NTFS On-Disk Structure

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

Volumes

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

A disk can have one volume or several. NTFS handles each volume independently of the others. Three sample disk configurations for a 150-MB hard disk are illustrated in Figure 12-23.

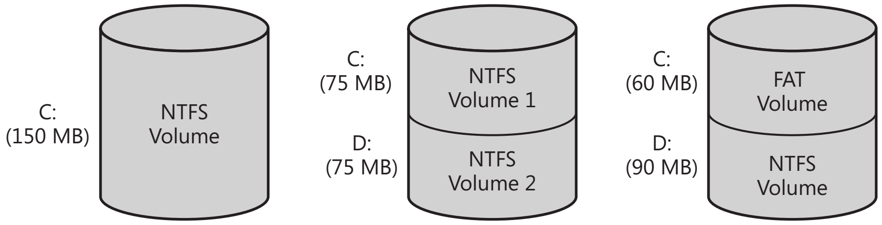


Figure 12-23 Sample disk configurations

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

Note The on-disk format of NTFS volumes on Windows 2000 is version 3.0, and because of minor changes to the format, it is version 3.1 on Windows XP and Windows Server 2003. The version number of a volume is stored in its $Volume metadata file.

Clusters

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

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

NTFS refers to physical locations on a disk by means of *logical cluster numbers* (LCNs). LCNs are simply the numbering of all clusters from the beginning of the volume to the end. To convert an LCN to a physical disk address, NTFS multiplies the LCN by the cluster factor to get the physical byte offset on the volume, as the disk driver interface requires. NTFS refers to the data within a file by means of *virtual cluster numbers* (VCNs). VCNs number the clusters belonging to a particular file from 0 through *m*. VCNs aren't necessarily physically contiguous, however; they can be mapped to any number of LCNs on the volume.

Master File Table

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

The MFT is the heart of the NTFS volume structure. The MFT is implemented as an array of file records. The size of each file record is fixed at 1 KB, regardless of cluster size. (The structure of a file record is described in the "File Records" section later in this chapter.) Logically, the MFT contains one record for each file on the volume, including a record for the MFT itself. In addition to the MFT, each NTFS volume includes a set of metadata files containing the information that's used to implement the file system structure. Each of these NTFS metadata files has a name that begins with a dollar sign ($), although the signs are hidden. For example, the filename of the MFT is $Mft. The rest of the files on an NTFS volume are normal user files and directories, as shown in Figure 12-24.

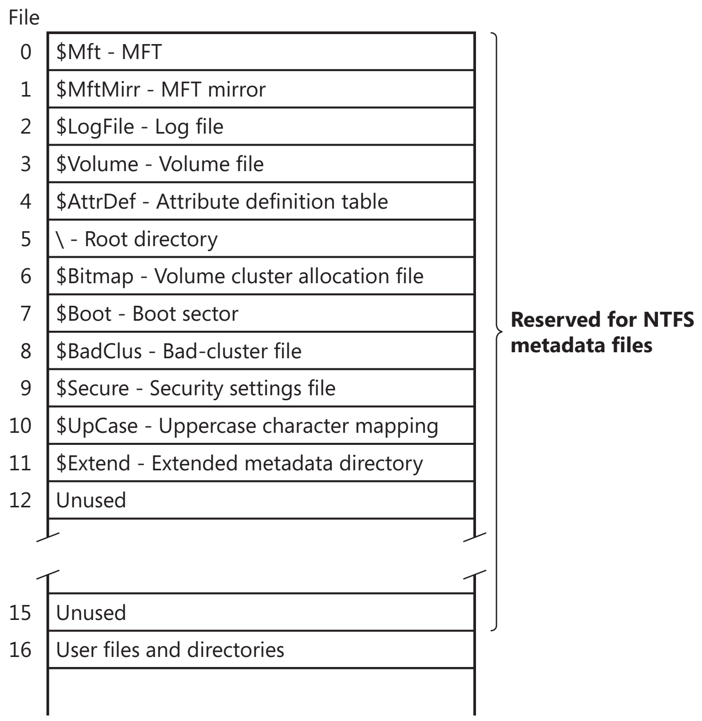

File
0 $Mft - MFT
1 $MftMirr - MFT mirror
2 $LogFile - Log file
3 $Volume - Volume file
4 $AttrDef - Attribute definition table
5 \ - Root directory
6 $Bitmap - Volume cluster allocation file
7 $Boot - Boot sector
8 $BadClus - Bad-cluster file
9 $Secure - Security settings file
10 $UpCase - Uppercase character mapping
11 $Extend - Extended metadata directory
12 Unused
15 Unused
16 User files and directories
Reserved for NTFS metadata files

Figure 12-24 File records for NTFS metadata files in the MFT

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

EXPERIMENT: Viewing the MFT

The Nfi utility included in the OEM Support Tools (part of the Windows debugging tools and available for download at *support.microsoft.com/support/kb/articles/Q253/0/66.asp*) allows you to dump the contents of an NTFS volume's MFT as well as to translate a volume cluster number or physical-disk sector number (on non-RAID volumes only) to the file that contains it, if it's part of a file. The first 16 entries of the MFT are reserved for metadata files, but optional metadata files (which are present only if a volume uses an associated feature) fall outside this area: \$Extend\$Quota, \$Extend\$ObjId, \$Extend\$UsnJrnl, and \$Extend\$Reparse. The following dump was performed on a volume that uses reparse points ($Reparse), quotas ($Quota), and object IDs ($ObjId):

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

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

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

File 2
Log File ($LogFile)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048736-2073343 (0x1f42e0-0x1fa2ff)
File 3
DASD ($Volume)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $OBJECT_ID (resident)
    $SECURITY_DESCRIPTOR (resident)
    $VOLUME_NAME (resident)
    $VOLUME_INFORMATION (resident)
    $DATA (resident)

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

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

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

File 7
Boot Sectors ($Boot)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 0-15 (0x0-0xf)
File 8
Bad Cluster List ($BadClus)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (resident)
    $DATA $Bad (nonresident)

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

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

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

File 12
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)
File 13
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

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

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

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

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

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

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

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

Note For the sake of clarity, the text and diagrams in this chapter depict a run as including a VCN, LCN, and run length. NTFS actually compresses this information on disk into an LCN/next-VCN pair. Given a starting VCN, NTFS can determine the length of a run by subtracting the starting VCN from the next VCN.

