[Previous] [Next]

Chapter 12

File Systems

In this chapter, we present an overview of the file system formats supported by Microsoft Windows 2000. We then describe the types of file system drivers and their basic operation, including how they interact with other system components such as the memory manager and the cache manager. Windows 2000 includes a native file system format, called the NTFS file system. In the balance of the chapter, we focus on the on-disk layout of NTFS volumes and the advanced features of NTFS, such as compression, recoverability, quotas, and encryption.

To fully understand this chapter, you should be familiar with the terminology introduced in Chapter 10, including the terms *volume* and *partition*. You'll also need to be acquainted with these additional terms:

- *Sectors* are hardware-addressable blocks on a storage medium. Hard disks for x86 systems almost always define a 512-byte sector size. Thus, if the operating system wants to modify the 632nd byte on a disk, it must write a 512-byte block of data to the second sector on the disk.
- File system formats define the way that file data is stored on storage media and impact a file system's features. For example, a format that doesn't allow user permissions to be associated with files and directories can't support security. A file system format can also impose limits on the sizes of files and storage devices that the file system supports. Finally, some file system formats efficiently implement support for either large or small files or for large or small disks.
- Clusters are the addressable blocks that many file system formats use. Cluster size is always a multiple of the sector size, as shown in Figure 12-1. File system formats use clusters to manage disk space more efficiently; a cluster size that is larger than the sector size divides a disk into more manageable blocks. The potential trade-off of a larger cluster size is wasted disk space, or internal fragmentation, that results because file sizes typically aren't perfect multiples of cluster sizes.

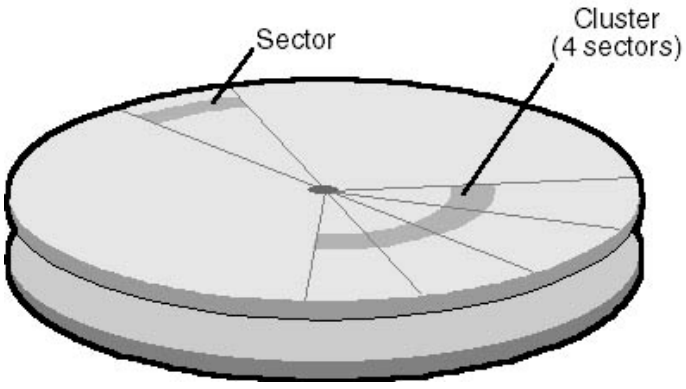


Figure 12-1 *Sectors and a cluster on a disk*

- *Metadata* is data stored on a volume in support of file system format management. It isn't typically made accessible to applications. Metadata includes the data that defines the placement of files and directories on a volume, for example.

[Previous] [Next]

Windows 2000 File System Formats

Windows 2000 includes support for the following file system formats:

- CDFS
- UDF
- FAT12, FAT16, and FAT32
- NTFS

Each of these formats is best suited for certain environments, as you'll see in the following sections.

CDFS

CDFS, or CD-ROM File System, is a relatively simple format that was defined in 1988 as the read-only formatting standard for CD-ROM media. Windows 2000 implements ISO 9660-compliant CDFS in \Winnt\System32\Drivers\Cdfs.sys, with long filename support defined by Level 2 of the ISO 9660 standard. Because of its simplicity, the CDFS format has a number of restrictions:

- Directory and file names must be fewer than 32 characters long.
- Directory trees can be no more than eight levels deep.

CDFS is considered a legacy format because the industry has adopted the Universal Disk Format (UDF) as the standard for read-only media.

UDF

The Windows 2000 UDF file system implementation is ISO 13346-compliant and supports UDF versions 1.02 and 1.5. OSTA (Optical Storage Technology Association) defined UDF in 1995 as a format to replace CDFS for magneto-optical storage media, mainly DVD-ROM. UDF is included in the DVD specification and is more flexible than CDFS. UDF file systems have the following traits:

- Filenames can be 255 characters long.
- The maximum path length is 1023 characters.
- Filenames can be upper and lower case.

Although the UDF format was designed with rewritable media in mind, the Windows 2000 UDF driver (\Winnt\System32\Drivers\Udfs.sys) provides read-only support.

FAT12, FAT16, and FAT32

Windows 2000 supports the FAT file system primarily to enable upgrades from other versions of Windows, for compatibility with other operating systems in multiboot systems, and as a floppy disk format. The Windows 2000 FAT file system driver is implemented in \Winnt\System32 \Drivers\Fastfat.sys.

Each FAT format includes a number that indicates the number of bits the format uses to identify clusters on a disk. FAT12's 12-bit cluster identifier limits a partition to storing a maximum of 2^{12} (4096) clusters. Windows 2000 uses cluster sizes from 512 bytes to 8 KB in size, which limits a

FAT12 volume size to 32 MB. Therefore, Windows 2000 uses FAT12 as the format for all 5¼-inch floppy disks and 3.5-inch floppy disks, which store up to 1.44 MB of data.

FAT16, with a 16-bit cluster identifier, can address 2^{16} (65,536) clusters. On Windows 2000, FAT16 cluster sizes range from 512 bytes (the sector size) to 64 KB, which limits FAT16 volume sizes to 4 GB. The cluster size Windows 2000 uses depends on the size of a volume. The various sizes are listed in Table 12-1. If you format a volume that is less than 16 MB as FAT by using the *format* command or the Disk Management snap-in, Windows 2000 uses the FAT12 format instead of FAT16.

Table 12-1 *Default FAT16 Cluster Sizes in Windows 2000*

Volume Size	Cluster Size
0-32 MB	512 bytes
33 MB-64 MB	1 KB
65 MB-128 MB	2 KB
129 MB-256 MB	4 KB
257 MB-511 MB	8 KB
512 MB-1023 MB	16 KB
1024 MB-2047 MB	32 KB
2048 MB-4095 MB	64 KB

A FAT volume is divided into several regions, which are shown in Figure 12-2. The file allocation table, which gives the FAT file system format its name, has one entry for each cluster on a volume. Because the file allocation table is critical to the successful interpretation of a volume's contents, the FAT format maintains two copies of the table so that if a file system driver or consistency-checking program (such as Chkdsk) can't access one (because of a bad disk sector, for example) it can read from the other.

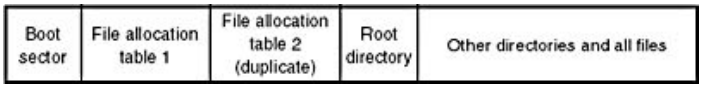


Figure 12-2 *FAT format organization*

Entries in the file allocation table define file-allocation chains (shown in Figure 12-3) for files and directories, where the links in the chain are indexes to the next cluster of a file's data. A file's directory entry stores the starting cluster of the file. The last entry of the file's allocation chain is the reserved value of 0xFFFF for FAT16 and 0xFFF for FAT12. The FAT entries for unused clusters have a value of 0. You can see in Figure 12-3 that FILE1 is assigned clusters 2, 3, and 4; FILE2 is fragmented and uses clusters 5, 6, and 8; and FILE3 uses only cluster 7.

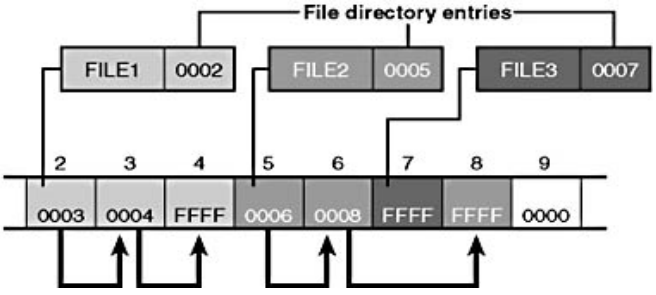


Figure 12-3 *Example FAT file-allocation chains*

The root directory of FAT12 and FAT16 volumes are preassigned enough space at the start of a volume to store 256 directory entries, which places an upper limit on the number of files and directories that can be stored in the root directory. (There's no preassigned space or size limit on FAT32 root directories.) A FAT directory entry is 32 bytes and stores a file's name, size, starting cluster, and time stamp (last-accessed, created, and so on) information. If a file has a name that is Unicode or that doesn't follow the MS-DOS 8.3 naming convention, additional directory entries are allocated to store the long filename. The supplementary entries precede the file's main entry. Figure 12-4 shows an example directory entry for a file named "The quick brown fox." The system has created a THEQUI~1.FOX 8.3 representation of the name (you don't see a "." in the directory entry because it is assumed to come after the eighth character) and used two more directory entries to store the Unicode long filename. Each row in the figure is made up of 16 bytes.

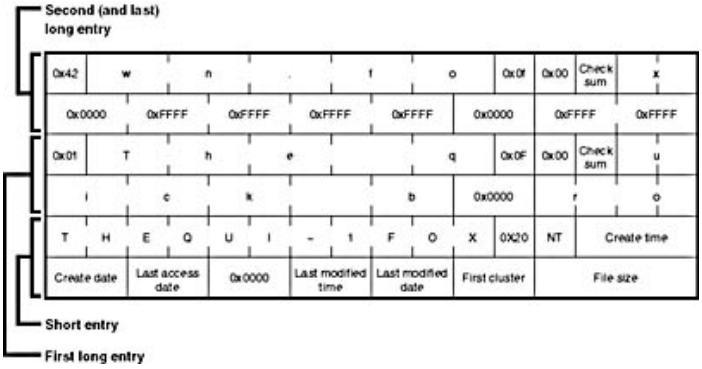


Figure 12-4 *FAT directory entry*

FAT32 is the most recently defined FAT-based file system format, and it's included with Windows 95 OSR2, Windows 98, and Windows Millennium Edition. FAT32 uses 32-bit cluster identifiers but reserves the high 4 bits, so in effect it has 28-bit cluster identifiers. Because FAT32 cluster sizes can be as large as 32 KB, FAT32 has a theoretical ability to address 8-terabyte (TB) volumes. Although Windows 2000 works with existing FAT32 volumes of larger sizes (created in other operating systems), it limits new FAT32 volumes to a maximum of 32 GB. FAT32's higher potential cluster numbers let it more efficiently manage disks than FAT16; it can handle up to 128-MB volumes with 512-byte clusters. Table 12-2 shows default cluster sizes for FAT32 volumes.

Table 12-2 *Default Cluster Sizes for FAT32 Volumes*

Partition Size	Cluster Size
32 MB to 8 GB	4 KB
8 GB to 16 GB	8 KB
16 GB to 32 GB	16 KB
32 GB	32 KB

Besides the higher limit on cluster numbers, other advantages FAT32 has over FAT12 and FAT16 include the fact that the FAT32 root directory isn't stored at a predefined location on the volume, the root directory doesn't have an upper limit on its size, and FAT32 stores a second copy of the boot sector for reliability. A limitation FAT32 shares with FAT16 is that the maximum file size is 4 GB, because directories store file sizes as 32-bit values.

NTFS

As we said at the beginning of the chapter, the NTFS file system is the native file system format of Windows 2000. NTFS uses 64-bit cluster indexes. This capacity gives NTFS the ability to address volumes of up to 16 exabytes (16 billion GB); however, Windows 2000 limits the size of an NTFS volume to that addressable with 32-bit clusters, which is 128 TB (using 64-KB clusters). Table 12-3 shows the default cluster sizes for NTFS volumes. (You can override the default when you format an NTFS volume.)

Table 12-3 *Default Cluster Sizes for NTFS Volumes*

Volume Size	Default Cluster Size
512 MB or less	512 bytes
513 MB-1024 MB (1 GB)	1 KB
1025 MB-2048 MB (2 GB)	2 KB
Greater than 2048 MB	4 KB

NTFS includes a number of advanced features, such as file and directory security, disk quotas, file compression, directory-based symbolic links, and encryption. One of its most significant features is *recoverability*. If a system is halted unexpectedly, the metadata of a FAT volume can be left in an inconsistent state, leading to the corruption of large amounts of file and directory data. NTFS logs changes to metadata in a transactional manner so that file system structures can be repaired to a consistent state with no loss of file or directory structure information. (File data can be lost, however.)

We'll describe NTFS data structures and advanced features in detail later in this chapter.

[Previous] [Next]

File System Driver Architecture

File system drivers (FSDs) manage file system formats. Although FSDs run in kernel mode, they differ in a number of ways from standard kernel-mode drivers. Perhaps most significant, they must register as an FSD with the I/O manager and they interact more extensively with the memory manager and the cache manager. Thus, they use a superset of the exported Ntoskrnl functions that standard drivers use. Whereas you need the Windows 2000 DDK in order to build standard kernel-mode drivers, you must have the Windows 2000 Installable File System (IFS) Kit to build file system drivers. (See Chapter 1 for more information on the DDK, and see *www.microsoft.com/ddk/ifskit* for more information on the IFS Kit.)

Windows 2000 has two different types of file system drivers:

- *Local FSDs* manage volumes directly connected to the computer.
- *Network FSDs* allow users to access data volumes connected to remote computers.

Local FSDs

Local FSDs include Ntfs.sys, Fastfat.sys, Udfs.sys, Cdfs.sys, and the Raw FSD (integrated in Ntoskrnl.exe). Figure 12-5 shows a simplified view of how local FSDs interact with the I/O manager and storage device drivers. As we described in the section "Volume Mounting" in Chapter 10, a local FSD is responsible for registering with the I/O manager. Once the FSD is registered, the I/O manager can call on it to perform volume recognition when applications or the system initially

access the volumes. Volume recognition involves an examination of a volume's boot sector and often, as a consistency check, the file system metadata.

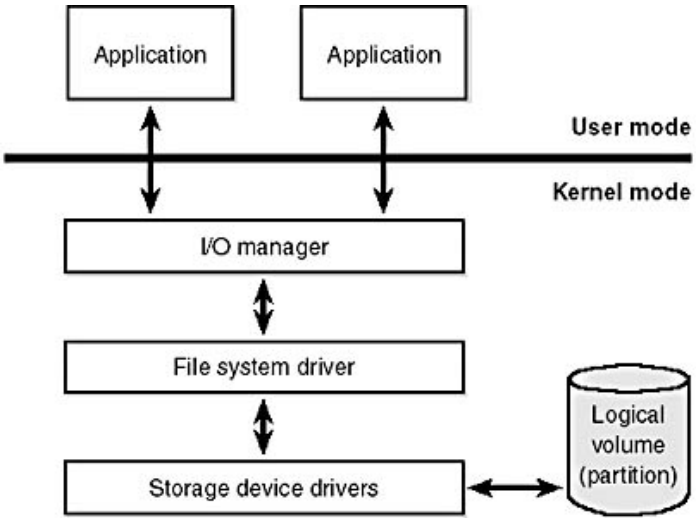


Figure 12-5 *Local FSD*

The first sector of every Windows 2000-supported file system format is reserved as the volume's boot sector. A boot sector contains enough information so that a local FSD can both identify the volume on which the sector resides as containing a format that the FSD manages and locate any other metadata necessary to identify where metadata is stored on the volume.

When a local FSD recognizes a volume, it creates a device object that represents the mounted file system format. The I/O manager makes a connection through the volume parameter block (VPB) between the volume's device object (which is created by a storage device) and the device object that the FSD created. The VPB's connection results in the I/O manager redirecting I/O requests targeted at the volume device object to the FSD device object. (See Chapter 10 for more information on VPBs.)

To improve performance, local FSDs usually use the cache manager to cache file system data, including metadata. They also integrate with the memory manager so that mapped files are implemented correctly. For example, they must query the memory manager whenever an application attempts to truncate a file in order to verify that no processes have mapped the part of the file beyond the truncation point. Windows 2000 doesn't permit file data that is mapped by an application to be deleted either through truncation or file deletion.

Local FSDs also support file system dismount operations, which permit the system to disconnect the FSD from the volume object. A dismount occurs whenever an application requires raw access to the on-disk contents of a volume or the media associated with a volume is changed. The first time an application accesses the media after a dismount, the I/O manager reinitiates a volume mount operation for the media.

Remote FSDs

Remote FSDs consist of two components: a client and a server. A client-side remote FSD allows applications to access remote files and directories. The client FSD accepts I/O requests from applications and translates them into network file system protocol commands that the FSD sends across the network to a server-side remote FSD. A server-side FSD listens for commands coming from a network connection and fulfills them by issuing I/O requests to the local FSD that manages the volume on which the file or directory that the command is intended for resides. Figure 12-6 shows the relationship between the client and server sides of a remote FSD interaction.

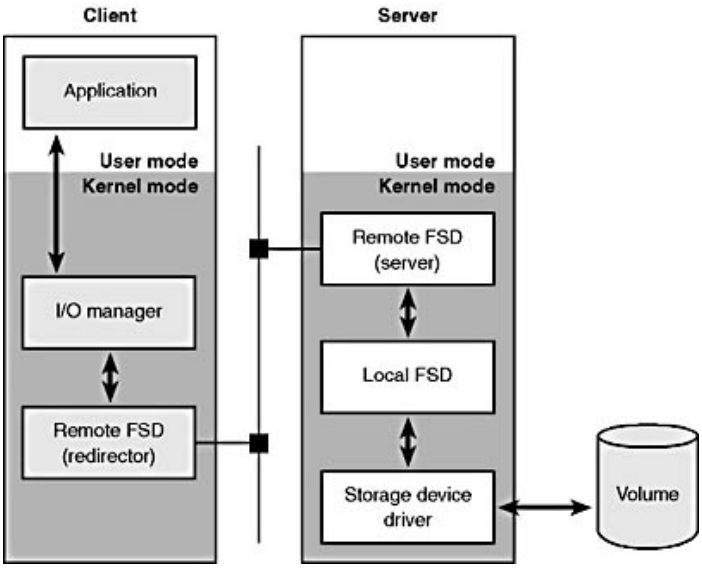


Figure 12-6 *Remote FSD operation*

Windows 2000 includes a client-side remote FSD named LANMan Redirector (redirector) and a server-side remote FSD server named LANMan Server (server). The redirector is implemented as a port/miniport driver combination, where the port driver (\Winnt\System32\Drivers\Rdbss.sys) is implemented as a driver subroutine library and the miniport (\Winnt\System32\Drivers\Mrxsmb.sys) uses services implemented by the port driver. The port/miniport model simplifies redirector development because the port driver, which all remote FSD miniport drivers share, handles many of the mundane details involved with interfacing a client-side remote FSD to the Windows 2000 I/O manager. In addition to the FSD components, both LANMan Redirector and LANMan Server include Win32 services named Workstation and Server, respectively.

Windows 2000 relies on the Common Internet File System (CIFS) protocol to format messages exchanged between the redirector and the server. CIFS is an enhanced version of Microsoft's Server Message Block (SMB) protocol. (For more information on CIFS, go to *www.cifs.com*.)

Like local FSDs, client-side remote FSDs usually use cache manager services to locally cache file data belonging to remote files and directories. However, client-side remote FSDs must implement a distributed cache coherency protocol, called *oplocks* (opportunistic locking), so that the data an application sees when it accesses a remote file is the same as the data applications running on other computers that are accessing the same file see. Although server-side remote FSDs participate in maintaining cache coherency across their clients, they don't cache data from the local FSDs, because local FSDs cache their own data. (Oplocks are described further in the section "Distributed File Caching" in Chapter 13.)

NOTE

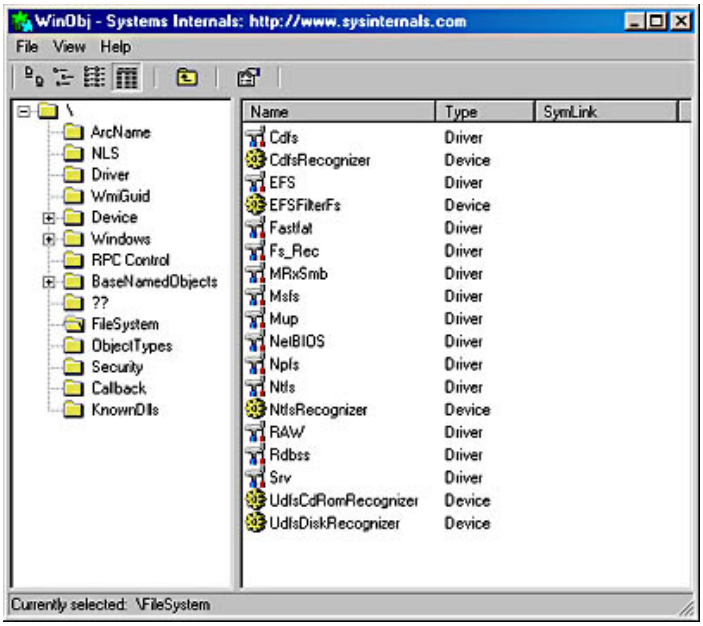
A filter driver that layers over a file system driver is called *file-system filter driver*. The ability to see all file system requests and optionally modify or complete them enables a range of applications, including on-access virus scanners and remote file replication services. Filemon, on the companion CD as \Sysint\Filemon, is an example of a file-system filter driver that is a *pass-through* filter. Filemon displays file system activity in real time without modifying the requests it sees.

EXPERIMENT

Viewing the List of Registered File Systems

When the I/O manager loads a device driver into memory, it typically names the driver

object it creates to represent the driver so that it's placed in the \Drivers object manager directory. The driver objects for any driver the I/O manager loads that have a Type attribute value of SERVICE_FILE_SYSTEM_DRIVER (2) are placed in the \FileSystem directory by the I/O manager. Thus, using a tool like Winobj (on the companion CD in \Sysint\Winobj.exe), you can see the file systems that have registered on a system, as shown in the following screen shot. (Note that some file system drivers also place device objects in the \FileSystem directory.)



File System Operation

Applications and the system access files in two ways: directly, via file I/O functions (such as *ReadFile* and *WriteFile*), and indirectly, by reading or writing a portion of their address space that represents a mapped file section. (See Chapter 7 for more information on mapped files.) Figure 12-7 is a simplified diagram that shows the components involved in these file system operations and the ways in which they interact. As you can see, an FSD can be invoked through several paths:

- From a user or system thread performing explicit file I/O
- From the memory manager's modified page writer
- Indirectly from the cache manager's lazy writer
- Indirectly from the cache manager's read-ahead thread
- From the memory manager's page fault handler

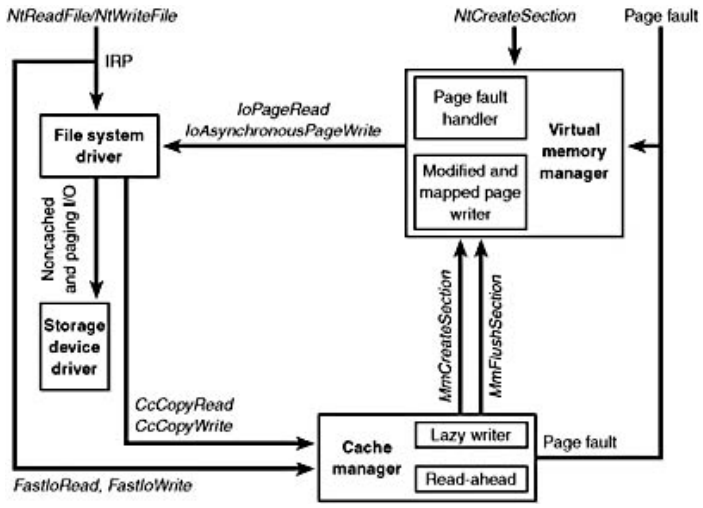


Figure 12-7 *Components involved in file system I/O*

The following sections describe the circumstances surrounding each of these scenarios and the steps FSDs typically take in response to each one. You'll see how much FSDs rely on the memory manager and the cache manager.

