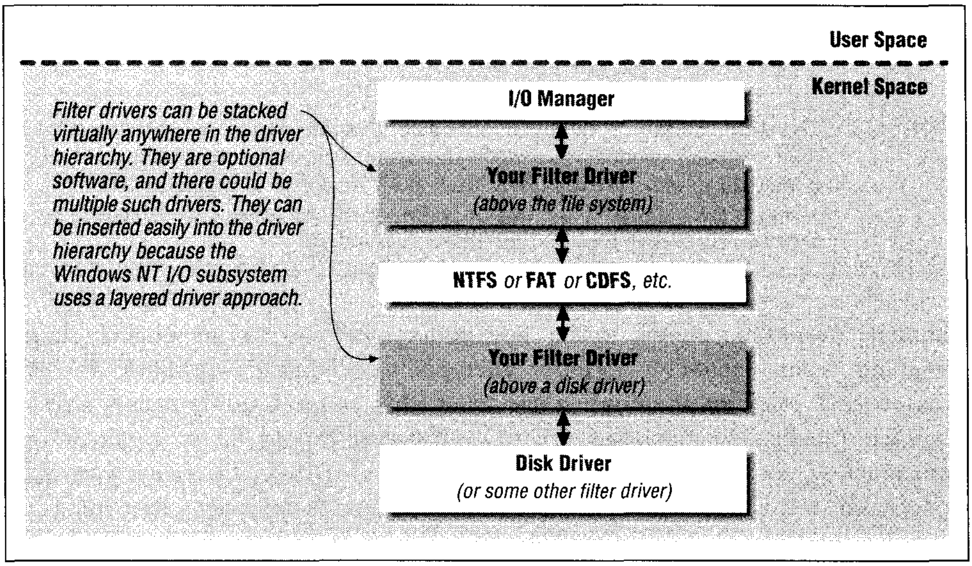
User Space
Kernel Space
Filter drivers can be stacked virtually anywhere in the driver hierarchy. They are optional software, and there could be multiple such drivers. They can be inserted easily into the driver hierarchy because the Windows NT I/O subsystem uses a layered driver approach.
I/O Manager
Your Filter Driver (above the file system)
NTFS or FAT or CDFS, etc.
Your Filter Driver (above a disk driver)
Disk Driver (or some other filter driver)

Figure 2-6. Filter drivers in the driver hierarchy

driver) so that they can continue to provide functionality, such as managing the mounted logical volume or transferring data to or from the physical disks.

So if you insert your filter driver so that it intercepts I/O requests dispatched to a file system driver, you can encrypt the data before it is passed into the file system for transfer to secondary storage, and you can decrypt it after the file system has retrieved the encrypted data from secondary storage, before it sends the data back to the user.

If, however, you decide to intercept requests below the file system, then you would follow the same methodology, except that now you would get a chance to modify the buffer only after it had passed through the file system and either before it is written out to disk (or across the network), or immediately after it has been retrieved from disk (or from across the network), but before it is returned to the file system.

It is relatively easy to insert a filter driver into the existing driver hierarchy in either of these two places, without having to redesign all other existing Windows NT file system, disk, and other intermediate drivers, because all drivers in the I/O subsystem must conform to a well-defined, layered driver interface.

This means, for example, that all drivers must respond to a standard set of requests that the I/O Manager could issue. Furthermore, there is a standard method by which a kernel-mode driver (or the I/O Manager itself) requests the services provided by another driver in the calling hierarchy. Every driver in the hierarchy must also respond to an I/O request in the expected manner, regardless of the caller.

NOTE The I/O subsystem does not mandate that all drivers implement their dispatch routines in exactly the same way; the only condition is that the drivers are aware of their own response to standard I/O Manager requests and are therefore aware of the impact they have by inserting themselves into the driver hierarchy.

Although everything seems to be just perfect for you to immediately begin designing your incredibly secure encryption/decryption algorithm for the Windows NT platform, there are some details that you will unfortunately have to consider. Ideally, the Windows NT I/O subsystem would be so modular that implementing your functionality should be a piece of cake. In reality, you must understand some subtle interactions that manifest themselves, depending on where in the driver hierarchy you decide to insert your filter driver. Chapter 12, *Filter Drivers,* focuses exclusively on the issues involved in designing a filter driver for the Windows NT platform.

Common Driver Development Issues

This book discusses many issues that kernel-mode file system and filter driver designers should understand thoroughly. There are some common development issues, however, that I would like to briefly discuss in this section. These include how to allocate and free memory in your kernel-mode driver, and how to implement some rudimentary debugging support in your driver.

Consult the Microsoft Driver Development Kit (DDK) documentation for additional details on some of the functions described here. Some of the material in this section uses terms that will be defined later in the book. Therefore, skim through the material during your first reading of this book and then come back to reread it after you have read through at least Chapter 4, *The NT'I/O Manager.*

Working with Kernel Memory

In Chapter 5, *The NT Virtual Memory Manager,* you will read about the NT VMM in considerable detail. However, there are some fairly common issues involved with driver development and the need for kernel memory that I will describe here. The code fragments presented later in this book assume that you have a good understanding of how to allocate and free kernel memory.

You must answer the following questions as you begin designing a kernel-mode driver:

- Does my driver occupy paged or nonpaged memory?
- Can I page out driver code?
- How do I allocate kernel memory on demand?
- How do I free previously allocated memory?
- Are there any issues I must be aware of when attempting to acquire or free kernel memory?

Pageable kernel-mode drivers

By default, the kernel loader will load all driver executables and any global data that you may have defined in your driver into nonpaged memory. Therefore, if you want your driver to reside in nonpaged memory, there is nothing further you need to do besides compiling, linking, and loading the driver.

Furthermore, the kernel loads the entire driver executable (and any associated dynamic link libraries) all at once, before invoking any driver initialization routines. Although it may not make much sense to you at this time, after loading the executable into memory, the kernel loader closes the executable file, allowing a user to delete even the currently executing driver image.

It is possible to specify to the loader the portions of your driver that you wish to make pageable. This can be done by using the following compiler directive in your driver code:*

```
•ifdef ALLOC_PRAGMA
•pragma         alloc_text(PAGE,  function_namel)
#pragma          alloc_text(PAGE,  function_name2)
// You can list additional functions at this point just as the two
// functions are listed above …
•endif // ALLOC_PRAGMA
```

Be careful, though, that you never allow any routine that could possibly be invoked at a high IRQL to be paged out. File system drivers can never allow any code or data to be paged out that might be required to satisfy page fault requests from the NT Virtual Memory Manager.

It is also possible for a kernel-mode driver to determine at run-time whether certain sections of driver code and/or data should be paged out or locked into memory. To do this, the driver must perform the following actions:

---

* The functions referenced in a pragma statement must he defined in the same compilation unit as the pragma.

- To make a code section pageable, use the following compiler directive in your code,

  ```
  #ifdef ALLOC_PRAGMA
  #pragma alloc_text(PAGExxxx, function_namel)
  #pragma alloc_text(PAGExxxx, function_name2)
  …
  fendif
  ```

  where xxxx is an optional, four-character, unique identifier for the driver's pageable section.