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

Another entry in the MFT is reserved for the *root directory* (also known as "\"). Its file record contains an index of the files and directories stored in the root of the NTFS directory structure. When NTFS is first asked to open a file, it begins its search for the file in the root directory's file record. After opening a file, NTFS stores the file's MFT file reference so that it can directly access the file's MFT record when it reads and writes the file later.

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

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

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

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

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

EXPERIMENT: Viewing NTFS Information

In Windows 2000, you can use the NTFSInfo tool from *www.sysinternals.com* to view information about an NTFS volume, including the placement and size of the MFT and MFT zone; and in Windows XP and Windows Server 2003, you can use the built-in Fsutil.exe command-line program:

```
C:\Windows\System32>fsutil fsinfo ntfsinfo c:
NTFS Volume Serial Number :       0xe82828e72828b68a
Version :                         3.1
Number Sectors :                  0x0000000001e461b7
Total Clusters :                  0x00000000003c8c36
Free Clusters  :                  0x00000000000164c8
Total Reserved :                  0x00000000000001b0
Bytes Per Sector  :               512
Bytes Per Cluster :               4096
Bytes Per FileRecord Segment    : 1024
Clusters Per FileRecord Segment : 0
Mft Valid Data Length :           0x0000000006413800
Mft Start Lcn  :                  0x00000000000c5294
Mft2 Start Lcn :                  0x000000000002f427
Mft Zone Start :                  0x00000000003bf7e0
Mft Zone End   :                  0x00000000003bf800
```

File Reference Numbers

A file on an NTFS volume is identified by a 64-bit value called a *file reference*. The file reference consists of a file number and a sequence number. The file number corresponds to the position of the file's file record in the MFT minus 1 (or to the position of the base file record minus 1 if the file has more than one file record). The file reference sequence number, which is incremented each time an MFT file record position is reused, enables NTFS to perform internal consistency checks. A file reference is illustrated in Figure 12-25.

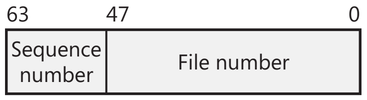


Figure 12-25 File reference

File Records

Instead of viewing a file as just a repository for textual or binary data, NTFS stores files as a collection of attribute/value pairs, one of which is the data it contains (called the *unnamed data attribute*). Other attributes that comprise a file include the filename, time stamp information, and possibly additional named data attributes. Figure 12-26 illustrates an MFT record for a small file.

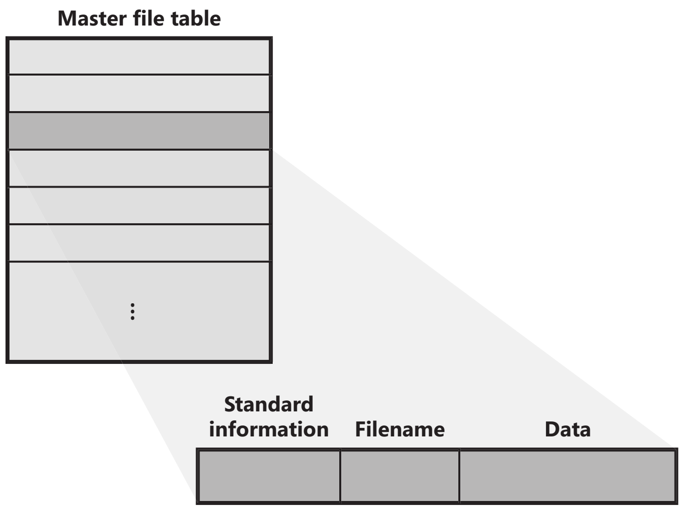


Figure 12-26 MFT record for a small file

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

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

Table 12-4 Attributes for NTFS Files

Attribute	Attribute Name	Description
Volume information	$VOLUME_INFORMATION, $VOLUME_NAME	These attributes are present only in the $Volume metadata file. They store volume version and label information.
Standard information	$STANDARD_INFORMATION	File attributes such as read-only, archive, and so on; time stamps, including when the file was created or last modified; and how many directories point to the file (its *hard link count*).

Table 12-4 **Attributes for NTFS Files**

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

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

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

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

Filenames

Both NTFS and FAT allow each filename in a path to be as many as 255 characters long. Filenames can contain Unicode characters as well as multiple periods and embedded spaces. However, the FAT file system supplied with MS-DOS is limited to 8 (non-Unicode) characters for its filenames, followed by a period and a 3-character extension. Figure 12-27 provides a visual representation of the different file namespaces Windows supports and shows how they intersect.

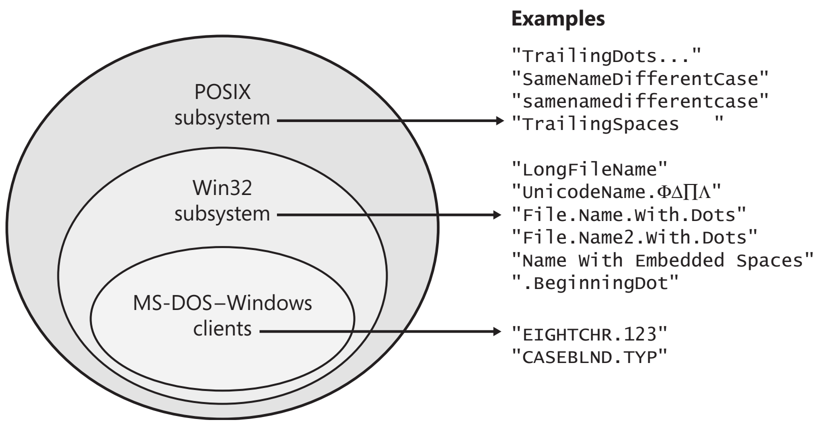


Figure 12-27 Windows file namespaces

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

The relationship between 32-bit Windows (Windows) applications and MS-DOS Windows applications is a much closer one, however. The Windows area in Figure 12-26 represents filenames that the Windows subsystem can create on an NTFS volume but that MS-DOS and 16-bit Windows applications can't see. This group includes filenames longer than the 8.3 format of MS-DOS names, those containing Unicode (international) characters, those with multiple period characters or a beginning period, and those with embedded spaces. When a file is created with such a name, NTFS automatically generates an alternate, MS-DOS-style filename for the file. Windows displays these short names when you use the */x* option with the *dir* command.