Explicit File I/O

The most obvious way an application accesses files is by calling Win32 I/O functions such as *CreateFile*, *ReadFile*, and *WriteFile*. An application opens a file with *CreateFile* and then reads, writes, or deletes the file by passing the handle returned from *CreateFile* to other Win32 functions. The *CreateFile* function, which is implemented in the Kernel32.dll Win32 client-side DLL, invokes the native function *NtCreateFile*, forming a complete root-relative pathname for the path that the application passed to it (processing .."" and "." symbols in the pathname) and prepending the path with "\??" (for example, \??\C:\Susan\Todo.txt).

The *NtCreateFile* system service uses *ObOpenObjectByName* to open the file, which parses the name starting with the object manager root directory and the first component of the path name ("??"). \?? is a subdirectory that contains symbolic links representing volumes that are assigned drive letters (and symbolic links to serial ports and other device objects that Win32 applications access directly), so the "C:" component of the name resolves to the \??\C: symbolic link. The symbolic link points to a volume device object under \Device, so when the object manager encounters the volume object, the object manager hands the rest of the pathname to the parse function that the I/O manager has registered for device objects, *IopParseDevice*. (In volumes on dynamic disks, a symbolic link points to an intermediary symbolic link, which points to a volume device object.) Figure 12-8 shows how volume objects are accessed through the object manager namespace. The figure shows how the \??\C: symbolic link points to the \Device\HarddiskVolume1 volume device object.

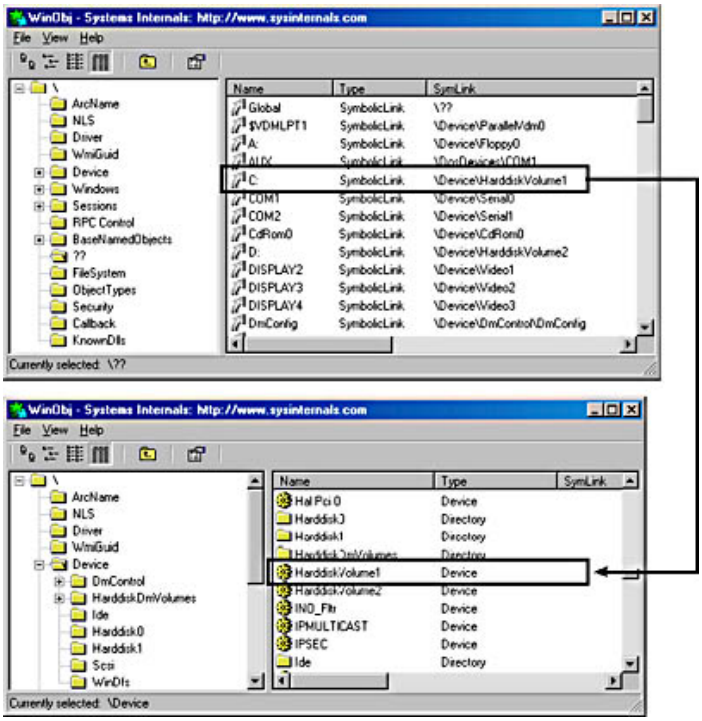


Figure 12-8 *Drive-letter name resolution*

After locking the caller's security context and obtaining security information from the caller's token, *IopParseDevice* creates an I/O request packet (IRP) of type IRP_MJ_CREATE, creates a file object that stores the name of the file being opened, follows the VPB of the volume device object to find the volume's mounted file system device object, and uses *IoCallDriver* to pass the IRP to the file system driver that owns the file system device object.

When an FSD receives an IRP_MJ_CREATE IRP, it looks up the specified file, performs security validation, and if the file exists and the user has permission to access the file in the way requested, returns a success code. The object manager creates a handle for the file object in the process's handle table and the handle propagates back through the calling chain, finally reaching the application as a return parameter from *CreateFile*. If the file system fails the create, the I/O manager deletes the file object it created for it.

We've skipped over the details of how the FSD locates the file being opened on the volume, but a *ReadFile* function call operation shares many of the FSD's interactions with the cache manager and storage driver. The path into the kernel taken as the result of a call to *ReadFile* is the same as for a call to *CreateFile*, but the *NtReadFile* system service doesn't need to perform a name lookup—it calls on the object manager to translate the handle passed from *ReadFile* into a file object pointer. If the handle indicates that the caller obtained permission to read the file when the file was opened, *NtReadFile* proceeds to create an IRP of type IRP_MJ_READ and sends it to the FSD on which the file resides. *NtReadFile* obtains the FSD's device object, which is stored in the file object, and calls *IoCallDriver*, and the I/O manager locates the FSD from the device object and gives the IRP to the FSD.

If the file being read can be cached (the FILE_FLAG_NO_BUFFERING flag wasn't passed to *CreateFile* when the file was opened), the FSD checks to see whether caching has already been initiated for the file object. The *PrivateCacheMap* field in a file object points to a private cache map data structure (which we described in Chapter 11), if caching is initiated for a file object. If the FSD hasn't initialized caching for the file object (which it does the first time a file object is read from or written to), the *PrivateCacheMap* field will be null. The FSD calls the cache manager *CcInitializeCacheMap* function to initialize caching, which involves the cache manager creating a private cache map and, if another file object referring to the same file hasn't initiated caching, a

shared cache map and a section object.

After it has verified that caching is enabled for the file, the FSD copies the requested file data from the cache manager's virtual memory to the buffer that the thread passed to *ReadFile*. The file system performs the copy within a try/except block so that it catches any faults that are the result of an invalid application buffer. The function the file system uses to perform the copy is the cache manager's *CcCopyRead* function. *CcCopyRead* takes as parameters a file object, file offset, and length.

When the cache manager executes *CcCopyRead*, it retrieves a pointer to a shared cache map, which is stored in the file object. Recall from Chapter 11 that a shared cache map stores pointers to virtual address control blocks (VACBs), with one VACB entry per 256-KB block of the file. If the VACB pointer for a portion of a file being read is null, *CcCopyRead* allocates a VACB, reserving a 256-KB view in the cache manager's virtual address space, and maps (using *MmCreateSection* and *MmMapViewOfSection*) the specified portion of the file into the view. Then *CcCopyRead* simply copies the file data from the mapped view to the buffer it was passed (the buffer originally passed to *ReadFile*). If the file data isn't in physical memory, the copy operation generates page faults, which are serviced by *MmAccessFault*.

When a page fault occurs, *MmAccessFault* examines the virtual address that caused the fault and locates the virtual address descriptor (VAD) in the VAD tree of the process that caused the fault. (See Chapter 7 for more information on VAD trees.) In this scenario, the VAD describes the cache manager's mapped view of the file being read, so *MmAccessFault* calls *MiDispatchFault* to handle a page fault on a valid virtual memory address. *MiDispatchFault* locates the control area (which the VAD points to) and through the control area finds a file object representing the open file. (If the file has been opened more than once, there might be a list of file objects linked through pointers in their private cache maps.)

With the file object in hand, *MiDispatchFault* calls the I/O manager function *IoPageRead* to build an IRP (of type IRP_MJ_READ) and sends the IRP to the FSD that owns the device object the file object points to. Thus, the file system is reentered to read the data that it requested via *CcCopyRead*, but this time the IRP is marked as noncached and paging I/O. These flags signal the FSD that it should retrieve file data directly from disk, and it does so by determining which clusters on disk contain the requested data and sending IRPs to the volume manager that owns the volume device object on which the file resides. The volume parameter block (VPB) field in the FSD's device object points to the volume device object.

The virtual memory manager waits for the FSD to complete the IRP read and then returns control to the cache manager, which continues the copy operation that was interrupted by a page fault. When the *CcCopyRead* completes, the FSD returns control to the thread that called *NtReadFile*, having copied the requested file data—with the aid of the cache manager and the virtual memory manager—to the thread's buffer.

The path for *WriteFile* is similar except that the *NtWriteFile* system service generates an IRP of type IRP_MJ_WRITE and the FSD calls *CcCopyWrite* instead of *CcCopyRead*. *CcCopyWrite*, like *CcCopyRead*, ensures that the portions of the file being written are mapped into the cache and then copies to the cache the buffer passed to *WriteFile*.

If a file's data is already stored in the system's working set, there are several variants on the scenario we've just described. If a file's data is already stored in the cache, *CcCopyRead* doesn't incur page faults. Also, under certain conditions, *NtReadFile* and *NtWriteFile* call an FSD's fast I/O entry point instead of immediately building and sending an IRP to the FSD. Some of these conditions follow: the portion of the file being read must reside in the first 4 GB of the file, the file can have no locks, and the portion of the file being read or written must fall within the file's currently allocated size.

The fast I/O read and write entry points for most FSDs call the cache manager's *CcFastCopyRead* and *CcFastCopyWrite* functions. These variants on the standard copy routines ensure that the file's data is mapped in the file system cache before performing a copy operation. If this condition isn't met, *CcFastCopyRead* and *CcFastCopyWrite* indicate that fast I/O isn't possible. When fast I/O isn't possible, *NtReadFile* and *NtWriteFile* fall back on creating an IRP.

Memory Manager's Modified and Mapped Page Writer

The memory manager's modified and mapped page writer threads wake up periodically to flush modified pages. The threads call *IoAsynchronousPageWrite* to create IRPs of type IRP_MJ_WRITE and write pages to either a paging file or a file that was modified after being mapped. Like the IRPs that *MiDispatchFault* creates, these IRPs are flagged as noncached and paging I/O. Thus, an FSD bypasses the file system cache and issues IRPs directly to a storage driver to write the memory to disk.

Cache Manager's Lazy Writer

The cache manager's lazy writer thread also plays a role in writing modified pages because it periodically flushes views of file sections mapped in the cache that it knows are dirty. The flush operation, which the cache manager performs by calling *MmFlushSection*, triggers the memory manager to write any modified pages in the portion of the section being flushed to disk. Like the modified and mapped page writers, *MmFlushSection* uses *IoAsynchronousPageWrite* to send the data to the FSD.

Cache Manager's Read-Ahead Thread

The cache manager includes a thread that is responsible for attempting to read data from files before an application, a driver, or a system thread explicitly requests it. The read-ahead thread uses the history of read operations that were performed on a file, which are stored in a file object's private cache map, to determine how much data to read. When the thread performs a read-ahead, it simply maps the portion of the file it wants to read into the cache (allocating VACBs as necessary) and touches the mapped data. The page faults caused by the memory accesses invoke the page fault handler, which reads the pages into the system's working set.

Memory Manager's Page Fault Handler

We described how the page fault handler is used in the context of explicit file I/O and cache manager read-ahead, but it is also invoked whenever any application accesses virtual memory that is a view of a mapped file and encounters pages that represent portions of a file that aren't part of the application's working set. The memory manager's *MmAccessFault* handler follows the same steps it does when the cache manager generates a page fault from *CcCopyRead* or *CcCopyWrite*, sending IRPs via *IoPageRead* to the file system on which the file is stored.

[Previous] [Next]

NTFS Design Goals and Features

In the following section, we'll look at the requirements that drove the design of NTFS. Then in the subsequent section, we'll examine the advanced features of NTFS.

High-End File System Requirements

From the start, NTFS was designed to include features required of an enterprise-class file system. To minimize data loss in the face of an unexpected system outage or crash, a file system must ensure

that the integrity of the file system's metadata be guaranteed at all times, and to protect sensitive data from unauthorized access, a file system must have an integrated security model. Finally, a file system must allow for software-based data redundancy as a low-cost alternative to hardware-redundant solutions for protecting user data. In this section, you'll find out how NTFS implements each of these capabilities.

Recoverability

To address the requirement for reliable data storage and data access, NTFS provides file system recovery based on the concept of an *atomic transaction*. Atomic transactions are a technique for handling modifications to a database so that system failures don't affect the correctness or integrity of the database. The basic tenet of atomic transactions is that some database operations, called *transactions*, are all-or-nothing propositions. (A transaction is defined as an I/O operation that alters file system data or changes the volume's directory structure.) The separate disk updates that make up the transaction must be executed *atomically*; that is, once the transaction begins to execute, all its disk updates must be completed. If a system failure interrupts the transaction, the part that has been completed must be undone, or *rolled back*. The rollback operation returns the database to a previously known and consistent state, as if the transaction had never occurred.

NTFS uses atomic transactions to implement its file system recovery feature. If a program initiates an I/O operation that alters the structure of an NTFS drive—that is, changes the directory structure, extends a file, allocates space for a new file, and so on—NTFS treats that operation as an atomic transaction. It guarantees that the transaction is either completed or, if the system fails while executing the transaction, rolled back. The details of how NTFS does this are explained in the section "NTFS Recovery Support."

In addition, NTFS uses redundant storage for vital file system information so that if a sector on the disk goes bad, NTFS can still access the volume's critical file system data. This redundancy of file system data contrasts with the on-disk structures of both the FAT file system and the HPFS file system (OS/2's native file system format), which have single sectors containing critical file system data. On these file systems, if a read error occurs in one of those sectors an entire volume is lost.

Security

Security in NTFS is derived directly from the Windows 2000 object model. Files and directories are protected from being accessed by unauthorized users. (For more information on Windows 2000 security, see Chapter 8.) An open file is implemented as a file object with a security descriptor stored on disk as a part of the file. Before a process can open a handle to any object, including a file object, the Windows 2000 security system verifies that the process has appropriate authorization to do so. The security descriptor, combined with the requirement that a user log on to the system and provide an identifying password, ensures that no process can access a file unless given specific permission to do so by a system administrator or by the file's owner. (For more information about security descriptors, see the section "Security Descriptors and Access Control" in Chapter 8, and for more details about file objects, see the section "File Objects" in Chapter 9.)

Data Redundancy and Fault Tolerance

In addition to recoverability of file system data, some customers require that their own data not be endangered by a power outage or catastrophic disk failure. The NTFS recovery capabilities do ensure that the file system on a volume remains accessible, but they make no guarantees for complete recovery of user files. Protection for applications that can't risk losing file data is provided through data redundancy.

Data redundancy for user files is implemented via the Windows 2000 layered driver model (explained in Chapter 9), which provides fault tolerant disk support. NTFS communicates with a

volume manager, which in turn communicates with a hard disk driver to write data to disk. A volume manager can *mirror*, or duplicate, data from one disk onto another disk so that a redundant copy can always be retrieved. This support is commonly called *RAID level 1*. Volume managers also allow data to be written in *stripes* across three or more disks, using the equivalent of one disk to maintain parity information. If the data on one disk is lost or becomes inaccessible, the driver can reconstruct the disk's contents by means of exclusive-OR operations. This support is called *RAID level 5*. (See Chapter 10 for more information on striped volumes, mirrored volumes, and RAID-5 volumes.)

Advanced Features of NTFS

In addition to NTFS being recoverable, secure, reliable, and efficient for mission-critical systems, it includes the following advanced features that allow it to support a broad range of applications. Some of these features are exposed as APIs for applications to leverage, and others are internal features:

- Multiple data streams
- Unicode-based names
- General indexing facility
- Dynamic bad-cluster remapping
- Hard links and junctions
- Compression and sparse files
- Change logging
- Per-user volume quotas
- Link tracking
- Encryption
- POSIX support
- Defragmentation

The following sections provide an overview of these features.

Multiple Data Streams

In NTFS, each unit of information associated with a file, including its name, its owner, its time stamps, its contents, and so on, is implemented as a file attribute (NTFS object attribute). Each attribute consists of a single *stream*, that is, a simple sequence of bytes. This generic implementation makes it easy to add more attributes (and therefore more streams) to a file. Because a file's data is "just another attribute" of the file and because new attributes can be added, NTFS files (and file directories) can contain multiple data streams.

An NTFS file has one default data stream, which has no name. An application can create additional, named data streams and access them by referring to their names. To avoid altering the Microsoft Win32 I/O APIs, which take a string as a filename argument, the name of the data stream is specified by appending a colon (:) to the filename. Because the colon is a reserved character, it can serve as a separator between the filename and the data stream name, as illustrated in this example:

```
myfile.dat:stream2
```

Each stream has a separate allocation size (how much disk space has been reserved for it), actual size (how many bytes the caller has used), and valid data length (how much of the stream has been initialized). In addition, each stream is given a separate file lock that is used to lock byte ranges and to allow concurrent access.

One component in Windows 2000 that uses multiple data streams is the Apple Macintosh file server support that comes with Windows 2000 Server. Macintosh systems use two streams per file—one to store data and the other to store resource information, such as the file type and the icon used to represent the file. Because NTFS allows multiple data streams, a Macintosh user can copy an entire Macintosh folder to a Windows 2000 Server, and another Macintosh user can copy the folder from the server without losing resource information.

Windows Explorer is another application that uses streams. When you right-click on an NTFS file and select Properties, the Summary tab of the resulting dialog box lets you associate information with the file, such as a title, subject, author, and keywords. Windows Explorer stores the information in an alternate stream it adds to the file, named "Summary Information."

Other applications can use the multiple data stream feature as well. A backup utility, for example, might use an extra data stream to store backup-specific time stamps on files. Or an archival utility might implement hierarchical storage in which files that are older than a certain date or that haven't been accessed for a specified period of time are moved to tape. The utility could copy the file to tape, set the file's default data stream to 0, and add a data stream that specifies the name and location of the tape on which the file is stored.

EXPERIMENT

Looking at Streams

Most Windows 2000 applications aren't designed to work with alternate named streams, but both the *echo* and the *more* commands are. Thus, a simple way to view streams in action is to create a named stream using *echo* and then display it using *more*. The following command sequence creates a file named test with a stream named stream:

```
C:\>echo hello > test:stream
C:\>more < test:stream
hello
C:\>
```

If you perform a directory listing, test's file size doesn't reflect the data stored in the alternate stream because NTFS returns the size of only the unnamed data stream for file query operations, including directory listings.

```
C:\>dir test
 Volume in drive C is WINDOWS
 Volume Serial Number is 3991-3040

 Directory of C:\

08/01/00  02:37p                     0 test
               1 File(s)              0 bytes
                             112,558,080 bytes free
C:\>
```

Unicode-Based Names

Like Windows 2000 as a whole, NTFS is fully Unicode enabled, using Unicode characters to store names of files, directories, and volumes. Unicode, a 16-bit character-coding scheme, allows each character in each of the world's major languages to be uniquely represented, which aids in moving data easily from one country to another. Unicode is an improvement over the traditional representation of international characters—using a double-byte coding scheme that stores some characters in 8 bits and others in 16 bits, a technique that requires loading various code pages to establish the available characters. Because Unicode has a unique representation for each character, it doesn't depend on which code page is loaded. Each directory and filename in a path can be as many as 255 characters long and can contain Unicode characters, embedded spaces, and multiple periods.

General Indexing Facility

The NTFS architecture is structured to allow indexing of file attributes on a disk volume. This structure enables the file system to efficiently locate files that match certain criteria—for example, all the files in a particular directory. The FAT file system indexes filenames but doesn't sort them, making lookups in large directories slow.

Several NTFS features take advantage of general indexing, including consolidated security descriptors, in which the security descriptors of a volume's files and directories are stored in a single internal stream, have duplicates removed, and are indexed using an internal security identifier that NTFS defines.

Dynamic Bad-Cluster Remapping

Ordinarily, if a program tries to read data from a bad disk sector, the read operation fails and the data in the allocated cluster becomes inaccessible. If the disk is formatted as a fault tolerant NTFS volume, however, the Windows 2000 fault tolerant driver dynamically retrieves a good copy of the data that was stored on the bad sector and then sends NTFS a warning that the sector is bad. NTFS allocates a new cluster, replacing the cluster in which the bad sector resides, and copies the data to the new cluster. It flags the bad cluster and no longer uses it. This data recovery and dynamic bad-cluster remapping is an especially useful feature for file servers and fault tolerant systems or for any application that can't afford to lose data. If the volume manager isn't loaded when a sector goes bad, NTFS still replaces the cluster and doesn't reuse it, but it can't recover the data that was on the bad sector.

Hard Links and Junctions

A hard link allows multiple paths to refer to the same file or directory. If you create a hard link named C:\Users\Documents\Spec.doc that refers to the existing file C:\My Documents\Spec.doc, the two paths link to the same on-disk file and you can make changes to the file using either path. Processes can create hard links with the Win32 *CreateHardLink* function or the *ln* POSIX function.

EXPERIMENT

Creating a Hard Link

Although applications can use the Win32 function *CreateHardLink* to create a hard link, no tools use this function. However, you can create a hard link by using the POSIX *ln* utility in the Windows 2000 resource kits. The POSIX tools can't be installed through the resource kit setup program, so you'll need to copy them manually from the \Apps\Posix directory in the resource kit CDs.

In addition to hard links, NTFS supports another type of redirectioncalled junctions. Junctions, also

called symbolic links, allow a directory to redirect file or directory pathname translation to an alternate directory. For example, if the path C:\Drivers is a junction that redirects to C:\Winnt\System32\Drivers, an application reading C:\Drivers\Ntfs.sys actually reads C:\Winnt\System\Drivers\Ntfs.sys. Junctions are a useful way to lift directories that are deep in a directory tree to a more convenient depth without disturbing the original tree's structure or contents. The example just cited lifts the drivers directory to the volume's root directory, reducing the directory depth of Ntfs.sys from three levels to one when Ntfs.sys is accessed through the junction. You can't use junctions to link to remote directories—only to directories on local volumes.

