

Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

The evolution of storage management in Windows NT begins with DOS, Microsoft's first OS. As hard disks became larger, DOS needed to accommodate them. To do so, one of the first steps Microsoft took was to let DOS create multiple partitions, or logical disks, on a physical disk. DOS could format each partition with a different file-system type (i.e., FAT12 or FAT16) and assign each partition a different drive letter.

DOS 3 and 4 were severely limited in the size and number of partitions they could create, but in DOS 5 the partitioning scheme fully matured. DOS 5 was able to divide a disk into any number of partitions of any size. NT borrowed the partitioning scheme that evolved in DOS to provide disk compatibility with DOS and Windows 3.x, and to let the NT development team rely on proven tools for disk management. In NT, Microsoft extended the basic concepts of DOS disk partitioning to support storage-management features that an enterprise-class OS requires: disk spanning and fault tolerance. Starting with the first version of NT, systems administrators have been able to create volumes that comprise multiple partitions. This capability lets large volumes consist of partitions from multiple physical disks and allows implementation of fault tolerance through software-based data redundancy.

Although NT's partitioning support is flexible enough to support most storage-management tasks, this support suffers from several drawbacks. One drawback is that most disk-configuration changes require a reboot to take effect. In today's world of servers that must remain online for months or even years at a time, any reboot—even a planned reboot—is a major inconvenience. Another drawback is that the NT Registry stores advanced disk-configuration information. This arrangement means that moving configuration information is onerous when you move disks between systems, and losing configuration information is easy when you need to reinstall NT. Finally, NT's requirement that each volume have a unique drive letter in the A through Z range places an upper limit on the number of possible local and remote volumes that users can create.

Windows 2000 (Win2K) eases many NT 4.0 storage-management restrictions with a slew of new storage-management enhancements. From volume mount points that remove the limit on the number of possible volumes, to integrated support for file migration to offline storage, to disk management without reboots, Win2K takes NT storage management to a level on par with most advanced UNIX systems. This month, I begin a two-part series that looks at storage-management internals in NT and Win2K. I begin by describing NT 4.0 disk-management implementation, including partitioning, drive-letter assignment, the mount process, and details of NT's software-based fault-tolerant drives. I'll conclude the series by looking at Win2K storage-management changes such as the Logical Disk Manager (LDM) and mount points.

Partitioning

As I've stated, the foundation of NT 4.0 disk management is the partitioning scheme that NT inherited from DOS 5. Before I delve into partitioning, let me define the terminology I use. A disk is a physical storage device such as a hard disk, a 3.5" disk, or a CD-ROM. A disk's hardware divides the disk into sectors, addressable blocks of fixed size. All x86-processor hard-disk sectors are 512 bytes, whereas CD-ROM sectors are typically 2048 bytes. A volume is an object that represents sectors from the same or different partitions that a file system manages as one unit. A volume is typically associated with one partition, but if you create a spanned volume or a volume that has data redundancy, the volume consists of more than one partition, possibly spread across different disks. People use the word drive to refer sometimes to disks and sometimes to volumes. To avoid confusion, I won't use drive.

When you install NT on a computer, one of the first things the OS requires you to do is to create a partition on the system's primary physical disk. NT defines the system volume on this partition and

Inside Storage Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

stores the files that it invokes early in the boot process on the system volume. In addition, NT Setup requires you to create a partition onto which the setup program installs the NT system files and creates the system directory. This partition serves as the home for the boot volume, which is where NT Setup installs the NT system files and creates the NT system directory (\winnt). The system and boot volume can be the same volume. The nomenclature that Microsoft defines for system and boot volumes is a little confusing. The system volume is where NT places boot files, including NT Loader (NTLDR) and NTDETECT, and the boot volume is where NT stores OS files such as ntoskrnl.exe, the core kernel file.

The standard BIOS implementations that x86 hardware uses dictate one requirement of NT's partitioning format—that the first sector of the primary disk contain the Master Boot Record (MBR). When an x86 processor boots, the computer's BIOS reads the MBR and treats part of the sector's contents as executable code. The BIOS invokes the MBR code to initiate an OS boot process after the BIOS performs preliminary configuration of the computer's hardware. In the case of Microsoft OSs, including NT, the MBR also contains a partition table. A partition table consists of four entries that define the locations of as many as four primary partitions on a disk. Numerous predefined partition types exist, and a partition's type, which the partition table records, specifies which file system the partition includes. For example, partition types exist for FAT32 and NTFS. A special partition type, extended partition (or Extended Boot Record—EBR), contains another MBR with its own partition table. By using extended partitions, Microsoft's OSs overcome the apparent limit of four partitions per disk, as an MBR's partition table defines. In general, the recursion that extended partitions permit can continue indefinitely, which means that no upper limit exists to the number of possible partitions on a disk. [Figure 1](#) shows an example disk-partitioning scenario.

