

11.7 THE WINDOWS 2000 FILE SYSTEM

Windows 2000 supports several file systems, the most important of which are **FAT-16**, **FAT-32**, and **NTFS (NT File System)**. FAT-16 is the old MS-DOS file system. It uses 16-bit disk addresses, which limits it to disk partitions no larger than 2 GB. FAT-32 uses 32-bit disk addresses and supports disk partitions up to 2 TB. NTFS is a new file system developed specifically for Windows NT and carried over to Windows 2000. It uses 64-bit disk addresses and can (theoretically) support disk partitions up to 2^{64} bytes, although other considerations limit it to smaller sizes. Windows 2000 also supports read-only file systems for CD-ROMs and DVDs. It is possible (even common) to have the same running system have access to multiple file system types available at the same time.

In this chapter we will treat the NTFS file system because it is a modern file system unencumbered by the need to be fully compatible with the MS-DOS file system, which was based on the CP/M file system designed for 8-inch floppy disks more than 20 years ago. Times have changed and 8-inch floppy disks are not quite state of the art any more. Neither are their file systems. Also, NTFS differs both in user interface and implementation in a number of ways from the UNIX file system, which makes it a good second example to study. NTFS is a large and complex system and space limitations prevent us from covering all of its features, but the material presented below should give a reasonable impression of it.

11.7.1 Fundamental Concepts

Individual file names in NTFS are limited to 255 characters; full paths are limited to 32,767 characters. File names are in Unicode, allowing people in countries not using the Latin alphabet (e.g., Greece, Japan, India, Russia, and Israel) to write file names in their native language. For example, $\phi\iota\lambda\epsilon$ is a perfectly legal file name. NTFS fully supports case sensitive names (so *foo* is different from *Foo* and *FOO*). Unfortunately, the Win32 API does not fully support case-sensitivity for file names and not at all for directory names, so this advantage is lost to programs restricted to using Win32 (e.g., for Windows 98 compatibility).

An NTFS file is not just a linear sequence of bytes, as FAT-32 and UNIX files are. Instead, a file consists of multiple attributes, each of which is represented by a stream of bytes. Most files have a few short streams, such as the name of the file and its 64-bit object ID, plus one long (unnamed) stream with the data. However, a file can also have two or more (long) data streams as well. Each stream has a name consisting of the file name, a colon, and the stream name, as in *foo:stream1*. Each stream has its own size and is lockable independently of all the other streams. The idea of multiple streams in a file was borrowed from the Apple Macintosh, in which files have two streams, the data fork and the resource fork. This concept was incorporated into NTFS to allow an NTFS server be able

to serve Macintosh clients.

File streams can be used for purposes other than Macintosh compatibility. For example, a photo editing program could use the unnamed stream for the main image and a named stream for a small thumbnail version. This scheme is simpler than the traditional way of putting them in the same file one after another. Another use of streams is in word processing. These programs often make two versions of a document, a temporary one for use during editing and a final one when the user is done. By making the temporary one a named stream and the final one the unnamed stream, both versions automatically share a file name, security information, timestamps, etc. with no extra work.

The maximum stream length is 2^{64} bytes. To get some idea of how big a 2^{64} -byte stream is, imagine that the stream were written out in binary, with each of the 0s and 1s in each byte occupying 1 mm of space. The 2^{67} -mm listing would be 15 light-years long, reaching far beyond the solar system, to Alpha Centauri and back. File pointers are used to keep track of where a process is in each stream, and these are 64 bits wide to handle the maximum length stream, which is about 18.4 exabytes.

The Win32 API function calls for file and directory manipulation are roughly similar to their UNIX counterparts, except most have more parameters and the security model is different. Opening a file returns a handle, which is then used for reading and writing the file. For graphical applications, no file handles are predefined. Standard input, standard output, and standard error have to be acquired explicitly if needed; in console mode they are preopened, however. Win32 also has a number of additional calls not present in UNIX.

11.7.2 File System API Calls in Windows 2000

The principal Win32 API functions for file management are listed in Fig. 11-1. There are actually many more, but these give a reasonable first impression of the basic ones. Let us now examine these calls briefly. `CreateFile` can be used to create a new file and return a handle to it. This API function must also be used to open existing files as there is no `FileOpen` API function. We have not listed the parameters for the API functions because they are so voluminous. As an example, `CreateFile` has seven parameters, which are roughly summarized as follows:

1. A pointer to the name of the file to create or open.
2. Flags telling whether the file can be read, written, or both.
3. Flags telling whether multiple processes can open the file at once.
4. A pointer to the security descriptor, telling who can access the file.
5. Flags telling what to do if the file exists/does not exist.
6. Flags dealing with attributes such as archiving, compression, etc.