Junctions are based on an NTFS mechanism called *reparse points.* (Reparse points are discussed further in the section "Reparse Points" later in this chapter.) A reparse point is a file or directory that has a block of data called *reparse data* associated with it. Reparse data is user-defined data about the file or directory, such as its state or location, that can be read from the reparse point by the application that created the data, a file system filter driver, or the I/O manager. When NTFS encounters a reparse point during a file or directory lookup, it returns a *reparse status code*, which signals file system filter drivers that are attached to the volume, and the I/O manager, to examine the reparse data. Each reparse point type has a unique *reparse tag.* The reparse tag allows the component responsible for interpreting the reparse point's reparse data to recognize the reparse point without having to check the reparse data. A reparse tag owner, either a file system filter driver or the I/O manager, can choose one of the following options when it recognizes reparse data:

- The reparse tag owner can manipulate the pathname specified in the file I/O operation that crosses the reparse point and let the I/O operation reissue with the altered pathname. Junctions take this approach to redirect a directory lookup, for example.
- The reparse tag owner can remove the reparse point from the file, alter the file in some way, and then reissue the file I/O operation. The Windows 2000 Hierarchical Storage Management (HSM) system uses reparse points in this way. HSM archives files by moving their contents to tape, leaving reparse points in their place. When a process accesses a file that has been archived, the HSM filter driver (\Winnt\System32\Drivers\Rsfilter.sys) removes the reparse point from the file, reads the file's data from the archival media, and reissues the access. Thus, the retrieval of the offline data is transparent to a process accessing an archived file.

There are no Win32 functions for creating reparse points. Instead, processes must use the FSCTL_SET_REPARSE_POINT file system control code with the Win32 *DeviceIoControl* function. A process can query a reparse point's contents with the FSCTL_GET_REPARSE_POINT file system control code. The FILE_ATTRIBUTE_REPARSE_POINT flag is set in a reparse point's file attributes, so applications can check for reparse points by using the Win32 *GetFileAttributes* function.

EXPERIMENT

Creating a Junction

Windows 2000 doesn't include any tools for creating junctions, but you can create a junction with either the Junction tool on the companion CD (\Sysint\Junction.exe) or the Windows 2000 resource kits tool Linkd. The Linkd tool also lets you view the definition of existing junctions, and Junction lets you view information about junctions and other reparse point tags.

Compression and Sparse Files

NTFS supports compression of file data. Because NTFS performs compression and decompression procedures transparently, applications don't have to be modified to take advantage of this feature. Directories can also be compressed, which means that any files subsequently created in the directory

are compressed.

Applications compress and decompress files by passing *DeviceIoControl* the FSCTL_SET_COMPRESSION file system control code. They query the compression state of a file or directory with the FSCTL_GET_COMPRESSION file system control code. A file or directory that is compressed has the FILE_ATTRIBUTE_COMPRESSED flag set in its attributes, so applications can also determine a file or directory's compression state with *GetFileAttributes*.

A second type of compression is known as *sparse files*. If a file is marked as sparse, NTFS doesn't allocate space on a volume for portions of the file that an application designates as empty. NTFS returns 0-filled buffers when an application reads from empty areas of a sparse file. This type of compression can be useful for client/server applications that implement circular-buffer logging, in which the server records information to a file and clients asynchronously read the information. Because the information that the server writes isn't needed after a client has read it, there's no need to store the information in the file. By making such a file sparse, the client can specify the portions of the file it reads as empty, freeing up space on the volume. The server can continue to append new information to the file, without fear that the file will grow to consume all available space on the volume.

As for compressed files, NTFS manages sparse files transparently. Applications specify a file's sparseness state by passing the FSCTL_SET_SPARSE file system control code to *DeviceIoControl*. To set a range of a file to empty, they use the FSCTL_SET_ZERO_DATA code, and they can ask NTFS for a description of what parts of a file are sparse by using FSCTL_QUERY_ALLOCATED_RANGES. One application of sparse files is the NTFS *change journal*, described next.

Change Logging

Many types of applications need to monitor volumes for file and directory changes. For example, an automatic backup program might perform an initial full backup and then incremental backups based on file changes. An obvious way for an application to monitor a volume for changes is for it to scan the volume, recording the state of files and directories, and on a subsequent scan detect differences. This process can adversely affect system performance, however, especially on computers with thousands or tens of thousands of files.

An alternate approach is for an application to register a directory notification by using the *FindFirstChangeNotification* or *ReadDirectoryChangesW* Win32 functions. As an input parameter, the application specifies the name of a directory it wants to monitor, and the function returns whenever the contents of the directory changes. Although this approach is more efficient than volume scanning, it requires the application to be running at all times. Using these functions can also require an application to scan directories, because *FindFirstChangeNotification* doesn't indicate what changed—just that something in the directory has changed. An application can pass a buffer to *ReadDirectoryChangesW* that the FSD fills in with change records. If the buffer overflows, however, the application must be prepared to fall back on scanning the directory.

NTFS provides a third approach that overcomes the drawbacks of the first two: an application can configure the NTFS change journal facility by using the *DeviceIoControl* function's FSCTL_CREATE_USN_JOURNAL file system control code to have NTFS record information about file and directory changes to an internal file called the *change journal*. A change journal is usually large enough to virtually guarantee that applications get a chance to process changes without missing any. Applications use the FSCTL_QUERY_USN_JOURNAL file system control to read records from a change journal, and they can specify that the *DeviceIoControl* function not complete until new records are available.

Per-User Volume Quotas

Systems administrators often need to track or limit user disk space usage on shared storage volumes, so NTFS includes quota-management support. NTFS quota-management support allows for per-user specification of quota enforcement, which is useful for usage tracking and tracking when a user reaches warning and limit thresholds. NTFS can be configured to log an event indicating the occurrence to the system Event Log if a user surpasses his warning limit. Similarly, if a user attempts to use more volume storage then her quota limit permits, NTFS can log an event to the system Event Log and fail the application file I/O that would have caused the quota violation with a "disk full" error code.

NTFS tracks a user's volume usage by relying on the fact that it tags files and directories with the security ID (SID) of the user who created them. (See Chapter 8 for a definition of SIDs.) The logical sizes of files and directories a user owns count against the user's administrator-defined quota limit. Thus, a user can't circumvent his or her quota limit by creating an empty sparse file that is larger than the quota would allow and then filling the file with nonzero data. Similarly, whereas a 50-KB file might compress to 10 KB, the full 50 KB is used for quota accounting.

By default, volumes don't have quota tracking enabled. You need to use the Quota tab of a volume's Properties dialog box, shown in Figure 12-9, to enable quotas, to specify default warning and limit thresholds, and to configure the NTFS behavior that occurs when a user hits the warning or limit threshold. The Quota Entries tool, which you can launch from this dialog box, enables an administrator to specify different limits and behavior for each user. Applications that want to interact with NTFS quota management use COM quota interfaces, including *IDiskQuotaControl*, *IDiskQuotaUser* and *IDiskQuotaEvents*.

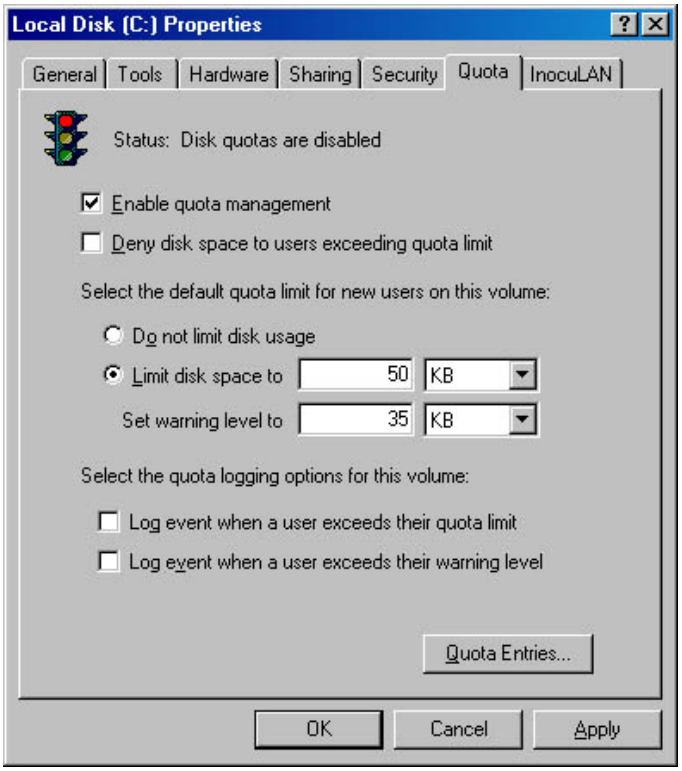


Figure 12-9 *Volume Properties dialog box*

Link Tracking

Shell shortcuts allow users to place files in their shell namespace (on their desktop, for example) that link to files located in the file system namespace. The Windows 2000 Start menu uses shell shortcuts extensively. Similarly, object linking and embedding (OLE) links allow documents from one application to be transparently embedded in the documents of other applications. The products of the

Microsoft Office 2000 suite, including PowerPoint, Excel, and Word, use OLE linking.

Although shell and OLE links provide an easy way to connect files with one another and with the shell namespace, they have in the past been difficult to manage. If a user moves the source of a shell or OLE link (a link source is the file or directory to which a link points) in Windows NT 4, Windows 95, or Windows 98, the link will be broken and the system has to rely on heuristics to attempt to locate the link's source. NTFS in Windows 2000 includes support for a service application called distributed link-tracking, which maintains the integrity of shell and OLE links when link targets move. Using the NTFS link-tracking support, if a link source located on an NTFS volume moves to any other NTFS volume within the originating volume's domain, the link-tracking service can transparently follow the movement and update the link to reflect the change.

NTFS link-tracking support is based on an optional file attribute known as an *object ID*. An application can assign an object ID to a file by using the FSCTL_CREATE_OR_GET_OBJECT_ID (which assigns an ID if one isn't already assigned) and FSCTL_SET_OBJECT_ID file system control codes. Object IDs are queried with the FSCTL_CREATE_OR_GET_OBJECT_ID and FSCTL_GET_OBJECT_ID file system control codes. The FSCTL_DELETE_OBJECT_ID file system control code lets applications delete object IDs from files.

Encryption

Corporate users often store sensitive information on their computers. Although data stored on company servers is usually safely protected with proper network security settings and physical access control, data stored on laptops can be exposed when a laptop is lost or stolen. NTFS file permissions don't offer protection because NTFS volumes can be fully accessed without regard to security by using NTFS file-reading software that doesn't require Windows 2000 to be running. Furthermore, NTFS file permissions are rendered useless when an alternate Windows 2000 installation is used to access files from an administrator account. Recall from Chapter 8 that the administrator account has the take-ownership and backup privileges, both of which allow it to access any secured object by overriding the object's security settings.

NTFS includes a facility called the Encrypting File System (EFS), which users can use to encrypt sensitive data. The operation of the EFS, as that of file compression, is completely transparent to applications, which means that file data is automatically decrypted when an application running in the account of a user authorized to view the data reads it and is automatically encrypted when an authorized application changes the data.

> **NOTE**
>
> NTFS doesn't permit the encryption of files located in the system volume's root directory or under the \Winnt directory because many of the files in these locations are required during the boot process and the EFS isn't active during the boot process.

The EFS relies on cryptographic services supplied by Windows 2000 in user mode, and so it consists of both a kernel-mode device driver that tightly integrates with NTFS as well as user-mode DLLs that communicate with the Local Security Authority Subsystem (Lsass) and cryptographic DLLs.

Files that are encrypted can be accessed only by using the private key of an account's EFS private/public key pair, and private keys are locked using an account's password. Thus, EFS-encrypted files on lost or stolen laptops can't be accessed using any means (other than a brute-force cryptographic attack) without the password of an account that is authorized to view the data.

Applications can use the *EncryptFile* and *DecryptFile* Win32 API functions to encrypt and decrypt files, and *FileEncryptionStatus* to retrieve as file or directory's EFS-related attributes, like whether the file or directory is encrypted.

POSIX Support

As explained in Chapter 2, one of the mandates for Windows 2000 was to fully support the POSIX 1003.1 standard. In the file system area, the POSIX standard requires support for case-sensitive file and directory names, traversal permissions (where security for each directory of a path is used when determining whether a user has access to a file or directory), a "file-change-time" time stamp (which is different than the MS-DOS "time-last-modified" stamp), and hard links (multiple directory entries that point to the same file). NTFS implements each of these features.

Defragmentation

A common myth that many people have held since the introduction of NTFS is that it automatically optimizes file placement on disk so as not to fragment the files. A file is fragmented if its data occupies discontiguous clusters. For example, Figure 12-10 shows a fragmented file consisting of three fragments. However, like most file systems (including versions of FAT on Windows 2000), NTFS makes no special efforts to keep files contiguous, other than to reserve a region of disk space known as the master file table (MFT) zone for the MFT. (NTFS lets other files allocate from the MFT zone when volume free space runs low.) Keeping an area free for the MFT can help it stay contiguous, but it, too, can become fragmented. (See the section "Master File Table (MFT)" later in this chapter for more information on MFTs.)

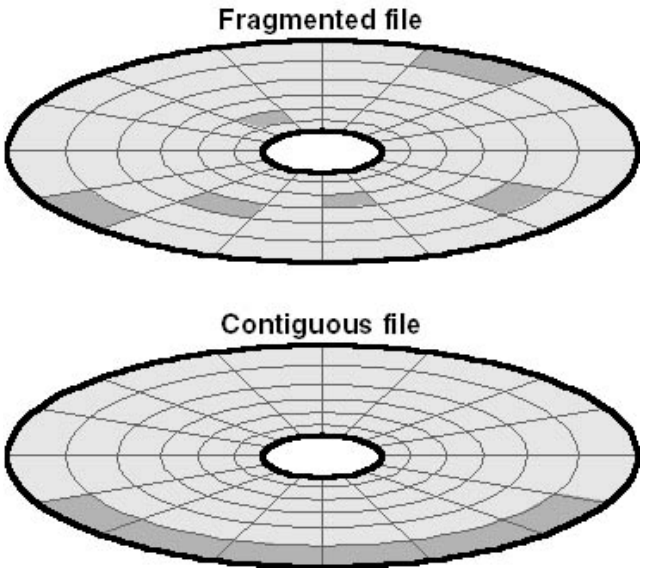


Figure 12-10 *Fragmented and contiguous files*

To facilitate the development of third-party disk defragmentation tools, Windows 2000 includes a defragmentation API that such tools can use to move file data so that files occupy contiguous clusters. The API consists of file system controls that let applications obtain a map of a volume's free and in-use clusters (FSCTL_GET_VOLUME_BITMAP), obtain a map of a file's cluster usage (FSCTL_GET_RETRIEVAL_POINTERS), and move a file (FSCTL_MOVE_FILE).

Windows 2000 includes a built-in defragmentation tool that is accessible by using the Disk Defragmenter utility (\Winnt\System32\Dfrg.msc). The built-in defragmentation tool has a number of limitations, such as an inability to be run from the command prompt or to be automatically scheduled. Third-party disk defragmentation products typically offer a richer feature set.

[Previous] [Next]

NTFS File System Driver

As described in Chapter 9, in the framework of the Windows 2000 I/O system, NTFS and other file systems are loadable device drivers that run in kernel mode. They are invoked indirectly by applications that use Win32 or other I/O APIs (such as POSIX). As Figure 12-11 shows, the Windows 2000 environment subsystems call Windows 2000 system services, which in turn locate the appropriate loaded drivers and call them. (For a description of system service dispatching, see the section "System Service Dispatching" in Chapter 3.)

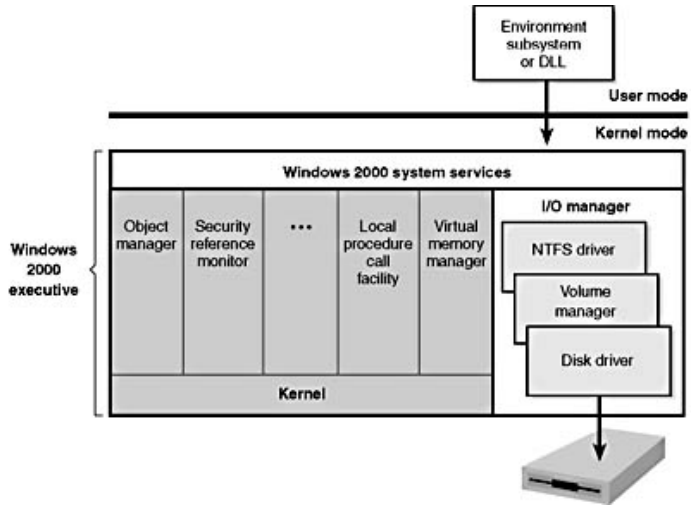


Figure 12-11 *Components of the Windows 2000 I/O system*

The layered drivers pass I/O requests to one another by calling the Windows 2000 executive's I/O manager. Relying on the I/O manager as an intermediary allows each driver to maintain independence so that it can be loaded or unloaded without affecting other drivers. In addition, the NTFS driver interacts with the three other Windows 2000 executive components, shown in the left side of Figure 12-12, that are closely related to file systems.

The log file service (LFS) is the part of NTFS that provides services for maintaining a log of disk writes. The log file LFS writes is used to recover an NTFS-formatted volume in the case of a system failure. (See the section "Log File Service (LFS)" for more information on LFS.)

The cache manager is the component of the Windows 2000 executive that provides systemwide caching services for NTFS and other file system drivers, including network file system drivers (servers and redirectors). All file systems implemented for Windows 2000 access cached files by mapping them into system address space and then accessing the virtual memory. The cache manager provides a specialized file system interface to the Windows 2000 memory manager for this purpose. When a program tries to access a part of a file that isn't loaded into the cache (a *cache miss*), the memory manager calls NTFS to access the disk driver and obtain the file contents from disk. The cache manager optimizes disk I/O by using its lazy writer threads to call the memory manager to flush cache contents to disk as a background activity (asynchronous disk writing). (For a complete description of the cache manager, see Chapter 11.)

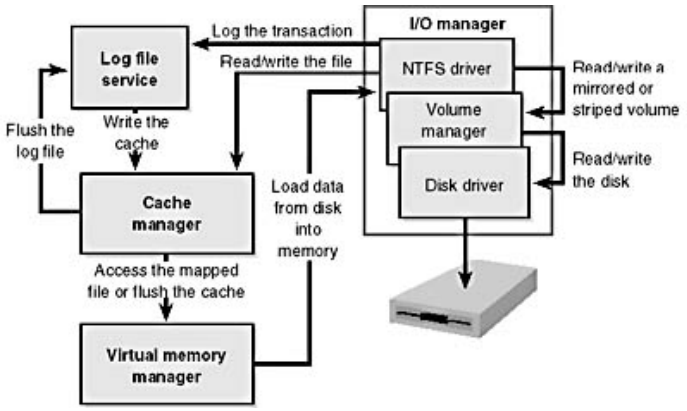


Figure 12-12 *NTFS and related components*

NTFS participates in the Windows 2000 object model by implementing files as objects. This implementation allows files to be shared and protected by the object manager, the component of Windows 2000 that manages all executive-level objects. (The object manager is described in the section "Object Manager" in Chapter 3.)

An application creates and accesses files just as it does other Windows 2000 objects: by means of object handles. By the time an I/O request reaches NTFS, the Windows 2000 object manager and security system have already verified that the calling process has the authority to access the file object in the way it is attempting to. The security system has compared the caller's access token to the entries in the access-control list for the file object. (See Chapter 8 for more information about access-control lists.) The I/O manager has also transformed the file handle into a pointer to a file object. NTFS uses the information in the file object to access the file on disk.

Figure 12-13 shows the data structures that link a file handle to the file system's on-disk structure.

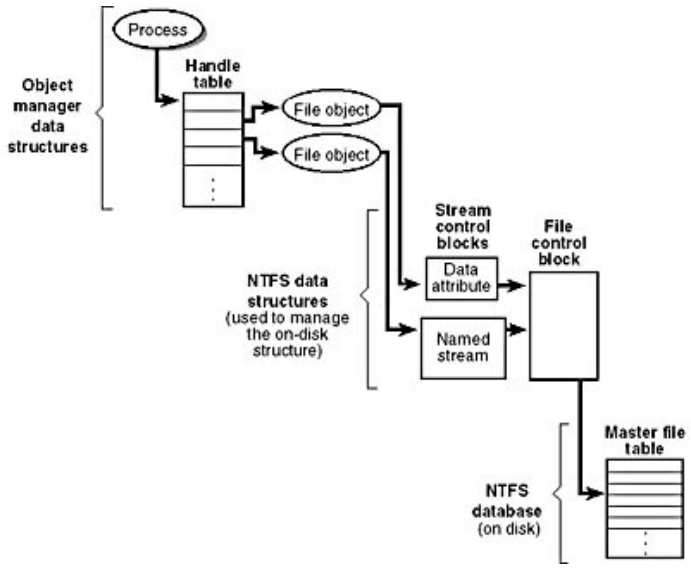


Figure 12-13 *NTFS data structures*

NTFS follows several pointers to get from the file object to the location of the file on disk. As Figure 12-13 shows, a file object, which represents a single call to the open-file system service, points to a *stream control block* (SCB) for the file attribute that the caller is trying to read or write. In Figure 12-13, a process has opened both the unnamed data attribute and a named stream (alternate data attribute) for the file. The SCBs represent individual file attributes and contain information about how to find specific attributes within a file. All the SCBs for a file point to a common data structure called a *file control block* (FCB). The FCB contains a pointer (actually, a file reference,

explained in the section "File Reference Numbers" later in this chapter) to the file's record in the disk-based master file table (MFT), which is described in detail in the following section.

[Previous] [Next]

NTFS On-Disk Structure

This section describes the on-disk structure of an NTFS volume, including how disk space is divided and organized into clusters, how files are organized into directories, how the actual file data and attribute information is stored on disk, and finally, how NTFS data compression works.

Volumes

The structure of NTFS begins with a volume. A *volume* corresponds to a logical partition on a disk, and it is created when you format a disk or part of a disk for NTFS. You can also create a RAID volume that spans multiple disks by using the Windows 2000 Disk Management MMC snap-in.

A disk can have one volume or several. NTFS handles each volume independently of the others. Three sample disk configurations for a 150-MB hard disk are illustrated in Figure 12-14.

Figure 12-14 *Sample disk configurations*

A volume consists of a series of files plus any additional unallocated space remaining on the disk partition. In the FAT file system, a volume also contains areas specially formatted for use by the file system. An NTFS volume, however, stores all file system data, such as bitmaps and directories, and even the system bootstrap, as ordinary files.

Clusters

The cluster size on an NTFS volume, or the *cluster factor*, is established when a user formats the volume with either the *format* command or the Disk Management MMC snap-in. The default cluster factor varies with the size of the volume, but it is an integral number of physical sectors, always a power of 2 (1 sector, 2 sectors, 4 sectors, 8 sectors, and so on). The cluster factor is expressed as the number of bytes in the cluster, such as 512 bytes, 1 KB, or 2 KB.