NT's boot first makes evident the distinction between primary and extended partitions. The system must mark one primary partition of the primary disk as active. The NT code in the MBR will transfer control to the code that the first sector of the active partition stores after the MBR code loads the sector's code into memory. The active partition is the NT system volume. NT designates the first sector of a partition the boot sector because, in the primary partition's case, the sector plays a role in the computer's boot process. However, every partition formatted with a file system has a boot sector that stores information about the structure of the file system on that partition. The second instance in which NT makes evident the distinction between primary and extended partitions is drive-letter assignment, which I discuss shortly.

Storage Drivers and Device Objects

NTLDR is the NT OS file that conducts the first portion of the NT process. NTLDR resides on the system volume; the boot sector code on the system volume executes NTLDR. NTLDR reads the boot.ini file from the system volume and presents the computer's boot choices to the user. The partition names that boot.ini designates are in the form multi(0)disk(0)rdisk(0)partition(1). These names are Advanced RISC Computing (ARC) names because they're part of a standard partition-naming scheme that Alpha firmware and other RISC processors use. NTLDR translates the name of the boot.ini boot entry that a user selects to the appropriate boot partition, then loads the NT system files (starting with the Registry, ntoskrnl.exe, and the boot drivers) into memory to continue the boot process.

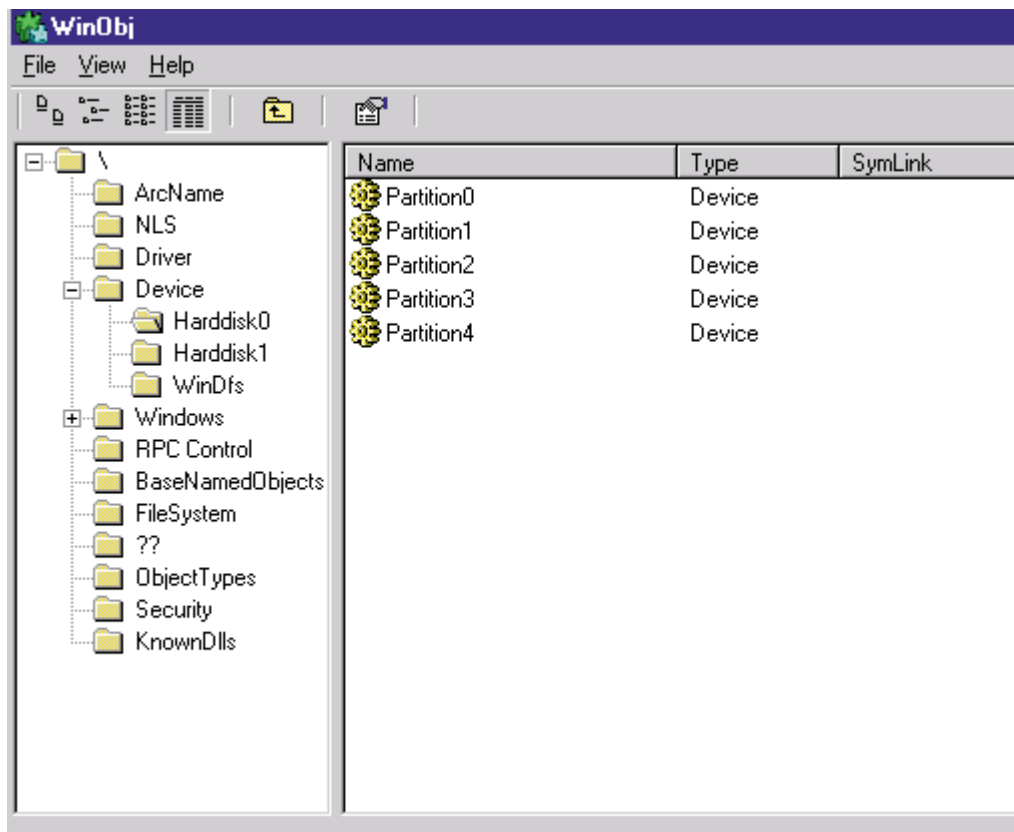
During initialization, the NT kernel starts the hard disk storage drivers. Storage drivers in NT follow a class/port/miniport architecture, in which Microsoft supplies a class driver that implements functionality common to all storage devices and a port driver that implements functionality common to a particular bus (e.g., SCSI, IDE). OEMs supply miniport drivers that plug into the port driver to interface NT to a particular implementation. For example, Adaptec supplies SCSI miniport drivers for the company's

Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

various SCSI controllers. This division's benefits are twofold. First, a miniport developer writes to the miniport environment rather than to the more complex NT driver environment. Second, Microsoft provides class and port drivers for Windows 9x so that miniport drivers that developers write for NT run on Win9x, and vice versa.