Win32 API function	UNIX	Description
CreateFile	open	Create a file or open an existing file; return a handle
DeleteFile	unlink	Destroy an existing file
CloseHandle	close	Close a file
ReadFile	read	Read data from a file
WriteFile	write	Write data to a file
SetFilePointer	lseek	Set the file pointer to a specific place in the file
GetFileAttributes	stat	Return the file properties
LockFile	fcntl	Lock a region of the file to provide mutual exclusion
UnlockFile	fcntl	Unlock a previously locked region of the file

Figure 11-1. The principal Win32 API functions for file I/O. The second column gives the nearest UNIX equivalent.

7. The handle of a file whose attributes should be cloned for the new file.

The next six API functions in Fig. 11-1 are fairly similar to the corresponding UNIX system calls. The last two allow a region of a file to be locked and unlocked to permit a process to get guaranteed mutual exclusion to it.

Using these API functions, it is possible to write a procedure to copy a file, analogous to the UNIX version of Fig. 6-5. Such a code fragment (without any error checking) is shown in Fig. 11-2. It has been designed to mimic our UNIX version. In practice, one would not have to program a copy file program since CopyFile is an API function (which executes something close to this program as a library procedure).

```

/* Open files for input and output. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);

/* Copy the file. */
do {
    s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
    if (s && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Close the files. */
CloseHandle(inhandle);
CloseHandle(outhandle);

```

Figure 11-2. A program fragment for copying a file using the Windows 2000 API functions.

Windows 2000 NTFS is a hierarchical file system, similar to the UNIX file system. The separator between component names is \ however, instead of /, a fossil inherited from MS-DOS. There is a concept of a current working directory and path names can be relative or absolute. Hard and symbolic links are supported, the former implemented by having multiple directory entries, as in UNIX, and the latter implemented using reparse points (discussed later in this chapter). In addition, compression, encryption, and fault tolerance are also supported. These features and their implementations will be discussed later in this chapter.

The major directory management API functions are given in Fig. 11-3, again along with their nearest UNIX equivalents. The functions should be self explanatory.

Win32 API function	UNIX	Description
CreateDirectory	mkdir	Create a new directory
RemoveDirectory	rmdir	Remove an empty directory
FindFirstFile	opendir	Initialize to start reading the entries in a directory
FindNextFile	readdir	Read the next directory entry
MoveFile	rename	Move a file from one directory to another
SetCurrentDirectory	chdir	Change the current working directory

Figure 11-3. The principal Win32 API functions for directory management. The second column gives the nearest UNIX equivalent, when one exists.

11.7.3 Implementation of the Windows 2000 File System

NTFS is a highly complex and sophisticated file system. It was designed from scratch, rather than being an attempt to improve the old MS-DOS file system. Below we will examine a number of its features, starting with its structure, then moving on to file name lookup, file compression, and file encryption.

File System Structure

Each NTFS volume (e.g., disk partition) contains files, directories, bitmaps, and other data structures. Each volume is organized as a linear sequence of blocks (clusters in Microsoft's terminology), with the block size being fixed for each volume and ranging from 512 bytes to 64 KB, depending on the volume size. Most NTFS disks use 4-KB blocks as a compromise between large blocks (for efficient transfers) and small blocks (for low internal fragmentation). Blocks are referred to by their offset from the start of the volume using 64-bit numbers.

The main data structure in each volume is the **MFT (Master File Table)**, which is a linear sequence of fixed-size 1-KB records. Each MFT record describes one file or directory. It contains the file's attributes, such as its name and

timestamps, and the list of disk addresses where its blocks are located. If a file is extremely large, it is sometimes necessary to use two or more MFT records to contain the list of all the blocks, in which case the first MFT record, called the **base record**, points to the other MFT records. This overflow scheme dates back to CP/M, where each directory entry was called an extent. A bitmap keeps track of which MFT entries are free.

The MFT is itself a file and as such can be placed anywhere within the volume, thus eliminating the problem with defective sectors in the first track. Furthermore, the file can grow as needed, up to a maximum size of 2^{48} records.

