This article may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist. To maintain the flow of the article, we've left these URLs in the text, but disabled the links.

NT Internals

Inside Win2K NTFS, Part 1

New features improve efficiency, optimize disk utilization, and enable developers to add functionality

By Mark Russinovich

TFS, the native file-system format for Windows 2000, has continuously evolved since its release with Windows NT 3.1. Although NTFS's original features made it suitable as a high-end file-system format, the extensive and significant enhancements that Microsoft added for Win2K address enterprise-level requirements that Microsoft identified as more organizations adopted NT. One feature, consolidated security information, improves the efficiency of NTFS in everyday operation; others, such as quota management, must be leveraged by applications or enabled and managed by an administrator.

In this two-part series, I take you inside the NTFS features introduced by NTFS 5.0 (NTFS5), the version of NTFS included with Win2K. I don't describe administrative interfaces to the features or list programming APIs that let you access them; instead, I discuss how NTFS implements the features behaviorally and in its on-disk format. The features I cover in this column include consolidated security information, reparse points (including mount points and junctions), and quota management. In Part 2, I will conclude with a look at sparse-file support, the change journal, link tracking, and encryption.

Before you proceed, you should have a firm understanding of basic NTFS on-disk organization, including Master File Table (MFT) entries, attribute types, and resident and nonresident attributes. If you're not familiar with these concepts, you might want to read "Inside NTFS," January 1998, as a primer. For more background information, see "Related Articles in Previous Issues," page 46. The sidebar "Exploring NTFS On-disk Structures," page 46, describes several tools for viewing and sources of information about NTFS internal data structures such as those I describe in this column.

General Indexing

Several new NTFS5 features rely on a fundamental NTFS feature called *attribute indexing*. Attribute indexing consists of sorting entries of a particular attribute type, using an efficient storage mechanism for fast lookups. Pre-Win2K versions of NTFS support attribute indexing for only $I30, the index attribute that stores directory entries. The attribute indexing process sorts directory entries by name and stores the entries as a B+ tree (a form of a binary tree that stores multiple items at each node in the tree). Figure 1, page 47, illustrates the MFT entry of a directory that contains nine entries stored in three nodes, each with three entries. The *index root* attribute contains the root of the B+ tree. Because the nine entries don't all fit in the directory's MFT entry, NTFS must store some of the entries elsewhere. Consequently, NTFS allocates two *index allocation* buffers to store two of the entries. (Index root and index allocation buffers typically can store entries for more than three files, depending on the length of the filenames.) As I explain in "Inside NTFS," an MFT entry is 1KB in size and

index allocation buffers are 4KB in size.

The red arrows in the figure emphasize the way NTFS stores entries that are alphabetically sequential. If you ran a program that opened the file e.bak in the directory that the figure illustrates, NTFS would read the index root attribute, which contains entries for d.new, h.txt, and i.doc, and compare the string e.bak with the name in the first entry, d.new. NTFS would conclude that e.bak is alphabetically greater than d.new and would therefore proceed to the next entry, h.txt. After performing the same comparison, NTFS would find that e.bak is alphabetically less than h.txt. NTFS would then look in h.txt's directory entry for the virtual cluster number (VCN) of the index buffer that contains directory entries alphabetically less than h.txt (but greater than d.new). VCNs represent a cluster's order within a file or directory. NTFS uses mapping information to translate a VCN to a Logical Cluster Number (LCN), which is the number of a cluster relative to the start of a volume. If the directory entry for h.txt didn't store a VCN for an index buffer, NTFS would immediately know that the h.txt directory doesn't contain e.bak and would indicate that the lookup failed.

After obtaining the VCN of the starting cluster of the index buffer that NTFS will examine next, NTFS reads the index allocation buffer and scans it for a match. In Figure 1, the index buffer's first entry is the one NTFS is searching for, so NTFS reads the number of e.bak's MFT entry from e.bak's directory entry. Directory entries also store other information, such as the file's timestamps (e.g., created, last modified), size, and attributes. Although NTFS also stores this information in the file's MFT entry, duplicating the information in a directory entry saves NTFS the trouble of reading the file's MFT entry to obtain the information when listing directories and doing simple file queries.

Directory entries are sorted alphabetically, which explains why NTFS files are always printed alphabetically in directory listings. In contrast, the FAT file system doesn't sort directories, so FAT listings aren't sorted. Further, because NTFS stores entries as a B+ tree, lookups for particular files in a large directory are very efficienttypically, NTFS needs to scan only a fraction of a directory. This approach contrasts with FAT's linear lookups, which require FAT to potentially examine every entry in a directory while searching for a specific name.

Whereas pre-Win2K NTFS implements indexing only for filenames, NTFS5 implements general indexing, which lets NTFS5 store arbitrary data in indexes and sort the data entries by something other than a name. NTFS5 uses general indexing to manage security descriptors, quota information, reparse points, and file object identifiers features that I explain in this series.

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-Win2K 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 Win2K Server Terminal Services and NT Server 4.0, Terminal Server Edition (WTS), in which security descriptors might contain entries for multiple accounts. NTFS5 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 $SIIand a data-stream attribute named $SDS, as Figure 2 shows. NTFS5 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 NTFS5 security IDs to the security descriptor's location in the $SDS data attribute.

