Chapter 12

File Systems

In this chapter, we present an overview of the file system formats supported by Microsoft Windows. 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. Following that is a description of how to use Filemon (from *www.sysinternals.com*) to troubleshoot a wide variety of file system access problems. In the balance of the chapter, we focus on the on-disk layout of the native Windows file system format called NTFS 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 they affect 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.

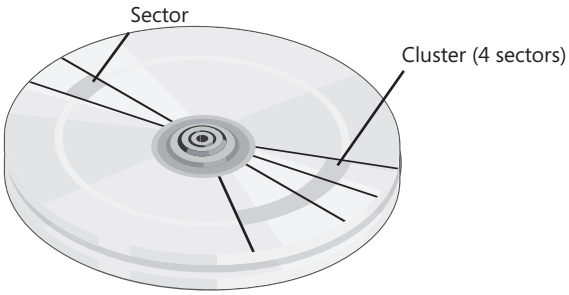


Figure 12-1 Sectors and a cluster on a disk

- *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.

Windows File System Formats

Windows 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 (\Windows\System32\Drivers\Cdfs.sys), or CD-ROM file system, is a read-only file system driver that supports a superset of the ISO-9660 format as well as a superset of the Joliet disk format. While the ISO-9660 format is relatively simple and has limitations such as ASCII uppercase names with a maximum length of 32 characters, Joliet is more flexible and supports UNICODE names of arbitrary length, for example. If structures for both formats are present on a disk (to offer maximum compatibility), CDFS uses the Joliet format. CDFS has a number of restrictions:

- A maximum file size of 4 GB
- A maximum of 65,535 directories

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 UDF file system implementation is OSTA (Optical Storage Technology) UDF-compliant. (UDF is a subset of the ISO-13346 format with extensions for formats such as CD-R and DVD-R/RW.) OSTA defined UDF in 1995 as a format to replace the ISO-9660 format for magneto-optical storage media, mainly DVD-ROM. UDF is included in the DVD specification and is more flexible than CDFS. The UDF driver supports UDF versions 1.02, version 1.5 on Windows 2000, and versions 2.0 and 2.01 on Windows XP and Windows Server 2003. UDF file systems have the following traits:

- Directory and filenames can be 254 ASCII or 127 UNICODE characters long.
- Files can be sparse. (Sparse files are defined later in the chapter.)
- File sizes are specified with 64-bits.

Although the UDF format was designed with rewritable media in mind, the Windows UDF driver (\Windows\System32\Drivers\Udfs.sys) provides read-only support. In addition, Windows does not implement support for other UDF features, including named streams, access control lists, or extended attributes.

FAT12, FAT16, and FAT32

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

The name of 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 212 (4096) clusters. Windows uses cluster sizes from 512 bytes to 8 KB in size, which limits a FAT12 volume size to 32 MB. Therefore, Windows uses FAT12 as the format for all 5-inch floppy disks and 3.5-inch floppy disks, which store up to 1.44 MB of data.

Note All FAT file system types reserve the first two clusters and the last 16 clusters of a volume, so the number of usable clusters for a FAT12 volume, for instance, is slightly less than 4096.

FAT16, with a 16-bit cluster identifier, can address 216 (65,536) clusters. On Windows, 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 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 uses the FAT12 format instead of FAT16.

Table 12-1 Default FAT16 Cluster Sizes in Windows

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.

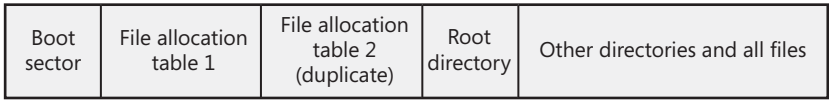


Figure 12-2 FAT format organization

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. Reading a file from a FAT volume can involve reading large portions of a file allocation table to traverse the file's allocation chains.

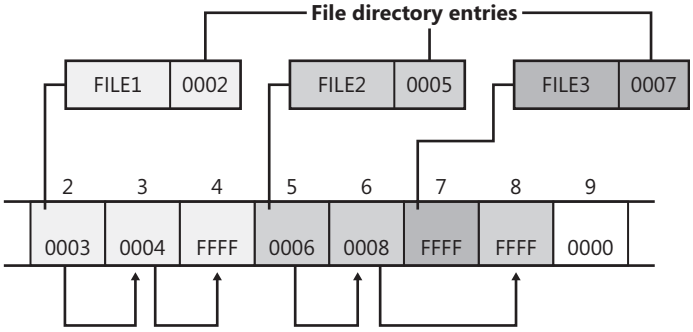


Figure 12-3 Sample FAT file-allocation chains

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 a sample directory entry for a file named "The quick brown fox." The system has created a THEQUI~1.FOX 8.3 representation of the name (that is, 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.

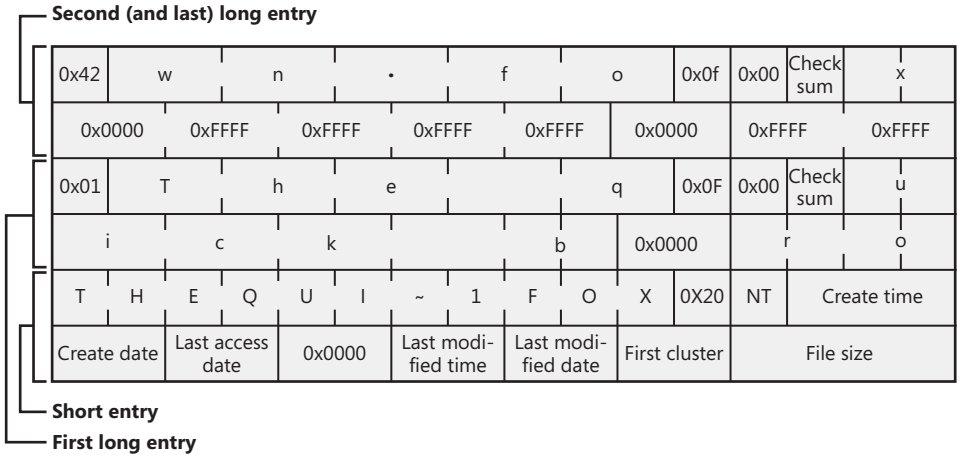


Figure 12-4 FAT directory entry

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 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 manage disks more efficiently 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–8 GB	4 KB
8 GB–16 GB	8 KB
16 GB–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.

Note Windows XP introduces support for FAT32 on DVD-RAM devices.