The MFT is shown in Fig. 11-4. Each MFT record consists of a sequence of (attribute header, value) pairs. Each attribute begins with a header telling which attribute this is and how long the value is because some attribute values are variable length, such as the file name and the data. If the attribute value is short enough to fit in the MFT record, it is placed there. If it is too long, it is placed elsewhere on the disk and a pointer to it is placed in the MFT record.

The first 16 MFT records are reserved for NTFS metadata files, as shown in Fig. 11-4. Each of the records describes a normal file that has attributes and data blocks, just like any other file. Each of these files has a name that begins with a dollar sign to indicate that it is a metadata file. The first record describes the MFT file itself. In particular, it tells where the blocks of the MFT file are located so the system can find the MFT file. Clearly, Windows 2000 needs a way to find the first block of the MFT file in order to find the rest of the file system information. The way it finds the first block of the MFT file is to look in the boot block, where its address is installed at system installation time.

Record 1 is a duplicate of the early part of the MFT file. This information is so precious that having a second copy can be critical in the event one of the first blocks of the MFT ever goes bad. Record 2 is the log file. When structural changes are made to the file system, such as adding a new directory or removing an existing one, the action is logged here before it is performed, in order to increase the chance of correct recovery in the event of a failure during the operation. Changes to file attributes are also logged here. In fact, the only changes not logged here are changes to user data. Record 3 contains information about the volume, such as its size, label, and version.

As mentioned above, each MFT record contains a sequence of (attribute header, value) pairs. The *\$AttrDef* file is where the attributes are defined. Information about this file is in MFT record 4. Next comes the root directory, which itself is a file and can grow to arbitrary length. It is described by MFT record 5.

Free space on the volume is kept track of with a bitmap. The bitmap is itself a file and its attributes and disk addresses are given in MFT record 6. The next MFT record points to the bootstrap loader file. Record 8 is used to link all the bad blocks together to make sure they never occur in a file. Record 9 contains the security information. Record 10 is used for case mapping. For the Latin letters A-Z case mapping is obvious (at least for people who speak Latin). Case

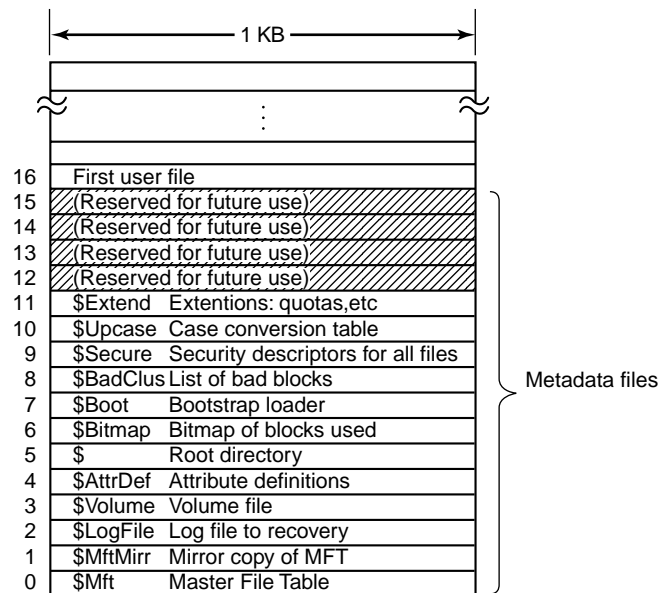


Figure 11-4. The NTFS master file table.

mapping for other languages, such as Greek, Armenian, or Georgian (the country, not the state), is less obvious to Latin speakers, so this file tells how to do it. Finally, record 11 is a directory containing miscellaneous files for things like disk quotas, object identifiers, reparse points, and so on. The last 4 MFT records are reserved for future use.

Each MFT record consists of a record header followed by a sequence of (attribute header, value) pairs. The record header contains a magic number used for validity checking, a sequence number updated each time the record is reused for a new file, a count of references to the file, the actual number of bytes in the record used, the identifier (index, sequence number) of the base record (used only for extension records), and some other miscellaneous fields. Following the record header comes the header of the first attribute, then the first attribute value, the second attribute header, the second attribute value, and so on.