As miniport drivers present to the class driver the disks that they identify early in the boot, NT's I/O Manager includes the `IoReadPartitionTable` function, which the class driver invokes for each disk. `IoReadPartitionTable` then invokes sector-level disk I/O, which the class, port, and miniport drivers provide to read a disk's partition table and construct an internal representation of the disk's partitioning. The Disk class driver creates device objects to represent each primary partition (including primary partitions within extended partitions) that the driver obtains from `IoReadPartitionTable`. Device drivers use device objects to represent both physical and logical devices, including disks, keyboards, and driver management interfaces. The device drivers can name a device object so that other device drivers or applications can open the object and send I/O requests to the object. The Disk class driver creates a `HarddiskX` directory (in which X is the disk number NT assigns the disk) in the NT Object Manager namespace for each hard disk. The Disk class driver then places partition device objects in the `HarddiskX` directory of the disk on which the objects reside. For example, the figure shows the contents of the `\device\harddisk0` directory on a computer whose primary hard disk has four partitions, which the four numbered partition device objects in the right-hand pane represent.



The Disk class driver gives the name `partition0` to the device object that represents the entire physical disk.

Whenever a device driver or application sends an I/O request to a device object, the NT I/O Manager routes the request (which comes in an I/O request packet—IRP—a self-contained package) to the

Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

device driver that created the target device object. Thus, if an application wants to read the boot sector of the second partition on harddisk0, the application first opens the device object `\device\harddisk0\partition2`, then sends the object a request to read 512 bytes starting at offset zero on the device. The NT I/O Manager routes the application's request to the Disk class driver, notifying the class driver that the IRP is aimed at the partition2 device object. Because partitions are logical conveniences that NT uses to represent contiguous areas on a physical disk, the class driver must translate offsets that are relative to a partition to offsets that are relative to the beginning of a disk. If partition2 begins 4096 sectors into the disk, the class driver would adjust the IRP's parameters to designate an offset with that value before passing the request to the miniport driver. The miniport driver carries out physical disk I/O and reads requested data into an application buffer designated in the IRP.

The Win32 API is unaware of the NT Object Manager namespace. NT reserves a couple of different namespace subdirectories for Win32's use and names one of these subdirectories `\??`. In this subdirectory, NT makes available device objects that Win32 applications interact with—including COM and parallel ports—as well as disks. Because disk objects actually reside in other subdirectories, NT uses symbolic links to connect names under `\??` with objects located elsewhere in the namespace. For each physical disk on a system, the I/O Manager creates a `\??\PhysicalDriveX` link that points to `\device\harddiskX\partition0` (numbers beginning with 0 replace X). Win32 applications that directly interact with the sectors on a disk open the disk by specifying the name `\\.\PhysicalDriveX` (in which X is the disk number) to invoke the Win32 CreateFile API. The Win32 application layer converts the name to `\??\PhysicalDriveX` before handing the name to the NT Object Manager.

Drive-Letter Assignments

After the I/O Manager initializes the disk storage drivers, it invokes the internal function `IoAssignDriveLetters`. This function creates a symbolic link under `\??` in the form of a drive letter for each disk partition, as well as for CD-ROMs and 3.5" disks. The drive-letter symbolic links refer to associated partition device objects. The I/O Manager's drive-letter assignment follows a default formula, but you can override the formula by explicitly assigning drive letters in Disk Administrator.

After you start Disk Administrator, the program scans the partitions on the system's hard disks and generates a random signature for each partition that Disk Administrator hasn't seen in previous executions. Disk Administrator stores a partition's signature in the partition's boot sector and also in the Registry value `HKEY_LOCAL_MACHINE\SYSTEM\DISK\Information`. The Information value includes a data structure for each disk partition that incorporates the Disk Administrator signature and the partition's drive letter, if you've assigned one. `IoAssignDriveLetters` reads the Information value and honors the drive letters you've specified before performing default assignments. The function reads partition signatures and matches them with the data that the Information value stores to correlate partitions with their assigned drive letters.