NTFS

As we said at the beginning of the chapter, the NTFS file system is the native file system format of Windows. NTFS uses 64-bit cluster numbers. This capacity gives NTFS the ability to address volumes of up to 16 exabytes (16 billion GB); however, Windows limits the size of an NTFS volume to that addressable with 32-bit clusters, which is slightly less than 256 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.) NTFS also supports 2^{32}-1 files per volume. The NTFS format allows for files that are 16 exabytes in size, but the implementation limits the maximum file size to 16 TB.

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.

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. For enhanced performance, file system drivers also usually rely on the services of the cache manager. Thus, they use a superset of the exported Ntoskrnl functions that standard drivers use. Whereas you need the Windows DDK in order to build standard

kernel-mode drivers, you must have the Windows Installable File System (IFS) Kit to build file system drivers. (See Chapter 1 for more information on the DDK, and see *www.microsoft.com/whdc/devtools/ifskit* for more information on the IFS Kit.)

Windows 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.

The first sector of every Windows-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.)

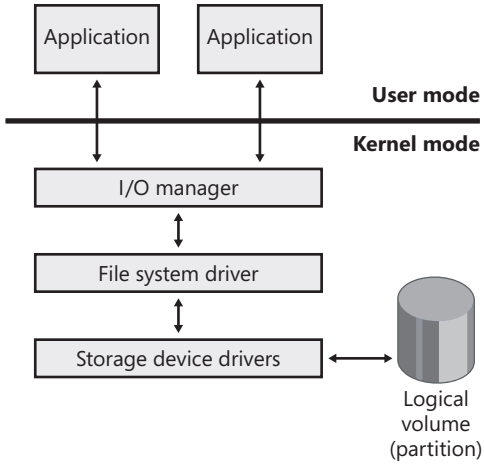


Figure 12-5 Local FSD

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 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 component, which is typically a 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.

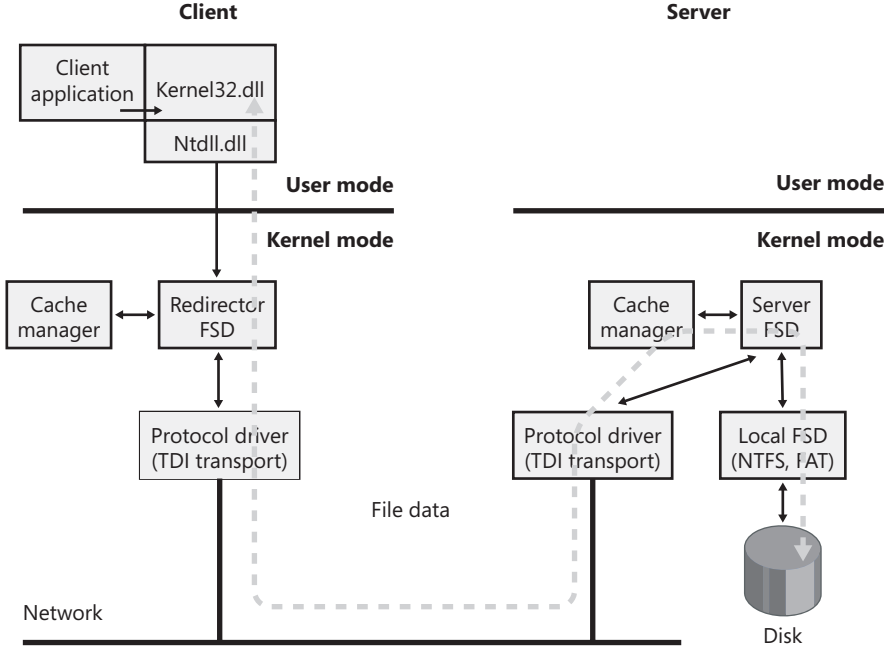


Figure 12-6 CIFS file sharing

Windows includes a client-side remote FSD named LANMan Redirector (redirector) and a server-side remote FSD server named LANMan Server (\Windows\System32\Drivers\Srv.sys - server). The redirector is implemented as a port/miniport driver combination, where the port driver (\Windows\System32\Drivers\Rdbss.sys) is implemented as a driver subroutine library and the miniport (\Windows\System32\Drivers\Mrxsmb.sys) uses services implemented by the port driver. Another redirector miniport driver is WebDAV (\Windows\System32\Drivers\Mrxdav.sys), which implements the client side of file access over HTTP. 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 I/O manager. In addition to the FSD components, both LANMan Redirector and LANMan Server include Windows services named Workstation and Server, respectively. Figure 12-6 shows the relationship bewteen a client accessing files remotely from a server through the redirector and server FSDs.

Windows relies on the Common Internet File System (CIFS) protocol to format messages exchanged between the redirector and the server. CIFS is a 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.

When a client wants to access a server file, it must first request an oplock. The client dictates the kind of caching that the client can perform based on the type of oplock that the server grants.

There are three main types of oplock:

- A Level I oplock is granted when a client has exclusive access to a file. A client holding this type of oplock for a file can cache both reads and writes on the client system.
- A Level II oplock represents a shared file lock. Clients that hold a Level II oplock can cache reads, but writing to the file invalidates the Level II oplock.
- A Batch oplock is the most permissive kind of oplock. A client with this oplock can cache both reads and writes to the file as well as open and close the file without requesting additional oplocks. Batch oplocks are typically used only to support the execution of batch files, which can open and close a file repeatedly as they execute.

If a client has no oplock, it can cache neither read nor write data locally and instead must retrieve data from the server and send all modifications directly to the server.

An example, shown in Figure 12-7, will help illustrate oplock operation. The server automatically grants a Level I oplock to the first client to open a server file for access. The redirector on the client caches the file data for both reads and writes in the file cache of the client machine. If a second client opens the file, it too requests a Level I oplock. However, because there are now two clients accessing the same file, the server must take steps to present a consistent view of the file's data to both clients. If the first client has written to the file, as is the case in Figure 12-7, the server revokes its oplock and grants neither client an oplock. When the first client's oplock is revoked, or *broken*, the client flushes any data it has cached for the file back to the server.