NTFS defines 13 attributes that can appear in MFT records. These are listed in Fig. 11-5. Each MFT record consists of a sequence of attribute headers, each of which identifies the attribute it is heading and gives the length and location of the value field along with a variety of flags and other information. Usually, attribute values follow their attribute headers directly, but if a value is too long to fit in the MFT record, it may be put in a separate disk block. Such an attribute is said to be a **nonresident attribute**. The data attribute is an obvious candidate. Some attributes, such as the name, may be repeated, but all attributes must appear

in a fixed order in the MFT record. The headers for resident attributes are 24 bytes long; those for nonresident attributes are longer because they contain information about where to find the attribute on disk.

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in \$Extend\$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in \$Volume)
Volume information	Volume version (used only in \$Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to \$LogFile
Data	Stream data; may be repeated

Figure 11-5. The attributes used in MFT records.

The standard information field contains the file owner, security information, the timestamps needed by POSIX, the hard link count, the read-only and archive bits, etc. It is a fixed-length field and is always present. The file name is variable length in Unicode. In order to make files with nonMS-DOS names accessible to old 16-bit programs, files can also have an 8 + 3 MS-DOS name. If the actual file name conforms to the MS-DOS 8 + 3 naming rule, a secondary MS-DOS name is not used.

In NT 4.0, security information could be put in an attribute, but in Windows 2000 it all goes into a single file so that multiple files can share the same security descriptions. The attribute list is needed in case the attributes do not fit in the MFT record. This attribute then tells where to find the extension records. Each entry in the list contains a 48-bit index into the MFT telling where the extension record is and a 16-bit sequence number to allow verification that the extension record and base records match up.

The object ID attribute gives the file a unique name. This is sometimes needed internally. The reparse point tells the procedure parsing the file name to do something special. This mechanism is used for mounting and symbolic links. The two volume attributes are only used for volume identification. The next three attributes deal with how directories are implemented. Small ones are just lists of files but large ones are implemented using B+ trees. The logged utility stream attribute is used by the encrypting file system.

Finally, we come to the attribute that everyone has been waiting for: the data. The stream name, if present, goes in this attribute header. Following the header is either a list of disk addresses telling which blocks the file contains, or for files of only a few hundred bytes (and there are many of these), the file itself. Putting the actual file data in the MFT record is called an **immediate file** (Mullender and Tanenbaum, 1987).

Of course, most of the time the data does not fit in the MFT record, so this attribute is usually nonresident. Let us now take a look at how NTFS keeps track of the location of nonresident attributes, in particular data.

The model for keeping track of disk blocks is that they are assigned in runs of consecutive blocks, where possible, for efficiency reasons. For example, if the first logical block of a file is placed in block 20 on the disk, then the system will try hard to place the second logical block in block 21, the third logical block in 22, and so on. One way to achieve these runs is to allocate disk storage several blocks at a time, if possible.

The blocks in a file are described by a sequence of records, each one describing a sequence of logically contiguous blocks. For a file with no holes in it, there will be only one such record. Files that are written in order from beginning to end all belong in this category. For a file with one hole in it (e.g., only blocks 0–49 and blocks 60–79 are defined), there will be two records. Such a file could be produced by writing the first 50 blocks, then seeking forward to logical block 60 and writing another 20 blocks. When a hole is read back, all the missing bytes are zeros.

Each record begins with a header giving the offset of the first block within the file. Next comes the offset of the first block not covered by the record. In the example above, the first record would have a header of (0, 50) and would provide the disk addresses for these 50 blocks. The second one would have a header of (60,80) and would provide the disk addresses for these 20 blocks.

Each record header is followed by one or more pairs, each giving a disk address and run length. The disk address is the offset of the disk block from the start of its partition; the run length is the number of blocks in the run. As many pairs as needed can be in the run record. Use of this scheme for a three-run, nine-block file is illustrated in Fig. 11-6.

In this figure we have an MFT record for a short file (short here means that all the information about the file blocks fits in one MFT record). It consists of the three runs of consecutive blocks on the disk. The first run is blocks 20-23, the second is blocks 64-65, and the third is blocks 80-82. Each of these runs is recorded in the MFT record as a (disk address, block count) pair. How many runs there are depends on how good a job the disk block allocator did in finding runs of consecutive blocks when the file was created. For a n -block file, the number of runs can be anything from 1 up to and including n .