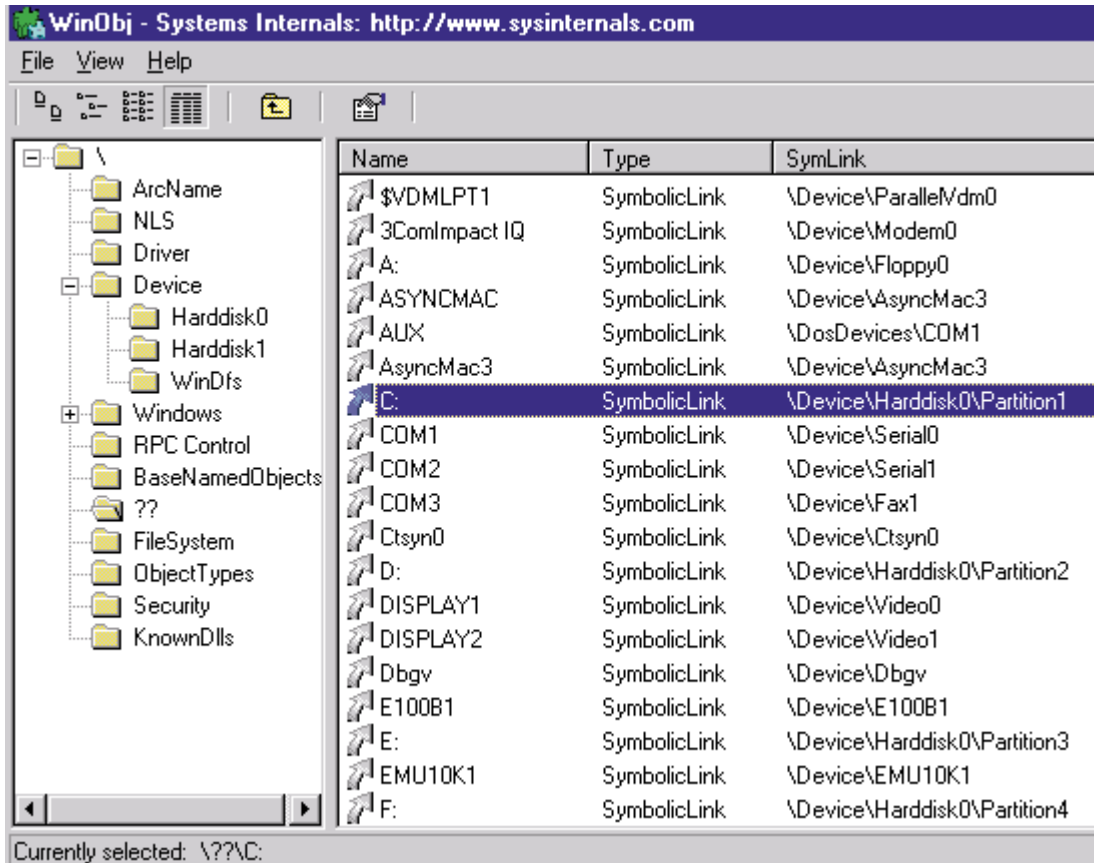
After `IoAssignDriveLetters` assigns explicitly specified drive letters, the function starts with the letter C (or the first unassigned letter higher than C) and assigns letters to the first active primary partition of each disk. If a disk has no active primary partition, `IoAssignDriveLetters` assigns a letter to the first primary partition. In the subsequent phase of assignment, `IoAssignDriveLetters` gives letters to each partition that is in each disk's extended partitions. Finally, `IoAssignDriveLetters` creates letters for the remaining unassigned primary partitions.

After `IoAssignDriveLetters` has created drive-letter symbolic links for hard disk partitions, the function gives letters to 3.5" disks and then to CD-ROMs. The first two 3.5" disks get the letters A and B, and any others receive the next available letter. You can assign letters to CD-ROMs in Disk Administrator,

Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

but rather than storing those assignments in the Information value, Disk Administrator stores the assignments in separate values that share the names of the device objects NT uses to represent the CD-ROMs. For example, a system with one CD-ROM that has an assigned drive letter will have a Registry value \device\cdrom0 beneath HKEY_LOCAL_MACHINE\SYSTEM\DISK that specifies the CD-ROM's assigned drive letter. The figure below shows the contents of a system's Object Manager \?? directory and highlights the C drive's symbolic link.



File-System Mounting

Because NT assigns a drive letter to a partition doesn't mean that the partition contains data that is organized by a file-system format NT recognizes. The volume-recognition process consists of a file system claiming ownership for a partition; that process takes place the first time the kernel, a device driver, or an application accesses a file or directory on a partition. After a file-system driver signals its responsibility for a partition, the I/O Manager directs all IRPs aimed at the partition to the owning driver. Mount operations in NT 4.0 consist of three components: file-system driver registration, Volume Parameter Blocks (VPBs), and mount requests.

The I/O Manager oversees the mount process and is aware of available file-system drivers because all file-system drivers register with the I/O Manager when they initialize. The I/O Manager provides the IoRegisterFileSystem function to local disk (rather than network) file-system drivers for this registration. When a file-system driver registers, the I/O Manager stores a reference to the driver in a list that the I/O Manager uses during mount operations.