The MS-DOS filenames are fully functional aliases for the NTFS files and are stored in the same directory as the long filenames. The MFT record for a file with an autogenerated MS-DOS filename is shown in Figure 12-28.

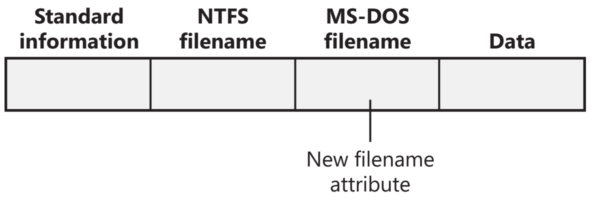


Figure 12-28 MFT file record with an MS-DOS filename attribute

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

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

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

1. Remove from the long name any characters that are illegal in MS-DOS names, including spaces and Unicode characters. Remove preceding and trailing periods. Remove all other embedded periods, except the last one.
2. Truncate the string before the period (if present) to six characters, and append the string "~*n*" (where *n* is a number, starting with 1, that is used to distinguish different files that truncate to the same name). Truncate the string after the period (if present) to three characters.

3. Put the result in uppercase letters. MS-DOS is case-insensitive, and this step guarantees that NTFS won't generate a new name that differs from the old only in case.
4. If the generated name duplicates an existing name in the directory, increment the ~*n* string.

Table 12-5 shows the long Windows filenames from Figure 12-26 and their NTFS-generated MS-DOS versions. The current algorithm and the examples in Figure 12-26 should give you an idea of what NTFS-generated MS-DOS–style filenames look like.

Note Although not generally recommended because it can cause incompatibilities with applications that rely on them, you can disable short name generation by setting HKLM\System\CurrentControlSet\Control\FileSystem\NtfsDisable8dot3NameCreation in the registry to a DWORD value of 1.

Table 12-5 NTFS-Generated Filenames

Windows Long Name	NTFS-Generated Short Name
LongFileName	LONGFI~1
UnicodeName.☞👎⚐☹	UNICOD~1
File.Name.With.Dots	FILENA~1.DOT
File.Name2.With.Dots	FILENA~2.DOT
Name With Embedded Spaces	NAMEWI~1
.BeginningDot	BEGINN~1

Resident and Nonresident Attributes

If a file is small, all its attributes and their values (its data, for example) fit in the file record. When the value of an attribute is stored directly in the MFT, the attribute is called a *resident attribute*. (In Figure 12-27, for example, all attributes are resident.) Several attributes are defined as always being resident so that NTFS can locate nonresident attributes. The standard information and index root attributes are always resident, for example.

Each attribute begins with a standard header containing information about the attribute, information that NTFS uses to manage the attributes in a generic way. The header, which is always resident, records whether the attribute's value is resident or nonresident. For resident attributes, the header also contains the offset from the header to the attribute's value and the length of the attribute's value, as Figure 12-29 illustrates for the filename attribute.

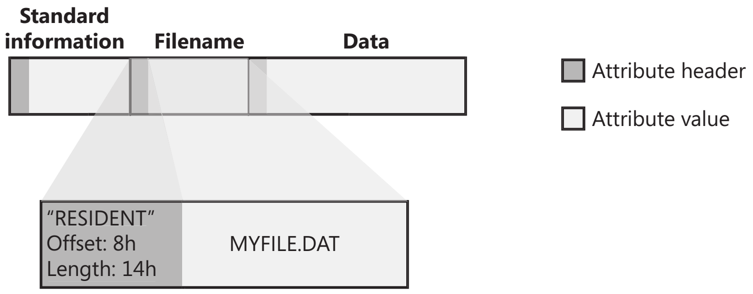


Figure 12-29 Resident attribute header and value

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

The attributes for a small directory, as well as for a small file, can be resident in the MFT, as Figure 12-30 shows. For a small directory, the index root attribute contains an index of file references for the files and the subdirectories in the directory.

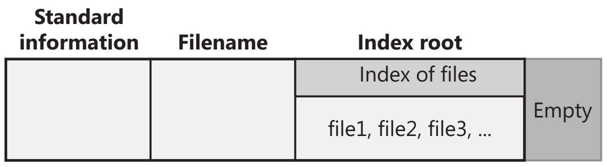


Figure 12-30 MFT file record for a small directory

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

When an attribute is nonresident, as the data attribute for a large file might be, its header contains the information NTFS needs to locate the attribute's value on the disk. Figure 12-31 shows a nonresident data attribute stored in two runs.

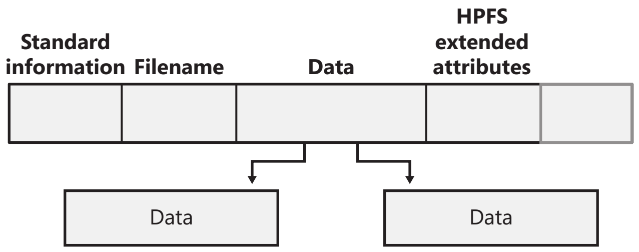


Figure 12-31 MFT file record for a large file with two data runs

Among the standard attributes, only those that can grow can be nonresident. For files, the attributes that can grow are the data and the attribute list (not shown in Figure 12-31). The standard information and filename attributes are always resident.

A large directory can also have nonresident attributes (or parts of attributes), as Figure 12-32 shows. In this example, the MFT file record doesn't have enough room to store the index of files that make up this large directory. A part of the index is stored in the index root attribute, and the rest of the index is stored in nonresident runs called *index buffers*. The index root, index allocation, and bitmap attributes are shown here in a simplified form. They are described in more detail in the next section. The standard information and filename attributes are always resident. The header and at least part of the value of the index root attribute are also resident for directories.