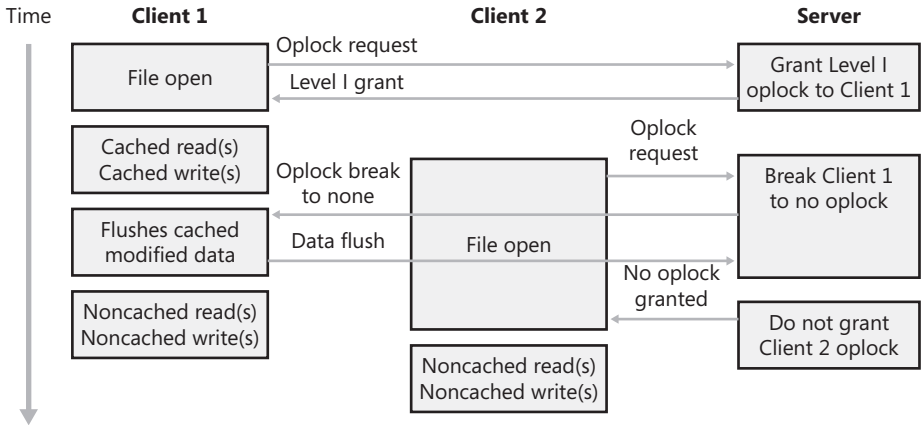
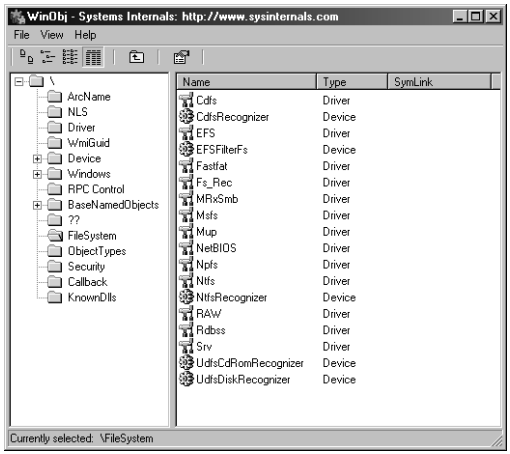


Figure 12-7 Oplock example

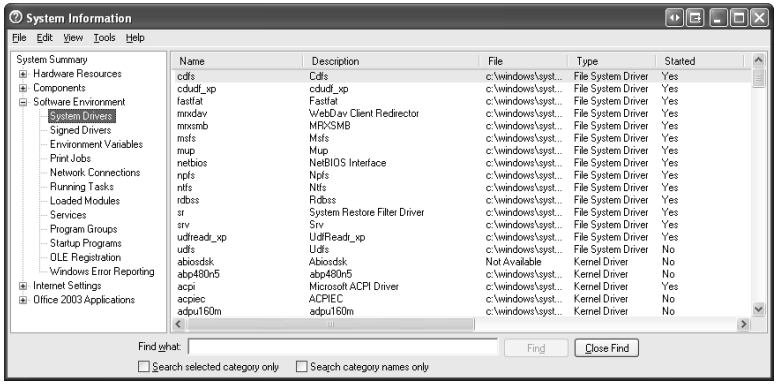
If the first client hadn't written to the file, the first client's oplock would have been broken to a Level II oplock, which is the same type of oplock the server grants to the second client. Now both clients can cache reads, but if either writes to the file, the server revokes their oplocks so that noncached operation commences. Once oplocks are broken, they aren't granted again for the same open instance of a file. However, if a client closes a file and then reopens it, the server reassesses what level of oplock to grant the client based on what other clients have the file open and whether or not at least one of them has written to the file.

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 such as Winobj (from *www.sysinternals.com*), 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.)



Another way to see registered file systems is to run the System Information viewer. On Windows 2000, run the Computer Management MMC snap-in and select Drivers under the Software Environment in the System Information node; on Windows XP and Windows Server 2003, run Msinfo32 from the Start menu's Run dialog box and select System Drivers under Software Environment. Sort the list of drivers by clicking the Type column and drivers with a Type attribute of SERVICE_FILE_SYSTEM_DRIVER group together.



Note that just because a driver registers as a file system driver type doesn't mean that it is a local or remote FSD. For example, Npfs (Named Pipe File System), which is visible in the list just shown, is a network API driver that supports named pipes but implements a private namespace, and therefore, is in some ways like a file system driver. See Chapter 13 for an experiment that reveals the Npfs namespace.

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 12-8 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 and mapped page writers
- 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

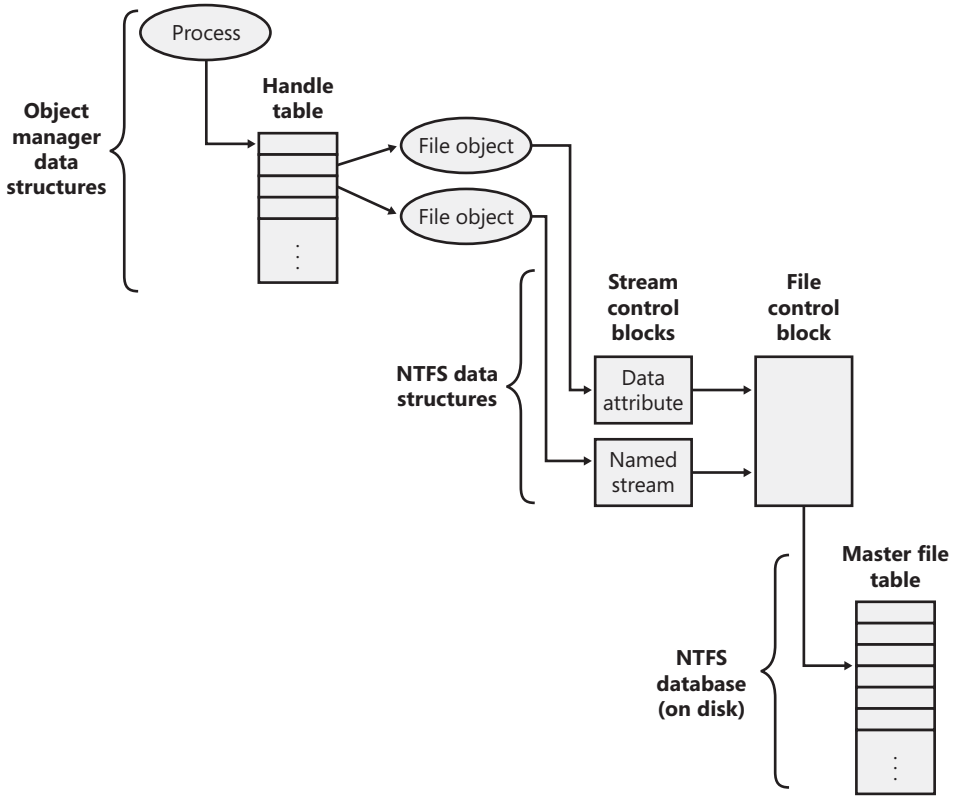


Figure 12-8 Components involved in file system I/O

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 Windows 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 Windows functions. The *CreateFile* function, which is implemented in the Kernel32.dll Windows 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:\Daryl\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 ("??"). Chapter 3 includes a thorough description of object manager name resolution and its use of process device maps, but we'll review the steps it follows here with a focus on volume drive letter lookup.