Internally, NTFS refers only to clusters. (However, NTFS forms low-level volume I/O operations such that it is sector-aligned and its length is a multiple of the sector size.) NTFS uses the cluster as its unit of allocation to maintain its independence from physical sector sizes. This independence allows NTFS to efficiently support very large disks by using a larger cluster factor or to support nonstandard disks that have a sector size other than 512 bytes. On a larger volume, use of a larger cluster factor can reduce fragmentation and speed allocation, at a small cost in terms of wasted disk space. Both the *format* command available from the Windows 2000 Command Prompt and the Format menu option under the All Tasks option on the Action menu in the Disk Management MMC snap-in choose a default cluster factor based on the volume size, but you can override this size.

NTFS refers to physical locations on a disk by means of *logical cluster numbers* (LCNs). LCNs are simply the numbering of all clusters from the beginning of the volume to the end. To convert an LCN to a physical disk address, NTFS multiplies the LCN by the cluster factor to get the physical

byte offset on the volume, as the disk driver interface requires. NTFS refers to the data within a file by means of *virtual cluster numbers* (VCNs). VCNs number the clusters belonging to a particular file from 0 through *m*. VCNs aren't necessarily physically contiguous, however; they can be mapped to any number of LCNs on the volume.

Master File Table (MFT)

In NTFS, all data stored on a volume is contained in files, including the data structures used to locate and retrieve files, the bootstrap data, and the bitmap that records the allocation state of the entire volume (the NTFS metadata). Storing everything in files allows the file system to easily locate and maintain the data, and each separate file can be protected by a security descriptor. In addition, if a particular part of the disk goes bad, NTFS can relocate the metadata files to prevent the disk from becoming inaccessible.

The *master file table* (MFT) is the heart of the NTFS volume structure. The MFT is implemented as an array of file records. The size of each file record is fixed at 1 KB, regardless of cluster size. (The structure of a file record is described in the "File Records" section.) Logically, the MFT contains one record for each file on the volume, including a record for the MFT itself. In addition to the MFT, each NTFS volume includes a set of metadata files containing the information that's used to implement the file system structure. Each of these NTFS metadata files has a name that begins with a dollar sign ($), although the signs are hidden. For example, the filename of the MFT is $Mft. The rest of the files on an NTFS volume are normal user files and directories, as shown in Figure 12-15.

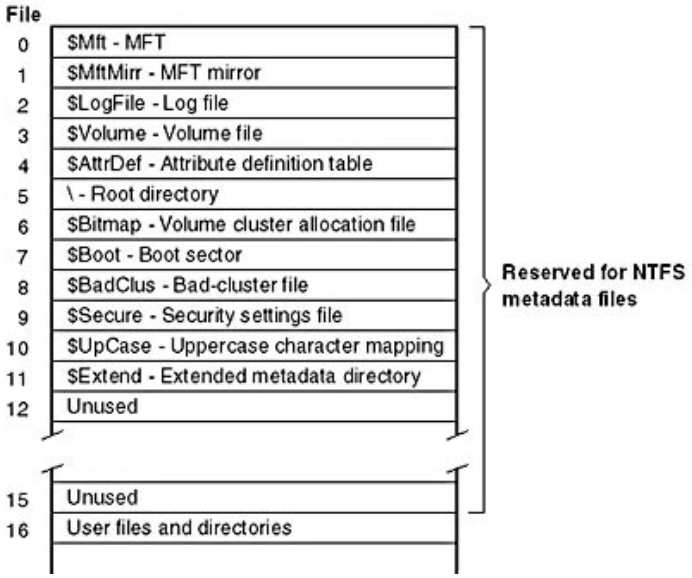


Figure 12-15 *File records for NTFS metadata files in the MFT*

Usually, each MFT record corresponds to a different file. If a file has a large number of attributes or becomes highly fragmented, however, more than one record might be needed for a single file. In such cases, the MFT first record, which stores the locations of the others, is called the *base file record*.

EXPERIMENT

Viewing the MFT

The Nfi utility included in the OEM Support Tools (part of the Windows 2000 debugging tools and available for download at *support.microsoft.com/support/kb/articles/Q253/0/66.asp*) allows you to dump the contents of an NTFS volume's MFT as well as to translate a volume cluster number or

physical-disk sector number (on non-RAID volumes only) to the file that contains it, if it's part of a file. The first 16 entries of the MFT are reserved for metadata files, but optional metadata files (which are present only if a volume uses an associated feature) fall outside this area: \$Extend\$Quota, \$Extend\$ObjId, \$Extend\$UsnJrnl, and \$Extend\$Reparse. The following dump was performed on a volume that uses reparse points ($Reparse), quotas ($Quota), and object IDs ($ObjId):

```
C:\>nfi G:\
NTFS File Sector Information Utility.
Copyright (C) Microsoft Corporation 1999. All rights reserved.

File 0
Master File Table ($Mft)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 32-52447 (0x20-0xccdf)
    $BITMAP (nonresident)
        logical sectors 16-23 (0x10-0x17)

File 1
Master File Table Mirror ($MftMirr)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048728-2048735 (0x1f42d8-0x1f42df)

File 2
Log File ($LogFile)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048736-2073343 (0x1f42e0-0x1fa2ff)

File 3
DASD ($Volume)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $OBJECT_ID (resident)
    $SECURITY_DESCRIPTOR (resident)
    $VOLUME_NAME (resident)
    $VOLUME_INFORMATION (resident)
    $DATA (resident)

File 4
Attribute Definition Table ($AttrDef)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 512256-512263 (0x7d100-0x7d107)

File 5
Root Directory
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $INDEX_ROOT $I30 (resident)
    $INDEX_ALLOCATION $I30 (nonresident)
        logical sectors 2073416-2073423 (0x1fa348-0x1fa34f)
    $BITMAP $I30 (resident)
```

```
File 6
Volume Bitmap ($BitMap)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2073424-2073675 (0x1fa350-0x1fa44b)

File 7
Boot Sectors ($Boot)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 0-15 (0x0-0xf)

File 8
Bad Cluster List ($BadClus)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (resident)
    $DATA $Bad (nonresident)

File 9
Security ($Secure)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA $SDS (nonresident)
        logical sectors 2073932-2074447 (0x1fa54c-0x1fa74f)
        logical sectors 523160-523163 (0x7fb98-0x7fb9b)
    $INDEX_ROOT $SDH (resident)
    $INDEX_ROOT $SII (resident)
    $INDEX_ALLOCATION $SDH (nonresident)
        logical sectors 1876152-1876159 (0x1ca0b8-0x1ca0bf)
    $INDEX_ALLOCATION $SII (nonresident)
        logical sectors 24-31 (0x18-0x1f)
    $BITMAP $SDH (resident)
    $BITMAP $SII (resident)

File 10
Upcase Table ($UpCase)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2073676-2073931 (0x1fa44c-0x1fa54b)

File 11
Extend Table ($Extend)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $I30 (resident)

File 12
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 13
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
```

```
        $SECURITY_DESCRIPTOR (resident)
        $DATA (resident)

    File 14
    (unknown/unnamed)
        $STANDARD_INFORMATION (resident)
        $SECURITY_DESCRIPTOR (resident)
        $DATA (resident)

    File 15
    (unknown/unnamed)
        $STANDARD_INFORMATION (resident)
        $SECURITY_DESCRIPTOR (resident)
        $DATA (resident)

    File 24
    \$Extend\$Quota
        $STANDARD_INFORMATION (resident)
        $FILE_NAME (resident)
        $INDEX_ROOT $O (resident)
        $INDEX_ROOT $Q (resident)

    File 25
    \$Extend\$ObjId
        $STANDARD_INFORMATION (resident)
        $FILE_NAME (resident)
        $INDEX_ROOT $O (resident)

    File 26
    \$Extend\$Reparse
        $STANDARD_INFORMATION (resident)
        $FILE_NAME (resident)
        $INDEX_ROOT $R (resident)
```

When it first accesses a volume, NTFS must *mount* it—that is, read metadata from the disk and construct internal data structures so that it can process application file system accesses. To mount the volume, NTFS looks in the boot sector to find the physical disk address of the MFT. The MFT's own file record is the first entry in the table; the second file record points to a file located in the middle of the disk called the *MFT mirror* (filename $MftMirr) that contains a copy of the first few rows of the MFT. This partial copy of the MFT is used to locate metadata files if part of the MFT file can't be read for some reason.

Once NTFS finds the file record for the MFT, it obtains the VCN-to-LCN mapping information in the file record's data attribute and stores it in memory. Each run has a VCN-to-LCN mapping and a run length because that's all the information necessary to locate an LCN for any VCN. This mapping information tells NTFS where the runs composing the MFT are located on the disk. (Runs are explained later in this chapter in the section "Resident and Nonresident Attributes.") NTFS then processes the MFT records for several more metadata files and opens the files. Next, NTFS performs its file system recovery operation (described in the section "Recovery"), and finally, it opens its remaining metadata files. The volume is now ready for user access.

As the system runs, NTFS writes to another important metadata file, the *log file* (filename $LogFile). NTFS uses the log file to record all operations that affect the NTFS volume structure, including file creation or any commands, such as Copy, that alter the directory structure. The log file is used to recover an NTFS volume after a system failure.

Another entry in the MFT is reserved for the *root directory* (also known as "\"). Its file record contains an index of the files and directories stored in the root of the NTFS directory structure.

When NTFS is first asked to open a file, it begins its search for the file in the root directory's file record. After opening a file, NTFS stores the file's MFT file reference so that it can directly access the file's MFT record when it reads and writes the file later.

NTFS records the allocation state of the volume in the *bitmap file* (filename $Bitmap). The data attribute for the bitmap file contains a bitmap, each of whose bits represents a cluster on the volume, identifying whether the cluster is free or has been allocated to a file.

The *security file* (filename $Secure) stores the volumewide security descriptor database. NTFS files and directories have individually settable security descriptors, but to conserve space, NTFS stores the settings in a common file, which allows files and directories that have the same security settings to reference the same security descriptor. In most environments, entire directory trees have the same security settings, so this optimization provides a significant savings.

Another system file, the *boot file* (filename $Boot), stores the Windows 2000 bootstrap code. For the system to boot, the bootstrap code must be located at a specific disk address. During formatting, however, the *format* command defines this area as a file by creating a file record for it. Creating the boot file allows NTFS to adhere to its rule of making everything on the disk a file. The boot file as well as NTFS metadata files can be individually protected by means of the security descriptors that are applied to all Windows 2000 objects. Using this "everything on the disk is a file" model also means that the bootstrap can be modified by normal file I/O, although the boot file is protected from editing.

NTFS also maintains a *bad-cluster file* (filename $BadClus) for recording any bad spots on the disk volume and a file known as the *volume file* (filename $Volume), which contains the volume name, the version of NTFS for which the volume is formatted, and a bit that when set signifies that a disk corruption has occurred and must be repaired by the Chkdsk utility. (The Chkdsk utility is covered in more detail later in the chapter.) The *uppercase file* (filename $UpCase) includes a translation table between lowercase and uppercase characters. NTFS maintains a file containing an *attribute definition table* (filename $AttrDef) that defines the attribute types supported on the volume and indicates whether they can be indexed, recovered during a system recovery operation, and so on.

NTFS stores several metadata files in the *extensions* (directory name $Extend) metadata directory, including the *object identifier file* (filename $ObjId), the *quota file* (filename $Quota), the *change journal file* (filename $UsnJrnl), and the *reparse point file* (filename $Reparse). These files store information related to optional features of NTFS. The object identifier file stores file object IDs, the quota file stores quota limit and behavior information on volumes that have quotas enabled, the change journal file records file and directory changes, and the reparse point file stores information about which files and directories on the volume include reparse point data.

File Reference Numbers

A file on an NTFS volume is identified by a 64-bit value called a *file reference*. The file reference consists of a file number and a sequence number. The file number corresponds to the position of the file's file record in the MFT minus 1 (or to the position of the base file record minus 1 if the file has more than one file record). The file reference sequence number, which is incremented each time an MFT file record position is reused, enables NTFS to perform internal consistency checks. A file reference is illustrated in Figure 12-16.

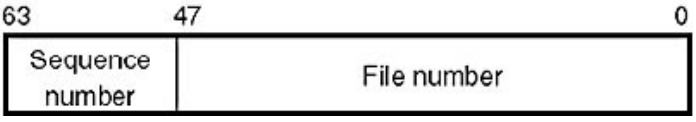


Figure 12-16 *File reference*

File Records

Instead of viewing a file as just a repository for textual or binary data, NTFS stores files as a collection of attribute/value pairs, one of which is the data it contains (called the *unnamed data attribute*). Other attributes that comprise a file include the filename, time stamp information, and possibly additional named data attributes. Figure 12-17 illustrates an MFT record for a small file.

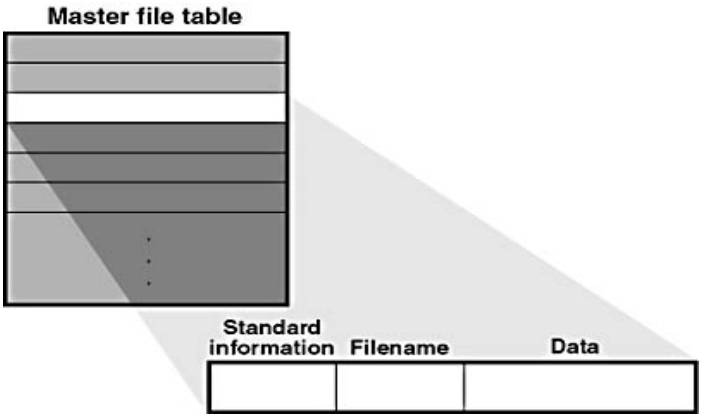


Figure 12-17 *MFT record for a small file*

Each file attribute is stored as a separate stream of bytes within a file. Strictly speaking, NTFS doesn't read and write files—it reads and writes attribute streams. NTFS supplies these attribute operations: create, delete, read (byte range), and write (byte range). The read and write services normally operate on the file's unnamed data attribute. However, a caller can specify a different data attribute by using the named data stream syntax.

Table 12-4 lists the attributes for files on an NTFS volume. (Not all attributes are present for every file.)

Table 12-4 *Attributes for NTFS Files*

Attribute	Attribute Name	Description
Volume information	$VOLUME_INFORMATION, $VOLUME_NAME	These attributes are present only in the $Volume metadata file. They store volume version sand label information.
Standard information	$STANDARD_INFORMATION	File attributes such as read-only, archive, and so on; time stamps, including when the file was created or last modified; and how many directories point to the file (its *hard link count*).
Filename	$FILE_NAME	The file's name in Unicode characters. A file can have multiple filename attributes, as it does when a hard link to a file exists or when a file with a long name has an automatically generated "short name" for access by MS-DOS and 16-bit Microsoft Windows applications.
Security descriptor	$SECURITY_DESCRIPTOR	This attribute is present for backward compatibility with previous versions of

		NTFS. The Windows 2000 version of NTFS stores all security descriptors in the $Secure metadata file, sharing descriptors among files and directories that have the same settings. Previous versions of NTFS stored private security descriptor information with each file and directory.
Data	$DATA	The contents of the file. In NTFS, a file has one default unnamed data attribute and can have additional named data attributes; that is, a file can have multiple data streams. A directory has no default data attribute but can have optional named data attributes.
Index root, index allocation, and index bitmap	$INDEX_ROOT, $INDEX_ALLOCATION, $BITMAP	Three attributes used to implement filename allocation and bitmap indexes for large directories (directories only).
Attribute list	$ATTRIBUTE_LIST	A list of the attributes that make up the file and the file reference of the MFT file record in which each attribute is located. This seldom-used attribute is present when a file requires more than one MFT file record.
Object ID	$OBJECT_ID	A 64-byte identifier for a file or directory, with the lowest 16 bytes (128 bits) unique to the volume. The link-tracking service assigns object IDs to shell shortcut and OLE link source files. NTFS provides APIs so that files and directories can be opened with their object ID rather than their filename.
Reparse information	$REPARSE_POINT	This attribute stores a file's reparse point data. NTFS junctions and mount points include this attribute.
Extended attributes	$EA, $EA_INFORMATION	Extended attributes aren't actively used but are provided for backward compatibility with OS/2 applications.
EFS information	$LOGGED_UTILITY_STREAM	EFS stores data in this attribute that's used to manage a file's encryption, such as the encrypted version of the key needed to decrypt the file and a list of users that are authorized to access the file. The word *logged* is in the attribute's name because changes to this attribute are recorded in the volume log file (described later in this chapter) for recoverability.

Table 12-4 shows attribute names; however, attributes actually correspond to numeric type codes, which NTFS uses to order the attributes within a file record. The file attributes in an MFT record are ordered by these type codes (numerically in ascending order), with some attribute types appearing more than once—if a file has multiple data attributes, for example, or multiple filenames.

Each attribute in a file record is identified with its attribute type code and has a value and an optional name. An attribute's value is the byte stream composing the attribute. For example, the value of the $FILE_NAME attribute is the file's name; the value of the $DATA attribute is whatever bytes the user stored in the file.

Most attributes never have names, though the index-related attributes and the $DATA attribute often do. Names distinguish among multiple attributes of the same type that a file can include. For example, a file that has a named data stream has two $DATA attributes: an unnamed $DATA attribute storing the default unnamed data stream and a named $DATA attribute having the name of the alternate stream and storing the named stream's data.

Filenames

Both NTFS and FAT allow each filename in a path to be as many as 255 characters long. Filenames can contain Unicode characters as well as multiple periods and embedded spaces. However, the FAT file system supplied with MS-DOS is limited to 8 (non-Unicode) characters for its filenames, followed by a period and a 3-character extension. Figure 12-18 provides a visual representation of the different file namespaces Windows 2000 supports and shows how they intersect.

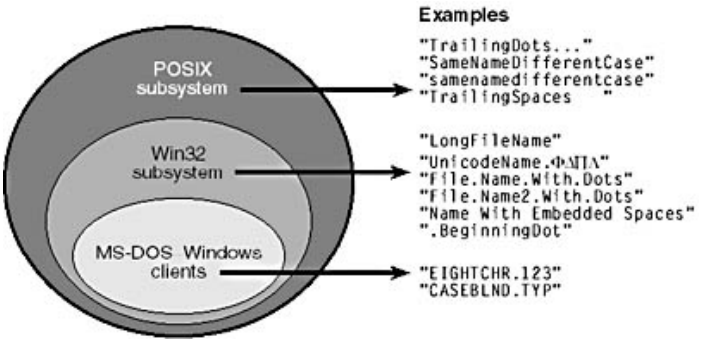


Figure 12-18 *Windows 2000 file namespaces*

The POSIX subsystem requires the biggest namespace of all the application execution environments that Windows 2000 supports, and therefore the NTFS namespace is equivalent to the POSIX namespace. The POSIX subsystem can create names that aren't visible to Win32 and MS-DOS applications, including names with trailing periods and trailing spaces. Ordinarily, creating a file using the large POSIX namespace isn't a problem because you would do that only if you intended the POSIX subsystem or POSIX client systems to use that file.

The relationship between 32-bit Windows (Win32) applications and MS-DOS Windows applications is a much closer one, however. The Win32 area in Figure 12-18 represents filenames that the Win32 subsystem can create on an NTFS volume but that MS-DOS and 16-bit Windows applications can't see. This group includes filenames longer than the 8.3 format of MS-DOS names, those containing Unicode (international) characters, those with multiple period characters or a beginning period, and those with embedded spaces. When a file is created with such a name, NTFS automatically generates an alternate, MSDOS-style filename for the file. Windows 2000 displays these short names when you use the */x* option with the *dir* command.

The MS-DOS filenames are fully functional aliases for the NTFS files and are stored in the same directory as the long filenames. The MFT record for a file with an autogenerated MS-DOS filename

is shown in Figure 12-19.

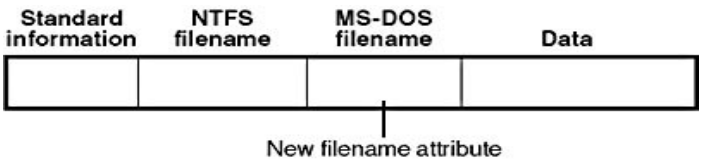


Figure 12-19 *MFT file record with an MS-DOS filename attribute*

The NTFS name and the generated MS-DOS name are stored in the same file record and therefore refer to the same file. The MS-DOS name can be used to open, read from, write to, or copy the file. If a user renames the file using either the long filename or the short filename, the new name replaces both the existing names. If the new name isn't a valid MS-DOS name, NTFS generates another MS-DOS name for the file.

NOTE

POSIX hard links are implemented in a similar way. When a hard link to a POSIX file is created, NTFS adds another filename attribute to the file's MFT file record. The two situations differ in one regard, however. When a user deletes a POSIX file that has multiple names (hard links), the file record and the file remain in place. The file and its record are deleted only when the last filename (hard link) is deleted. If a file has both an NTFS name and an autogenerated MSDOS name, however, a user can delete the file using either name.

Here's the algorithm NTFS uses to generate an MS-DOS name from a long filename:

1. Remove from the long name any characters that are illegal in MSDOS names, including spaces and Unicode characters. Remove preceding and trailing periods. Remove all other embedded periods, except the last one.

2. Truncate the string before the period (if present) to six characters, and append the string "~ n " (where *n* is a number, starting with 1, that is used to distinguish different files that truncate to the same name). Truncate the string after the period (if present) to three characters.

3. Put the result in uppercase letters. MS-DOS is case-insensitive, and this step guarantees that NTFS won't generate a new name that differs from the old only in case.

4. If the generated name duplicates an existing name in the directory, increment the ~*n* string.

Table 12-5 shows the long Win32 filenames from Figure 12-18 and their NTFS-generated MS-DOS versions. The current algorithm and the examples in Figure 12-18 should give you an idea of what NTFS-generated MS-DOSstyle filenames look like. Application developers shouldn't depend on this algorithm, though, because it might change in the future.

Table 12-5 *NTFS-Generated Filenames*

Win32 Long Name	NTFS-Generated Short Name
LongFileName	LONGFI~1
UnicodeName.FDPL	UNICOD~1
File.Name.With.Dots	FILENA~1.DOT
File.Name2.With.Dots	FILENA~2.DOT

Name With Embedded Spaces	NAMEWI~1
.BeginningDot	BEGINN~1

Resident and Nonresident Attributes

If a file is small, all its attributes and their values (its data, for example) fit in the file record. When the value of an attribute is stored directly in the MFT, the attribute is called a *resident attribute*. (In Figure 12-17, for example, all attributes are resident.) Several attributes are defined as always being resident so that NTFS can locate nonresident attributes. The standard information and index root attributes are always resident, for example.