Several comments are worth making here. First, there is no upper limit to the size of files that can be represented this way. In the absence of address

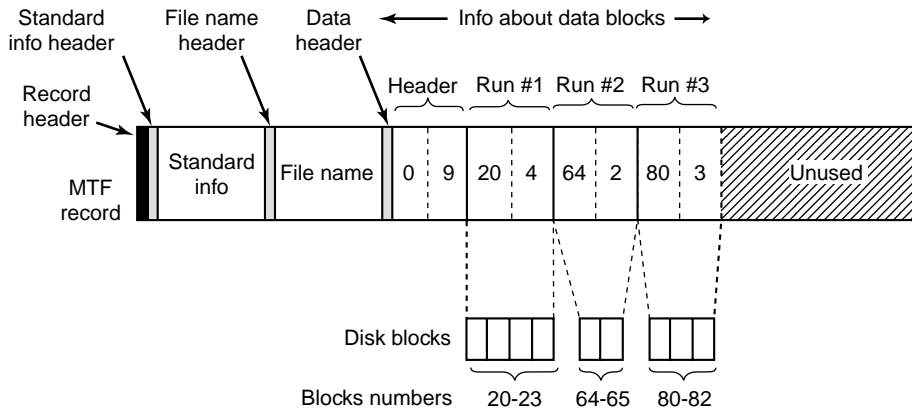


Figure 11-6. An MFT record for a three-run, nine-block file.

compression, each pair requires two 64-bit numbers in the pair for a total of 16 bytes. However, a pair could represent 1 million or more consecutive disk blocks. In fact, a 20-MB file consisting of 20 separate runs of 1 million 1-KB blocks each fits easily in one MFT record, whereas a 60-KB file scattered into 60 isolated blocks does not.

Second, while the straightforward way of representing each pair takes 2×8 bytes, a compression method is available to reduce the size of the pairs below 16. Many disk addresses have multiple high-order zero-bytes. These can be omitted. The data header tells how many are omitted, that is, how many bytes are actually used per address. Other kinds of compression are also used. In practice, the pairs are often only 4 bytes.

Our first example was easy: all the file information fit in one MFT record. What happens if the file is so large or highly fragmented that the block information does not fit in one MFT record? The answer is simple: use two or more MFT records. In Fig. 11-7 we see a file whose base record is in MFT record 102. It has too many runs for one MFT record, so it computes how many extension records it needs, say, two, and puts their indices in the base record. The rest of the record is used for the first k data runs.

Note that Fig. 11-7 contains some redundancy. In theory, it should not be necessary to specify the end of a sequence of runs because this information can be calculated from the run pairs. The reason for “overspecifying” this information is to make seeking more efficient: to find the block at a given file offset, it is only necessary to examine the record headers, not the run pairs.

When all the space in record 102 has been used up, storage of the runs continues with MFT record 105. As many runs are packed in this record as fit. When this record is also full, the rest of the runs go in MFT record 108. In this way many MFT records can be used to handle large fragmented files.

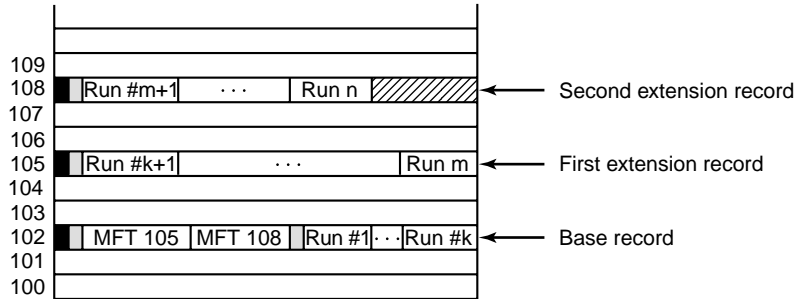


Figure 11-7. A file that requires three MFT records to store all its runs.

A problem arises if so many MFT records are needed that there is no room in the base MFT to list all their indices. There is also a solution to this problem: the list of extension MFT records is made nonresident (i.e., stored on disk instead of in the base MFT record). Then it can grow as large as needed.

An MFT entry for a small directory is shown in Fig. 11-8. The record contains a number of directory entries, each of which describes one file or directory. Each entry has a fixed-length structure followed by a variable-length file name. The fixed part contains the index of the MFT entry for the file, the length of the file name, and a variety of other fields and flags. Looking for an entry in a directory consists of examining all the file names in turn.