The first step the object manager takes is to translate \?? to the process's per-session namespace directory that the DosDevicesDirectory field of the device map structure in the process object references. On Windows 2000 systems without Terminal Services, the DosDevicesDirectory field references the \?? directory; and on Windows 2000 systems with Terminal Services, the device map references a per-session directory in which symbolic link objects representing all valid volume drive letters are stored. On Windows XP and Windows Server 2003, however, only volume names for network shares are typically stored in the per-session directory, so on those systems when a name (C: in this example) is not present in the per-session directory, the object manager restarts its search in the directory referenced by the GlobalDosDevicesDirectory field of the device map associated with the per-session directory. The GlobalDosDevicesDirectory always points at the \Global?? directory, which is where Windows XP and Windows Server 2003 store volume drive letters for local volumes. (See the section "Session Namespace" in Chapter 3 for more information.)

The symbolic link for a volume drive letter 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 12-9 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 on Windows 2000 without Terminal Services.

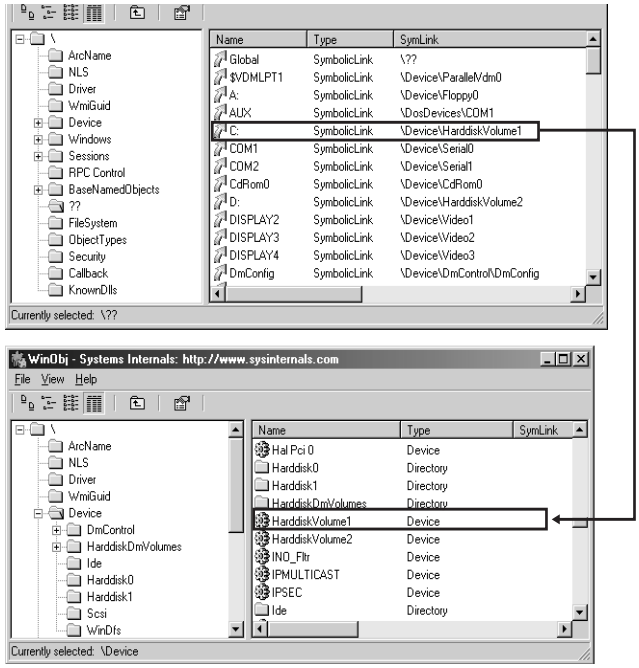


Figure 12-9 Drive-letter name resolution

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 (that is, 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 the *ReadFile* function. 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 *MmMapViewInSystemCache*) 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 *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. (See the section "Fast I/O" in Chapter 11 for a more complete description of fast I/O.)

Memory Manager's Modified and Mapped Page Writer

The memory manager's modified and mapped page writer threads wake up periodically and when available memory runs low 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 *IoSynchronousPageWrite* 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.

File System Filter Drivers

A filter driver that layers over a file system driver is called a *file system filter driver*. (See Chapter 9 for more information on filter drivers.) The ability to see all file system requests and optionally modify or complete them enables a range of applications, including remote file replication services, file encryption, efficient backup, and licensing. Every commercial on-access virus scanner includes a file system filter driver that intercepts IRPs that deliver IRP_MJ_CREATE commands that issue whenever an application opens a file. Before propagating the IRP to the file system driver to which the command is directed, the virus scanner examines the file being opened to ensure it's clean of a virus. If the file is clean, the virus scanner passes the IRP on, but if the file is infected the virus scanner communicates with its associated Windows service process to quarantine or clean the file. If the file can't be cleaned, the driver fails the IRP (typically with an access-denied error) so that the virus cannot become active.

In this section, we'll describe the operation of two specific file system filter drivers: Filemon and System Restore. Filemon, a file system activity monitoring utility from *www.sysinternals.com* that has been used throughout this book, is an example of a passive filter driver, which is one that does not modify the flow of IRPs between applications and file system drivers. System Restore, a feature that was introduced in Windows XP, uses a file system filter driver to watch for changes to key system files and make backups so that the files can be returned to the state they had at particular points in time called *restore points*.

Note Windows XP Service Pack 2 and Windows Server 2003 include the Filesystem Filter Manager (\Windows\System32\Drivers\Fltmgr.sys), which will also be made available for Windows 2000, as part of a port/miniport model for file system filter drivers. The Filesystem Filter Manager greatly simplifies the development of filter drivers by interfacing a filter miniport driver to the Windows I/O system and providing services for querying filenames, attaching to volumes, and interacting with other filters. Vendors, including Microsoft, will write new file system filters and migrate existing filters to the framework provided by the Filesystem Filter Manager.

Filemon

Filemon works by extracting a file system filter device driver (Filem.sys) from its executable image (Filemon.exe) the first time you run it after a boot, installing the driver in memory, and then deleting the driver image from disk. Through the Filemon GUI, you can direct it to monitor file system activity on local volumes that have assigned drive letters, network shares, named pipes, and mail slots. When the driver receives a command to start monitoring a volume, it creates a filter device object and attaches it to the device object that represents a mounted file system on the volume. For example, if the NTFS driver had mounted a volume, Filemon's driver would attach, using the I/O manager function *IoAttachDeviceToDeviceStackSafe*, its own device object to that of NTFS. After an attach operation, the I/O manager redirects an IRP targeted at the underlying device object to the driver owning the attached device, in this case Filemon.

When the Filemon driver intercepts an IRP, it records information about the IRP's command, including target file name and other parameters specific to the command (such as read and write lengths and offsets) to a nonpaged kernel buffer. Twice a second the Filemon GUI sends an IRP to Filemon's interface device object, which requests a copy of the buffer containing the latest activity and displays the activity in its output window. Filemon's use is described further in the "Troubleshooting File System Problems" section later in this chapter.

System Restore

System Restore, which originally appeared in a more rudimentary form in Windows Me (Millennium Edition), provides a way to restore a Windows XP system to a previously known good point that would otherwise require you to reinstall an application or even the entire operating system. (System Restore is not available on Windows 2000 or Windows Server 2003.) For example, if you install one or more applications or make other system file changes, registry changes, or both that cause applications to fail, you can use System Restore to revert the system files and the Registry to the state it had before the change occurred. System Restore is especially useful when you install an application that makes changes you would like to undo. Windows XP–compatible setup applications integrate with System Restore to create a "restore point" before an installation begins.