Each attribute begins with a standard header containing information about the attribute, information that NTFS uses to manage the attributes in a generic way. The header, which is always resident, records whether the attribute's value is resident or nonresident. For resident attributes, the header also contains the offset from the header to the attribute's value and the length of the attribute's value, as Figure 12-20 illustrates for the filename attribute.

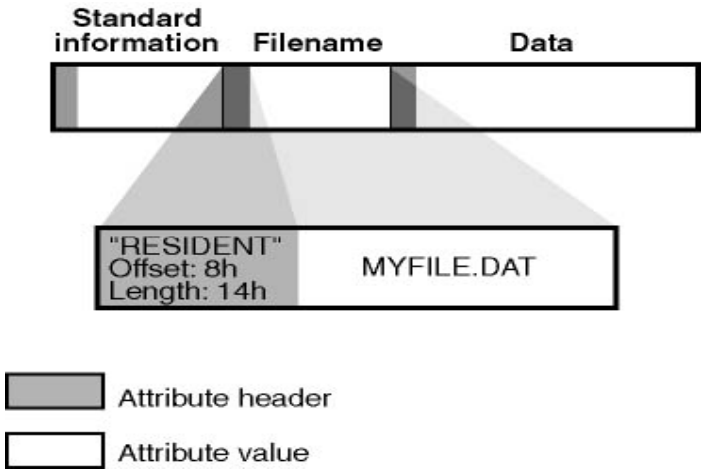


Figure 12-20 *Resident attribute header and value*

When an attribute's value is stored directly in the MFT, the time it takes NTFS to access the value is greatly reduced. Instead of looking up a file in a table and then reading a succession of allocation units to find the file's data (as the FAT file system does, for example), NTFS accesses the disk once and retrieves the data immediately.

The attributes for a small directory, as well as for a small file, can be resident in the MFT, as Figure 12-21 shows. For a small directory, the index root attribute contains an index of file references for the files and the subdirectories in the directory.

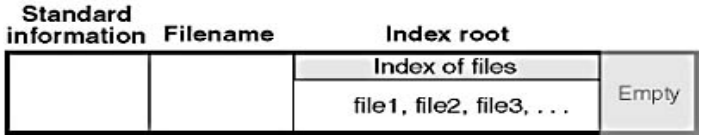


Figure 12-21 *MFT file record for a small directory*

Of course, many files and directories can't be squeezed into a 1-KB fixed-size MFT record. If a particular attribute, such as a file's data attribute, is too large to be contained in an MFT file record, NTFS allocates clusters for the attribute's data separate from the MFT. This area is called a *run* (or an *extent*). If the attribute's value later grows (if a user appends data to the file, for example), NTFS allocates another run for the additional data. Attributes whose values are stored in runs rather than in

the MFT are called *nonresident attributes.* The file system decides whether a particular attribute is resident or nonresident; the location of the data is transparent to the process accessing it.

When an attribute is nonresident, as the data attribute for a large file might be, its header contains the information NTFS needs to locate the attribute's value on the disk. Figure 12-22 shows a nonresident data attribute stored in two runs.

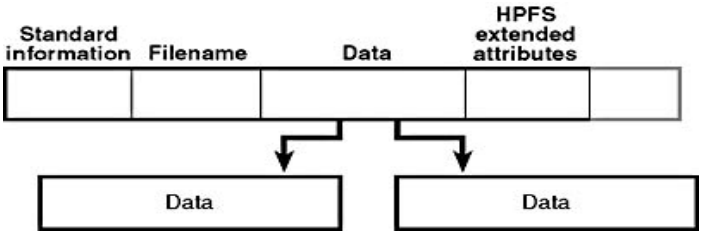


Figure 12-22 *MFT file record for a large file with two data runs*

Among the standard attributes, only those that can grow can be nonresident. For files, the attributes that can grow are the data and the attribute list (not shown in Figure 12-22). The standard information and filename attributes are always resident.

A large directory can also have nonresident attributes (or parts of attributes), as Figure 12-23 shows. In this example, the MFT file record doesn't have enough room to store the index of files that make up this large directory. A part of the index is stored in the index root attribute, and the rest of the index is stored in nonresident runs called *index buffers.* The index root, index allocation, and bitmap attributes are shown here in a simplified form. They are described in more detail in the next section. The standard information and filename attributes are always resident. The header and at least part of the value of the index root attribute are also resident for directories.

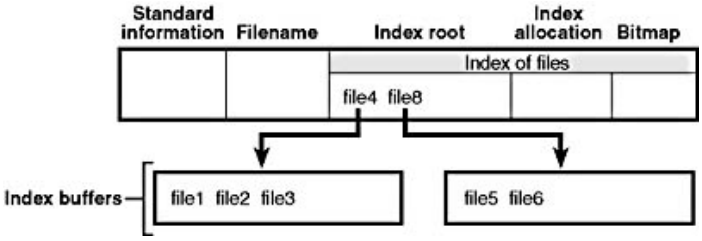


Figure 12-23 *MFT file record for a large directory with a nonresident filename index*

When a file's (or a directory's) attributes can't fit in an MFT file record and separate allocations are needed, NTFS keeps track of the runs by means of VCN-to-LCN mapping pairs. LCNs represent the sequence of clusters on an entire volume from 0 through *n.* VCNs number the clusters belonging to a particular file from 0 through *m*. For example, the clusters in the runs of a nonresident data attribute are numbered as shown in Figure 12-24.

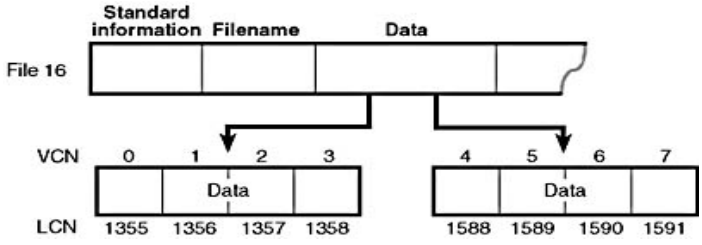


Figure 12-24 *VCNs for a nonresident data attribute*

If this file had more than two runs, the numbering of the third run would start with VCN 8. As

Figure 12-25 shows, the data attribute header contains VCN-to-LCN mappings for the two runs here, which allows NTFS to easily find the allocations on the disk.

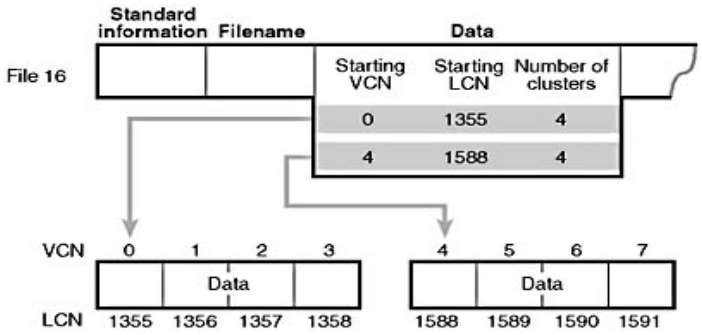


Figure 12-25 *VCN-to-LCN mappings for a nonresident data attribute*

Although Figure 12-25 shows just data runs, other attributes can be stored in runs if there isn't enough room in the MFT file record to contain them. And if a particular file has too many attributes to fit in the MFT record, a second MFT record is used to contain the additional attributes (or attribute headers for nonresident attributes). In this case, an attribute called the *attribute list* is added. The attribute list attribute contains the name and type code of each of the file's attributes and the file reference of the MFT record where the attribute is located. The attribute list attribute is provided for those cases in which a file grows so large or so fragmented that a single MFT record can't contain the multitude of VCN-to-LCN mappings needed to find all its runs. Files with more than 200 runs typically require an attribute list.

Indexing

In NTFS, a file directory is simply an index of filenames—that is, a collection of filenames (along with their file references) organized in a particular way for quick access. To create a directory, NTFS indexes the filename attributes of the files in the directory. The MFT record for the root directory of a volume is shown in Figure 12-26.

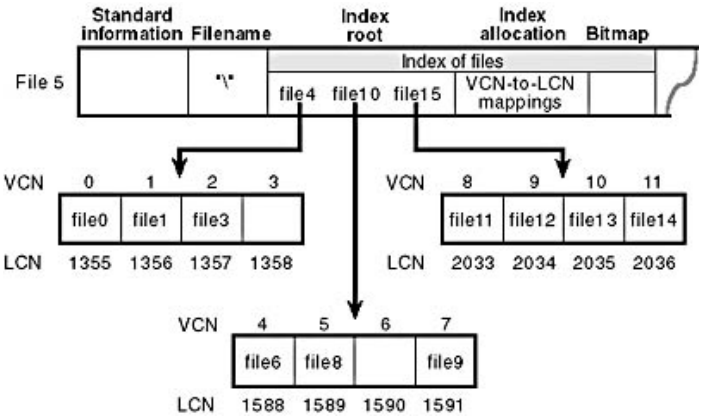


Figure 12-26 *Filename index for a volume's root directory*

Conceptually, an MFT entry for a directory contains in its index root attribute a sorted list of the files in the directory. For large directories, however, the filenames are actually stored in 4-KB fixed-size index buffers that contain and organize the filenames. Index buffers implement a *b+ tree* data structure, which minimizes the number of disk accesses needed to find a particular file, especially for large directories. The index root attribute contains the first level of the b+ tree (root subdirectories) and points to index buffers containing the next level (more subdirectories, perhaps, or files).

Figure 12-26 shows only filenames in the index root attribute and the index buffers (*file6*, for example), but each entry in an index also contains the file reference in the MFT where the file is described and time stamp and file size information for the file. NTFS duplicates the time stamp and file size information from the file's MFT record. This technique, which is used by FAT and NTFS, requires updated information to be written in two places. Even so, it's a significant speed optimization for directory browsing because it enables the file system to display each file's time stamps and size without opening every file in the directory.

The index allocation attribute maps the VCNs of the index buffer runs to the LCNs that indicate where the index buffers reside on the disk, and the bitmap attribute keeps track of which VCNs in the index buffers are in use and which are free. Figure 12-26 shows one file entry per VCN (that is, per cluster), but filename entries are actually packed into each cluster. Each 4-KB index buffer can contain about 20 to 30 filename entries.

The b+ tree data structure is a type of balanced tree that is ideal for organizing sorted data stored on a disk because it minimizes the number of disk accesses needed to find an entry. In the MFT, a directory's index root attribute contains several filenames that act as indexes into the second level of the b+ tree. Each filename in the index root attribute has an optional pointer associated with it that points to an index buffer. The index buffer it points to contains filenames with lexicographic values less than its own. In Figure 12-26, for example, *file4* is a first-level entry in the b+ tree. It points to an index buffer containing filenames that are (lexicographically) less than itself—the filenames *file0*, *file1*, and *file3*. Note that the names *file1*, *file2*, and so on that are used in this example are not literal filenames but names intended to show the relative placement of files that are lexicographically ordered according to the displayed sequence.

Storing the filenames in b+ trees provides several benefits. Directory lookups are fast because the filenames are stored in a sorted order. And when higher-level software enumerates the files in a directory, NTFS returns already-sorted names. Finally, because b+ trees tend to grow wide rather than deep, NTFS's fast lookup times don't degrade as directories grow.

NTFS also provides general support for indexing data besides filenames. As we stated earlier, a file can have an object ID assigned to it, which is stored in the file's $OBJECT _ID attribute. NTFS provides an API that allows applications to open a file by using the file's object ID instead of its name. NTFS therefore must make the process of translating an object ID to a file's file number an efficient one. To do so, it stores a mapping of all a volume's object IDs to their file reference numbers in the \$Extend\$ObjId metadata file. NTFS sorts the object IDs in the $ObjId's $O index. As are filenames in filename indexes, the object ID index is stored as a b+ tree.

Data Compression and Sparse Files

NTFS supports compression on a per-file, per-directory, or per-volume basis. (NTFS compression is performed only on user data, not file system metadata.) You can tell whether a volume is compressed by using the Win32 *GetVolumeInformation* function. To retrieve the actual compressed size of a file, use the Win32 *GetCompressedFileSize* function. Finally, to examine or change the compression setting for a file or directory, use the Win32 *DeviceIoControl* function. (See the FSCTL_GET_COMPRESSION and FSCTL_SET_COMPRESSION file system control codes.) Keep in mind that although setting a file's compression state compresses (or decompresses) the file right away, setting a directory's or volume's compression state doesn't cause any immediate compression or decompression. Instead, setting a directory's or volume's compression state sets a default compression state that will be given to all newly created files and subdirectories within that directory or volume.

The following section introduces NTFS compression by examining the simple case of compressing sparse data. The subsequent sections extend the discussion to the compression of ordinary files and sparse files.

Compressing Sparse Data

Sparse data is often large but contains only a small amount of nonzero data relative to its size. A sparse matrix is one example of sparse data.As described earlier, NTFS uses VCNs, from 0 through *m*, to enumerate the clusters of a file. Each VCN maps to a corresponding LCN, which identifies the disk location of the cluster. Figure 12-27 illustrates the runs (disk allocations) of a normal, noncompressed file, including its VCNs and the LCNs they map to.

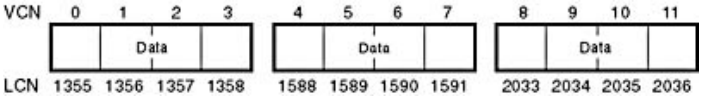


Figure 12-27 *Runs of a noncompressed file*

This file is stored in 3 runs, each of which is 4 clusters long, for a total of 12 clusters. Figure 12-28 shows the MFT record for this file. As described earlier, to save space, the MFT record's data attribute, which contains VCN-to-LCN mappings, records only one mapping for each run, rather than one for each cluster. Notice, however, that each VCN from 0 through 11 has a corresponding LCN associated with it. The first entry starts at VCN 0 and covers 4 clusters, the second entry starts at VCN 4 and covers 4 clusters, and so on. This entry format is typical for a noncompressed file.

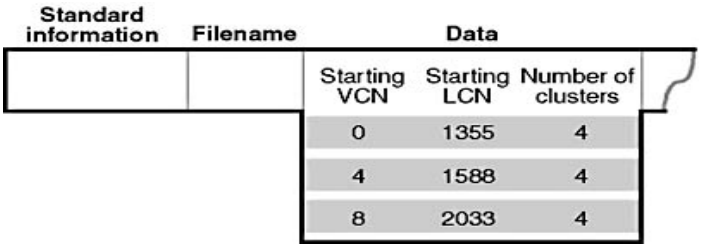


Figure 12-28 *MFT record for a noncompressed file*

When a user selects a file on an NTFS volume for compression, one NTFS compression technique is to remove long strings of zeros from the file. If the file's data is sparse, it typically shrinks to occupy a fraction of the disk space it would otherwise require. On subsequent writes to the file, NTFS allocates space only for runs that contain nonzero data.

Figure 12-29 depicts the runs of a compressed file containing sparse data. Notice that certain ranges of the file's VCNs (16-31 and 64-127) have no disk allocations.

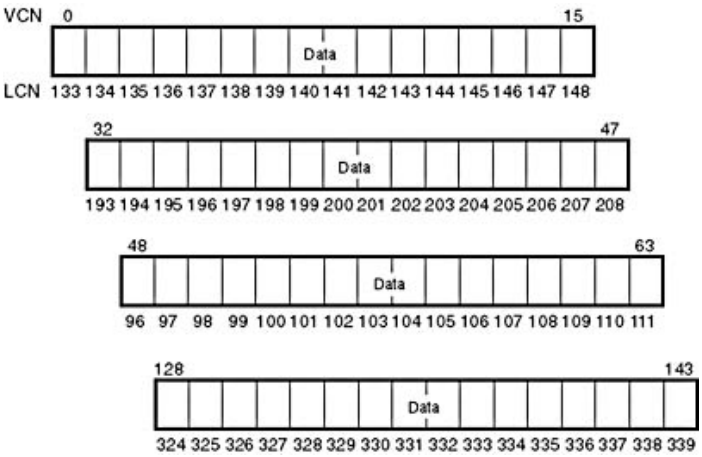


Figure 12-29 *Runs of a compressed file containing sparse data*

The MFT record for this compressed file omits blocks of VCNs that contain zeros and therefore

have no physical storage allocated to them. The first data entry in Figure 12-30, for example, starts at VCN 0 and covers 16 clusters. The second entry jumps to VCN 32 and covers 16 clusters.

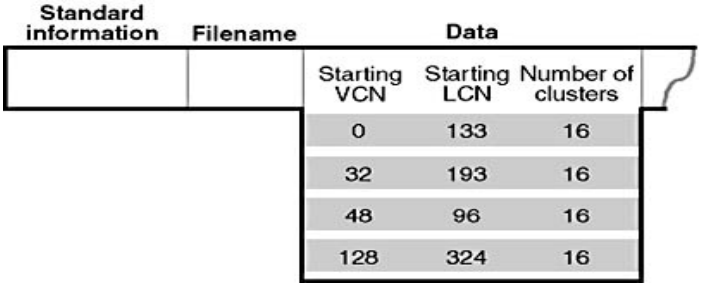


Figure 12-30 *MFT record for a compressed file containing sparse data*

When a program reads data from a compressed file, NTFS checks the MFT record to determine whether a VCN-to-LCN mapping covers the location being read. If the program is reading from an unallocated "hole" in the file, it means that the data in that part of the file consists of zeros, so NTFS returns zeros without accessing the disk. If a program writes nonzero data to a "hole," NTFS quietly allocates disk space and then writes the data. This technique is very efficient for sparse file data that contains a lot of zero data.

Compressing Nonsparse Data

The preceding example of compressing a sparse file is somewhat contrived. It describes "compression" for a case in which whole sections of a file were filled with zeros but the remaining data in the file wasn't affected by the compression. The data in most files isn't sparse, but it can still be compressed by the application of a compression algorithm.

In NTFS, users can specify compression for individual files or for all the files in a directory. (New files created in a directory marked compressed are automatically compressed—existing files must be compressed individually.) When it compresses a file, NTFS divides the file's unprocessed data into *compression units* 16 clusters long (equal to 8 KB for a 512-byte cluster, for example). Certain sequences of data in a file might not compress much, if at all; so for each compression unit in the file, NTFS determines whether compressing the unit will save at least 1 cluster of storage. If compressing the unit won't free up at least 1 cluster, NTFS allocates a 16-cluster run and writes the data in that unit to disk without compressing it. If the data in a 16-cluster unit will compress to 15 or fewer clusters, NTFS allocates only the number of clusters needed to contain the compressed data and then writes it to disk. Figure 12-31 illustrates the compression of a file with four runs. The unshaded areas in this figure represent the actual storage locations that the file occupies after compression. The first, second, and fourth runs were compressed; the third run wasn't. Even with one noncompressed run, compressing this file saved 26 clusters of disk space, or 41 percent.

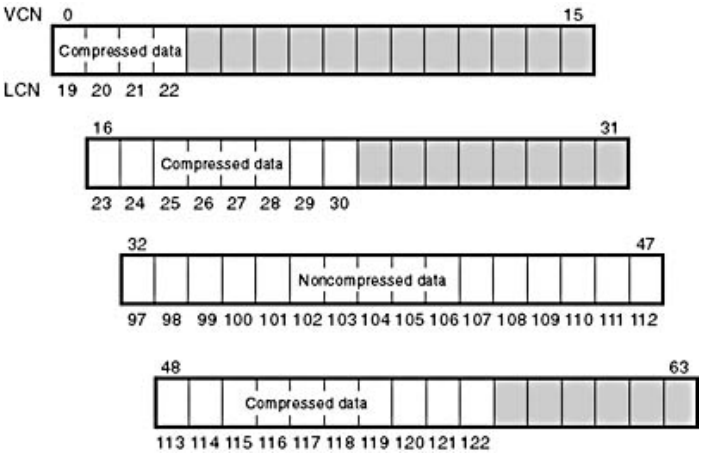


Figure 12-31 *Data runs of a compressed file*

NOTE

Although the diagrams in this chapter show contiguous LCNs, a compression unit need not be stored in physically contiguous clusters. Runs that occupy noncontiguous clusters produce slightly more complicated MFT records than the one shown in Figure 12-32.

When it writes data to a compressed file, NTFS ensures that each run begins on a virtual 16-cluster boundary. Thus the starting VCN of each run is a multiple of 16, and the runs are no longer than 16 clusters. NTFS reads and writes at least one compression unit at a time when it accesses compressed files. When it writes compressed data, however, NTFS tries to store compression units in physically contiguous locations so that it can read them all in a single I/O operation. The 16-cluster size of the NTFS compression unit was chosen to reduce internal fragmentation: the larger the compression unit, the less the overall disk space needed to store the data. This 16-cluster compression unit size represents a trade-off between producing smaller compressed files and slowing read operations for programs that randomly access files. The equivalent of 16 clusters must be decompressed for each cache miss. (A cache miss is more likely to occur during random file access.) Figure 12-32 shows the MFT record for the compressed file shown in Figure 12-31.

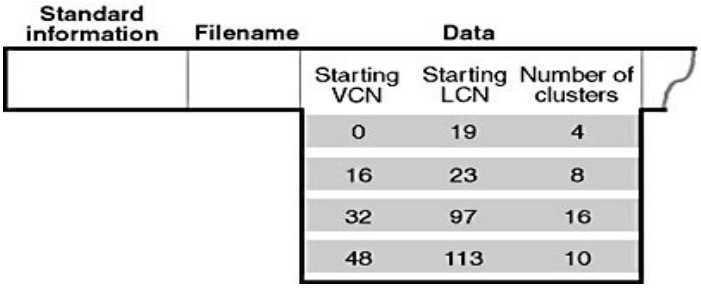


Figure 12-32 *MFT record for a compressed file*

One difference between this compressed file and the earlier example of a compressed file containing sparse data is that three of the compressed runs in this file are less than 16 clusters long. Reading this information from a file's MFT file record enables NTFS to know whether data in the file is compressed. Any run shorter than 16 clusters contains compressed data that NTFS must decompress when it first reads the data into the cache. A run that is exactly 16 clusters long doesn't contain compressed data and therefore requires no decompression.

If the data in a run has been compressed, NTFS decompresses the data into a scratch buffer and then copies it to the caller's buffer. NTFS also loads the decompressed data into the cache, which makes subsequent reads from the same run as fast as any other cached read. NTFS writes any updates to the file to the cache, leaving the lazy writer to compress and write the modified data to disk asynchronously. This strategy ensures that writing to a compressed file produces no more significant delay than writing to a noncompressed file would.