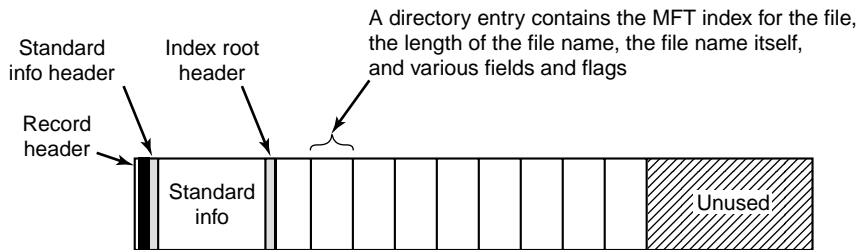


Figure 11-8. The MFT record for a small directory.

Large directories use a different format. Instead of listing the files linearly, a B+ tree is used to make alphabetical lookup possible and to make it easy to insert new names in the directory in the proper place.

File Name Lookup

We now have enough information to see how file name lookup occurs. When a user program wants to open a file, it typically makes a call like

```
CreateFile("C:\maria\web.htm", ...)
```

This call goes to the user-level shared library, *kernel32.dll*, where `\??` is prepended to the file name giving

```
\??\C:\maria\web.htm
```

It is this name that is passed as a parameter to the system call `NtFileCreate`.

Then the operating system starts the search at the root of the object manager's name space (see Fig. 11-0). It then looks in the directory `\??` to find `C:`, which it will find. This file is a symbolic link to another part of the object manager's name space, the directory `\Device`. The link typically ends at an object whose name is something like `\Device\HarddiskVolume1`. This object corresponds to the first partition of the first hard disk. From this object it is possible to determine which MFT to use, namely the one on this partition. These steps are shown in Fig. 11-9.

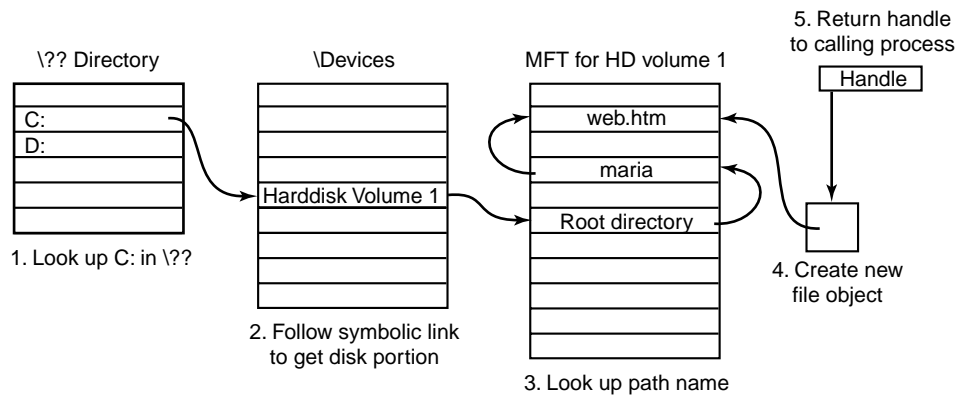


Figure 11-9. Steps in looking up the file `C:\maria\web.htm`.

The parsing of the file name continues now at the root directory, whose blocks can be found from entry 5 in the MFT (see Fig. 11-4). The string "maria" is now looked up in the root directory, which returns the index into the MFT for the directory *maria*. This directory is then searched for the string "web.htm". If successful, the result is a new object created by the object manager. The object, which is unnamed, contains the index of the MFT record for the file. A handle to this object is returned to the calling process. On subsequent `ReadFile` calls, the handle is provided, which allows the object manager to find the index and then the contents of the MFT record for the file. If a thread in a second process opens the file again, it gets a handle to a new file object.

In addition to regular files and directories, NTFS supports hard links in the UNIX sense, and also symbolic links using a mechanism called **reparse points**. It is possible to tag a file or directory as a reparse point and associate a block of data with it. When the file or directory is encountered during a file name parse, exception processing is triggered and the block of data is interpreted. It can do various things, including redirecting the search to a different part of the directory

hierarchy or even to a different partition. This mechanism is used to support both symbolic links and mounted file systems.

File Compression

NTFS supports transparent file compression. A file can be created in compressed mode, which means that NTFS automatically tries to compress the blocks as they are written to disk and automatically uncompresses them when they are read back. Processes that read or write compressed files are completely unaware of the fact that compression and decompression are going on.