- To make a data section pageable, use the following compiler directive in your code:

  ```
  #ifdef ALLOC_PRAGMA
  #pragma data_seg(PAGE)...
  // Define your pageable data section module here.
  #pragma data_seg() // Ends the pageable data section.
  ```

- Invoke MmLockPagableCodeSection() and MmLockPagableCodeSectionByHandle () to lock code sections that were marked as pageable in memory.
- Invoke `MmLockPagableDataSection()` and MmLockPagableDataSectionByHandle () to lock data sections that were marked as pageable.
- Invoke MmUnlockPagableImageSection() to unlock any code or data section that may have been locked using the functions listed above.

There are two additional routines provided by the VMM that you should be aware of (and look up in the DDK documentation) if you wish to page out the entire driver or reset paging attributes back to their original settings:

`MmPageEntireDriver()`

> This routine will make the entire driver pageable, overriding any section page attributes that were declared earlier using compiler directives.

`MmResetDriverPaging()`

> This function will reset the paging attributes back to the initially declared attributes.

Finally, to automatically have the Memory Manager discard sections of code that you won't need once the driver has been initialized, use the following compiler directives:

```
•ifdef ALLOC_PRAGMA
•pragma alloc_text(INIT, DriverEntry)
•pragma alloc_text(INIT, functionl_called_by_driver_entry)
…
•endif // ALLOC_PRAGMA
```

Be careful to specify only those functions that can be safely discarded and will never again be required once the driver initialization has been completed.

Allocating kernel memory

Every kernel-mode driver requires memory to store private data. Typically, your driver will request memory from the NT Virtual Memory Manager. Whenever your driver requests memory, it must determine whether it needs paged or nonpaged memory. If your driver can afford to incur page faults during execution when accessing allocated memory, try to use paged memory whenever possible.

NOTE Most lower-level disk and network drivers typically can't use pageable data because their code often executes at high IRQ levels that do not allow page faults.* However, file systems (which are often considerably larger and more resource intensive than disk drivers) do sometimes have the opportunity to allocate certain memory from the paged pool. If you can use pageable memory in your driver, always take the extra effort to identify the memory that could be pageable and specify the paged pool type when requesting memory from the Virtual Memory Manager.

Nonpaged memory is a limited resource available to the entire system. Though the amount of memory reserved for nonpaged pool depends upon the type of system used (and the amount of physical memory available on the system), it is definitely something to be conservatively used.†

The following support routines are provided by the Windows NT Executive to kernel-mode drivers for allocating memory:

- `ExAllocatePool()`
- `ExAllocatePoolWithQuota()`
- `ExAllocatePoolWithTag()`
- `ExAllocatePoolWithQuotaTag()`

---

* See Chapter 5 for a detailed discussion on page fault handling performed by the Windows NT Virtual Memory Manager. This chapter also further explains why kernel-mode drivers must not incur page faults at high IRQ levels.

† The NT Virtual Memory Manager uses a private algorithm to determine the total amount of nonpaged pool reserved on a node. This algorithm uses the total amount of physical memory on the system as the determining factor to compute the amount of nonpaged pool. The Virtual Memory Manager also attempts to increase the amount of nonpaged pool (if required) up to a precomputed maximum value. Finally, although the initially allocated nonpaged pool is contiguous, it tends to get fragmented, and the Virtual Memory Manager makes no attempts to ensure that the pool stays contiguous when expanding it.

Note that all of the pool allocation support routines are nonblocking in Windows NT. In other words, the memory allocation function invoked will return memory if it is currently available; otherwise, the functions will return NULL (indicating that memory could not be allocated). On many other operating system platforms (e.g., many UNIX derivatives), kernel-mode components are allowed to specify whether the memory allocation function should block (wait) for memory to become available, or return failure immediately.

Whenever your driver invokes one of these functions to request memory, it must specify the type of memory required:

`NonPagedPool`
: The pool allocation package will return either a pointer to nonpageable memory or NULL.

`PagedPool`
: Always specify this type if your application can handle a page fault when accessing the allocated memory. Never allocate paged memory if you have any synchronization structures (described in the next chapter) contained within the allocated memory.

`NonPagedPoolMustSucceed`
: If all else fails and you simply must get memory immediately, use this pool type. Note that the memory reserved for this type is an extremely scarce resource. It may be as low as 16KB on a system, though the amount is variable. If you request pool of this type (and only do that if you failed to get memory any other way), and if the Virtual Memory Manager cannot provide you with the requested memory, it will bugcheck the system (described later in this chapter) with an error code of MUST_SUCCEED_POOL_EMPTY.

`NonPagedPoolCacheAligned`
: This allocates nonpaged memory that is aligned on a CPU-specific boundary, determined by the data cache line size. Note that this option defaults to the NonPagedPool allocation type on Intel platforms.

`PagedPoolCacheAligned`
: A request to allocate pageable memory aligned along the CPU data cache line size.

`NonPagedPoolCacheAlignedMustSucceed`
: Once again, use this option to request nonpaged memory only as a last resort.

The pool allocation package initializes several lists, each containing blocks of a certain fixed size. Whenever you request memory using one of the `ExAllocatePool()` functions listed above, the support routine will try to allocate a fixed-size block that is closest in size (greater than or equal to) the requested amount.

If your request exceeds a page, however, or if the requested amount exceeds the size of the largest-size block in the various lists, or if there is no available block of the appropriate size in the preallocated lists, the Virtual Memory Manager will allocate the requested amount from any available system memory of the appropriate type.

NOTE When the lists of preallocated blocks are empty, the Virtual Memory Manager will allocate at least one page of memory, split it up, and put any remaining amount (after returning the requested amount of memory to the caller) on the appropriate block list.

Unfortunately, however, for requests for nonpaged pool where the requested amount is greater than PAGE_SIZE, the pool allocation support routine will not attempt to split up any unused amount. This wastes precious nonpaged memory, another reason why you should be extremely conservative in your requests for this type of memory.

If there is simply no memory available of the requested type, the Virtual Memory Manager will return NULL to the caller or bugcheck* if you request memory from the must-succeed pool.

It is also possible for your driver to use one of MmAllocateNonCachedMemory() or MmAllocateContiguousMemory()t to request nonpaged, or physically contiguous, memory, respectively. These routines are not typically used by file system or filter drivers, which use either the Executive pool routines or other constructs, such as zones or lookaside lists (described below), for memory management.

Using zones

Kernel-mode drivers can fragment the physical memory available to the system if they repeatedly allocate and free small amounts of memory (less than 1 PAGE_SIZE). This can cause all sorts of problems for the rest of the system, including degradation of system performance.

---

* To bugcheck the system is to bring down (halt) the system in a controlled manner. Typically, the KeBugCheck() function is used, which will bring clown the system while displaying the bugcheck code and possibly more information on the reason for the bugcheck operation. You should bugcheck a system only when your driver discovers an unrecoverable inconsistency that will corrupt the system.

t The contiguous memory is allocated from the list of nonpaged memory pages reserved at system initialization time. Note that there is no way to ensure that the system will have the amount of contiguous memory requested, because the nonpaged pool tends to become fragmented due to expansion and usage. The only advice typically given to kernel-mode driver designers that develop drivers requiring contiguous nonpaged memory is to load the memory early in the system boot cycle and to retain the contiguous memory given to the driver by the Virtual Memory Manager.

One way you can avoid this situation of fragmenting system memory is by preallocating a reasonably sized chunk of memory and then doing some of your own memory management, allocating and freeing smaller-sized blocks from this preallocated chunk as necessary. This method avoids system fragmentation, because the Virtual Memory Manager is usually out of the picture once you have preallocated your fixed-sized chunk. You only need to go back to the Virtual Memory Manager when you run out of memory in your chunk and need to expand its size.