System Restore's core lies in a service named SrService, which executes from a DLL (\Windows\System32\Srsvc.dll) running in an instance of a generic service host (\Windows\System32\Svchost.exe) process. (See Chapter 4 for a description of Svchost.) The service's role is both to automatically create restore points and to export an API so that other applications—such as setup programs—can manually initiate restore point creation. System Restore reads its configuration parameters from HKLM\Software\Microsoft\System Restore, including ones that specify how much disk space must be available for it to operate and at what interval automated restore-point creation occurs. By default, the service creates a restore point prior to the installation of an unsigned device driver and tries to create an automatic checkpoint every 24 hours. (See Chapter 9 for information on driver signing.) If the DWORD registry value RPGlobalInterval is set under System Restore's parameter key, HKLM\System\CurrentControlSet\Services\SR\Parameters, it overrides this interval and specifies the minimum time interval in seconds between automatic restore points.

When the System Restore service creates a new restore point, it creates a restore point directory and then "snapshots" a set of critical system files, including the system and user-profile Registry hives, WMI configuration information, the IIS metabase file (if IIS is installed), and the COM registration database. Then the system restore file system filter driver, \Windows\System32\Drivers\Sr.sys, begins to track changes to files and directories, saving copies of files that are being deleted or modified in the restore point, and noting other changes, such as directory creation and deletion, in a restore point tracking log.

Restore point data is maintained on a per-volume basis, so tracking logs and saved files are stored under the \System Volume Information_restore{*XX-XXX-XXX* } directory (where the *X*s represent the computer's system-assigned GUID) of a file's original volume. The restore directory contains restore-point subdirectories having names in the form RP*n*, where *n* is a restore point's unique identifier. Files that make up a restore point's initial snapshot are stored under a restore point's Snapshot directory.

Backup files copied by the System Restore driver are given unique names, such as A0000135.dll, in an appropriate restore-point directory that reflect the assignment of an identifier and the preservation of the file's original extension. A restore point can have multiple tracking logs, each having a name like change.log.*N*, where *N* is a unique tracking log ID. A tracking log contains records that store enough information regarding a change to a file or directory for the change to be undone. For example, if a file was deleted, the tracking log entry for that operation would store the copy's name in the restore point (for example, A0000135.dll) and the file's original long and short file names. The System Restore driver starts a new tracking log when a current one grows larger than 1 MB. Figure 12-10 depicts the flow of file system requests as the System Restore driver updates a restore point in response to modifications.

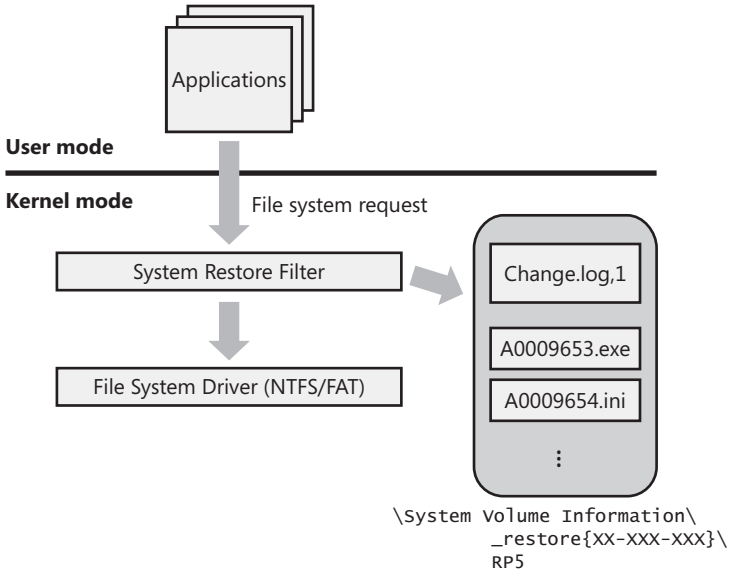


Figure 12-10 System Restore filter driver operation

Figure 12-11 shows a screen shot of a System Restore directory, which includes several restore point subdirectories, as well as the contents of the subdirectory corresponding to restore point 1. Note that the \System Volume Information directories are not accessible by user or even administrator accounts, but they are by the local system account. To view the contents of this folder, follow these steps with the PsExec utility from *www.sysinternals.com*:

```
C:\WINDOWS\SYSTEM32>psexec -s cmd

PsExec v1.55 - Execute processes remotely
Copyright (C) 2001-2004 Mark Russinovich
Sysinternals - www.sysinternals.com


Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>cd \system*

C:\System Volume Information>cd _rest*

C:\System Volume Information\_restore{987E0331-0F01-427C-A58A-7A2E4AABF84D}>
```

Once in the System Restore directory, you can examine its contents with the DIR command or navigate into directories associated with restore points.

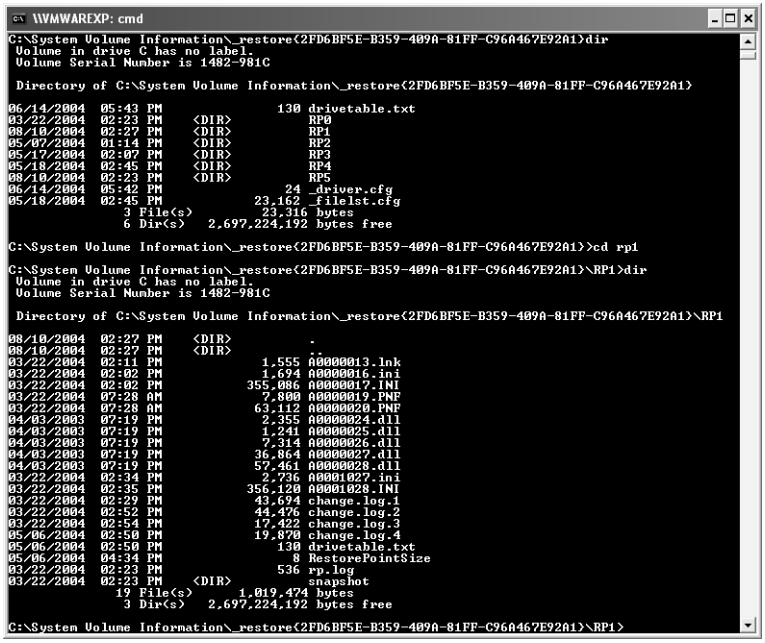


Figure 12-11 System Restore directory and restore point contents

The restore point directory on the boot volume also stores a file named _filelst.cfg, which is a binary file that includes the extensions of files for which changes should be stored in a restore point and the list of directories—such as those that store temporary files—for which changes should be ignored. This list, which is documented in the Platform SDK, directs System Restore to track only nondata files. For example, you wouldn't want an important Microsoft Word document to be deleted just because you rolled back the system to correct an application configuration problem.