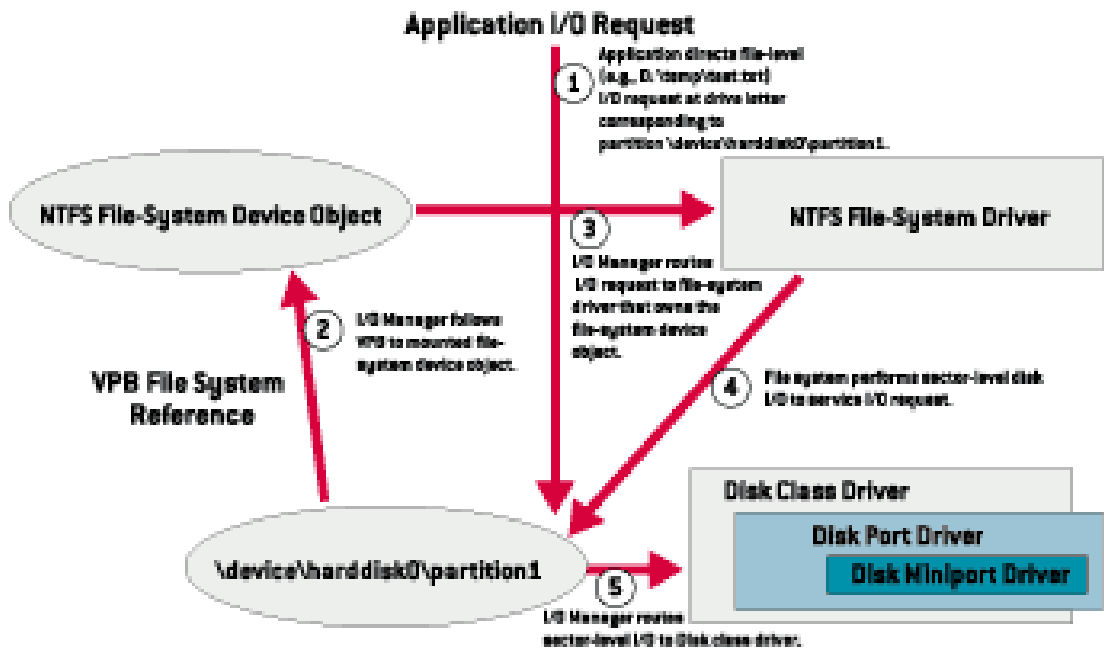
Every device object contains a VPB data structure, but the I/O Manager treats VPBs as meaningful only for partition device objects. A VPB serves as the link between a partition device object and the

Inside Storage Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

device object that a file-system driver creates to represent a mounted file-system instance for that partition. If a VPB's file-system reference is empty, then no file system has mounted the partition. The I/O Manager checks a partition device object's VPB whenever an open API that specifies a filename or directory name on a partition device object executes. For example, if the I/O Manager assigns drive letter D to the second partition on a system's first hard disk, IoAssignDriveLetters creates a \??\D: symbolic link that resolves to the device object \device\harddisk0\partition2. A Win32 application that attempts to open the \test file on the D drive specifies the name D:\test, which the Win32 subsystem converts to \??\D:\test before invoking NtCreateFile, the kernel's file-open routine. NtCreateFile uses the Object Manager to parse the name, and the Object Manager encounters the \device\harddisk0\partition2 device object with the path \test still unresolved. At that point, the I/O Manager checks to see whether \device\harddisk0\partition2's VPB references a file system. If not, the I/O Manager uses a mount request to ask each registered file-system driver whether the driver recognizes the format of the partition in question as the driver's own. If a file-system driver signals affirmatively, the I/O Manager fills in the VPB and passes the open request with the remaining path (i.e., \test) to the file-system driver. The file-system driver completes the request by using its file-system format to interpret the data that the partition stores. After a mount fills in a partition device object's VPB, the I/O Manager hands subsequent open requests aimed at the partition to the mounted file-system driver. If no file-system driver claims a partition, then RAW—a file-system driver built into NT—claims the partition and fails all requests to open files on the partition. The figure below shows a simplified example (i.e., the figure omits the file-system driver's interactions with the NT Cache Manager) of the path that I/O that is directed at a mounted partition follows.



Aside from the boot volume, which a driver mounts while the kernel is initializing, file-system drivers mount most volumes when Chkdsk runs during the blue-screen portion of the boot sequence. Chkdsk accesses each drive letter to see whether the volume associated with the letter requires a consistency check. Mounting can occur more than once for the same disk with removable media (e.g., 3.5" disk device). CD-ROM File System (CDFFS) and FAT, NT's two file-system drivers that support removable media, respond to media changes by querying the disk's volume identifier. If either driver sees the volume identifier change, the driver dismounts the disk and attempts to remount it.

Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

Fault Tolerance

NT's I/O architecture permits a powerful feature: dynamic layering of device objects. A device driver can create a device object and attach it to a target device object. The I/O Manager routes requests directed at a target device object to the object's attached device object, if one exists. Device drivers use this mechanism to monitor or change the behavior of device objects that belong to other device drivers. A driver that relies on layering is a filter driver, and when a filter driver receives an IRP aimed at a target device, the filter has full control over the request. The filter can fail the request, create new subrequests, or pass the unmodified request to the target device. NT storage drivers commonly use layering in three places. At the highest level, file-system filter drivers attach to the target device objects that represent mounted partitions that file-system drivers create. A file-system filter driver therefore intercepts requests aimed at mounted volumes so that the driver can implement functionality such as monitoring, encryption, or on-access virus scanning.