NTFS keeps disk allocations for a compressed file contiguous whenever possible. As the LCNs indicate, the first two runs of the compressed file shown in Figure 12-31 are physically contiguous, as are the last two. When two or more runs are contiguous, NTFS performs disk read-ahead, as it does with the data in other files. Because the reading and decompression of contiguous file data take place asynchronously before the program requests the data, subsequent read operations obtain the data directly from the cache, which greatly enhances read performance.

Sparse Files

Sparse files (the NTFS file type, as opposed to files that consist of sparse data, described earlier) are

essentially compressed files for which NTFS doesn't apply compression to the file's nonsparse data. However, NTFS manages the run data of a sparse file's MFT record the same way it does for compressed files that consist of sparse and nonsparse data.

Reparse Points

A reparse point is a block of up to 16 KB of application-defined reparse data and a 32-bit reparse tag that are stored in the $REPARSE_POINT attribute of a file or directory. Whenever an application creates or deletes a reparse point, NTFS updates the \$Extend\$Reparse metadata file, in which NTFS stores entries that identify the file record numbers of files and directories that contain reparse points. Storing the records in a central location enables NTFS to provide interfaces for applications to enumerate all a volume's reparse points or just specific types of reparse points, such as mount points. (See Chapter 10 for more information on mount points.) The \$Extend\$Reparse file uses the general indexing facility of NTFS by collating the file's entries (in an index named $R) by reparse point tags.

The Change Journal File

The change journal file, \$Extend\$UsnJrnl, is a sparse file that NTFS creates only when an application enables change logging. The journal stores change entries in the $J data stream. Entries include the following information about a file or directory change:

- The time of the change
- The change type (delete, rename, size extend, and so on)
- The file or directory's attributes
- The file or directory's name
- The file or directory's file reference number
- The file reference number of the file's parent directory

The journal is sparse so that it never overflows; when the journal's on-disk size exceeds the maximum defined for the file, NTFS simply begins zeroing the file data that precedes the window of change information having a size equal to the maximum journal size, as shown in Figure 12-33. To prevent constant resizing when an application is continuously exceeding the journal's size, NTFS shrinks the journal only when its size is twice an application-defined value over the maximum configured size.

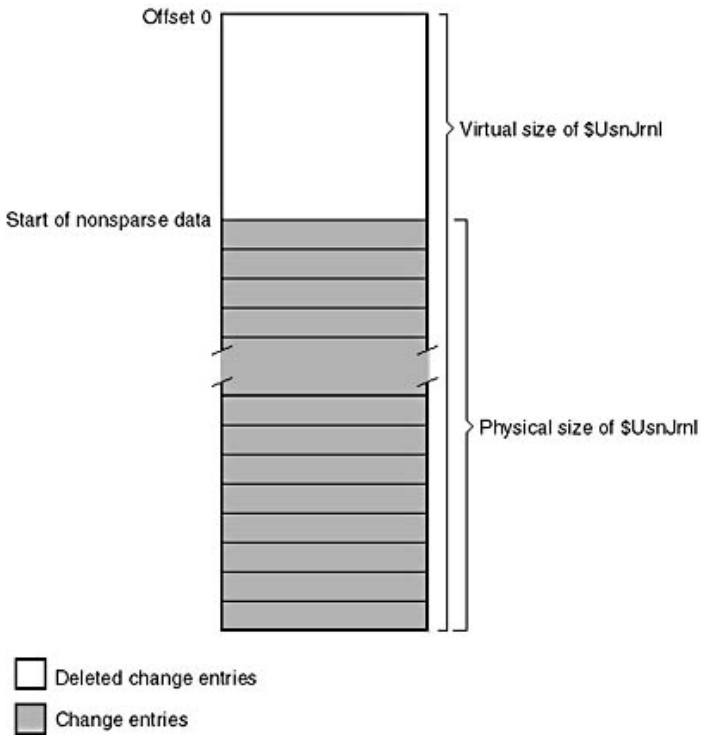


Figure 12-33 *Change journal ($UsnJrnl) space allocation*

Object IDs

In addition to storing the object ID assigned to a file or directory in the $OBJECT_ID attribute of its MFT record, NTFS also keeps the correspondence between object IDs and their file reference numbers in the $O index of the \$Extend\$ObjId metadata file. The index collates entries by object ID, making it easy for NTFS to quickly locate a file based on its ID. This feature allows applications, using undocumented native API functionality, to open a file or directory using its object ID.

Quota Tracking

The NTFS quota-tracking facility associates an owner ID with each user who creates files and stores the user's owner ID with each file or directory the user creates. To determine whether a user has been assigned an ID, NTFS uses the user's SID as a key to index the $O index of the \$Extend\$Quota metadata file. If an ID isn't located, NTFS allocates a unique ID for the user and records the association in the $O index.

\$Extend\$Quota also contains an index named $Q that NTFS uses to store per-user quota information entries, collating the entries by owner ID. When a user attempts to allocate space on a volume, NTFS uses the owner ID to look up the user's quota entry and determine whether there is sufficient disk space left in the user's quota to allow the allocation.

Consolidated Security

Another example of general indexing is seen in the \$Secure metadata file, which stores security descriptors for all the files and directories on a volume. NTFS assigns each unique security descriptor an NTFS security ID. (These are different than the SIDs described in Chapter 8.)

When a process applies a security descriptor to a file or directory, NTFS obtains a 32-bit hash of the descriptor and looks up the corresponding security ID in an index named $SDH that is stored in the

\$Secure file. Multiple security descriptors can hash to the same value, so NTFS compares the security descriptor being applied with any that have the same hash to verify an exact match. If it locates the applied security descriptor in the $SDH index, NTFS assigns the file the associated security ID. Otherwise, it allocates a new security ID, updates the $SDH index, and adds the security descriptor to the $SII index. The $SII index is collated by security ID so that when a user attempts to open a file or directory, NTFS can quickly locate the file or directory's security descriptor by using the file or directory's security ID.

[Previous] [Next]

NTFS Recovery Support

NTFS recovery support ensures that if a power failure or a system failure occurs, no file system operations (transactions) will be left incomplete and the structure of the disk volume will remain intact without the need to run a disk repair utility. The NTFS Chkdsk utility is used to repair catastrophic disk corruption caused by I/O errors (bad disk sectors, electrical anomalies, or disk failures, for example) or software bugs. But with the NTFS recovery capabilities in place, Chkdsk is rarely needed.

As mentioned earlier (in the section "Recoverability"), NTFS uses a transaction-processing scheme to implement recoverability. This strategy ensures a full disk recovery that is also extremely fast (on the order of seconds) for even the largest disks. NTFS limits its recovery procedures to file system data to ensure that at the very least the user will never lose a volume because of a corrupted file system; however, unless an application takes specific action (such as flushing cached files to disk), NTFS doesn't guarantee user data to be fully updated if a crash occurs. Transaction-based protection of user data is available in most of the database products available for Windows 2000, such as Microsoft SQL Server. The decision not to implement user data recovery in the file system represents a trade-off between a fully fault tolerant file system and one that provides optimum performance for all file operations.

The following sections describe the evolution of file system reliability as a context for an introduction to recoverable file systems, detail the transaction-logging scheme NTFS uses to record modifications to file system data structures, and explain how NTFS recovers a volume if the system fails.

Evolution of File System Design

The development of a recoverable file system was a step forward in the evolution of file system design. In the past, two techniques were common for constructing a file system's I/O and caching support: *careful write* and *lazy write*. The file systems developed for Digital Equipment Corporation's (now Compaq's) VAX/VMS and for some other proprietary operating systems employed a careful write algorithm, while OS/2 HPFS and most older UNIX file systems used a lazy write file system scheme.

The next two subsections briefly review these two types of file systems and their intrinsic trade-offs between safety and performance. The third subsection describes NTFS's recoverable approach and explains how it differs from the other two strategies.

Careful Write File Systems

When an operating system crashes or loses power, I/O operations in progress are immediately, and often prematurely, interrupted. Depending on what I/O operation or operations were in progress and how far along they were, such an abrupt halt can produce inconsistencies in a file system. An inconsistency in this context is a file system corruption—a filename appears in a directory listing,

for example, but the file system doesn't know the file is there or can't access the file. The worst file system corruptions can leave an entire volume inaccessible.

A careful write file system doesn't try to prevent file system inconsistencies. Rather, it orders its write operations so that, at worst, a system crash will produce predictable, noncritical inconsistencies, which the file system can fix at its leisure.

When any kind of file system receives a request to update the disk, it must perform several suboperations before the update will be complete. In a file system that uses the careful write strategy, the suboperations are always written to disk serially. When allocating disk space for a file, for example, the file system first sets some bits in its bitmap and then allocates the space to the file. If the power fails immediately after the bits are set, the careful write file system loses access to some disk space—to the space represented by the set bits—but existing data isn't corrupted.

Serializing write operations also means that I/O requests are filled in the order in which they are received. If one process allocates disk space and shortly thereafter another process creates a file, a careful write file system completes the disk allocation before it starts to create the file because interleaving the suboperations of the two I/O requests could result in an inconsistent state.

NOTE

The FAT file system uses a *write-through* algorithm that causes disk modifications to be immediately written to the disk. Unlike the careful write approach, the write-through technique doesn't require the file system to order its writes to prevent inconsistencies.

The main advantage of a careful write file system is that in the event of a failure the volume stays consistent and usable without the need to immediately run a slow volume repair utility. Such a utility is needed to correct the predictable, nondestructive disk inconsistencies that occur as the result of a system failure, but the utility can be run at a convenient time, typically when the system is rebooted.

Lazy Write File Systems

A careful write file system sacrifices speed for the safety it provides. A lazy write file system improves performance by using a *write-back* caching strategy; that is, it writes file modifications to the cache and flushes the contents of the cache to disk in an optimized way, usually as a background activity.

The performance improvements associated with the lazy write caching technique take several forms. First, the overall number of disk writes is reduced. Because serialized, immediate disk writes aren't required, the contents of a buffer can be modified several times before they are written to disk. Second, the speed of servicing application requests is greatly increased because the file system can return control to the caller without waiting for disk writes to be completed. Finally, the lazy write strategy ignores the inconsistent intermediate states on a file volume that can result when the suboperations of two or more I/O requests are interleaved. It is thus easier to make the file system multithreaded, allowing more than one I/O operation to be in progress at a time.

The disadvantage of the lazy write technique is that it creates intervals during which a volume is in too inconsistent a state to be corrected by the file system. Consequently, lazy write file systems must keep track of the volume's state at all times. In general, lazy write file systems gain a performance advantage over careful write systems—at the expense of greater risk and user inconvenience if the system fails.

Recoverable File Systems

A recoverable file system such as NTFS tries to exceed the safety of a careful write file system while achieving the performance of a lazy write file system. A recoverable file system ensures volume

consistency by using logging techniques (sometimes called *journaling*) originally developed for transaction processing. If the operating system crashes, the recoverable file system restores consistency by executing a recovery procedure that accesses information that has been stored in a log file. Because the file system has logged its disk writes, the recovery procedure takes only seconds, regardless of the size of the volume.

The recovery procedure for a recoverable file system is exact, guaranteeing that the volume will be restored to a consistent state. In NTFS, none of the inadequate restorations associated with lazy write file systems can happen.

A recoverable file system incurs some costs for the safety it provides. Every transaction that alters the volume structure requires that one record be written to the log file for each of the transaction's suboperations. This logging overhead is ameliorated by the file system's "batching" of log records—writing many records to the log file in a single I/O operation. In addition, the recoverable file system can employ the optimization techniques of a lazy write file system. It can even increase the length of the intervals between cache flushes because the file system can be recovered if the system crashes before the cache changes have been flushed to disk. This gain over the caching performance of lazy write file systems makes up for, and often exceeds, the overhead of the recoverable file system's logging activity.

Neither careful write nor lazy write file systems guarantee protection of user file data. If the system crashes while an application is writing a file, the file can be lost or corrupted. Worse, the crash can corrupt a lazy write file system, destroying existing files or even rendering an entire volume inaccessible.

The NTFS recoverable file system implements several strategies that improve its reliability over that of the traditional file systems. First, NTFS recoverability guarantees that the volume structure won't be corrupted, so all files will remain accessible after a system failure.

Second, although NTFS doesn't guarantee protection of user data in the event of a system crash—some changes can be lost from the cache—applications can take advantage of the NTFS write-through and cache-flushing capabilities to ensure that file modifications are recorded on disk at appropriate intervals. Both *cache write-through*—forcing write operations to be immediately recorded on disk—and *cache flushing*—forcing cache contents to be written to disk—are efficient operations. NTFS doesn't have to do extra disk I/O to flush modifications to several different file system data structures because changes to the data structures are recorded—in a single write operation—in the log file; if a failure occurs and cache contents are lost, the file system modifications can be recovered from the log. Furthermore, unlike the FAT file system, NTFS guarantees that user data will be consistent and available immediately after a write-through operation or a cache flush, even if the system subsequently fails.

Logging

NTFS provides file system recoverability by means of a transaction-processing technique called *logging*. In NTFS logging, the suboperations of any transaction that alters important file system data structures are recorded in a log file before they are carried through on the disk. That way, if the system crashes, partially completed transactions can be redone or undone when the system comes back online. In transaction processing, this technique is known as *write-ahead logging*. In NTFS, transactions include writing to the disk or deleting a file and can be made up of several suboperations.

Log File Service (LFS)

The log file service (LFS) is a series of kernel-mode routines inside the NTFS driver that NTFS uses to access the log file. Although originally designed to provide logging and recovery services for

more than one client, the LFS is used only by NTFS. The caller—NTFS in this case—passes the LFS a pointer to an open file object, which specifies a log file to be accessed. The LFS either initializes a new log file or calls the Windows 2000 cache manager to access the existing log file through the cache, as shown in Figure 12-34.

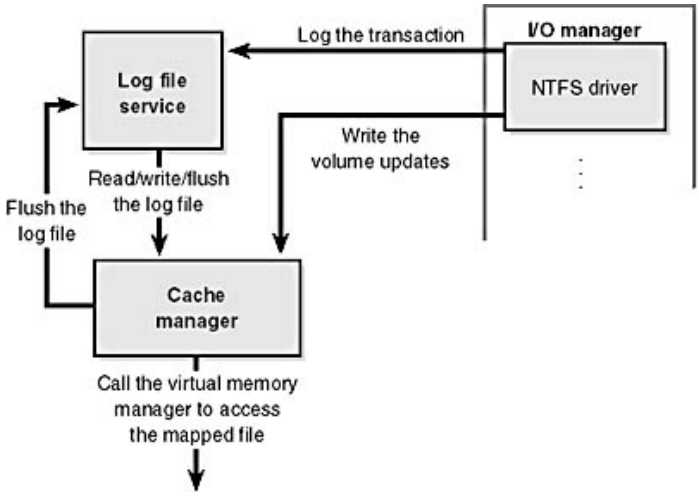


Figure 12-34 *Log file service (LFS)*

The LFS divides the log file into two regions: a *restart area* and an "infinite" *logging area,* as shown in Figure 12-35.

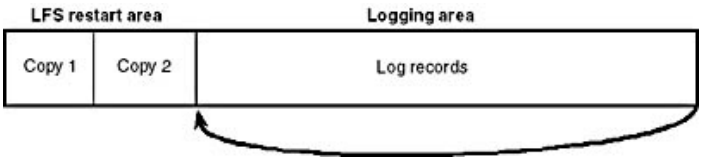


Figure 12-35 *Log file regions*

NTFS calls the LFS to read and write the restart area. NTFS uses the restart area to store context information such as the location in the logging area at which NTFS will begin to read during recovery after a system failure. The LFS maintains a second copy of the restart data in case the first becomes corrupted or otherwise inaccessible. The remainder of the log file is the logging area, which contains transaction records NTFS writes in order to recover a volume in the event of a system failure. The LFS makes the log file appear infinite by reusing it circularly (while guaranteeing that it doesn't overwrite information it needs). The LFS uses *logical sequence numbers* (LSNs) to identify records written to the log file. As the LFS cycles through the file, it increases the values of the LSNs. NTFS uses 64 bits to represent LSNs, so the number of possible LSNs is so large as to be virtually infinite.

NTFS never reads transactions from or writes transactions to the log file directly. The LFS provides services NTFS calls to open the log file, write log records, read log records in forward or backward order, flush log records up to a particular LSN, or set the beginning of the log file to a higher LSN. During recovery, NTFS calls the LFS to perform the following actions: read forward through the log records to redo any transactions that were recorded in the log file but weren't flushed to disk at the time of the system failure; read backward through the log records to undo, or roll back, any transactions that weren't completely logged before the crash; and set the beginning of the log file to a record with a higher LSN when NTFS no longer needs the older transaction records in the log file.

Here's how the system guarantees that the volume can be recovered:

1. NTFS first calls the LFS to record in the (cached) log file any transactions that will modify the

volume structure.

2. NTFS modifies the volume (also in the cache).

3. The cache manager prompts the LFS to flush the log file to disk. (The LFS implements the flush by calling the cache manager back, telling it which pages of memory to flush. Refer back to the calling sequence shown in Figure 12-34.)

4. After the cache manager flushes the log file to disk, it flushes the volume changes (the metadata operations themselves) to disk.

These steps ensure that if the file system modifications are ultimately unsuccessful, the corresponding transactions can be retrieved from the log file and can be either redone or undone as part of the file system recovery procedure.

File system recovery begins automatically the first time the volume is used after the system is rebooted. NTFS checks whether the transactions that were recorded in the log file before the crash were applied to the volume, and if they weren't, it redoes them. NTFS also guarantees that transactions not completely logged before the crash are undone so that they don't appear on the volume.

Log Record Types

The LFS allows its clients to write any kind of record to their log files. NTFS writes several types of records. Two types, *update records* and *checkpoint records*, are described here.

Update records Update records are the most common type of record NTFS writes to the log file. Each update record contains two kinds of information:

- **Redo information** How to reapply one suboperation of a fully logged ("committed") transaction to the volume if a system failure occurs before the transaction is flushed from the cache

- **Undo information** How to reverse one suboperation of a transaction that was only partially logged ("not committed") at the time of a system failure

Figure 12-36 shows three update records in the log file. Each record represents one suboperation of a transaction, creating a new file. The redo entry in each update record tells NTFS how to reapply the suboperation to the volume, and the undo entry tells NTFS how to roll back (undo) the suboperation.

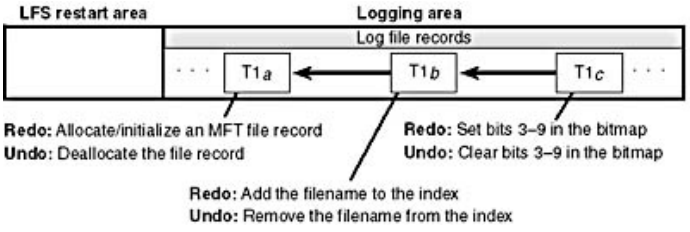


Figure 12-36 *Update records in the log file*

After logging a transaction (in this example, by calling the LFS to write the three update records to the log file), NTFS performs the suboperations on the volume itself, in the cache. When it has finished updating the cache, NTFS writes another record to the log file, recording the entire transaction as complete—a suboperation known as *committing* a transaction. Once a transaction is committed, NTFS guarantees that the entire transaction will appear on the volume, even if the operating system subsequently fails.

When recovering after a system failure, NTFS reads through the log file and redoes each committed transaction. Although NTFS completed the committed transactions before the system failure, it doesn't know whether the cache manager flushed the volume modifications to disk in time. The updates might have been lost from the cache when the system failed. Therefore, NTFS executes the committed transactions again just to be sure that the disk is up to date.

After redoing the committed transactions during a file system recovery, NTFS locates all the transactions in the log file that weren't committed at failure and rolls back (undoes) each suboperation that had been logged. In Figure 12-36, NTFS would first undo the $T1_c$ suboperation and then follow the backward pointer to $T1_b$ and undo that suboperation. It would continue to follow the backward pointers, undoing suboperations, until it reached the first suboperation in the transaction. By following the pointers, NTFS knows how many and which update records it must undo to roll back a transaction.

Redo and undo information can be expressed either physically or logically. Physical descriptions specify volume updates in terms of particular byte ranges on the disk that are to be changed, moved, and so on. Logical descriptions express updates in terms of operations such as "delete file A.dat." As the lowest layer of software maintaining the file system structure, NTFS writes update records with physical descriptions. Transaction-processing or other application-level software might benefit from writing update records in logical terms, however, because logically expressed updates are more compact than physically expressed ones. Logical descriptions necessarily depend on NTFS to understand what operations, such as deleting a file, involve.

NTFS writes update records (usually several) for each of the following transactions:

- Creating a file
- Deleting a file
- Extending a file
- Truncating a file
- Setting file information
- Renaming a file
- Changing the security applied to a file

The redo and undo information in an update record must be carefully designed because although NTFS undoes a transaction, recovers from a system failure, or even operates normally, it might try to redo a transaction that has already been done or, conversely, to undo a transaction that never occurred or that has already been undone. Similarly, NTFS might try to redo or undo a transaction consisting of several update records, only some of which are complete on disk. The format of the update records must ensure that executing redundant redo or undo operations is *idempotent*, that is, has a neutral effect. For example, setting a bit that is already set has no effect, but toggling a bit that has already been toggled does. The file system must also handle intermediate volume states correctly.

Checkpoint records In addition to update records, NTFS periodically writes a checkpoint record to the log file, as illustrated in Figure 12-37.

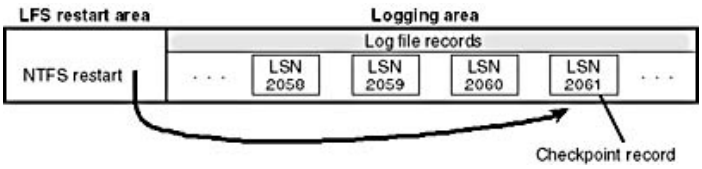


Figure 12-37 *Checkpoint record in the log file*

A checkpoint record helps NTFS determine what processing would be needed to recover a volume if a crash were to occur immediately. Using information stored in the checkpoint record, NTFS knows, for example, how far back in the log file it must go to begin its recovery. After writing a checkpoint record, NTFS stores the LSN of the record in the restart area so that it can quickly find its most recently written checkpoint record when it begins file system recovery after a crash occurs.

Although the LFS presents the log file to NTFS as if it were infinitely large, it isn't. The generous size of the log file and the frequent writing of checkpoint records (an operation that usually frees up space in the log file) make the possibility of the log file's filling up a remote one. Nevertheless, the LFS accounts for this possibility by tracking several numbers:

- The available log space
- The amount of space needed to write an incoming log record and to undo the write, should that be necessary
- The amount of space needed to roll back all active (noncommitted) transactions, should that be necessary

If the log file doesn't contain enough available space to accommodate the total of the last two items, the LFS returns a "log file full" error and NTFS raises an exception. The NTFS exception handler rolls back the current transaction and places it in a queue to be restarted later.