EXPERIMENT: Looking at System Restore Filter Device Objects

To monitor changes to files and directories, the System Restore filter driver must attach filter device objects to the FAT and NTFS device objects representing volumes. In addition, it attaches a filter device object to the device objects representing file system drivers so that it can become aware of new volumes as they are mounted by the file system and then subsequently attach filter device objects to them. You can see System Restore's device objects with a kernel debugger:

```
lkd> !drvobj \filesystem\sr
Driver object (81543850) is for:
 \FileSystem\sr
Driver Extension List: (id , addr)

Device Object list:
814ee370  81542dd0  81543728
```

In this sample output, the System Restore driver has three device objects. The last one in the list is named *SystemRestore*, so it serves as the interface to which the user-mode components of System Restore direct commands:

```
lkd> !devobj 81543728
Device object (81543728) is for:
 SystemRestore \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e128feac DevExt 00000000 DevObjExt 815437e0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
Device queue is not busy.
```

The first and second objects are attached to NTFS file system device objects:

```
lkd> !devobj 814ee370
Device object (814ee370) is for:
  \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 814ee428 DevObjExt 814ee570
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedTo (Lower) 81532020 \FileSystem\Ntfs
Device queue is not busy.
lkd> !devobj 81542dd0
Device object (81542dd0) is for:
  \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 81542e88 DevObjExt 81542fd0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedTo (Lower) 815432e8 \FileSystem\Ntfs
Device queue is not busy.
```

One of the NTFS device objects is the NTFS file system driver's interface device because its name is NTFS:

```
lkd> !devobj 815432e8
Device object (815432e8) is for:
 Ntfs \FileSystem\Ntfs DriverObject 81543410
Current Irp 00000000 RefCount 1 Type 00000008 Flags 00000040
Dacl e1297154 DevExt 00000000 DevObjExt 815433a0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedDevice (Upper) 81542dd0 \FileSystem\sr
Device queue is not busy.
```

The other represents the mounted NTFS volume on C:, the system's only volume, so it does not have a name:

```
lkd> !devobj 81532020
Device object (81532020) is for:
  \FileSystem\Ntfs DriverObject 81543410
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 815320d8 DevObjExt 81532880
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedDevice (Upper) 814ee370 \FileSystem\sr
Device queue is not busy.
```

When the user directs the system to perform a restore, the System Restore Wizard (\Windows\System32\Restore\Rstrui.exe) creates a DWORD value named *RestoreInProgress* under the System Restore parameters key and sets it to 1. Then it initiates a system shutdown with reboot by calling the Windows *ExitWindowsEx* function. After the reboot, the WinLogon process (\Windows\System32\Winlogon.exe) realizes that it should perform a restore, and then copies saved files from the Restore Point's directory to their original locations and uses the log files to undo file system changes to files and directories. When the process is complete, the boot continues. Besides making restores safer, the reboot is necessary to activate restored Registry hives.

The Platform SDK documents two System Restore–related APIs, *SRSetRestorePoint* and *SRRemoveRestorePoint*, for use by installation programs, and developers should examine the file extensions that their applications use in light of System Restore. Files that store user data should not have extensions matching those protected by System Restore; otherwise, users could lose data when rolling back to a restore point.

Troubleshooting File System Problems

Chapter 4 describes the way that the system and applications store data in the registry, making registry-related problems such as misconfigured security and missing registry values and keys the source of many system and application failures. The system and applications also use files to store data, and they access executable and DLL image files. Misconfigured NTFS security and missing files or directories are therefore also a common source of system and application failures. This is because the system and applications often make assumptions about what they should be able to access and then misbehave in unexpected ways when the assumptions are violated.

Filemon shows all file activity as it occurs, which makes it an ideal tool for troubleshooting file system–related system and application failures. Filemon's user interface is virtually identical to Regmon's, and Filemon includes the same filtering, highlighting and search features as Regmon. To run Filemon the first time on a system, an account must have the same privileges as to run Regmon: Load Driver and Debug. After loading, the driver remains resident, so subsequent executions require only the Debug privilege.

Filemon Basic vs. Advanced Modes

When you run Filemon, it starts in basic mode, which shows the file system activity most often useful for troubleshooting. When in basic mode, Filemon omits certain file system operations from display, including:

- Accesses to NTFS metadata files
- Activity that occurs in the System process
- I/O to the paging file

- I/O generated by the Filemon process
- Fast I/O failures

While in basic mode, Filemon also reports file I/O operations with friendly names rather than the IRP types used to represent them. For example, both IRP_MJ_WRITE and FASTIO_WRITE operations display as Write, and IRP_MJ_CREATE operations show as Open if they represent an open operation and as Create for the creation of new files.

EXPERIMENT: Viewing File System Activity on an Idle System

Windows file system drivers implement support for *file change notification*, which enables applications to request notification of file system changes without polling for them. The Windows functions for doing so include *ReadDirectoryChangesW* and the *FindFirstChangeNotification*, *FindNextChangeNofification* pair. When you run Filemon on a system that's idle, you should therefore not see the repeated accesses to files or directories because that activity unnecessarily negatively affects a system's overall performance.

Run Filemon, and after several seconds examine the output log to see whether you can spot polling behavior. Right-click on an output line associated with polling and choose Process Properties from the context menu to view details of the process performing the activity.

Filemon Troubleshooting Techniques

The two basic Filemon troubleshooting techniques are identical to those of Regmon: looking at the last thing in a Filemon trace that an application did before it failed, or comparing a Filemon trace of a failing application with a trace from a working system. See the section "Regmon Troubleshooting Techniques" in Chapter 4 for more information on these techniques.