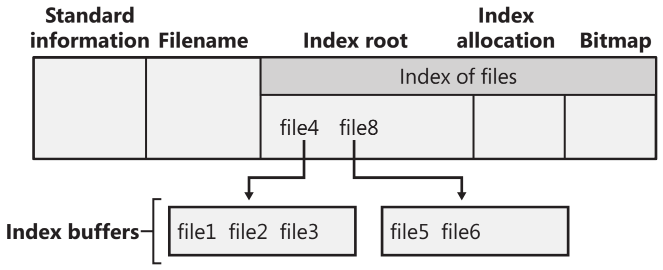


Figure 12-32 MFT file record for a large directory with a nonresident filename index

When a file's (or a directory's) attributes can't fit in an MFT file record and separate allocations are needed, NTFS keeps track of the runs by means of VCN-to-LCN mapping pairs. LCNs represent the sequence of clusters on an entire volume from 0 through *n*. VCNs number the clusters belonging to a particular file from 0 through *m*. For example, the clusters in the runs of a nonresident data attribute are numbered as shown in Figure 12-33.

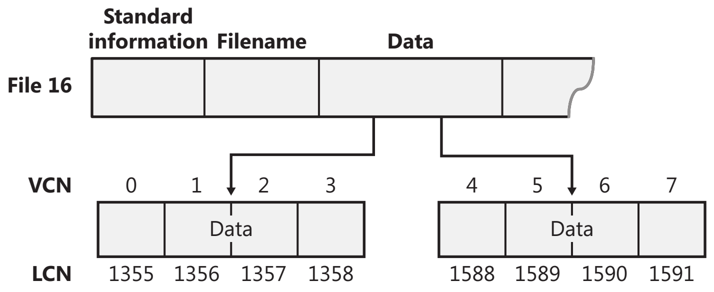


Figure 12-33 VCNs for a nonresident data attribute

If this file had more than two runs, the numbering of the third run would start with VCN 8. As Figure 12-34 shows, the data attribute header contains VCN-to-LCN mappings for the two runs here, which allows NTFS to easily find the allocations on the disk.

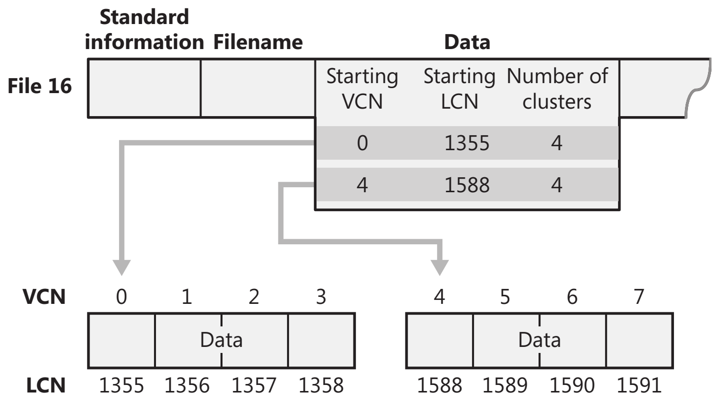


Figure 12-34 VCN-to-LCN mappings for a nonresident data attribute

Although Figure 12-33 shows just data runs, other attributes can be stored in runs if there isn't enough room in the MFT file record to contain them. And if a particular file has too many attributes to fit in the MFT record, a second MFT record is used to contain the additional attributes (or attribute headers for nonresident attributes). In this case, an attribute called the *attribute list* is added. The attribute list attribute contains the name and type code of each of the file's attributes and the file reference of the MFT record where the attribute is located. The attribute list attribute is provided for those cases in which a file grows so large or so fragmented that a single MFT record can't contain the multitude of VCN-to-LCN mappings needed to find all its runs. Files with more than 200 runs typically require an attribute list.

Data Compression and Sparse Files

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

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

Compressing Sparse Data

Sparse data is often large but contains only a small amount of nonzero data relative to its size. A sparse matrix is one example of sparse data. As described earlier, NTFS uses VCNs, from 0 through *m*, to enumerate the clusters of a file. Each VCN maps to a corresponding LCN, which identifies the disk location of the cluster. Figure 12-35 illustrates the runs (disk allocations) of a normal, noncompressed file, including its VCNs and the LCNs they map to.

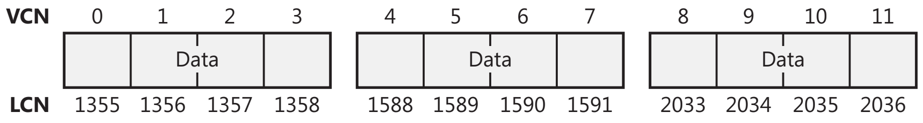


Figure 12-35 Runs of a noncompressed file

This file is stored in 3 runs, each of which is 4 clusters long, for a total of 12 clusters. Figure 12-36 shows the MFT record for this file. As described earlier, to save space, the MFT record's data attribute, which contains VCN-to-LCN mappings, records only one mapping for each run, rather than one for each cluster. Notice, however, that each VCN from 0 through 11 has a corresponding LCN associated with it. The first entry starts at VCN 0 and covers 4 clusters, the second entry starts at VCN 4 and covers 4 clusters, and so on. This entry format is typical for a noncompressed file.

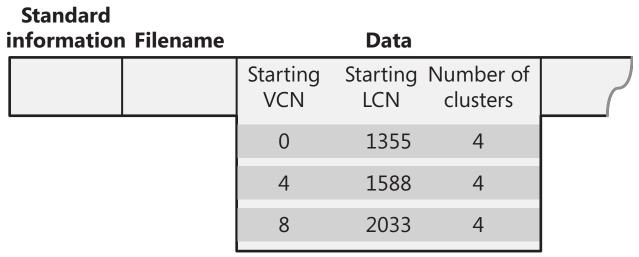


Figure 12-36 MFT record for a noncompressed file

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

Figure 12-37 depicts the runs of a compressed file containing sparse data. Notice that certain ranges of the file's VCNs (16–31 and 64–127) have no disk allocations.