To free up space in the log file, NTFS must momentarily prevent further transactions on files. To do so, NTFS blocks file creation and deletion and then requests exclusive access to all system files and shared access to all user files. Gradually, active transactions either are completed successfully or receive the "log file full" exception. NTFS rolls back and queues the transactions that receive the exception.

Once it has blocked transaction activity on files as just described, NTFS calls the cache manager to flush unwritten data to disk, including unwritten log file data. After everything is safely flushed to disk, NTFS no longer needs the data in the log file. It resets the beginning of the log file to the current position, making the log file "empty." Then it restarts the queued transactions. Beyond the short pause in I/O processing, the "log file full" error has no effect on executing programs.

This scenario is one example of how NTFS uses the log file not only for file system recovery but also for error recovery during normal operation. You'll find out more about error recovery in the following section.

Recovery

NTFS automatically performs a disk recovery the first time a program accesses an NTFS volume after the system has been booted. (If no recovery is needed, the process is trivial.) Recovery depends on two tables NTFS maintains in memory:

- The *transaction table* keeps track of transactions that have been started but aren't yet committed. The suboperations of these active transactions must be removed from the disk during recovery.

- The *dirty page table* records which pages in the cache contain modifications to the file system structure that haven't yet been written to disk. This data must be flushed to disk during recovery.

NTFS writes a checkpoint record to the log file once every 5 seconds. Just before it does, it calls the LFS to store a current copy of the transaction table and of the dirty page table in the log file. NTFS then records in the checkpoint record the LSNs of the log records containing the copied tables. When recovery begins after a system failure, NTFS calls the LFS to locate the log records containing the most recent checkpoint record and the most recent copies of the transaction and dirty page tables. It then copies the tables to memory.

The log file usually contains more update records following the last checkpoint record. These update records represent volume modifications that occurred after the last checkpoint record was written. NTFS must update the transaction and dirty page tables to include these operations. After updating the tables, NTFS uses the tables and the contents of the log file to update the volume itself.

To effect its volume recovery, NTFS scans the log file three times, loading the file into memory during the first pass to minimize disk I/O. Each pass has a particular purpose:

1. Analysis
2. Redoing transactions
3. Undoing transactions

Analysis Pass

During the *analysis pass*, as shown in Figure 12-38, NTFS scans forward in the log file from the beginning of the last checkpoint operation to find update records and use them to update the transaction and dirty page tables it copied to memory. Notice in the figure that the checkpoint operation stores three records in the log file and that update records might be interspersed among these records. NTFS therefore must start its scan at the beginning of the checkpoint operation.

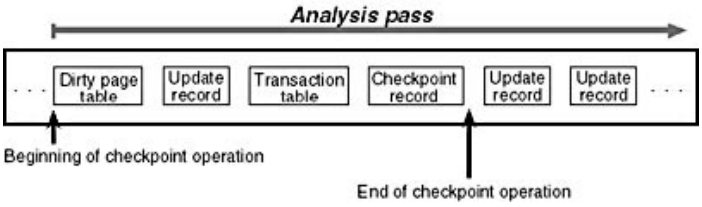


Figure 12-38 *Analysis pass*

Most update records that appear in the log file after the checkpoint operation begins represent a modification to either the transaction table or the dirty page table. If an update record is a "transaction committed" record, for example, the transaction the record represents must be removed from the transaction table. Similarly, if the update record is a "page update" record that modifies a file system data structure, the dirty page table must be updated to reflect that change.

Once the tables are up to date in memory, NTFS scans the tables to determine the LSN of the oldest update record that logs an operation that hasn't been carried out on disk. The transaction table contains the LSNs of the noncommitted (incomplete) transactions, and the dirty page table contains the LSNs of records in the cache that haven't been flushed to disk. The LSN of the oldest update record that NTFS finds in these two tables determines where the redo pass will begin. If the last checkpoint record is older, however, NTFS will start the redo pass there instead.

Redo Pass

During the *redo pass*, as shown in Figure 12-39, NTFS scans forward in the log file from the LSN of the oldest update record, which it found during the analysis pass. It looks for "page update" records, which contain volume modifications that were written before the system failure but that might not have been flushed to disk. NTFS redoes these updates in the cache.

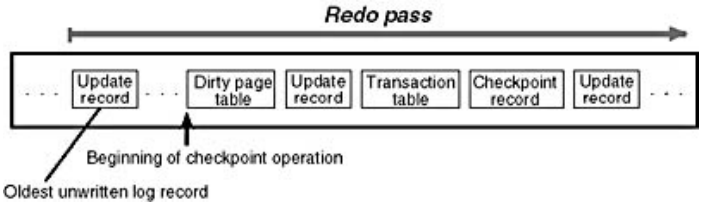


Figure 12-39 *Redo pass*

When NTFS reaches the end of the log file, it has updated the cache with the necessary volume modifications and the cache manager's lazy writer can begin writing cache contents to disk in the background.

Undo Pass

After it completes the redo pass, NTFS begins its *undo pass*, in which it rolls back any transactions that weren't committed when the system failed. Figure 12-40 shows two transactions in the log file; transaction 1 was committed before the power failure, but transaction 2 wasn't. NTFS must undo transaction 2.

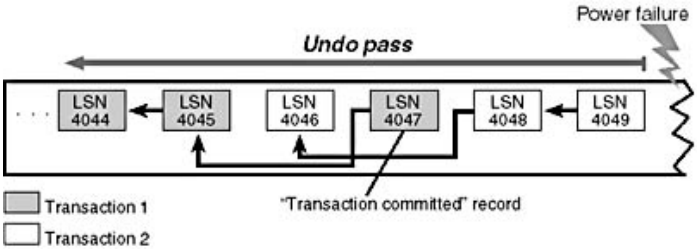


Figure 12-40 *Undo pass*

Suppose that transaction 2 created a file, an operation that comprises three suboperations, each with its own update record. The update records of a transaction are linked by backward pointers in the log file because they are usually not contiguous.

The NTFS transaction table lists the LSN of the last-logged update record for each noncommitted transaction. In this example, the transaction table identifies LSN 4049 as the last update record logged for transaction 2. As shown from right to left in Figure 12-41, NTFS rolls back transaction 2.

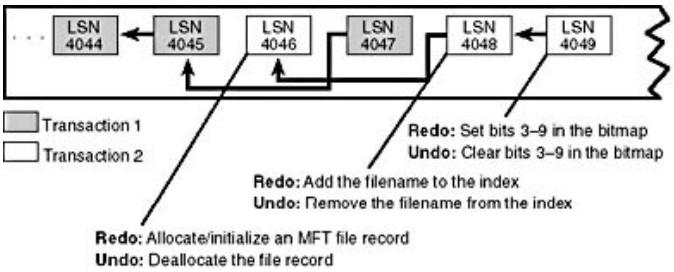


Figure 12-41 *Undoing a transaction*

After locating LSN 4049, NTFS finds the undo information and executes it, clearing bits 3 through 9 in its allocation bitmap. NTFS then follows the backward pointer to LSN 4048, which directs it to

remove the new filename from the appropriate filename index. Finally, it follows the last backward pointer and deallocates the MFT file record reserved for the file, as the update record with LSN 4046 specifies. Transaction 2 is now rolled back. If there are other noncommitted transactions to undo, NTFS follows the same procedure to roll them back. Because undoing transactions affects the volume's file system structure, NTFS must log the undo operations in the log file. After all, the power might fail again during the recovery, and NTFS would have to redo its undo operations!

When the undo pass of the recovery is finished, the volume has been restored to a consistent state. At this point, NTFS flushes the cache changes to disk to ensure that the volume is up to date. NTFS then writes an "empty" LFS restart area to indicate that the volume is consistent and that no recovery need be done if the system should fail again immediately. Recovery is complete.

NTFS guarantees that recovery will return the volume to some preexisting consistent state, but not necessarily to the state that existed just before the system crash. NTFS can't make that guarantee because, for performance, it uses a "lazy commit" algorithm, which means that the log file isn't immediately flushed to disk each time a "transaction committed" record is written. Instead, numerous "transaction committed" records are batched and written together, either when the cache manager calls the LFS to flush the log file to disk or when the LFS writes a checkpoint record (once every 5 seconds) to the log file. Another reason the recovered volume might not be completely up to date is that several parallel transactions might be active when the system crashes and some of their "transaction committed" records might make it to disk whereas others might not. The consistent volume that recovery produces includes all the volume updates whose "transaction committed" records made it to disk and none of the updates whose "transaction committed" records didn't make it to disk.

NTFS uses the log file to recover a volume after the system fails, but it also takes advantage of an important "freebie" it gets from logging transactions. File systems necessarily contain a lot of code devoted to recovering from file system errors that occur during the course of normal file I/O. Because NTFS logs each transaction that modifies the volume structure, it can use the log file to recover when a file system error occurs and thus can greatly simplify its error handling code. The "log file full" error described earlier is one example of using the log file for error recovery.

Most I/O errors a program receives aren't file system errors and therefore can't be resolved entirely by NTFS. When called to create a file, for example, NTFS might begin by creating a file record in the MFT and then enter the new file's name in a directory index. When it tries to allocate space for the file in its bitmap, however, it could discover that the disk is full and the create request can't be completed. In such a case, NTFS uses the information in the log file to undo the part of the operation it has already completed and to deallocate the data structures it reserved for the file. Then it returns a "disk full" error to the caller, which in turn must respond appropriately to the error.

[Previous] [Next]

NTFS Bad-Cluster Recovery

The volume managers included with Windows 2000, FtDisk (for basic disks) and Logical Disk Manager (LDM, for dynamic disks), can recover data from a bad sector on a fault tolerant volume, but if the hard disk doesn't use the SCSI protocol or runs out of spare sectors, a volume manager can't perform sector sparing to replace the bad sector. (See Chapter 10 for more information on the volume managers.) When the file system reads from the sector, the volume manager instead recovers the data and returns the warning to the file system that there is only one copy of the data.

The FAT file system doesn't respond to this volume manager warning. Moreover, neither these file systems nor the volume managers keep track of the bad sectors, so a user must run the Chkdsk or Format utility to prevent a volume manager from repeatedly recovering data for the file system.

Both Chkdsk and Format are less than ideal for removing bad sectors from use. Chkdsk can take a long time to find and remove bad sectors, and Format wipes all the data off the partition it's formatting.

In the file system equivalent of a volume manager's sector sparing, NTFS dynamically replaces the cluster containing a bad sector and keeps track of the bad cluster so that it won't be reused. (Recall that NTFS maintains portability by addressing logical clusters rather than physical sectors.) NTFS performs these functions when the volume manager can't perform sector sparing. When a volume manager returns a bad-sector warning or when the hard disk driver returns a bad-sector error, NTFS allocates a new cluster to replace the one containing the bad sector. NTFS copies the data that the volume manager has recovered into the new cluster to reestablish data redundancy.

Figure 12-42 shows an MFT record for a user file with a bad cluster in one of its data runs as it exited before the cluster went bad. When it receives a bad-sector error, NTFS reassigns the cluster containing the sector to its bad-cluster file. This prevents the bad cluster from being allocated to another file. NTFS then allocates a new cluster for the file and changes the file's VCN-to-LCN mappings to point to the new cluster. This bad-cluster remapping (introduced earlier in this chapter) is illustrated in Figure 12-43. Cluster number 1357, which contains the bad sector, is replaced by a new cluster, number 1049.

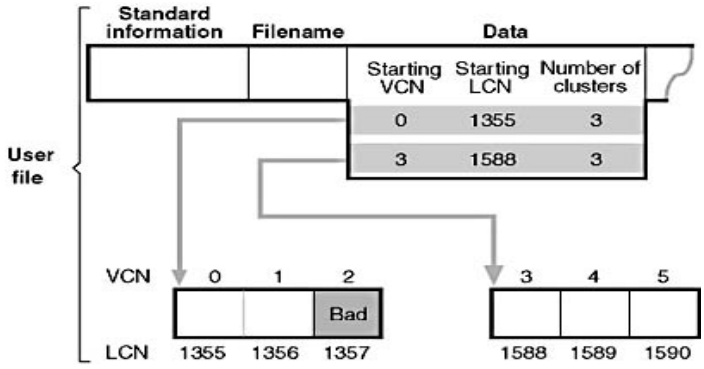


Figure 12-42 *MFT record for a user file with a bad cluster*

Bad-sector errors are undesirable, but when they do occur, the combination of NTFS and volume managers provides the best possible solution. If the bad sector is on a redundant volume, the volume manager recovers the data and replaces the sector if it can. If it can't replace the sector, it returns a warning to NTFS and NTFS replaces the cluster containing the bad sector.

If the volume isn't configured as a redundant volume, the data in the bad sector can't be recovered. When the volume is formatted as a FAT volume and the volume manager can't recover the data, reading from the bad sector yields indeterminate results. If some of the file system's control structures reside in the bad sector, an entire file or group of files (or potentially, the whole disk) can be lost. At best, some data in the affected file (often, all the data in the file beyond the bad sector) is lost. Moreover, the FAT file system is likely to reallocate the bad sector to the same or another file on the volume, causing the problem to resurface.

Like the other file systems, NTFS can't recover data from a bad sector without help from a volume manager. However, NTFS greatly contains the damage a bad sector can cause. If NTFS discovers the bad sector during a read operation, it remaps the cluster the sector is in, as shown in Figure 12-43. If the volume isn't configured as a redundant volume, NTFS returns a "data read" error to the calling program. Although the data that was in that cluster is lost, the rest of the file—and the file system—remains intact; the calling program can respond appropriately to the data loss; and the bad cluster won't be reused in future allocations. If NTFS discovers the bad cluster on a write operation rather than a read, NTFS remaps the cluster before writing and thus loses no data and generates no error.

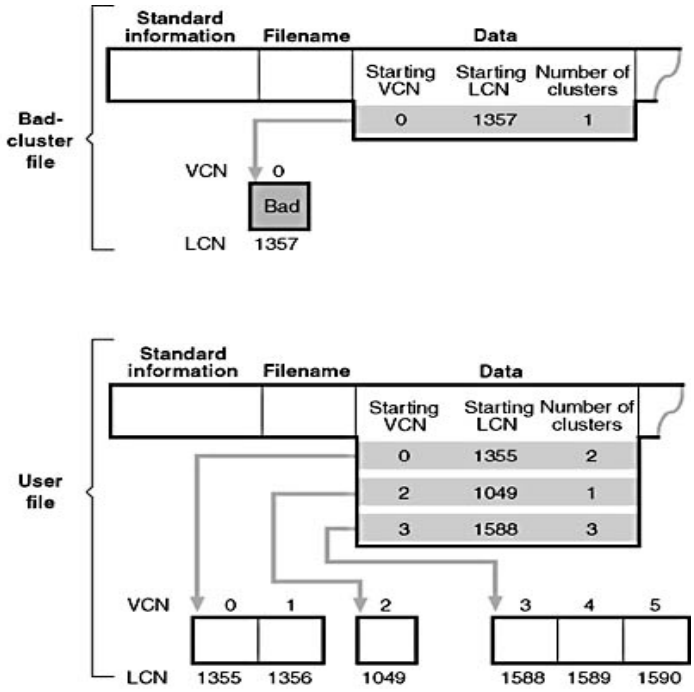


Figure 12-43 *Bad-cluster remapping*

The same recovery procedures are followed if file system data is stored in a sector that goes bad. If the bad sector is on a redundant volume, NTFS replaces the cluster dynamically, using the data recovered by the volume manager. If the volume isn't redundant, the data can't be recovered and NTFS sets a bit in the volume file that indicates corruption on the volume. The NTFS Chkdsk utility checks this bit when the system is next rebooted, and if the bit is set, Chkdsk executes, fixing the file system corruption by reconstructing the NTFS metadata.

In rare instances, file system corruption can occur even on a fault tolerant disk configuration. A double error can destroy both file system data and the means to reconstruct it. If the system crashes while NTFS is writing the mirror copy of an MFT file record, of a filename index, or of the log file, for example, the mirror copy of such file system data might not be fully updated. If the system were rebooted and a bad-sector error occurred on the primary disk at exactly the same location as the incomplete write on the disk mirror, NTFS would be unable to recover the correct data from the disk mirror. NTFS implements a special scheme for detecting such corruptions in file system data. If it ever finds an inconsistency, it sets the corruption bit in the volume file, which causes Chkdsk to reconstruct the NTFS metadata when the system is next rebooted. Because file system corruption is rare on a fault tolerant disk configuration, Chkdsk is seldom needed. It is supplied as a safety precaution rather than as a first-line data recovery strategy.

The use of Chkdsk on NTFS is vastly different from its use on the FAT file system. Before writing anything to disk, FAT sets the volume's dirty bit and then resets the bit after the modification is complete. If any I/O operation is in progress when the system crashes, the dirty bit is left set and Chkdsk runs when the system is rebooted. On NTFS, Chkdsk runs only when unexpected or unreadable file system data is found and NTFS can't recover the data from a redundant volume or from redundant file system structures on a single volume. (The system boot sector is duplicated, as are the parts of the MFT required for booting the system and running the NTFS recovery procedure. This redundancy ensures that NTFS will always be able to boot and recover itself.)

Table 12-6 summarizes what happens when a sector goes bad on a disk volume formatted for one of the Windows 2000-supported file systems according to various conditions that we've described in this section.

Table 12-6 *Summary of NTFS Data Recovery Scenarios*

Scenario	With a SCSI disk that has spare sectors	With a non-SCSI disk or a disk with no spare sectors*
Fault tolerant volume**	1. Volume manager recovers the data. 2. Volume manager performs *sector sparing* (replaces the bad sector). 3. File system remains unaware of the error.	1. Volume manager recovers the data. 2. Volume manager sends the data and a bad-sector error to the file system. 3. NTFS performs *cluster remapping.*
Non-fault-tolerant volume	1. Volume manager can't recover the data. 2. Volume manager sends a bad-sector error to the file system. 3. NTFS performs *cluster remapping.* Data is lost†.	1. Volume manager can't recover the data. 2. Volume manager sends a bad-sector error to the file system. 3. NTFS performs *cluster remapping.* Data is lost†.

* In neither of these cases can a volume manager perform sector sparing: (1) hard disks that don't use the SCSI protocol have no standard interface for providing sector sparing; (2) some hard disks don't provide hardware support for sector sparing, and SCSI hard disks that do provide sector sparing can eventually run out of spare sectors if a lot of sectors go bad.

** A fault tolerant volume is one of the following: a mirror set or a RAID-5 set.

† In a write operation, no data is lost: NTFS remaps the cluster before the write.

If the volume on which the bad sector appears is a fault tolerant volume (a mirrored or RAID-5 volume), and if the hard disk is one that supports sector sparing (and that hasn't run out of spare sectors), which file system you're using—FAT or NTFS—doesn't matter. The volume manager replaces the bad sector without the need for user or file system intervention.

If a bad sector is located on a hard disk that doesn't support sector sparing, the file system is responsible for replacing (remapping) the bad sector or—in the case of NTFS—the cluster in which the bad sector resides. The FAT file system doesn't provide sector or cluster remapping. The benefits of NTFS cluster remapping are that bad spots in a file can be fixed without harm to the file (or harm to the file system, as the case may be) and that the bad cluster won't be reallocated to the same or another file.

[Previous] [Next]

Encrypting File System Security

EFS security relies on Windows 2000 cryptography support, which Microsoft introduced in Windows NT 4. The first time a file is encrypted, EFS assigns the account of the user performing the encryption a private/public key pair for use in file encryption. Users can encrypt files via Windows Explorer by opening a file's Properties dialog box, pressing Advanced, and selecting the Encrypt Contents To Secure Data option, as shown in Figure 12-44. Users can also encrypt files via a command-line utility named *cipher*. Windows 2000 automatically encrypts files that reside in directories that are designated as encrypted directories. When a file is encrypted, EFS generates a random number for the file that EFS calls the file's file encryption key (FEK). EFS uses the FEK to encrypt the file's contents with a stronger variant of the Data Encryption Standard (DES) algorithm—DESX. EFS stores the file's FEK with the file but encrypts the file with the user's EFS public key by using the RSA public key-based encryption algorithm. After EFS completes these

steps, the file is secure: other users can't decrypt the data without the file's decrypted FEK, and they can't decrypt the FEK without the private key.

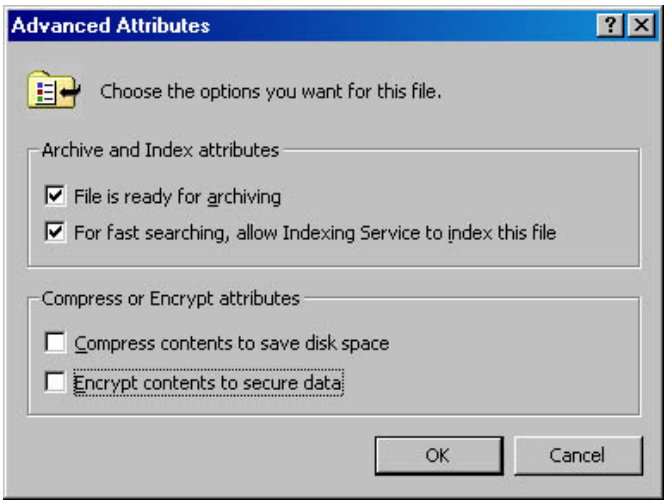


Figure 12-44 *Encrypt files by using the Advanced Attributes dialog box*

EFS uses a private/public key algorithm to encrypt FEKs. To encrypt file data, EFS uses DESX because DESX is a *symmetric encryption algorithm*, which means it uses the same key to encrypt and decrypt data. Symmetric encryption algorithms are typically very fast, which makes them suitable for encrypting large amounts of data, such as file data. However, symmetric encryption algorithms have a weakness: you can bypass their security if you obtain the key. If multiple users want to share one encrypted file protected only by DESX, each user would require access to the file's FEK. Leaving the FEK unencrypted would obviously be a security problem, but encrypting the FEK once would require all the users to share the same FEK decryption key—another potential security problem.

Keeping the FEK secure is a difficult problem, which EFS addresses with the public key-based half of its encryption architecture. Encrypting a file's FEK for individual users who access the file lets multiple users share an encrypted file. EFS can encrypt a file's FEK with each user's public key and can store each user's encrypted FEK with the file. Anyone can access a user's public key, but no one can use a public key to decrypt the data that the public key encrypted. The only way users can decrypt a file is with their private key, which the operating system must access and typically stores in a secure location. A user's private key decrypts the user's encrypted copy of a file's FEK. Windows 2000 stores private keys on a computer's hard disk (which isn't terribly secure), but subsequent releases of the operating system will let users store their private key on portable media such as smart cards. Public key-based algorithms are usually slow, but EFS uses these algorithms only to encrypt FEKs. Splitting key management between a publicly available key and a private key makes key management a little easier than symmetric encryption algorithms do and solves the dilemma of keeping the FEK secure.