To help you incorporate this method of memory management into your driver, the Windows NT Executive provides a set of support utilities. These functions work on a *zone*, for which your driver must have preallocated memory. Another requirement is that the size of each block that can be allocated from the zone is fixed at the time of zone initialization. Therefore, if you have a fixed-size data structure that is smaller than the size of one page, and if you know that you will be repeatedly allocating and freeing memory for structures of this type, you should seriously consider using the zone method (or the lookaside list discussed later) to perform the memory allocation and deallocation.

Note that the method used here requires your driver to retain a preallocated piece of memory. The trade-off is a possible waste of kernel memory, since you would typically allocate the chunk at driver-initialization time (especially when your driver does not require the memory for a long time), against the possibility of fragmenting the kernel pool of available pages.

Here is the sequence of operations you must follow to use the zones method:

1. Determine the size of the memory chunk you are likely to need.

   Be careful not to allocate either too much or too little memory for the zone. Allocating too much memory is simply being wasteful, and allocating too little will result in having to allocate more, leading to memory fragmentation, something you wish to avoid.

TIP Determining the optimal amount of memory that should be preallocated for a zone is often an iterative task. However, as a general rule, you should be conservative with the amount of memory reserved for a zone. If you allocate too little memory, under most circumstances the worst-case scenario will be that your driver has to go back to the VMM for more memory at run-time. If you allocate too much memory (more than you will ever use), you will have effectively denied access to the excess memory to all components in the system and could thereby even cause some components to fail.

---

2. Allocate the zone using one of the ExAllocatePool() routines listed previously.

You have a choice of allocating from nonpaged or paged pool. Note that the base address of your piece of memory must be aligned on a 8-byte boundary (i.e., the base address should be a multiple of 8).

3. Allocate and initialize a spin lock or use some other synchronization mechanism to protect modifications to the list.

   Synchronization structures, including Executive spin locks, are discussed extensively in the next chapter.

4. Define a structure of type ZONE_HEADER somewhere in global memory (or in a driver object extension).

   Driver object extensions are discussed in Chapter 4. The ZONE_HEADER structure serves as a control structure for the zone, used by the zone management support routines to allocate and free entries from the zone.

5. Invoke ExInitializeZone () to initialize the zone header.

   You will also have to pass in (as arguments to the routine) a pointer to the zone you allocated in Step 2 and the size of the structures that you expect to allocate from the zone. The size of the structures you expect to allocate must be aligned on a 8-byte boundary.

   Also note that a ZONE_SEGMENT_HEADER-sized block of memory from the chunk of memory you supply will be used by the zone manipulation routines to maintain some additional control information. The rest of the preallocated memory will be carved out into the fixed-size blocks (of the size specified by you) for use by your driver.

Now the zone is ready for use by your driver. Whenever you need a new structure from the zone, use either the ExAllocateFromZone () or the ExInterlockedAllocateFromZone () functions. The only difference between these two functions is that the interlocked version accepts a pointer to the Executive spin lock structure that you previously initialized, and will automatically guarantee list consistency by using the spin lock to provide synchronization. If you decide to use the noninterlocked version instead, you are responsible for ensuring that the list does not get corrupted due to concurrent access and modification by multiple threads. Therefore, you must use some appropriate synchronization method in your driver.

To return a previously allocated structure to a zone, use either the ExFreeToZone () or the ExInterlockedFreeToZone () support routines provided.

Do not use the zone manipulation routines at an IRQL greater than DISPATCH_LEVEL, because you will not be able to use the synchronization structures (spin locks or another) at a higher IRQL.

In the event that you do need to extend the size of a zone, you must use the ExExtendZone ( ) function provided. Once again, you must pass a newly allocated chunk of memory that will be used to extend the zone. Remember that the base address of this memory must also be aligned along a 8-byte boundary.

Unfortunately, there is no routine provided that decreases the size of a previously extended zone. Therefore, any chunk allocated and used when you initialize or extend the zone will be unavailable to the rest of the system until the machine is rebooted. This places the responsibility on your driver to ensure that you are fairly accurate in your estimates of how much memory should be reserved for the zone.

The file system example code provided in Part 3 uses zones for memory management. Examine the source code for the sample file system driver on the accompanying diskette for examples of using this method in your driver.

Using ***lookaside lists***

Although using zones helps to reduce fragmentation of system memory, there are some disadvantages you must be aware of when you use zones.

- Your driver must preallocate the memory for the zone, usually at driver initialization time, even though this memory may not be used until much later.
- You must be fairly accurate about your memory requirements; you cannot release any excess memory that you may have allocated during peak driver utilization.

  When you design and use your driver, you will see that there are periods when your driver is simply overwhelmed with requests. At such times, naturally, your memory requirements will increase. If you use zones, there is a distinct probability that your zone will get depleted at such times. Then, you must either allocate memory directly from the system or extend the zone.

  Extending the zone means that the newly allocated memory cannot be released until a system reboot—not a very appealing prospect. Allocating directly from the system means that you have to maintain some sort of flag in your allocated structure indicating where the memory came from so that you could release it appropriately (either back to the zone if it came from the zone, or back to the system if you allocated using a direct invocation to an ExAllocatePool ( ) routine).
- You must use either some private synchronization mechanism or, more typically, a spin lock to synchronize access to the zone.

The lookaside list is a new structure defined in Windows NT 4.0, and with the associated support routines, it addresses the limitations of the zone method.

When you invoke the ExInitializeNPagedLookasideList() or the ExInitializePagedlookasideList() functions to initialize the list, no memory is preallocated. Instead, entries are allocated on an as-needed basis when you actually require the memory. Although your driver is free to supply pointers to your driver-specific *allocate* and *free* functions when initializing the list header, this is optional and the Windows NT Executive pool management package will use the ExAllocatePoolWithTag() function (and the corresponding free routine) by default.

Second, you are required to specify a *list depth* at initialization time. This depth specifies the maximum number of entries of the desired size that will be queued on the list. Note that the list becomes populated with available entries as you allocate and then subsequently free the memory.

Therefore, when you start requiring memory and the package begins allocating some on your behalf, any freed entries will not be given back to the system but will instead be queued onto the list head until the *depth* number of entries have been queued. Any entries allocated and released beyond this value will automatically be returned to the system.

This allows your driver to increase your memory consumption during peak usage periods without having either to retain the memory until the next boot cycle or maintain the state information (using flags) in your allocated structures to determine where to return the memory when you release it.

Finally, on architectures that provide Windows NT with the appropriate instruction support, the ExAllocateFromNPagedLookasideList() (or the ExAllocateFromPagedLookasideList()) function and the corresponding release functions will use an atomic 8-byte compare-exchange operation to synchronize access to the list instead of using the FAST_MUTEX or KSPIN_LOCK (described in the next chapter) associated with the list. This is a considerably more efficient method of synchronization.

Remember to always allocate the NPAGED_LOOKASIDE_LIST list header or the PAGED_LOOKASIDE_LiIST list header from nonpaged memory.

Available **kernel stack**

Each thread executing on the Windows NT platform has both a user stack, used when the thread is executing in user mode, and a kernel stack, used only when the thread is executing in kernel mode.