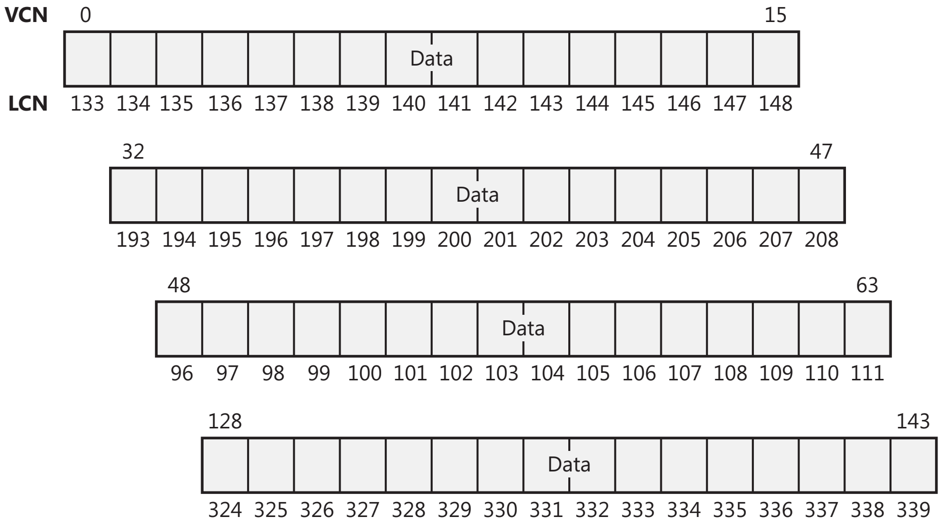
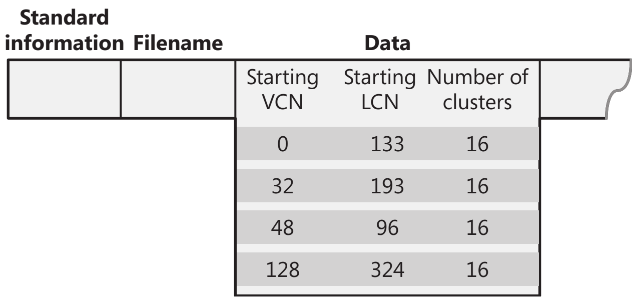


Figure 12-37 Runs of a compressed file containing sparse data

The MFT record for this compressed file omits blocks of VCNs that contain zeros and therefore have no physical storage allocated to them. The first data entry in Figure 12-38, for example, starts at VCN 0 and covers 16 clusters. The second entry jumps to VCN 32 and covers 16 clusters.



Starting VCN	Starting LCN	Number of clusters
0	133	16
32	193	16
48	96	16
128	324	16

Figure 12-38 MFT record for a compressed file containing sparse data

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

Compressing Nonsparse Data

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

In NTFS, users can specify compression for individual files or for all the files in a directory. (New files created in a directory marked compressed are automatically compressed—existing files must be compressed individually.) When it compresses a file, NTFS divides the file's unprocessed data into *compression units* 16 clusters long (equal to 8 KB for a 512-byte cluster, for example). Certain sequences of data in a file might not compress much, if at all; so for each compression unit in the file, NTFS determines whether compressing the unit will save at least 1 cluster of storage. If compressing the unit won't free up at least 1 cluster, NTFS allocates a 16-cluster run and writes the data in that unit to disk without compressing it. If the data in a 16-cluster unit will compress to 15 or fewer clusters, NTFS allocates only the number of clusters needed to contain the compressed data and then writes it to disk. Figure 12-39 illustrates the compression of a file with four runs. The unshaded areas in this figure represent the actual storage locations that the file occupies after compression. The first, second, and fourth runs were compressed; the third run wasn't. Even with one noncompressed run, compressing this file saved 26 clusters of disk space, or 41 percent.

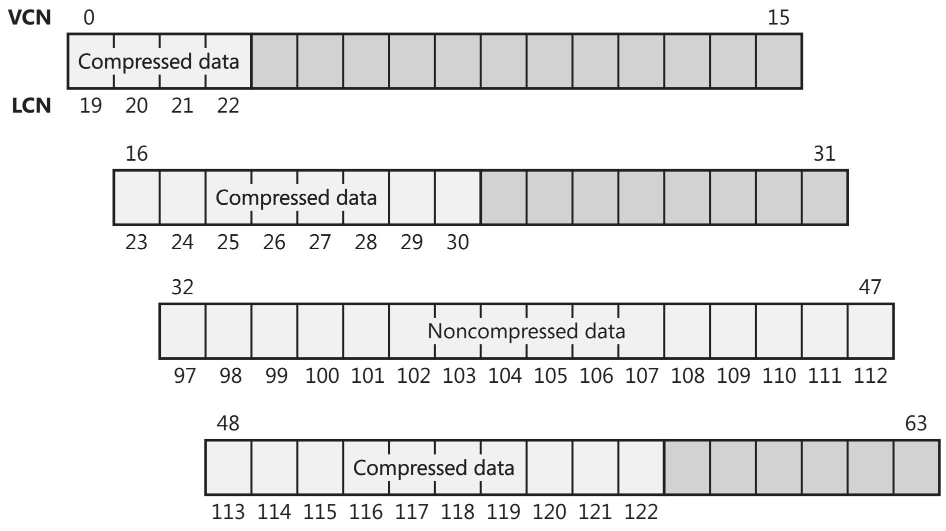
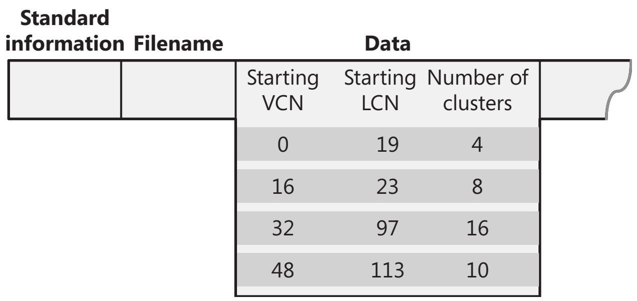


Figure 12-39 Data runs of a compressed file

Note Although the diagrams in this chapter show contiguous LCNs, a compression unit need not be stored in physically contiguous clusters. Runs that occupy noncontiguous clusters produce slightly more complicated MFT records than the one shown in Figure 12-39.