Entries in a Filemon trace that have values of FILE NOT FOUND, NO SUCH FILE, PATH NOT FOUND, SHARING VIOLATION, and ACCESS DENIED in the Result column are ones that you should investigate. The first three are reported when an application or the system attempts to open a nonexistent file or directory. In many cases, these errors do not indicate a serious problem. When you execute a program from the Start menu's Run dialog box without specifying its full path, for instance, Explorer will search the directories listed in the system PATH environment variable for the image file until it locates the file or has searched all the listed directories. Each attempt to find the image in a directory that does not contain it results in a Filemon output line similar to this:

```
5:28:26 PMEXPLORER.EXE:1568FASTIO_QUERY_OPENC:\Documents and Settings\mark.AUSTIN\Start Menu
\test.exe    FILE NOT FOUND    Attributes: Error
```

Access-denied errors are a common source of file system–related application failures, and they occur when an application does not have permission to open the file or directory for the

access types it desires. Some applications do not check error codes or perform error recovery, and they fail by crashing or terminating; others display misleading error messages that mask the root cause of the error.

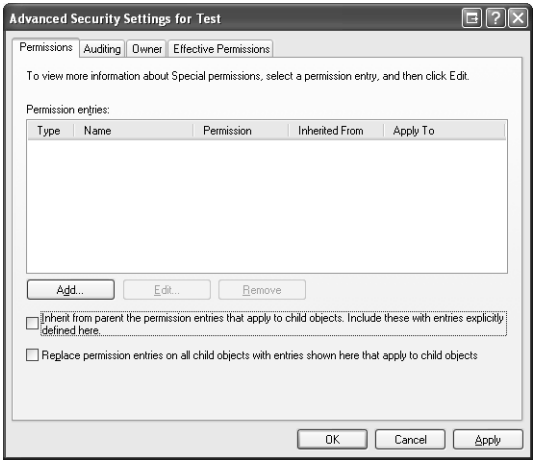
Buffer-overflow exploits are a serious security concern, but a code result of BUFFER OVERFLOW is simply a file system driver's way to indicate to an application that the buffer it specified to store result data was too small to hold the data. Application developers use this behavior to determine how large a buffer should be because the file system driver also returns the size of the buffer required to store the data. Operations with a buffer overflow result are usually followed by the same operation with a successful result.

EXPERIMENT: Seeing an Error's Root Cause with Filemon

Applications sometimes present error messages in response to an error condition that do not reveal the root cause of the error. These error messages can be frustrating because they might lead you to spend time diagnosing or resolving problems that do not exist. If the error message is related to a file system issue, Filemon will show you what underlying errors might have occurred prior to the appearance of an error message.

In this experiment, you'll set permission on a directory and then perform a file save operation in Notepad that results in a misleading error message. Filemon's trace shows the actual error and the source of the message displayed in Notepad's error dialog box.

1. Run Filemon, and set the include filter to "notepad.exe".
2. Open Explorer, and create a directory named "Test" in a directory on an NTFS volume. (In this example, the root directory was used.)
3. Edit the security permissions on the Test directory to remove all access. This might require you to open the Advanced Security Settings dialog box and use the settings on the Permissions tab to remove inherited security.



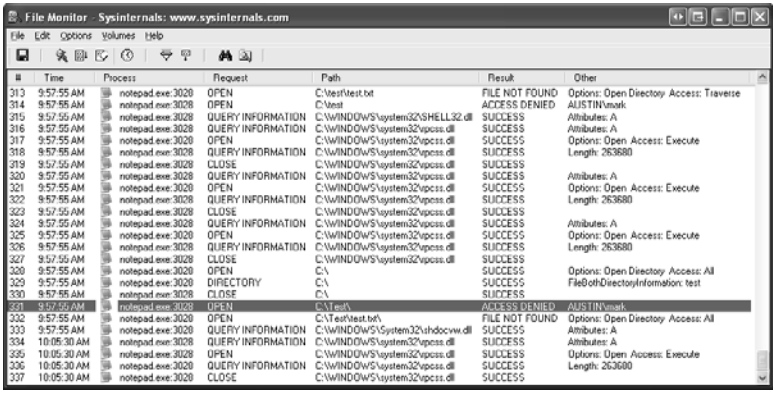
When you apply the modified security, Explorer should warn you that no one will have access to the folder.

4. Run Notepad, and enter some text into its window. Then select the Save entry in the File menu. In the File Name field of the Save dialog box, enter **c:\test\test.txt** (assuming the folder you created is on the volume C:).

5. Notepad will display the following error message:

6. The message implies that C:\Test does not exist.

7. The Filemon trace you see should look something like this:



The output at the bottom of the figure was generated immediately prior to the appearance of the error message. Line 331 shows that Notepad tried to open C:\Test and got an access-denied error. Immediately afterward on line 332, it tried to open the C:\Test\Test.txt directory (as indicated by the Directory option in the Other column of that line) and received a file-not-found error because the directory does not exist. The error message Notepad displays, "Path does not exist", is consistent with a file-not-found error, not an access-denied error. So it appears that Notepad first tried to open the directory, and when that failed it assumed for some reason that the name C:\Test\Test.txt was the name of a directory instead of a file. When it couldn't open that directory, Notepad presented the error message, but the root cause, which Filemon reveals, is the access denied error.

Filemon has been used extensively within Microsoft and other organizations to solve difficult or nearly impossible-to-diagnose problems. One example of Filemon being used to troubleshoot a problem revealed the root cause of a misleading error message generated by the

Windows Installer service. When the user tried to install an application using its Windows Installer Package file, the Windows Installer service reported the error shown in Figure 12-12, which states that the service cannot write to the Temp folder. The user verified that, contrary to the error message's claims, his profile's temporary directory (that he obtained by typing **set temp** in a console window) was located on a volume with adequate free space and had default permissions.

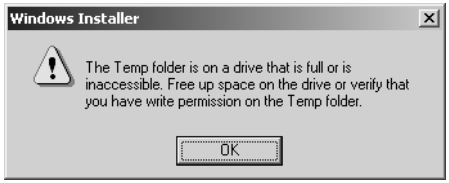


Figure 12-12 Microsoft Windows Installer error message

The user ran Filemon to capture a trace of the file system activity leading up to the error and identified the highlighted line in Figure 12-13 as the root cause of the error. The conclusion that the user drew from the trace is that the Windows Installer service was referring to \Windows\Installer, not his profile's temporary directory, as the Temp folder in the error message. The access-denied line in the output reports that the Windows Installer service was running in the local system account, so the user modified the permissions on \Windows\Installer to allow the local system account write access and resolved the problem.

msiexec.exe:1648	CLOSE	C:\	SUCCESS	
msiexec.exe:1648	QUERY INFORMATION	C:\WINNT\Installer\34f62b5.msi	FILE NOT FOUND	Attributes: Error
msiexec.exe:1648	CREATE	C:\WINNT\Installer\34f62b5.msi	ACCESS DENIED	NT AUTHORITY\SYSTEM
msiexec.exe:1648	CLOSE	D:\ACCESSRT.MSI	SUCCESS	
msiexec.exe:1880	QUERY INFORMATION	C:\WINNT\System32\SAGE.DLL	FILE NOT FOUND	Attributes: Error

Figure 12-13 Microsoft Windows Installer Service Filemon trace

Another Filemon troubleshooting example involved Microsoft Word. The user in this case would launch Word and type for a few seconds, only to have the Word window close without notice. A Filemon trace of the scenario, part of which is shown in Figure 12-14, shows Word repeatedly reading from the same part of a file named Mssp3es.lex immediately before exiting. (When a process exits, the system closes all its handles, which you can see happening from line 25460 onward.) The user determined that .lex files are related to Microsoft Office Proofing Tools and reinstalled that component, which resolved the problem.