Whenever a thread requests system services causing a switch to kernel mode, the *trap* mechanism always switches stacks and replaces the user-space stack with the kernel-space stack allocated for the thread. This kernel stack is of fixed size and is therefore a limited resource. On Windows NT 3.51 and earlier, the kernel stack was limited to two pages of memory; therefore, on Intel architectures, each thread was restricted to an 8KB kernel stack. Beginning with Windows NT 4.0, the kernel stack size has been increased to 12KB. However, this is not sufficient in itself for your driver to be extravagant in its use of available stack space.

There is a lot of recursive behavior exhibited by the higher level drivers in Windows NT, especially with the file system drivers, the NT Virtual Memory Manager, and the NT Cache Manager. This can lead to situations where the kernel stack gets depleted rather rapidly. Furthermore, the highly layered model of drivers within the I/O subsystem can cause the kernel stack to be depleted if the driver hierarchy becomes too deep and if one or more drivers in the hierarchy are not careful about their stack usage.

Be warned that the kernel stack cannot be increased dynamically. Therefore, always be prudent in your usage of local variables that reside on the stack. If you develop a filter driver that inserts itself into a driver hierarchy, be extremely frugal with your usage of the stack space, because you may inadvertently push the stack consumption beyond the limit and bring down the system unexpectedly.

Working with Unicode Strings

All character strings are represented internally by the Windows NT operating system as Unicode (16-bit wide) characters (also called *wide characters).* This allows the system to more easily accommodate and work with languages not based on the Latin alphabet.

When you design your driver, be prepared to receive strings in Unicode and to be able to manipulate such strings. Each Unicode string is represented using the UNICODE_STRING structure defined by the system. This structure consists of the following fields:

`Length`

This is the length of the string in bytes (not characters). It does not include the terminating NULL character if the string is null-terminated.

`MaximumLength`

This is the actual length of the buffer in bytes. Note that it is possible to have a maximum length that is much greater than the Length field.

`Buffer`

The is a pointer to the actual wide-character string constant. Wide-character strings do not necessarily have to be null terminated since the `Length` field above describes the number of valid bytes contained in the string.

Any string you wish to store in the associated `Buffer` must have a length (in bytes) that is less than or equal to the MaximumLength.

NOTE To use a null-terminated wide-character string in a UNICODE_STRING structure, initialize the `Length` field to the number of bytes contained in the wide-character string constant, excluding the UNICODE_NULL character; initialize the MaximumLength field to the size of the string constant (this should include the entire buffer including the space allocated for the UNICODE_NULL character).

There are a variety of support routines provided to facilitate manipulation of Unicode strings. The DDK header files contain the function declarations:

`RtlInitUnicodeString`
: This function initializes a counted Unicode string. You can either pass in an optional wide-character null-terminated source string or NULL. The target Unicode string `Buffer` will either be initialized to point to the `Buffer` field in the null-terminated source string (if supplied) or will be initialized to NULL. The `Length` and MaximumLength fields will be appropriately initialized.*

`RtlAnsiStringToUnicodeString`
: Given a source ANSI string, this routine will convert the string to Unicode and initialize the contents of the target string to contain the converted character string. You can either request the routine to allocate memory for the target wide-character string or supply the memory yourself by initializing MaximumLength in the target Unicode string structure to the length of your passed-in buffer. If you do request that the routine allocate memory for you, then remember to free the memory by invoking the `RtlFreeUnicodeStringO` function (see below).

`RtlUnicodeStringToAnsiString`
: This routine converts a source Unicode string to a target ANSI string.

`RtlCompareUnicodeString`
: A case-sensitive or case-insensitive comparison of two Unicode strings is performed. This function returns 0 if the strings are equal, a value less than 0 if the first Unicode string is less than the second one, and a value greater than 0 if the first Unicode string is greater than the second.

`RtlEqualUnicodeString`
: This function performs either a case-sensitive or a case-insensitive comparison of two Unicode strings. TRUE is returned if the strings are equal and FALSE otherwise.

---

* If a source wide-character string constant is supplied, the `Length` of the target string will be set to the number of non-null characters in the source string multiplied by `sizeof` (WCHAR). The MaximumLength field will be initialized to the value contained in the `Length` field + `sizeof` (UNICODE_NULL).

RtlPrefixUnicodeString

This function is defined as follows:

```
BOOLEAN
RtlPrefixUnicodeString(
    IN PUNICODE_STRING        Stringl,
    IN PUNICODE_STRING        String2,
    IN BOOLEAN                CaselnSensitive
)
```

This function will return TRUE if `Stringl` is a prefix of the counted string `String2`. If both strings are equal, this function will return TRUE.

RtlUpcaseUnicodeString

This function converts a copy of the source string into upper case Unicode characters and writes out the resulting string into the target string argument. It will also allocate memory for the target string if you request it to; otherwise you must pass in a target string with memory already allocated.

Use the RtlFreeUnicodeString() function to free the memory allocated for you by this function.

RtlDowncaseUnicodeString

This routine performs the converse of the RtlUpcaseUnicodeString() function above.

RtlCopyUnicodeString

A copy of the source Unicode string is put into the target string. As many Unicode characters as possible will be copied, given the MaximumLength field of the target string. The caller is always responsible for preallocating memory for the target of the copy operation.

RtlAppendUnicodeStringToString

This function will concatenate two Unicode strings. If the contents of the `Length` field in the target plus the `Length` of the source is greater than the value contained in the MaximumLength field in the target, the function will return STATUS_BUFFER_TOO_SMALL.

RtlAppendUnicodeToString

This is similar to the RtlAppendUnicodeStringToString() function except that the source Unicode string is simply a wide-character string instead of a buffered Unicode string.

RtlFreeUnicodeString

Any memory allocated by a previous invocation to RtlAnsiStringToUnicodeString() or RtlUpcaseUnicodeString() is released.

Declaring a wide-character (16-bit character set) string constant is a simple matter of appending an L before the string constant. For example, the ANSI string constant `"This is a string"` could easily be declared as a wide-character string as `L"This is a string"`. The size of each character comprising a wide-character string is computed as `sizeof(WCHAR)`. The wide-character string constant can then be used to create a UNICODE_STRING structure by initializing the `Buffer` field to point to the wide-character string constant and initializing the Length and MaximumLength fields appropriately.

Be careful not to treat Unicode characters as if they were simple ANSI. For example, you cannot assume that there is any kind of relationship between upper- and lowercase Unicode characters. Therefore, some of your assumptions (including allocating a fixed-sized table to contain the character set) will no longer be valid with respect to Unicode strings.

Linked-List Manipulation

Most drivers need to link together internal data structures, or create driver-specific queues. Typically, you will use linked lists to perform such functionality. The Windows NT Executive provides system-defined data structures and support functions for manipulating linked lists.

There are three types of linked list support functions and structures defined by the Windows NT DDK:

Singly linked lists

The DDK provides a predefined structure to use to create your own singly linked lists. The structure is defined as follows:

```
typedef struct _SINGLE_LIST_ENTRY {
    struct _SINGLE_LIST_ENTRY    *Next;
} SINGLE_LIST_ENTRY, *PSINGLE_LIST_ENTRY;
```

You should declare a variable of this type to serve as the list anchor. Initialize the `Next` field to NULL in the list anchor before attempting to use it. For example, you can have a field either in your driver extension structure or in global memory associated with the driver that is declared as follows:

```
SINGLE_LIST_ENTRY        PrivateListHead;
```

Each structure that you wish to link together using this list entry type should also contain a field of type SINGLE_LIST_ENTRY. For example, if you wish to queue structures of type SFsdPrivateDataStructure, you would define the data structure as follows:

```
typedef SFsdPrivateDataStructure {
    // Define all sorts of fields...
    SINGLE_LIST_ENTRY          NextPrivateStructure;
    // All sorts of other fields...
}
```

Now, whenever you wish to queue an instance of the `SFsdPrivateDataStructure` onto a linked list, use either of the following routines:

- `PushEntryList()`

  This function takes two arguments: a pointer to the list anchor for the linked list and a pointer to the field of type SINGLE_LIST_ENTRY in your data structure that you wish to queue. Therefore, if you have a variable called `SFsdAPrivateStructure` of type `SFsdPrivateDataStructure`, you can invoke this routine as follows:

  ```
  PushEntryList(&PrivateListHead,
                &(SFsdAPrivateStructure.NextPrivateStructure));
  ```

  You must ensure that this invocation is protected by some sort of internal synchronization mechanism that your driver uses.