When it writes data to a compressed file, NTFS ensures that each run begins on a virtual 16-cluster boundary. Thus the starting VCN of each run is a multiple of 16, and the runs are no longer than 16 clusters. NTFS reads and writes at least one compression unit at a time when it accesses compressed files. When it writes compressed data, however, NTFS tries to store compression units in physically contiguous locations so that it can read them all in a single I/O operation. The 16-cluster size of the NTFS compression unit was chosen to reduce internal fragmentation: the larger the compression unit, the less the overall disk space needed to store the data. This 16-cluster compression unit size represents a trade-off between producing smaller compressed files and slowing read operations for programs that randomly access files. The equivalent of 16 clusters must be decompressed for each cache miss. (A cache miss is more likely to occur during random file access.) Figure 12-40 shows the MFT record for the compressed file shown in Figure 12-39.



Starting VCN	Starting LCN	Number of clusters
0	19	4
16	23	8
32	97	16
48	113	10

Figure 12-40 MFT record for a compressed file

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

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

NTFS keeps disk allocations for a compressed file contiguous whenever possible. As the LCNs indicate, the first two runs of the compressed file shown in Figure 12-38 are physically contiguous, as are the last two. When two or more runs are contiguous, NTFS performs disk read-ahead, as it does with the data in other files. Because the reading and decompression of contiguous file data take place asynchronously before the program requests the data, subsequent read operations obtain the data directly from the cache, which greatly enhances read performance.

Sparse Files

Sparse files (the NTFS file type, as opposed to files that consist of sparse data, described earlier) are essentially compressed files for which NTFS doesn't apply compression to the file's nonsparse data. However, NTFS manages the run data of a sparse file's MFT record the same way it does for compressed files that consist of sparse and nonsparse data.

The Change Journal File

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

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

The journal is sparse so that it never overflows; when the journal's on-disk size exceeds the maximum defined for the file, NTFS simply begins zeroing the file data that precedes the window of change information having a size equal to the maximum journal size, as shown in Figure 12-41. To prevent constant resizing when an application is continuously exceeding the journal's size, NTFS shrinks the journal only when its size is twice an application-defined value over the maximum configured size.

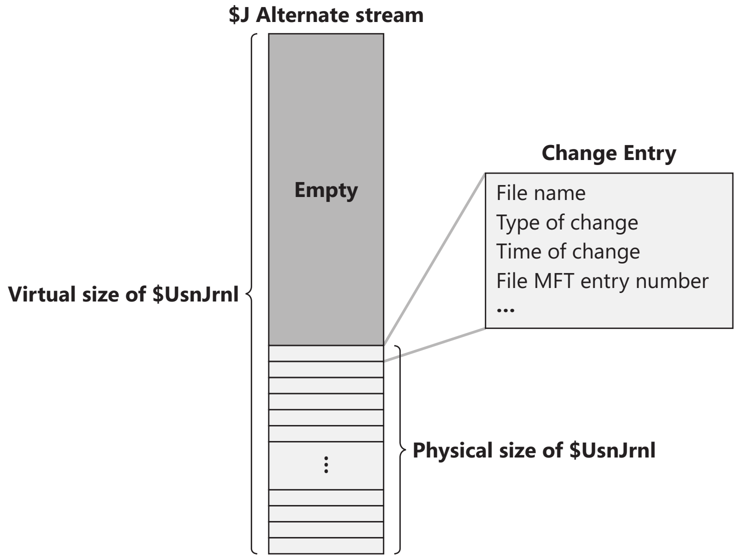


Figure 12-41 Change journal ($UsnJrnl) space allocation

Indexing

In NTFS, a file directory is simply an index of filenames—that is, a collection of filenames (along with their file references) organized in a particular way for quick access. To create a directory, NTFS indexes the filename attributes of the files in the directory. The MFT record for the root directory of a volume is shown in Figure 12-42.

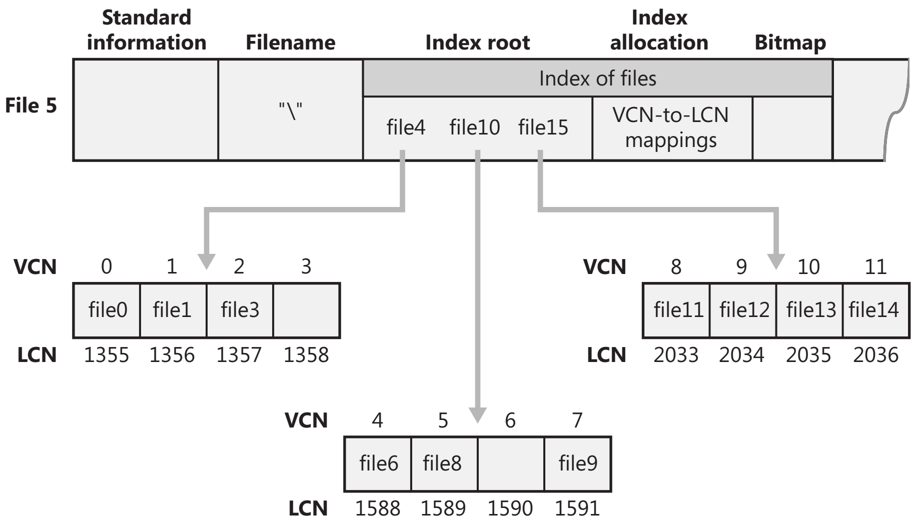


Figure 12-42 Filename index for a volume's root directory

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

Figure 12-42 shows only filenames in the index root attribute and the index buffers (*file6*, for example), but each entry in an index also contains the file reference in the MFT where the file is described and time stamp and file size information for the file. NTFS duplicates the time stamp and file size information from the file's MFT record. This technique, which is used by FAT and NTFS, requires updated information to be written in two places. Even so, it's a significant speed optimization for directory browsing because it enables the file system to display each file's time stamps and size without opening every file in the directory.