If you've installed NT disk performance counters by executing the Diskperf y command, then you've installed the DiskPerf filter driver. DiskPerf attaches to the device objects that represent physical disks (e.g., \device\harddisk0\partition0) so that DiskPerf can generate performance-related statistics for Performance Monitor to present. If you create a nonstandard volume—such as a volume set, mirrored drive, stripe set, or stripe set with parity—in Disk Administrator, you enable the FtDisk filter driver.

A volume set is a volume that uses two or more partitions to create the image of one contiguous partition. A systems administrator can use partitions from different disks to create a volume set that is larger than any given physical disk on a computer. A mirror is a volume that maintains copies of its data on two partitions. In a mirror, all write operations take place on both partitions, but read operations take place only from one partition. Mirrors tolerate single-disk failures; operation continues on the surviving half of the mirror. A stripe set is a multipartition volume whose data is interleaved between partitions. NT uses a stripe unit of 64KB. The system stores the first 64KB of file-system data on the first partition of the stripe, stores the second 64KB on the second partition, and so on, thus wrapping back to the first partition. Stripe sets can improve performance when the partitions are on different disks because I/O operations can proceed in parallel on different disks. Finally, stripe sets with parity are stripe sets with an extra 64KB block of data for each 64KB stripe spread across the set's partitions. The extra block stores parity information that NT can use to recover the data stored on one of the set's partitions if the disk on which the partition is located fails. Stripe sets with parity are also known as RAID 5 volumes.

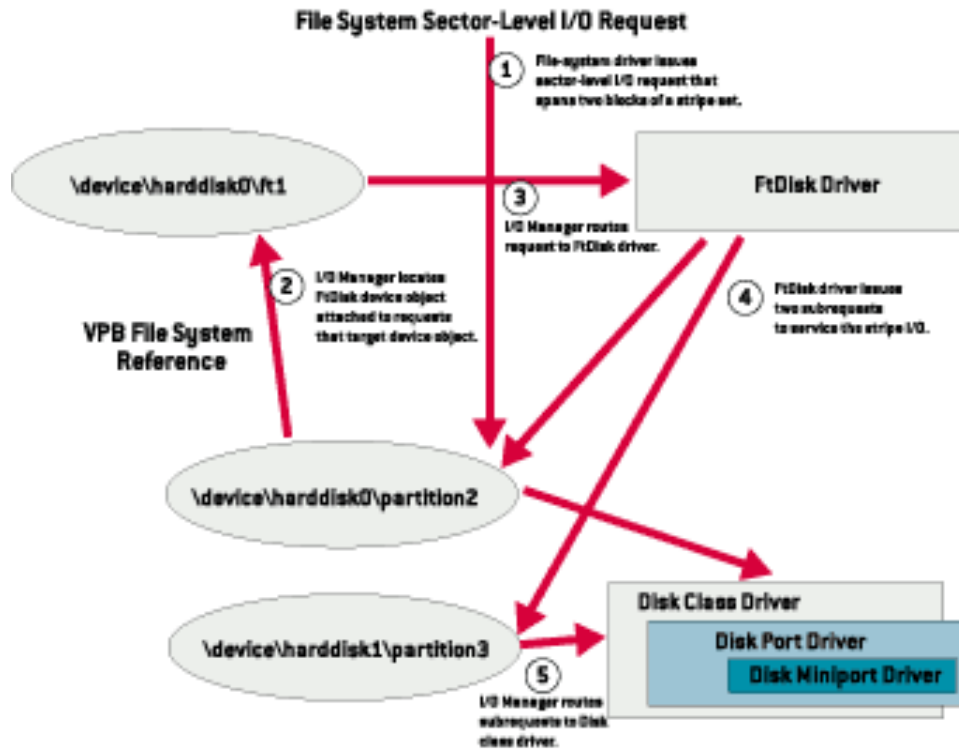
Disk Administrator stores advanced volume-configuration information in the HKEY_LOCAL_MACHINE\SYSTEM\DISK\Information value, with the partition drive-letter and signature information, and the FtDisk driver reads this information during the boot process. A data structure that FtDisk manages in the Information value associates partitions that belong to the same volume. Because file-system drivers expect a volume's contents to reside on one partition, without FtDisk, file-system drivers typically don't recognize a volume that consists of multiple partitions. FtDisk therefore attaches itself to every partition device object in a system to manipulate requests aimed at the device objects that constitute advanced volumes.

Some examples of FtDisk's operations will help clarify its role. If a striped volume consists of \device\harddisk0\partition2 and \device\harddisk1\partition3, as the figure below shows, and an administrator has assigned drive letter D to the stripe, then the I/O Manager defines the link \\??\D: to reference \device\harddisk0\partition2. If FtDisk were not present, an application opening a file on the stripe would receive an error because no file system would understand or mount the partial volume that \device\harddisk0\partition2 represents. With FtDisk present, an FtDisk device object intercepts file-system disk I/O aimed at \device\harddisk0\partition2, and the FtDisk driver adjusts the request

Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

before passing it to the disk class driver. The adjustment FtDisk makes configures the request to refer to the correct offset of the request's target stripe on either `\device\harddisk0\partition2` or `\device\harddisk1\partition3`.