- `ExInterlockedPushEntryList()`

  The only difference between this function call and the `PushEntryList()` function is that you must supply a pointer to an initialized variable of type KSPIN_LOCK when you invoke this function. Synchronization is automatically provided by the `ExInterlockedPushEntryList()` function via the spin lock that you provide.

  Note that you must ensure that all of the list entry structures you pass in to the `ExInterlockedPushEntryList()` have been allocated from nonpaged pool, because the system cannot take a page fault once a spin lock has been acquired.

Corresponding routines that unlink the first entry from the list are the `PopEntryListO` and the `ExInterlockedPopEntryList()` functions.

Doubly linked lists

The following structure type is predefined by the Windows NT operating system for supporting doubly linked lists:

```
typedef struct _LIST_ENTRY {
   struct _LIST_ENTRY * volatile Flink;
   struct _LIST_ENTRY * volatile Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
```

Just as in the case of singly linked lists, you must define a variable of type LIST_ENTRY to serve as your list anchor. You should use the `InitializeListHead(&SFsdListAnchorOfTypeListEntry)` macro to initialize the forward and backward pointers in the list anchor variable. Note that the forward and backward pointers are initialized to point to the list anchor; therefore, never expect to get a NULL list entry pointer when you traverse the list (the doubly linked list is organized as a circular list).

If you wish to link together structures of a particular type, ensure that a field of type LIST_ENTRY is associated with (typically contained in) the structure definition. For example, you can define a structure called SFsdPrivateDataStructure as follows:

```
typedef SFsdPrivateDataStructure {
    // Define all sorts of fields...
    LIST_ENTRY        NextPrivateStructure;
    // All sorts of other fields...
}
```

To queue an instance of a structure of type SFsdPrivateDataStructure, you can now use the following macros/functions:

– `InsertHeadList()`

  This macro takes as arguments a pointer to the list anchor (which must have been initialized using InitializeListHead() described above) and a pointer to the field of type LIST_ENTRY in the structure to be queued, and inserts the entry at the head of the list.

  For example, you can invoke this macro, as shown here, to queue an instance called SFsdAPrivateStructure of the SFsdPrivateDataStructure structure type:

```
InsertHeadList(&SFsdListAnchorOfTypeListEntry,
               &(SFsdAPrivateStructure.NextPrivateStructure));
```

– `InsertTailList()`

  Similar to the InsertHeadList described above except that it inserts the entry at the tail of the list.

– `RemoveHeadList() orRemoveTailList()`

  These macros simply require a pointer to the list anchor. The former will return a pointer to the entry removed from the head of the list and the latter will return a pointer to the entry removed from the tail of the list.

– `RemoveEntryList()`

  This macro takes as an argument a pointer to the LIST_ENTRY field in the structure to be removed.

There are interlocked versions (written as functions) of the macros described above. These functions take as an additional argument a pointer to an initialized variable of type KSPIN_LOCK, which is used to synchronize access to the list. The list entries must always be allocated from non-paged pool if you wish to use the interlocked functions to manipulate the linked list.

You should use the IsListEmpty() macro to determine whether a doubly linked list is empty. This macro returns TRUE if the Flink and Blink fields in the list anchor structure both point to the list anchor. Otherwise, the macro returns FALSE.

S-Lists

This is a new structure introduced in Windows NT 4.0 to support interlocked, singly linked lists efficiently. To use this structure, you should define a list anchor of the following type:

```
typedef union _SLIST_HEADER {
    ULONGLONG Alignment;
    struct {
        SINGLE_LIST_ENTRY  Next;
        USHORT             Depth;
        USHORT             Sequence;
    };
} SLIST_HEADER, *PSLIST_HEADER;
```

The ExInitializeSListHeadO function can be used to initialize a S-List linked list anchor. Your driver must supply a pointer to the list anchor structure when invoking this function. Ensure that the list anchor is allocated from nonpaged pool. Furthermore, you should allocate and initialize a spin lock to be used when you add or remove entries from the list.

The ExInterlockedPushEntrySList() and the ExInterlockedPopEntrySList () functions that are provided to add and remove list entries may not use the spin lock but may instead try to use an 8-byte atomic compare-exchange instruction on those architectures that support it.

All entries for the S-List linked list must be allocated from nonpaged pool.

You can also use the ExQueryDepthSListHead() to determine the number of entries currently on the list. This is convenient, since you no longer have to maintain a separate count of the number of entries (as you might have to if you use an anchor of type SINGLE_LIST_ENTRY structure instead).

Using the CONTAINING_RECORD macro

The Windows NT DDK provides the following macro, which is very useful to all kernel-mode driver developers:

```
#define CONTAINING_RECORD(address, type, field)    \
      ((type*)((PCHAR)(address) - (PCHAR)(&((type *)0)->field)))
```

This macro can be used to get the base address of any in-memory structure, as long as you know the address of the field contained in the structure. The macro definition is quite simple: your driver supplies the address to a field in the structure, the structure type, and the field name; the macro will compute the *field offset* (in bytes) for the supplied field in the structure and subtract the computed offset number of bytes from the supplied field pointer address to get the base address of the structure itself.

The CONTAINING_RECORD macro allows you the flexibility to place fields of type LIST_ENTRY and SINGLE_LIST_ENTRY anywhere in the containing data structure. You can use this macro whenever you need to determine the address of an in-memory data structure, if you know the address of a field contained in the structure.

As an example of how the CONTAINING_RECORD macro can be used by your driver, consider the following structure defined by a kernel-mode file system driver:

```
typedef struct _SFsdFileControlBlock {
    // Some fields that will be expanded upon later in this book.
    ...
    // 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;
    ...
} SFsdFCB, *PtrSFsdFCB;

LIST_ENTRY              SFsdAllLinkedFCBs;
```

The interesting field in the SFsdFCB structure is the NextFCB field. This field is of type LIST_ENTRY and will presumably be used to insert FCB structures onto a doubly linked list. The global variable SFsdAllLinkedFCBs is used to serve as the list anchor.