The index allocation attribute maps the VCNs of the index buffer runs to the LCNs that indicate where the index buffers reside on the disk, and the bitmap attribute keeps track of which VCNs in the index buffers are in use and which are free. Figure 12-42 shows one file entry per VCN (that is, per cluster), but filename entries are actually packed into each cluster. Each 4-KB index buffer can contain about 20 to 30 filename entries.

The b+ tree data structure is a type of balanced tree that is ideal for organizing sorted data stored on a disk because it minimizes the number of disk accesses needed to find an entry. In the MFT, a directory's index root attribute contains several filenames that act as indexes into the second level of the b+ tree. Each filename in the index root attribute has an optional pointer associated with it that points to an index buffer. The index buffer it points to contains filenames with lexicographic values less than its own. In Figure 12-42, for example, *file4* is a first-level entry in the b+ tree. It points to an index buffer containing filenames that are (lexicographically) less than itself—the filenames *file0*, *file1*, and *file3*. Note that the names *file1*, *file2*, and so on that are used in this example are not literal filenames but names intended to show the relative placement of files that are lexicographically ordered according to the displayed sequence.

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

NTFS also provides general support for indexing data besides filenames, and several NTFS features—including object IDs, quota tracking, and consolidated security—use indexing to manage internal data.

Object IDs

In addition to storing the object ID assigned to a file or directory in the $OBJECT_ID attribute of its MFT record, NTFS also keeps the correspondence between object IDs and their file reference numbers in the $O index of the \$Extend\$ObjId metadata file. The index collates entries by object ID, making it easy for NTFS to quickly locate a file based on its ID. This feature allows applications, using undocumented native API functionality, to open a file or directory using its object ID. Figure 12-43 demonstrates the correspondence of the $Objid metadata file and $OBJECT_ID attributes in MFT records.

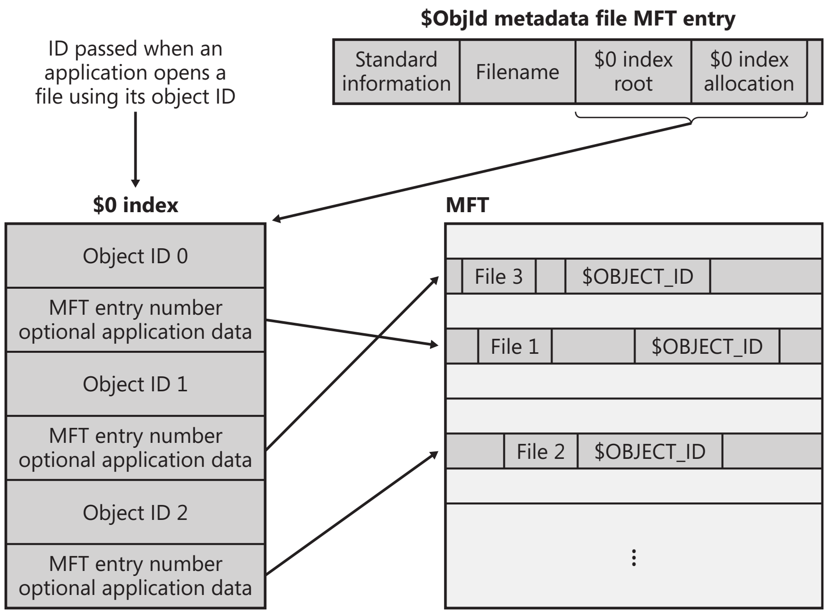

$ObjId metadata file MFT entry
Standard information
Filename
$0 index root
$0 index allocation
ID passed when an application opens a file using its object ID
$0 index
MFT
Object ID 0
MFT entry number optional application data
Object ID 1
MFT entry number optional application data
Object ID 2
MFT entry number optional application data
File 3
$OBJECT_ID
File 1
$OBJECT_ID
File 2
$OBJECT_ID
⋮

Figure 12-43 $Objid and $OBJECT_ID relationships

Quota Tracking

NTFS stores quota information in the \$Extend\$Quota metadata file, which consists of the indexes $O and $Q. Figure 12-44 shows the organization of these indexes. Just as NTFS assigns each security descriptor a unique internal security ID, NTFS assigns each user a unique user ID. When an administrator defines quota information for a user, NTFS allocates a user ID that corresponds to the user's SID. In the $O index, NTFS creates an entry that maps a SID to a user ID and sorts the index by user ID; in the $Q index, NTFS creates a quota control entry. A quota control entry contains the value of the user's quota limits, as well as the amount of disk space the user consumes on the volume.

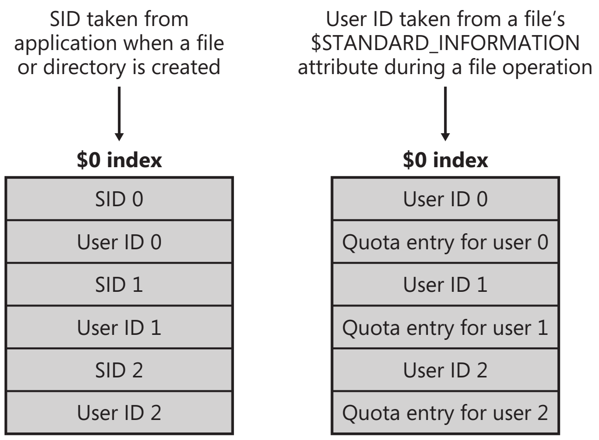


Figure 12-44 $Quota indexing

When an application creates a file or directory, NTFS obtains the application user's SID and looks up the associated user ID in the $O index. NTFS records the user ID in the new file or directory's $STANDARD_INFORMATION attribute, which counts all disk space allocated to the file or directory against that user's quota. Then NTFS looks up the quota entry in the $Q index and determines whether the new allocation causes the user to exceed his or her warning or limit threshold. When a new allocation causes the user to exceed a threshold, NTFS takes appropriate steps, such as logging an event to the System event log or not letting the user create the file or directory. As a file or directory changes size, NTFS updates the quota control entry associated with the user ID stored in the $STANDARD_INFORMATION attribute. NTFS uses general indexing to efficiently correlate user IDs with account SIDs, and, given a user ID, to efficiently look up a user's quota control information.