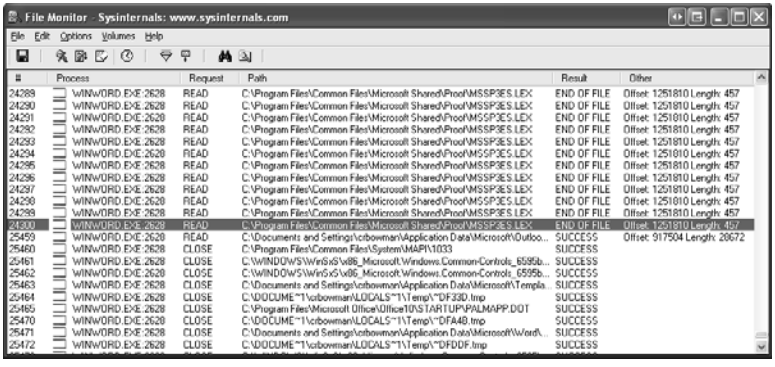


Figure 12-14 Microsoft Word reading a .lex file

In another example, a user encountered the error message shown in Figure 12-15 every time she started Microsoft Excel. The Filemon trace in Figure 12-16 that the user captured during Excel's startup reveals Excel reading from a file named 59403e20 in a directory named Xlstart in the Microsoft Office installation directory. The user investigated the problem and learned from Excel's documentation that Excel automatically tries to open files stored in the Xlstart directory. However, the file visible in the trace was not an Excel file, which resulted in the error message. Deleting the file caused the errors to cease.

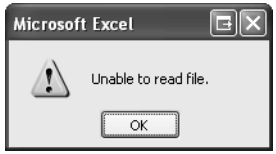


Figure 12-15 Microsoft Excel startup error message

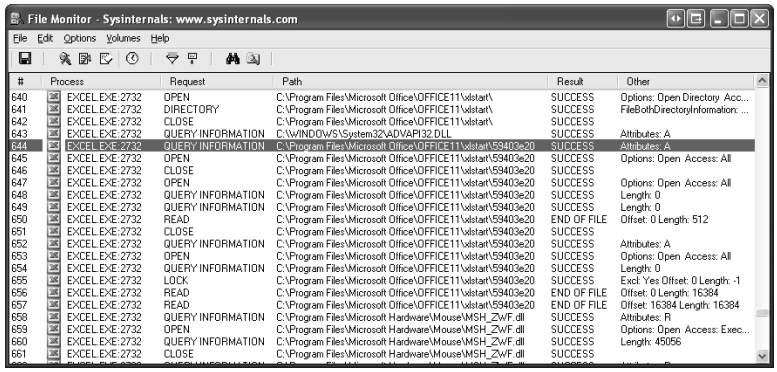


Figure 12-16 Filemon trace of Microsoft Excel startup

A final example of Filemon troubleshooting involves out-of-date DLLs. A particular user would run Microsoft Access 2000 and experience a hang when he tried to import a Microsoft Excel file. The user tried to import the same file on another system with Microsoft Access 2000 and was able to do so successfully. After capturing traces of the import operation on both systems and saving them to log files, the user compared the logs with Windiff. The results of the comparison are shown in Figure 12-17.

After discounting unimportant differences such as the different temporary file names seen in line 19 and the different casing of file names seen in line 26, the user determined that the first relevant difference between the traces is line 37. On the system where the import fails, Microsoft Access loads a copy of Accwiz.dll from the \Winnt\System32 directory, whereas on the system where the import succeeds it reads Accwiz.dll from \Progra~1\Files\Microsoft\Office. The user examined the Accwiz.dll copy in \Winnt\System32 and discovered that it was from an older version of Microsoft Access, but the system's DLL search order caused it to find that instance instead of the one in the Microsoft Access installation directory. Deleting that copy and registering the proper copy fixed the problem.

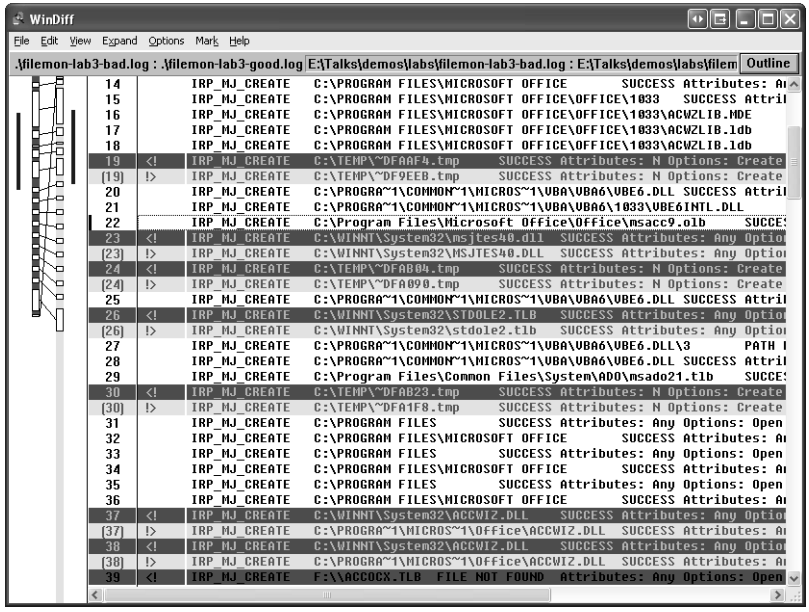


Figure 12-17 Comparison of Microsoft Access trace logs

These are just a few examples showing how Filemon can be used to discover the root cause of file system problems that might not be reported clearly by the application. The remainder of the chapter focuses on the native Windows file system, NTFS.

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 its metadata is 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