The interesting point to note is that the NextFCB field is *not* the first field in the SFsdFCB structure.* Rather, it is somewhere in the middle of the structure definition. However, given the address of the NextFCB field, the CONTAINING_RECORD macro is used to determine the address of the FCB structure itself. The following code fragment traverses and processes all FCB structures that are linked to the SFsdAllLinkedFCBs global variable:t

```
LIST_ENTRY        TmpListEntryPtr = NULL;
PtrSFsdFCB        PtrFCB = NULL;

TmpListEntryPtr = SFsdAllLinkedFCBs.Flink;
while (TmpListEntryPtr != &SFsdAllLinkedFCBs) {
    PtrFCB = CONTAINING_RECORD(TmpListEntryPtr, SFsdFCB, NextFCB);
    // Process the FCB now.
    ...;
    // Get a pointer to the next list entry.
```

---

* A common method of manipulating linked lists of structures is to place link pointers at the head of the structure and to cast the link pointer to the structure type when following pointers in the linked list.

t I have deliberately omitted any synchronization code to simply illustrate the use of the CONTAINING_RECORD macro.

```
    TmpListEntryPtr = TmpListEntryPtr->Flink;
}
```

Therefore, note once again that your driver is not required to place fields of type LIST_ENTRY and SINGLE_LIST_ENTRY at the head of the containing data structures, as long as you use the CONTAINING_RECORD (or some equivalent) macro to get a pointer to the base structure.

Preparing to Debug the Driver

Here are some simple points to keep in mind when designing your kernel-mode driver:

Insert debug breakpoints

Appendix D, *Debugging Support,* describes debugging the kernel-mode driver in greater detail. Note for now that if you have a debugger attached to your target machine, you can insert the DbgBreakPoint() function call in your code to break into the debugger when certain conditions occur.

Be careful to place appropriate #ifdef statements around your debug breakpoint statements so you can easily disable the break statements in a nondebug build of the driver. Here's a method I've used:

```
#if DBG
#define    SFsdBreakPoint()        DbgBreakPoint()
#else
#define    SFsdBreakPoint()
#endif
```

The DBG variable has a value of 1 when you compile your driver using a *checked* build environment. In this case, any SFsdBreakPoint() statements in your driver will be activated. The expectation is that you will only execute the debug version of your driver during the development and test phase and that you will always have a debug host node connected to the target machine executing your driver. However, if you compile the driver using the *free* (nondebug) build environment, the SFsdBreakPoint() statement will be rendered harmless.

The Windows NT DDK also provides a KdBreakPoint() function that is defined exactly as the SFsdBreakPoint() function described here. Therefore, you may choose to simply use KdBreakPoint() in your code and be assured that the breakpoint will be automatically rendered harmless in a nondebug build.

Insert debug print statements

You can use the KdPrint() macro that is defined to DbgPrint() in a debug version of the driver code. You can supply a formatted string to this function just as you would do with a printf() function call.

The KdPrint{} macro automatically becomes non-operational in the nondebug version of the driver executable.

Insert bugcheck (panic) calls in your driver

Never bring down the system unless you absolutely have to. And there are very few reasons indeed to bring down a live production system executing your code.* Instead, explore every alternative available if you detect inconsistencies in your code. Try to disable your driver if you can, stop processing requests, shut down the offending module, anything to avoid halting the system.

But there still might be situations (especially during development) when you may wish to bugcheck the system. There are two alternative function calls that you can invoke to bring down the system immediately in a controlled manner:

– `KeBugCheck()`

This function takes a single unsigned long argument (the BugCheckCode), which can be the reason that you have decided to terminate system execution. Internally, KeBugCheck() simply invokes KeBugCheckEx() described below.

– `KeBugCheckEx()`

This function takes a maximum of five possible arguments. The first is the BugCheckCode, the remaining four are optional arguments (each of type unsigned long) you may supply that provide more information to the user of the system and can possibly assist in postmortem analysis of the cause for the bugcheck.

There are no restrictions mandated by the system as to what the values of these four optional arguments should be.

If there is no debugger connected to the system, the system will do the following:

— Disable all interrupts on the node.

— Ask all other nodes (in a multiprocessor system) to stop execution.

— Use HalDisplayString() to print a message.

The user will see the infamous blue screen of death (BSOD) on their monitor. The message

```
STOP: 0x%lX (0x%lX, 0x%lX, 0x%lX, 0x%lX)
```

---

* Some exceptions that immediately come to mind are if continuing system execution could cause system security to be compromised or would lead to user data corruption. In such situations, it is preferable to bugcheck the system rather than to continue running.

will be displayed, with the bugcheck code displayed first, followed by each of the optional arguments supplied to KeBugCheckEx ( ) .

- If a message can be associated with the bugcheck code, invoke `HalDisplayString ( )` to print the descriptive message.
- The KeBugCheckEx ( ) function will then attempt to dump the machine state.

  If any of the bugcheck arguments is a valid code address, the system will try to print the name of the image file that contains the code address.

  The routine prints the version of the operating system executing on the node and then attempts to display the list of the node's loaded modules. The number of loaded module names displayed depends upon the number of lines of text that can be displayed on your monitor. Finally, the function will try to dump out some of the current stack frame. The system will then stop execution.

If, however, a debugger is connected to the system, the KeBugCheckEx ( ) function will display the message

```
Fatal System Error: Ox%lX (Ox%lX, Ox%lX, Ox%lX, Ox%lX)
```

on your debug host node, using the DbgPrint ( ) function call. Then, the system will break into the debugger using the DbgBreakPoint ( ) function call. You now have the opportunity to examine the system state to determine the cause of the error. If you ask the system to continue, the code sequence described above is executed.

Windows NT Object Name Space

As described in Chapter 1, *Windows NT System Components*, the designers of Windows NT have tried hard to make it an object-based system. There is a comprehensive set of object types that are defined by the system, and each object type has appropriate methods (or functions) associated with it to allow kernel-mode components to access and modify objects of the type.

Windows NT object types include adapter objects, controller objects, process objects, thread objects, driver objects, device objects, file objects, timer objects, and so on. One such special object type is the directory object. This object is simply a container object that, in turn, contains objects of other types.

The Object Manager allows each object to have an optional name associated with it. This facilitates the sharing of objects across processes, since more than one process can potentially open the same named object of a particular type. The Object Manager therefore manages a single, global name space for a node running the Windows NT operating system.

Following in the footsteps of most modern-day commercial file system implementations, the NT Object Manager presents a hierarchical name space to the rest of the system. There is a root directory object called \ for this global name space. All named objects can be located by specifying an absolute pathname for the object starting at the root of the object name space. Note that the Object Manager allows the creation of named object directories contained within directory objects, thereby providing a multilevel tree hierarchy.

The Object Manager also supports a special object type called the symbolic link object type. A symbolic link is simply an alias for another named object.

Figure 2-7 shows a typical name space presented by the NT Object Manager:

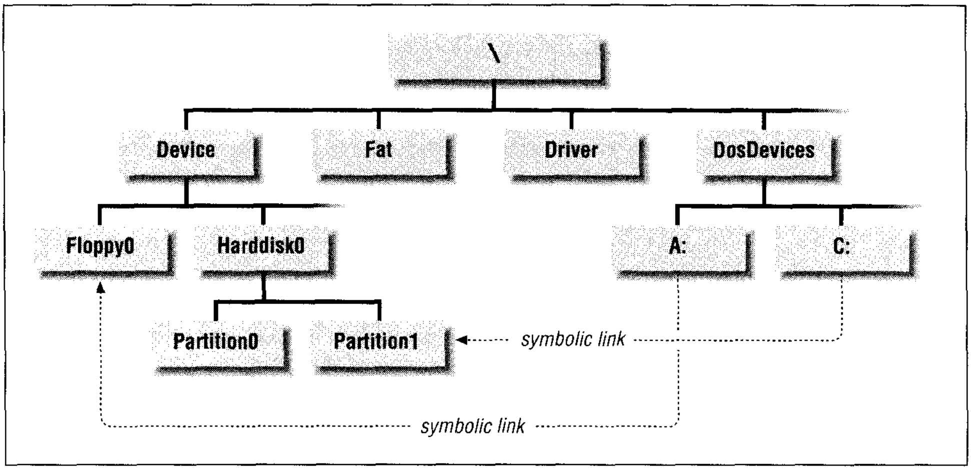

Figure 2-7. Name space presented by the Object Manager

The NT Object Manager defines object types when requested by other NT components. Certain object types are predefined by the Windows NT Object Manager. Whenever a Windows NT Executive component requests a new type to be defined by the NT Object Manager, the component has the option of providing pointers to the parse, close, and delete callback functions to be associated with all object instances of that particular type. The Object Manager remembers these function pointers and invokes the callback functions whenever a parse, close, or cleanup operation is being performed on an object instance of the particular type.