In the case of writes to a mirrored volume, FtDisk splits each request so that each half of the mirror receives the full write operation. For mirrored reads, FtDisk performs a read from half of a mirror, relying on the other half when a read operation fails.

The Dynamics of Win2K

I began this series with a look at Windows NT 4.0's storage management architecture. I reviewed that architecture's reliance on DOS-style partitioning, its storage-driver architecture, and its use of NT device layering to implement enhancements such as software-based fault tolerance and performance monitoring. NT 4.0 storage management has two significant limitations: an upper limit of 26 volumes and a requirement to reboot the system whenever you change advanced storage settings. To be accessible, an NT 4.0 volume needs a drive letter in the A to Z range, which is where the 26-volume limit originates. Whenever you create a volume set, mirrored volume, stripe set, or stripe set with parity, you change the advanced storage configuration, and NT requires you to reboot for the new settings to take effect.

Windows 2000's (Win2K's) storage architecture has changed dramatically from NT 4.0, and the biggest changes are the removal of the two limitations I've just described. Along the way, Win2K has picked up a handful of other new storage-management features, such as Hierarchical Storage Management (HSM) capability. Delving into the details of Win2K's storage architecture will show you how Win2K has changed and improved NT 4.0's storage architecture.

To understand these improvements, you need to be familiar with the concepts I covered in part 1, including device objects and Object Manager symbolic links. I rely on the same terminology I used in

Inside Storage Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

part 1: A disk is a physical storage device that hardware divides into discretely addressable chunks called sectors, a partition is a contiguous group of sectors that the system can assign to volumes, and a volume is a collection of one or more partitions that a file system manages as one object.

Basic Disks vs. Dynamic Disks

Win2K introduces the concept of Basic and Dynamic disks. Basic disks are disks that rely on the DOS-style partitioning scheme that I described in part 1. In a sense, Basic disks are NT legacy disks. Dynamic disks are new to Win2K and implement a partitioning scheme that I describe later in this article. The difference between Basic and Dynamic disks lies in their support for advanced (multipartition) volumes. The Registry stores advanced-volume configuration information for Basic disks; storage of advanced-volume configuration information for Dynamic disks is on-disk. Storing advanced-volume configuration on-disk ties the Dynamic disk to the storage it describes, so losing advanced volume configuration data is harder and moving disks with advanced volumes between systems is easier.

Win2K manages all disks as Basic disks unless you manually create Dynamic disks or convert existing Basic disks (with enough free space) to Dynamic disks. To encourage administrators to use Dynamic disks, Microsoft has imposed some usage limitations on Basic disks. For example, you can create new advanced volumes only on Dynamic disks (if you upgrade an NT 4.0 system, Win2K will support existing advanced volumes). Another limitation is that Win2K lets you dynamically grow NTFS volumes only on Dynamic disks. A disadvantage to Dynamic disks is that the partitioning format they use is proprietary and incompatible with other OSs, including all other versions of Windows. Thus, you can't access Dynamic disks in a dual-boot environment. For several reasons, including the fact that laptop disks typically don't move easily between computers, Win2K uses only Basic disks on laptops.

Basic Disks

Although the way Win2K partitions Basic disks hasn't changed from the way NT 4.0 partitions disks, the way Win2K's device drivers manage Basic disks has changed. As they did in NT 4.0, storage devices in Win2K follow the class-port-miniport model. Also as in NT 4.0, Microsoft supplies disk.sys, a class driver that implements functionality common to disks. Microsoft provides a handful of disk port drivers for Win2K. For example, scsiport.sys is the port driver for disks on SCSI buses, and pciidex.sys is a port driver for IDE-based systems. Win2K ships with several miniport drivers, including one—aha154x.sys—for Adaptec's 1540 family of SCSI controllers. On systems that have at least one ATAPI-based IDE device, one driver—atapi.sys—combines port and miniport functionality. Most Win2K installations include one or more of the drivers I've mentioned.

Before I outline disk management in Win2K, let me review NT 4.0 disk management. In NT 4.0, the Disk class driver creates a device object with a name in the form `\Device\HarddiskX\Partition0` to represent a physical disk; a number that uniquely identifies the disk replaces X. When the class driver detects a disk, the driver uses the I/O Manager function `IoReadPartitionTable` to scan the disk's partition table. For each partition it identifies, the class driver creates a device object under the disk's `PartitionY Harddisk` directory (Y is a number that identifies the partition). The I/O Manager function `IoAssignDriveLetters` creates symbolic links under the Object Manager's `\??` subdirectory for each partition device object, and file systems mount the partition device objects as the system and applications access the partitions.