Compression works as follows. When NTFS writes a file marked for compression to disk, it examines the first 16 (logical) blocks in the file, irrespective of how many runs they occupy. It then runs a compression algorithm on them. If the resulting data can be stored in 15 or fewer blocks, the compressed data are written to the disk, preferably in one run, if possible. If the compressed data still take 16 blocks, the 16 blocks are written in uncompressed form. Then blocks 16-31 are examined to see if they can be compressed to 15 blocks or less, and so on.

Figure 11-10(a) shows a file in which the first 16 blocks have successfully compressed to eight blocks, the second 16 blocks failed to compress, and the third 16 blocks have also compressed by 50%. The three parts have been written as three runs and stored in the MFT record. The “missing” blocks are stored in the MFT entry with disk address 0 as shown in Fig. 11-10(b). Here the header (0, 48) is followed by five pairs, two for the first (compressed) run, one for the uncompressed run, and two for the final (compressed) run.

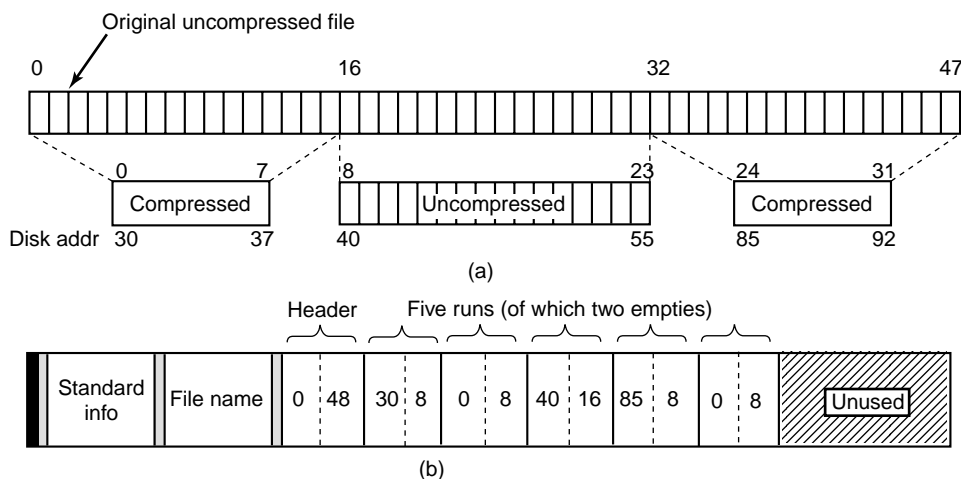


Figure 11-10. (a) An example of a 48-block file being compressed to 32 blocks. (b) The MFT record for the file after compression.

When the file is read back, NTFS has to know which runs are compressed and which are not. It sees that based on the disk addresses. A disk address of 0 indicates that it is the final part of 16 compressed blocks. Disk block 0 may not be used for storing data, to avoid ambiguity. Since it contains the boot sector, using it for data is impossible anyway.

Random access to compressed files is possible, but tricky. Suppose that a process does a seek to block 35 in Fig. 11-10. How does NTFS locate block 35 in a compressed file? The answer is that it has to read and decompress the entire run first. Then it knows where block 35 is and can pass it to any process that reads it. The choice of 16 blocks for the compression unit was a compromise. Making it shorter would have made the compression less effective. Making it longer would have made random access more expensive.

File Encryption

Computers are used nowadays to store all kinds of sensitive data, including plans for corporate takeovers, tax information, and love letters (love email?), which the owners do not especially want revealed to anyone. Information loss can happen when a laptop computer is lost or stolen, a desktop system is rebooted using an MS-DOS floppy disk to bypass Windows 2000 security, or a hard disk is physically removed from one computer and installed on another one with an insecure operating system. Even the simple act of going to the bathroom and leaving the computer unattended and logged in can be a huge security breach.

Windows 2000 addresses these problem by having an option to encrypt files, so even in the event the computer is stolen or rebooted using MS-DOS, the files will be unreadable. The normal way to use Windows 2000 encryption is to mark certain directories as encrypted, which causes all the files in them to be encrypted, and new files moved to them or created in them to be encrypted as well. The actual encryption and decryption is not done by NTFS itself, but by a driver called **EFS (Encrypting File System)**, which is positioned between NTFS and the user process. In this way, application programs are unaware of encryption and NTFS itself is only partially involved in it.

To understand how the encrypting file system works, it is necessary to understand how modern cryptography works. For this purpose, a brief review was given in Sec. 9.2. Readers not familiar with the basics of cryptography should read that section before continuing.