Whenever a user process or an application tries to open an object, it must supply an absolute pathname to the NT Object Manager. The Object Manager begins parsing the name, one token at a time. Whenever the Object Manager encounters an object that has a parsing callback function associated with it, the Object Manager suspends its own parsing of the name, and invokes the parsing function supplied for the object, passing it the remainder of the user-supplied pathname (the portion that has not yet been parsed).

So how is all of this relevant to file system drivers or network redirectors?

Consider what happens when a user process tries to open the file *C:\accounting\june-97.*

The user's open request is submitted to the Win32 subsystem, which translates the *C:* portion of the name to the string *\DosDevices\C:* before forwarding the request to the Windows NT Executive for further processing.* The complete name sent to the Windows NT kernel is *\DosDevices\C:\accounting\june-97.*

All create and open requests are directed initially to the NT Object Manager. The Object Manager receives the open request and begins parsing the filename. The first thing it notices is that the object *\DosDevices\C:* is really a symbolic link object (the *\DosDevices* portion of the name refers to a directory object type). Since symbolic link object types contain the name of the object they are linked to, the Object Manager replaces the symbolic link name (i.e., *\DosDevices\C:*) with the name of the linked object (i.e., *\Device\HarddiskO\PartitionT).*

NOTE Under Windows NT 4.0, the *\DosDevices* object type is itself a symbolic link to the directory object *\??.* Therefore, under Windows NT 4.0, the Object Manager will first replace the *\DosDevices* symbolic link name with \.?.?and then restart parsing of the name.

The complete name is now *\Device\HarddiskO\Partitionl\accounting\june-97.* Once the Object Manager has performed the name replacement, it begins the parsing of the pathname once again, beginning at the root of the object name, space. The object name space, including the portion managed by the file system is illustrated in Figure 2-8.

Now, the Object Manager traverses the global object name space until it encounters the *Partitionl* device object. This is a device object type defined by the Windows NT I/O Manager. The I/O Manager also supplies a parsing routine when creating this object type. Therefore, the Object Manager stops any further parsing of the pathname and instead forwards the open request to the Windows

---

* Note that the *C:* drive letter name is simply a shortcut provided by the Win32 subsystem to the *\DosDevices\C:* symbolic: link object type in the Windows NT object name space. Therefore, the Win32 subsystem is responsible for expanding the name before forwarding the request to the Windows NT Executive. This is also the reason why you cannot use the *C:\. . .* pathname if you try to open or create a filename from within the NT Executive (for example, from within your driver). You must instead use the Windows NT Object Manager recognizable pathname, beginning at the root of the Object Manager name space.

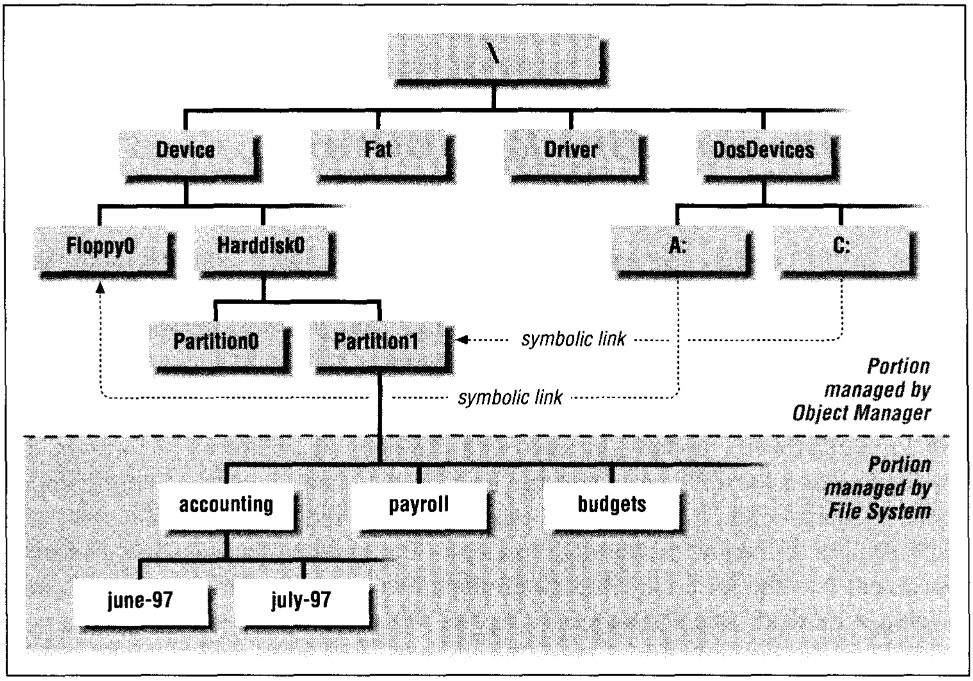

Figure 2-8. Object name space

NT I/O Manager's parsing routine. The string passed to the Windows NT I/O Manager is that portion of the pathname that has not yet been parsed by the Object Manager, namely *\accounting\june-97.* When invoking the parsing routine, the Object Manager also passes a pointer to the *Partitionl* device object to the NT I/O Manager.

The Windows NT I/O Manager now executes a reasonably complicated sequence of instructions to perform the open operation on behalf of the caller. This sequence is described in considerable detail in subsequent chapters. For now, you should note that the I/O Manager will typically identify the file system driver that is currently managing the mounted logical volume for the physical disk represented by *Partitionl,* the named device object. Once it has identified the appropriate file system driver, the I/O Manager will simply forward the open request to the file system driver's create/open dispatch routine.

Now, it is the responsibility of the file system driver to process the user request. Note that the filename passed to the file system driver is the portion that was not parsed by the NT Object Manager: *\accounting\june-97.*

This is how user open/create requests end up in a file system driver. Understanding the sequence of operations that lead to the invoking of the file system create/open dispatch entry point will be quite valuable when we begin to explore the implementation of the file system create/open dispatch entry point and the file system mount logical volume implementation in greater detail.

Filename Handling for Network Redirectors

Earlier in this chapter, we saw how a network redirector is a kernel-mode software module that presents a file system interface to local users, but in reality communicates with server modules on remote nodes to obtain data from the remote shared logical volumes.

The Multiple Provider Router (MPR) and the Multiple Universal Naming Convention Provider (MUP) modules interact with the network redirector to present the appearance of a local file system to the user on the client machine. These components, in conjunction with a kernel-mode network redirector module, have the responsibility of integrating the name space of the remote (shared) logical volume file system into the local name space on the client node. Therefore, to design and develop a network redirector module for the Windows NT operating system, you will have to understand both of these components fairly well.