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 and timestamp information. Win2K expands this attribute to accommodate additional data, such as the 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, then 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 NTFS security ID, and 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, 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. NTFS5 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.

NTFS5'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

Reparse points let an application associate a block of application data with a file or directory and let the Object Manager reparse, or reexecute a name lookup, when an application encounters a reparse point. (For information about the Object Manager's role in the OS's architecture, see "Inside NT's Object Manager," October 1997.) In addition to storing the reparse data, the reparse point stores a reparse code that identifies the reparse point as belonging to a particular application. Although not useful by themselves, reparse points let Win2K or third-party developers build functionality. Win2K provides several types of reparse-point functionality, including mount points, NTFS junctions, and Hierarchical Storage Management (HSM). I discuss the way each of these functionalities works, then delve into the implementation of reparse points.

To be accessible, all NT volumes must have a drive letter. NTFS5 mount points let you connect a volume at a mount point directory on a parent NTFS5 volume without assigning a drive letter to the child volume. This capability lets you consolidate multiple volumes under one drive letter. For example, if you mount a volume that contains the directory \articles to a mount point named C:\documents, you can use the path C:\articles\documents to access the \documents directory's files. A mount point is a reparse point whose data consists of a volume's internal name. This internal name has the form \??\Volume{XX-XX-XX-XX}, where the Xs are the numbers that make up the globally unique ID (GUID) that Win2K assigned to the volume.

When you open the file C:\articlesdocuments\column.doc, NTFS encounters the mount point associated with the \documents directory. NTFS reads the mount point's reparse data (the volume name) and returns a reparse status to the Object Manager. The I/O manager intercepts the reparse status, examines the reparse data, and determines that NTFS encountered a mount point. The I/O manager modifies the name being looked up and directs the Object Manager (the kernel component that guides name lookups) to reissue the lookup using the modified path \??\Volume {*GUID*}\documents\column.doc. The reissued lookup causes name parsing for \documents\column.doc to continue on the mounted volume. To create and list mount points, you can use the Microsoft Management Console (MMC) Disk Management snap-in or the Mountvol command-line tool that comes with Win2K.

NTFS junctions are similar to mount points, and the I/O manager and Object Manager implement junctions as they implement mount points. However, junctions reference directories rather than volumes. Junctions are the Win2K equivalent of UNIX symbol links (although unlike UNIX symbolic links, junctions can't be applied to files). If you create the junction C:\articles\documents that references D:\documents, you can access files stored in D:\documents by using the path C:\articles\documents. The junction's reparse point stores the redirected path information, and as for mount-point traversal, the I/O manager modifies the name and reissues the name lookup when NTFS encounters a junction. Figure 3 illustrates how junctions work. When the application opens C:\directory1\file, NTFS encounters a reparse point on C:\directory1 that points at C:\directory2. The I/O manager changes the name to C:\directory2\file, and the application ultimately opens C:directory2\file.

Win2K doesn't include tools for creating junctions, and Microsoft doesn't officially support such tools because some applications might not function properly when they use paths that contain junctions. However, you can use either the Linkd tool from the *Microsoft Windows 2000 Resource Kit* or the free Junction tool (http://www.sysinternals.com/misc.htm) to create and list junctions.

Not all reparse points rely on path reparsing functionality. The HSM system uses HSM reparse points to transparently migrate infrequently accessed files to offline storage. When HSM moves a file offline, the HSM system deletes the file's contents and creates a reparse point in the file's place. The reparse point data contains information the HSM system uses to locate the file's data on archival media. When an application later accesses an offline HSM file, the HSM driver RsFilter.sys (Remote Storage Filter) intercepts the reparse code that NTFS returns to the Object Manager. The driver deletes the reparse point, fetches the file data from archival storage, then reissues the original request. This time, NTFS accesses the file as it would any other, and the application doesn't realize that data shuffling occurred.

When a file or directory has a reparse point associated with it, NTFS creates an attribute named $Reparse for the reparse point. This attribute stores the reparse code and data. So that NTFS can easily locate all reparse points on a volume, a metadata file named \$Extend\$Reparse stores entries that connect the reparse point file and directory MFT entry numbers to their associated reparse point codes. NTFS sorts the entries by MFT entry number in the $R index.

Quota Tracking

One of the most popular roles that Win2K Server and NT Server systems play is that of a file server. A deficiency of NT is that systems administrators have no built-in tools to monitor or control the amount of server disk space that individual users consume. Several third-party quota-management products are available to fill this gap in NT's functionality. However, Win2K provides basic quota management that in many cases alleviates the need for third-party tools. Win2K quota management is based on NTFS quota support (FAT volumes don't support quota management) and is a per-volume, per-user model. In other words, an administrator can use an interface like the one that Figure 4 shows to specify warning and limit thresholds on the amount of disk space a user can consume on an NTFS volume.

NTFS stores quota information in the \$Extend\$Quota metadata file, which consists of the indexes $O and $Q. Figure 5 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.

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.

Yet More Features

This discussion of quota tracking brings us to the end of this issue's column. In my next column, I conclude this series about new NTFS features by describing the implementation of distributed link tracking support, sparse- file support, volume change tracking, and encryption.