Several components work together to make EFS work, as the diagram of EFS architecture in Figure 12-45 shows. As you can see, EFS is implemented as a device driver that runs in kernel mode and is tightly connected with the NTFS file system driver. Whenever NTFS encounters an encrypted file, NTFS executes functions in the EFS driver that the EFS driver registered with NTFS when EFS initialized. The EFS functions encrypt and decrypt file data as applications access encrypted files. Although EFS stores an FEK with a file's data, users' public keys encrypt the FEK. To encrypt or decrypt file data, EFS must decrypt the file's FEK with the aid of cryptography services that reside in user mode.

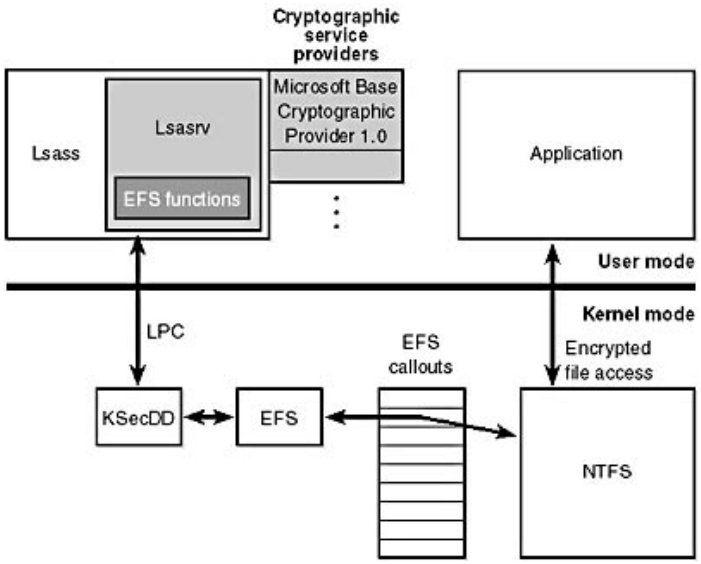


Figure 12-45 *EFS architecture*

The Local Security Authority Subsystem (Lsass - \Winnt\System32\Lsass.exe) manages logon sessions but also handles EFS key management chores. For example, when the EFS driver needs to decrypt an FEK in order to decrypt file data a user wants to access, the EFS driver sends a request to Lsass. EFS sends the request via a local procedure call (LPC). The KSecDD (\Winnt\System32 \Drivers\Ksecdd.sys) device driver exports functions for other drivers that need to send LPC messages to Lsass. The Local Security Authority Server (Lsasrv - \Winnt\System32\Lsasrv.dll) component of Lsass that listens for remote procedure call (RPC) requests passes requests to decrypt an FEK to the appropriate EFS-related decryption function, which also resides in Lsasrv. Lsasrv uses functions in Microsoft's CryptoAPI (also referred to as CAPI) to decrypt the FEK, which the EFS driver sent to Lsass in encrypted form.

CryptoAPI comprises cryptographic service provider (CSP) DLLs that make various cryptography services (such as encryption/decryption and hashing) available to applications. The CSP DLLs manage retrieval of user private and public keys, for example, so that Lsasrv doesn't need to concern itself with the details of how keys are protected or even with the details of the encryption algorithms. After Lsasrv decrypts an FEK, Lsasrv returns the FEK to the EFS driver via an LPC reply message. After EFS receives the decrypted FEK, EFS can use DESX to decrypt the file data for NTFS. Let's look at the details of how EFS integrates with NTFS and how Lsasrv uses CryptoAPI to manage keys.

Registering Callbacks

NTFS doesn't require the EFS driver's (Winnt\System32\Drivers\Efs.sys) presence to execute, but encrypted files won't be accessible if the EFS driver isn't present. NTFS has a plug-in interface for the EFS driver, so when the EFS driver initializes, it can attach itself to NTFS. The NTFS driver exports several functions for the EFS driver to use, including one that EFS calls to notify NTFS both of the presence of EFS and of the EFS-related APIs EFS is making available.

Encrypting a File for the First Time

The NTFS driver calls only the EFS functions that register when NTFS encounters an encrypted file. A file's attributes record that the file is encrypted in the same way that a file records that it is compressed (discussed earlier in this chapter). NTFS and EFS have specific interfaces for converting a file from nonencrypted to encrypted form, but user-mode components primarily drive the process. Windows 2000 lets you encrypt a file in two ways: by using the *cipher* command-line utility or by

checking the Encrypt Contents To Secure Data box in the Advanced Attibutes dialog box for a file in Windows Explorer. Both Windows Explorer and the *cipher* command rely on the *EncryptFile* Win32 API that the Advapi32.dll (Advanced Win32 APIs DLL) exports. Advapi32 loads another DLL, Feclient.dll (File Encryption Client DLL), to obtain APIs that Advapi32 can use to invoke EFS interfaces in Lsasrv via LPCs.

When Lsasrv receives an LPC message from Feclient to encrypt a file, Lsasrv uses the Windows 2000 impersonation facility to impersonate the user that ran the application (either *cipher* or Windows Explorer) that is encrypting the file. This procedure lets Windows 2000 treat the file operations that Lsasrv performs as if the user who wants to encrypt the file is performing them. Lsasrv usually runs in the System account. (The System account is described in Chapter 8.) In fact, if it doesn't impersonate the user, Lsasrv usually won't have permission to access the file in question.

Lsasrv next creates a log file in the volume's System Volume Information directory into which Lsasrv records the progress of the encryption process. The log file usually has the name efs0.log, but if other files are undergoing encryption, increasing numbers replace the 0 until a unique log file name for the current encryption is created.

CryptoAPI relies on information that a user's registry profile stores, so Lsasrv next uses the *LoadUserProfile* API function of Userenv.dll (User Environment DLL) to load the profile into the registry of the user it is impersonating. Typically, the user profile is already loaded, because Winlogon loads a user's profile when a user logs on. However, if a user uses the Windows 2000 *RunAs* command to log on to a different account, when you try to access encrypted files from that account, the account's profile might not load.

Lsasrv then generates an FEK for the file by using the RSA encryption facilities of the Microsoft Base Cryptographic Provider 1.0 CSP.

Constructing Key Rings

At this point, Lsasrv has an FEK and can construct EFS information to store with the file, including an encrypted version of the FEK. Lsasrv reads the HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash value of the user performing the encryption to obtain the user's public key signature. (Note that this key doesn't appear in the registry until a file or folder is encrypted.) Lsasrv uses the signature to access the user's public key and encrypt FEKs.

Lsasrv can now construct the information that EFS stores with the file. EFS stores only one block of information in an encrypted file, and that block contains an entry for each user sharing the file. These entries are called *key entries*, and EFS stores them in the Data Decryption Field (DDF) portion of the file's EFS data. A collection of multiple key entries is called a *key ring*, because, as mentioned earlier, EFS lets multiple users share encrypted files.

Figure 12-46 shows a file's EFS information format and key entry format. EFS stores enough information in the first part of a key entry to precisely describe a user's public key. This data includes the user's security ID (SID), the container name in which the key is stored, the cryptographic provider name, and the private/public keypair certificate hash. The second part of the key entry contains an encrypted version of the FEK. Lsasrv uses the CryptoAPI to encrypt the FEK with the RSA algorithm and the user's public key.

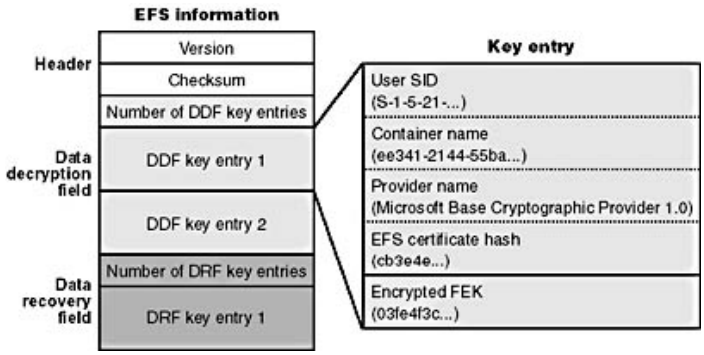


Figure 12-46 *Format of EFS information and key entries*

Next, Lsasrv creates another key ring that contains recovery key entries. EFS stores information about recovery key entries in a file's Data Recovery Field (DRF). The format of DRF entries is identical to the format of DDF entries. The DRF's purpose is to let designated accounts, or Recovery Agents, decrypt a user's file when administrative authority must have access to the user's data. For example, suppose a company employee used a CryptoAPI that let him store his private key on a smart card, and then he lost the card. Without Recovery Agents, no one could recover his encrypted data.

Recovery Agents are defined with the Encrypted Data Recovery Agents security policy of the local computer or domain. This policy is available from the Group Policy MMC snap-in, as shown in Figure 12-47. When you use the Recovery Agent Wizard (by right-clicking on Encrypted Data Recovery Agents and selecting Encrypted Recovery Agent from the New option), you can add Recovery Agents and specify which private/public key pairs (designated by their certificates) the Recovery Agents use for EFS recovery. Lsasrv interprets the recovery policy when it initializes and when it receives notification that the recovery policy has changed. EFS creates a DRF key entry for each Recovery Agent by using the cryptographic provider registered for EFS recovery. The default Recovery Agent provider is the RSA encryption facility of Base Cryptographic Provider 1.0—the same provider Lsasrv uses for user keys.

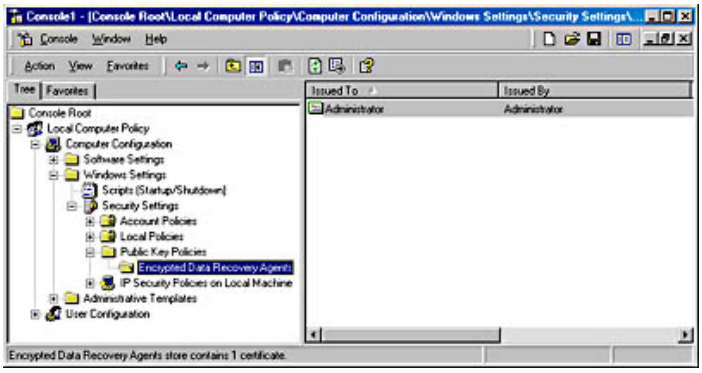


Figure 12-47 *Encrypted Data Recovery Agents group policy*

In the final step in creating EFS information for a file, Lsasrv calculates a checksum for the DDF and DRF by using the MD5 hash facility of Base Cryptographic Provider 1.0. Lsasrv stores the checksum's result in the EFS information header. EFS references this checksum during decryption to ensure that the contents of a file's EFS information haven't become corrupted or been tampered with.

Encrypting File Data

Figure 12-48 illustrates the flow of the encryption process. After Lsasrv constructs the necessary information for a file a user wants to encrypt, it can begin encrypting the file. Lsasrv creates a backup file, Efs0.tmp, for the file undergoing encryption. (Lsasrv uses higher numbers in the backup

filename if other backup files exist.) Lsasrv creates the backup file in the directory that contains the file undergoing encryption. Lsasrv applies a restrictive security descriptor to the backup file so that only the System account can access the file's contents. Lsasrv next initializes the log file that it created in the first phase of the encryption process. Finally, Lsasrv records in the log file that the backup file has been created. Lsasrv encrypts the original file only after the file is completely backed up.

Lsasrv next sends the EFS device driver, through NTFS, a command to add to the original file the EFS information that it just created. NTFS receives this command, but because NTFS doesn't understand EFS commands, NTFS calls the EFS driver. The EFS driver takes the EFS information that Lsasrv sent and uses exported NTFS functions to apply the information to the file. The exported NTFS functions let EFS add the $LOGGED_UTILITY_STREAM attribute to NTFS file. Execution returns to Lsasrv, which copies the contents of the file undergoing encryption to the backup file. When the backup copy is complete, including backups of all alternate data streams, Lsasrv records in the log file that the backup file is up to date. Lsasrv then sends another command to NTFS to tell NTFS to encrypt the contents of the original file.

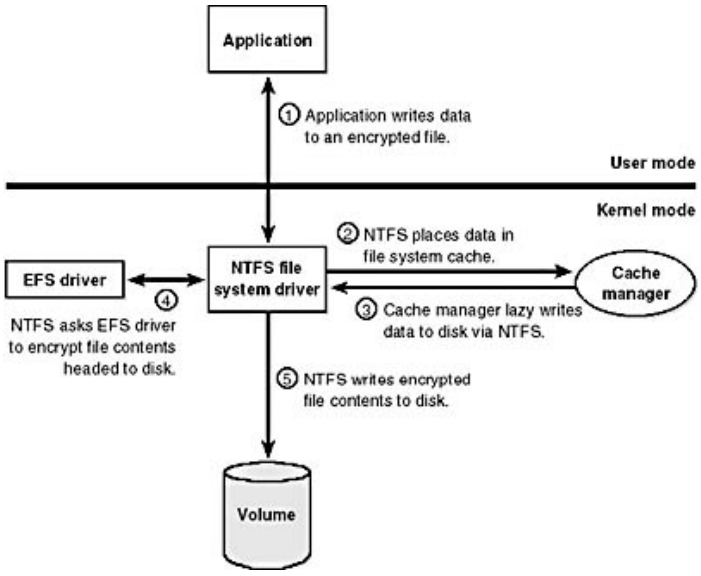


Figure 12-48 *Flow of EFS*

When NTFS receives the EFS command to encrypt the file, NTFS deletes the contents of the original file and copies the backup data to the file. After NTFS copies each section of the file, NTFS flushes the section's data from the file system cache, which prompts the cache manager to tell NTFS to write the file's data to disk. Because the file is marked as encrypted, at this point in the file-writing process, NTFS calls EFS to encrypt the data before NTFS writes the data to disk. EFS uses the unencrypted FEK that NTFS passes it to perform DESX encryption of the file, one sector (512 bytes) at a time.

On Windows 2000 versions approved for export outside the United States, the EFS driver implements a 56-bit key DESX encryption. For the U.S.-only version of Windows 2000, the key is 128 bits long.

After EFS encrypts the file, Lsasrv records in the log file that the encryption was successful and deletes the file's backup copy. Finally, Lsasrv deletes the log file and returns control to the application that requested the file's encryption.

Encryption Process Summary

The following list summarizes the steps EFS performs to encrypt a file:

1. The user profile is loaded if necessary.
2. A log file is created in the System Volume Information directory with the name Efs*x*.log, where *x* is a unique number (for example, Efs0.log). As subsequent steps are performed, records are written to the log so that the file can be recovered in case the system fails during the encryption process.
3. Base Cryptographic Provider 1.0 generates a random 128-bit FEK for the file.
4. A user EFS private/public key pair is generated or obtained. HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash identifies the user's key pair.
5. A DDF key ring is created for the file that has an entry for the user. The entry contains a copy of the FEK that has been encrypted with the user's EFS public key.
6. A DRF key ring is created for the file. It has an entry for each Recovery Agent on the system, with each entry containing a copy of the FEK encrypted with the agent's EFS public key.
7. A backup file with a name in the form Efs0.tmp is created in the same directory as the file to be encrypted.
8. The DDF and DRF key rings are added to a header and augment the file as its EFS attribute.
9. The backup file is marked encrypted, and the original file is copied to the backup.
10. The original file's contents are destroyed, and the backup is copied to the original. This copy operation results in the data in the original file being encrypted because the file is now marked as encrypted.
11. The backup file is deleted.
12. The log file is deleted.
13. The user profile is unloaded (if it was loaded in step 1).

If the system crashes during the encryption process, either the original file remains intact or the backup file contains a consistent copy. When Lsasrv initializes after a system crash, it looks for log files under the System Volume Information subdirectory on each NTFS volume on the system. If Lsasrv finds one or more log files, it examines their contents and determines how recovery should take place. Lsasrv deletes the log file and the corresponding backup file if the original file wasn't modified at the time of the crash; otherwise, Lsasrv copies the backup file over the original, partially encrypted file and then deletes the log and backup. After Lsasrv processes log files, the file system will be in a consistent state with respect to encryption, with no loss of user data.

The Decryption Process

The decryption process begins when a user opens an encrypted file. NTFS examines the file's attributes when opening the file and then executes a callback function in the EFS driver. The EFS driver reads the $LOGGED_UTILITY_STREAM attribute associated with the encrypted file. To read the attribute, the driver calls EFS support functions that NTFS exports for EFS's use. NTFS completes the necessary steps to open the file. The EFS driver ensures that the user opening the file has access privileges to the file's encrypted data (that is, that an encrypted FEK in either the DDF or DRF key rings corresponds to a private/public key pair associated with the user). As EFS performs this validation, EFS obtains the file's decrypted FEK to use in subsequent data operations the user might perform on the file.

EFS can't decrypt an FEK and relies on Lsasrv (which can use CryptoAPI) to perform FEK decryption. EFS sends an LPC message by way of the Ksecdd.sys driver to Lsasrv that asks Lsasrv to obtain the decrypted form of the encrypted FEK in the $LOGGED_UTILITY_STREAM attribute data (the EFS data) that corresponds to the user who is opening the file.

When Lsasrv receives the LPC message, Lsasrv executes the Userenv.dll (User Environment DLL) *LoadUserProfile* API function to bring the user's profile into the registry, if the profile isn't already loaded. Lsasrv proceeds through each key field in the EFS data, using the user's private key to try to decrypt each FEK. For each key, Lsasrv attempts to decrypt a DDF or DRF key entry's FEK. If the certificate hash in a key field doesn't refer to a key the user owns, Lsasrv moves on to the next key field. If Lsasrv can't decrypt any DDF or DRF key field's FEK, the user can't obtain the file's FEK. Consequently, EFS denies access to the application opening the file. However, if Lsasrv identifies a hash as corresponding to a key the user owns, it decrypts the FEK with the user's private key using CryptoAPI.

Because Lsasrv processes both DDF and DRF key rings when decrypting an FEK, it automatically performs file recovery operations. If a Recovery Agent that isn't registered to access an encrypted file (that is, it doesn't have a corresponding field in the DDF key ring) tries to access a file, EFS will let the Recovery Agent gain access because the agent has access to a key pair for a key field in the DRF key ring.

Decrypted FEK Caching

Traveling the path from the EFS driver to Lsasrv and back can take a relatively long time—in the process of decrypting an FEK, CryptoAPI uses results in more than 2000 registry API calls and 400 file system accesses on a typical system. The EFS driver, with the aid of NTFS, uses a cache to try to avoid this expense.

Decrypting File Data

After an application opens an encrypted file, the application can read from and write to the file. NTFS calls the EFS driver to decrypt file data as NTFS reads the data from the disk, and before NTFS places the data in the file system cache. Similarly, when an application writes data to a file, the data remains in unencrypted form in the file system cache until the application or the cache manager uses NTFS to flush the data back to disk. When an encrypted file's data writes back from the cache to the disk, NTFS calls the EFS driver to encrypt the data.

As stated earlier, the EFS driver performs encryption and decryption in 512-byte units. The 512-byte size is the most convenient for the driver because disk reads and writes occur in multiples of the 512-byte sector.

Backing Up Encrypted Files

An important aspect of any file encryption facility's design is that file data is never available in unencrypted form except to applications that access the file via the encryption facility. This restriction particularly affects backup utilities, in which archival media store files. EFS addresses this problem by providing a facility for backup utilities so that the utilities can back up and restore files in their encrypted states. Thus, backup utilities don't have to be able to decrypt file data, nor do they need to decrypt file data in their backup procedures.

Backup utilities use the new EFS API functions *OpenEncryptedFileRaw*, *ReadEncryptedFileRaw*, *WriteEncryptedFileRaw*, and *CloseEncryptedFileRaw* in Windows 2000 to access a file's encrypted contents. The Advapi32.dll library provides these API functions, which all use LPCs to invoke corresponding functions in Lsasrv. For example, after a backup utility opens a file for raw access during a backup operation, the utility calls *ReadEncryptedFileRaw* to obtain the file data. The

Lsasrv function *EfsReadFileRaw* issues control commands (which the EFS session key encrypts with DESX) to the NTFS driver to read the file's EFS attribute first and then the encrypted contents.

EfsReadFileRaw might have to perform multiple read operations to read a large file. As *EfsReadFileRaw* reads each portion of such a file, Lsasrv sends an RPC message to Advapi32.dll that executes a callback function that the backup program specified when it issued the *ReadEncryptedFileRaw* API function. *EfsReadFileRaw* hands the encrypted data it just read to the callback function, which can write the data to the backup media. Backup utilities restore encrypted files in a similar manner. The utilities call the *WriteEncryptedFileRaw* API function, which invokes a callback function in the backup program to obtain the unencrypted data from the backup media while Lsasrv's *EfsWriteFileRaw* function is restoring the file's contents.

EXPERIMENT

Viewing EFS Information

EFS has a handful of other API functions that applications can use to manipulate encrypted files. For example, applications use the *AddUsersToEncryptedFile* API function to give additional users access to an encrypted file and *RemoveUsersFromEncryptedFile* to revoke users' access to an encrypted file. Applications use the *QueryUsersOnEncryptedFile* function to obtain information about a file's associated DDF and DRF key fields. *QueryUsersOnEncryptedFile* returns the SID, certificate hash value, and display information that each DDF and DRF key field contains. The following output is from the EFSDump utility, included on the companion CD under \Sysint\Efsdump.exe, when an encrypted file is specified as a command-line argument:

```
C:\>efsdump test.txt
EFS Information Dumper v1.02
Copyright (C) 1999 Mark Russinovich
Systems Internals - http://www.sysinternals.com

test.txt:
DDF Entry:
    SUSANCOMP\Joe:
         CN=Joe,L=EFS,OU=EFS File Encryption Certificate
DRF Entry:
    SUSANCOMP\Administrator
         OU=EFS File Encryption Certificate, L=EFS, CN=Administrator
```

You can see that the file test.txt has one DDF entry for user Joe and one DRF entry for Administrator, which is the only Recovery Agent currently registered on the system.

[Previous] [Next]

Conclusion

As you saw in the introduction to this chapter, the overriding goal for NTFS was to provide a file system that wasn't only reliable but also fast. The performance of Windows 2000 disk I/O isn't due solely to the implementation of NTFS, however. It comes in large measure from synergy between NTFS and the Windows 2000 cache manager. Together, NTFS and the cache manager achieve respectable I/O performance while providing an unprecedented level of reliability and high-end data storage features for both workstation and server systems.