Multiple Provider Router (MPR)

The MPR module is a user-mode DLL executing on client nodes. It serves as a buffer between the common application utilities that are network-aware and the multiple network providers that may execute on the client node.

NOTE A network provider is a software module designed to work in close cooperation with the network redirector. The network provider serves as a sort of interface to the rest of the system, allowing network-aware applications to request some common functionality from the network redirector in a standard fashion, without having to develop code specific to each type of redirector that may be installed on the client node.

You may be wondering how there can be multiple network redirectors on a single client node. Having multiple redirectors installed and running on a client node is not really an unusual condition if you stop to think about it. The Windows NT operating system ships with the LAN Manager Redirector that is supplied with the operating system itself. In addition, there are commercially available implementations of the Network File System (NFS) protocol as well as the

Distributed File System (DPS) protocol that are also implemented as network redirectors. Then, think about all of the third-party developers like yourself who design and implement a network file system, and you could easily end up with a situation where a client node will have more than one network redirector installed.

So what exactly does the MPR do? Consider the net command that is available on your Windows NT client node. This command allows the user to create a new connection to a shared, remote network drive. Furthermore, it allows the user to obtain information about the connection to the remote node, browse shared network resources on remote nodes, delete the connection when it's no longer needed, and perform other similar tasks. As the user of the network or as an application developer who wishes to interact with the multiple network redirectors that may be installed on the machine, you would prefer to interact with the network redirectors in some standard manner, without dealing with the peculiarities of any particular network.

This is exactly what the MPR attempts to facilitate. The MPR has defined two sets of routines, each belonging to a distinct, well-defined interface. There is a set of network-independent APIs that are supported by the MPR DLL and are available to all Win32 application developers who wish to request services from a network redirector/provider. Similarly, there is another set of provider APIs that are invoked by the MPR DLL and must be implemented by the various network redirectors.

Therefore, a Win32 application trying to create a new network connection (for example) would invoke a standard Win32 API routine called WNetAddConnection()or WNetAddConnection2(). These functions are implemented within the MPR DLL. Upon receiving this request, the MPR DLL will invoke the NPAddConnection ( ) or an equivalent routine that must be provided by each network provider DLL that has registered itself with the MPR. Once such a request is received by the network provider DLL, the network provider can determine whether it will process the request, returning the results of the operation back to the MPR for subsequent forwarding onto the original requesting process, or whether it will allow the MPR to do the work. Note that in order to process requests, the network provider DLL will often invoke the kernel-mode network redirector software using file system control requests. Chapter 11, *Writing a File System Driver III,* explains how file system control requests are processed by the file system driver (redirector).

NOTE To register a network provider DLL with the MPR, the Registry on the client node must be modified. If you design and implement a network redirector and also decide to ship a network provider DLL with it, your installation program will probably perform all the appropriate modifications for you.

Appendix B, *MPR Support,* describes the modifications that must be made to the Registry in order to install your network provider DLL.

The order in which the various network provider DLLs are invoked is dependent upon the order in which the providers are listed in the Registry on the client node.

In the case of the NPAddConnection() request issued by the MPR to the network provider DLL, the DLL most likely submits the request to the kernel-mode redirector. The redirector attempts to contact the remote node specified in the arguments to the request, tries to locate the shared resource on the remote node, and also tries to make the connection on behalf of the requesting process.

If the request succeeds, and if the requesting process had specified it, the network provider DLL may also try to create a symbolic link as a drive letter (e.g., *X:*) to represent the newly created connection to the remote shared resource object. The symbolic link may refer either to a new device object created by the kernel-mode redirector, representing the new connection, or to the common redirector device object itself.* In either case, whenever the user's process attempts to access the name space below the *X:* drive letter, the request will be redirected by the I/O Manager to the network redirector in the kernel for further processing.

Consult Appendix B for a description of the functions that your network provider must implement in order to support the common Windows NT network-aware applications. If you implement a network provider DLL that supports the functions described, your network redirector will be able to take advantage of system-supplied utilities, such as the net command to add/delete/query connections to remote (shared) resources.

Multiple UNC Provider

The Windows NT platform also allows users to access remote (shared) resources using the Universal Naming Convention (UNC). This convention is pretty simple

---

* The network-provided DLL typically uses the Win32 function DefineDosDevices () to create the drive letter (symbolic link object type). Also note that most file systems and network redirectors create a named device object representing the file system device or the redirector device. Often, drive letters (symbolic links) for remote shared network drives refer to the network redirector device object.

in its design: each shared remote resource can be uniquely identified by the name \\server_name\shared_resource_name.

There are very few restrictions on the characters that can be used in either the server name and the shared resource name. You cannot use the "\" character as part of either the server name or the shared resource name, but most other common characters are allowed. The other restriction that you must be aware of is that the total length of the UNC name (including the name of the remote server and the name of the shared resource) cannot exceed 255 characters.

So when a user tries to access a remote shared resource by using a UNC name, how does the name get resolved?

Since UNC is Win32-specific, the Win32 subsystem is always looking for UNC names specified by a user process. Upon encountering such a name, the Win32 subsystem replaces the "\\" characters with the name \Device\UNC and then submits the request to the Windows NT Executive.

The \Device\UNC object type is really a symbolic link to the object \Device\Mup. The MUP driver is an extremely simple kernel-mode driver module (unlike the MPR module discussed above, which resides in user space) that has been described as a resource locator and is typically loaded automatically at system boot time. It creates a device object of type FILE_DEVICE_MULTI_UNC_PROVIDER during the driver initialization.* It also implements a create/open dispatch routine that is invoked whenever a create/open request targeted to the MUP driver is received, as in the case described above.

After the open request is received by the MUP driver, the MUP sends a special input/output control (IOCTL) to each network redirector that has registered itself with the MUP, asking the redirector whether it recognizes (and is willing to claim) some subset of the caller supplied name (i.e., *\server_name\shared_resource_name*\. . .).

Any redirector (or even more than one) can claim a portion of the remote resource name. The redirector recognizing the name must inform the MUP about the number of characters in the name string that it recognizes as a unique, valid, remote resource identifier. The first redirector that registers itself with the MUP has a higher priority than the next one to do so, and this ordering determines which redirector gets to process the user request, if more than one redirector recognizes the remote shared resource name.

---

* You will read in much greater detail about creating device objects and about device objects in general later in this book.

When any one redirector recognizes the name, the MUP prepends the name of the device object for the network redirector to the pathname string, replaces the name in the file object, and returns STATUS_REPARSE to the Object Manager. This time around, the request is directed to the network redirector that claimed the name for further processing. Now the MUP is completely out of the picture and will no longer be invoked for any operations pertaining to that particular create/open request.

The only other optimization performed by the MUP is to cache the portion of the name recognized by the redirector. The next time an open request is received beginning with the same string, the MUP checks its cache to see if the name is present, and if so, directly reroutes the request to the target network redirector device object without performing the tedious polling that it had done the first time around. Names are automatically discarded from the cache after some period of inactivity (typically if 15 minutes have elapsed since the name was last used in an open operation).

To work in conjunction with the MUP, your network redirector must do two things:

- Register itself with the MUP, using a system-supplied support routine called `FsRtlRegisterUncProvider()`. This typically is done by your driver at initialization.
- Respond to the special device control request issued by the MUP, asking your driver to check whether it recognizes a name.

Example code fragments are provided later in this book.

The next chapter discusses how you can incorporate structured exception handling and the various synchronization primitives available under Windows NT in your driver.