Now let us see how Windows 2000 encrypts files. When the user asks a file to be encrypted, a random 128-bit file key is generated and used to encrypt the file block by block using a symmetric algorithm parametrized by this key. Each new file encrypted gets a different 128-bit random file key, so no two files use the same encryption key, which increases security in case one key is compromised. The current encryption algorithm is a variant of **DES (Data Encryption Standard)**, but the EFS architecture supports the addition of new algorithms in the

future. Encrypting each block independently of all the others is necessary to make random access still possible.

The file key has to be stored somewhere so the file can be decrypted later. If it were just stored on the disk in plaintext, then someone who stole or found the computer could easily decrypt the file, defeating the purpose of encrypting the files. For this reason, the file keys must all be encrypted before they are stored on the disk. Public-key cryptography is used for this purpose.

After the file is encrypted, the location of the user's public key is looked up using information in the registry. There is no danger of storing the public key's location in the registry because if a thief steals the computer and finds the public key, there is no way to deduce the private key from it. The 128-bit random file key is now encrypted with the public key and the result stored on disk along with the file, as shown in Fig. 11-11.

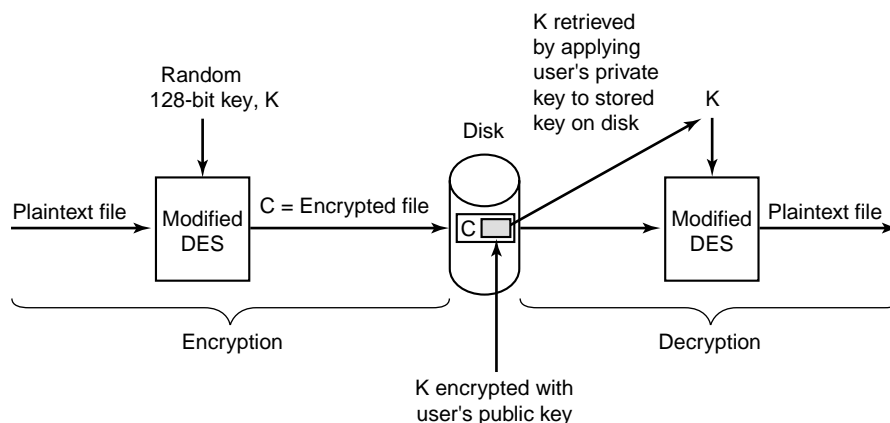


Figure 11-11. Operating of the encrypting file system.

To decrypt a file, the encrypted 128-bit random file key is fetched from disk. However, to decrypt it and retrieve the file key, the user must present the private key. Ideally, this should be stored on a smart card, external to the computer, and only inserted in a reader when a file has to be decrypted. Although Windows 2000 supports smart cards, it does not store private keys on them.

Instead, the first time a user encrypts a file using EFS, Windows 2000 generates a (private key, public key) pair and stores the private key on disk encrypted using a symmetric encryption algorithm. The key used for the symmetric algorithm is derived either from the user's login password or from a key stored on the smart card, if smart card login is enabled. In this way, EFS can decrypt the private key at login time and keep it within its own virtual address space during normal operation so it can decrypt the 128-bit file keys as needed without further disk accesses. When the computer is shut down, the private key is erased from EFS' virtual address space so anyone stealing the computer will not have access to the private key.

A complication occurs when multiple users need access to the same encrypted file. Currently the shared use of encrypted files by multiple users is not supported. However, the EFS architecture could support sharing in the future by encrypting each file's key multiple times, once with the public key of each authorized user. All of these encrypted versions of the file key could be attached to the file.

The potential need to share encrypted files is one reason why this two-key system is used. If all files were encrypted by their owner's key, there would be no way to share any files. By using a different key to encrypt each file, this problem can be solved.

Having a random file key per file but encrypting it with the owner's symmetric key does not work because having the symmetric encryption key just lying around in plain view would ruin the security—generating the decryption key from the encryption key is too easy. Thus (slow) public-key cryptography is needed to encrypt the file keys. Because the encryption key is public anyway, having it lying around is not dangerous.

The other reason the two-key system is used is performance. Using public-key cryptography to encrypt each file would be too slow. It is much more efficient to use symmetric-key cryptography to encrypt the data and public-key cryptography to encrypt the symmetric file key.