Consolidated Security

NTFS has always supported security, which lets an administrator specify which users can and can't access individual files and directories. In pre–Windows 2000 NTFS, every file and directory stores its security descriptor in its own security attribute. In most cases, administrators apply the same security settings to an entire directory tree, which results in duplication of security descriptors across all the files and subdirectories to which the settings apply. This duplication can intensively utilize disk space in multiuser environments, such as Windows 2000 Server Terminal Services, in which security descriptors might contain entries for multiple accounts. NTFS in Windows 2000 and later optimizes disk utilization for security descriptors by using a central metadata file named $Secure to store only one instance of each security descriptor on a volume.

The $Secure file contains two index attributes—$SDH and $SII—and a data-stream attribute named $SDS, as Figure 12-45 shows. NTFS assigns every unique security descriptor on a volume an internal NTFS security ID (not to be confused with a SID, which uniquely identifies computers and user accounts) and hashes the security descriptor according to a simple hash algorithm. A hash is a potentially nonunique shorthand representation of a descriptor. Entries in the $SDH index map the security descriptor hashes to the security descriptor's storage location within the $SDS data attribute, and the $SII index entries map NTFS security IDs to the security descriptor's location in the $SDS data attribute.

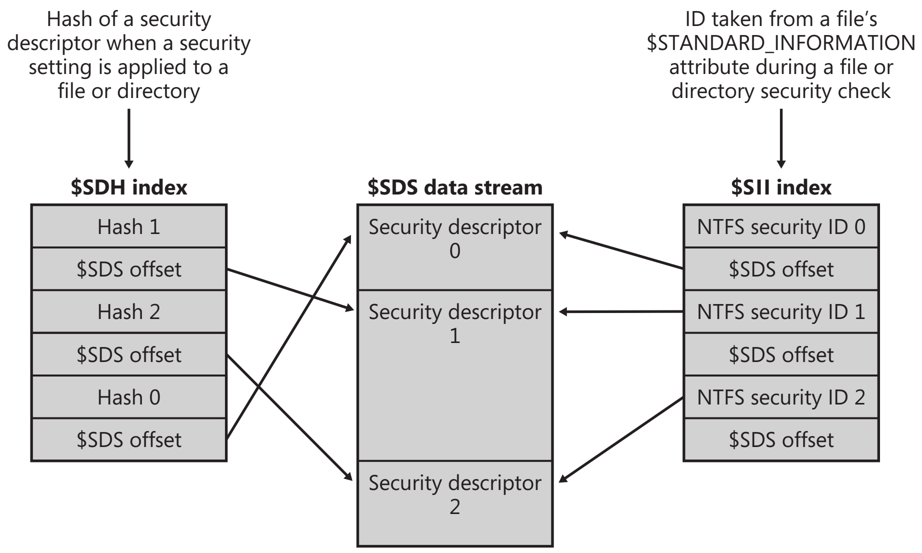


Figure 12-45 $Secure indexing

When you apply a security descriptor to a file or directory, NTFS obtains a hash of the descriptor and looks through the $SDH index for a match. NTFS sorts the $SDH index entries according to the hash of their corresponding security descriptor and stores the entries in a b+ tree. If NTFS finds a match for the descriptor in the $SDH index, NTFS locates the offset of the entry's security descriptor from the entry's offset value and reads the security descriptor from the $SDS attribute. If the hashes match but the security descriptors don't, NTFS looks for another matching entry in the $SDH index. When NTFS finds a precise match, the file or directory to which you're applying the security descriptor can reference the existing security descriptor in the $SDS attribute. NTFS makes the reference by reading the NTFS security identifier from the $SDH entry and storing it in the file or directory's $STANDARD_INFORMATION attribute. The NTFS $STANDARD_INFORMATION attribute, which all files and directories have, stores basic information about a file, including its attributes, timestamp information, and file's security identifier.

If NTFS doesn't find in the $SDH index an entry that has a security descriptor that matches the descriptor you're applying, the descriptor you're applying is unique to the volume and NTFS assigns the descriptor a new internal security ID. NTFS internal security IDs are 32-bit values, whereas SIDs are typically several times larger, so representing SIDs with NTFS security IDs saves space in the $STANDARD_ INFORMATION attribute. NTFS then adds the security descriptor to the $SDS attribute, which is sorted in a b+ tree by the NTFS security ID, and it adds to the $SDH and $SII indexes entries that reference the descriptor's offset in the $SDS data.

When an application attempts to open a file or directory, NTFS uses the $SII index to look up the file or directory's security descriptor. NTFS reads the file or directory's internal security ID from the MFT entry's $STANDARD_INFORMATION attribute. It then uses the $Secure file's

$SII index to locate the ID's entry in the $SDS attribute. The offset into the $SDS attribute lets NTFS read the security descriptor and complete the security check. NTFS stores the 32 most recently accessed security descriptors with their $SII indices in a cache so that it will access only the $Secure file when the $SII isn't cached.

NTFS doesn't delete entries in the $Secure file, even if no file or directory on a volume references the entry. Not deleting these entries doesn't significantly decrease disk space because most volumes, even those used for long periods, have relatively few unique security descriptors.

NTFS's use of general indexing lets files and directories that have the same security settings efficiently share security descriptors. The $SII index lets NTFS quickly look up a security descriptor in the $Secure file while performing security checks, and the $SDH index lets NTFS quickly determine whether a security descriptor being applied to a file or directory is already stored in the $Secure file and can be shared.

Reparse Points

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

NTFS Recovery Support

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

As mentioned earlier (in the section "Recoverability"), NTFS uses a transaction-processing scheme to implement recoverability. This strategy ensures a full disk recovery that is also extremely fast (on the order of seconds) for even the largest disks. NTFS limits its recovery procedures to file system data to ensure that at the very least the user will never lose a volume because of a corrupted file system; however, unless an application takes specific action (such as flushing cached files to disk), NTFS doesn't guarantee user data to be fully updated if a crash occurs. Transaction-based protection of user data is available in most of the database