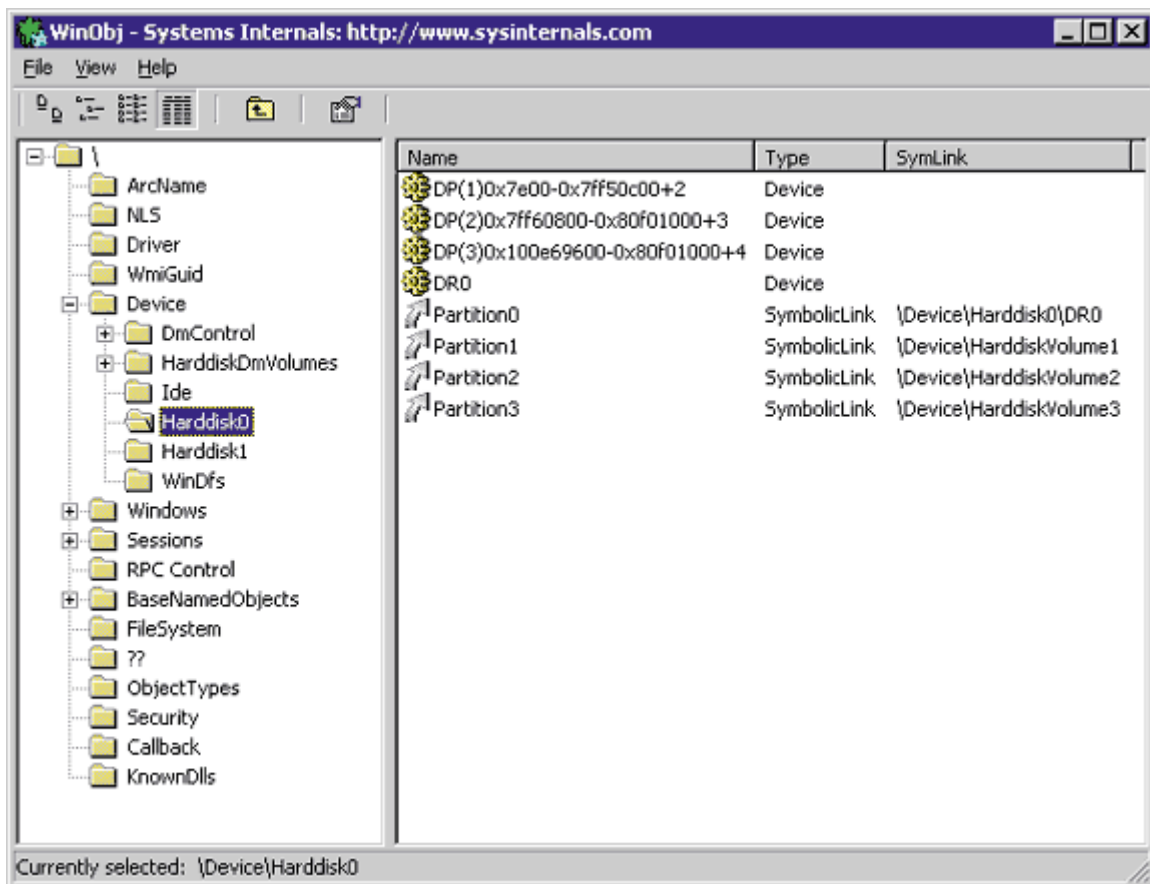
For Basic disks in Win2K, the Disk class driver still creates device objects that represent disks and partitions; however, the objects' naming and role are different than in NT 4.0. Device objects that represent disks have names of the form `\Device\HarddiskX\DRX`; the number that identifies the disk replaces both Xs. The class driver still uses `IoReadPartitionTable` to scan disks, but the partition

Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

device objects have more descriptive names. An example partition object name is `\Device\Harddisk0\DP(1)0x7e00-0x7ff50c00+2`. This name identifies the first partition on the first disk on a system. The two hexadecimal numbers (0x7e00 and 0x7ff50c00) designate the start and length of the partition, and the last number is an internal identifier that the class driver assigns.

To maintain compatibility with applications that are familiar with NT 4.0 naming conventions, the Disk class driver creates symbolic links with NT 4.0-formatted names that refer to the device objects the driver created. For example, the Disk class driver creates the link `\Device\Harddisk0\Partition0` to refer to `\Device\Harddisk0\DR0`, and `\Device\Harddisk0\Partition1` to refer to the first partition device object of the first disk. The class driver also creates the same Win32 symbolic links in Win2K that represent physical drives that it created in NT 4.0. Thus, for example, the link `\??\PhysicalDrive0` references `\Device\Harddisk0\DR0`. The figure below shows the WinObj utility viewing the contents of a Harddisk directory for a Basic disk. (You can download a copy of WinObj from <http://sysinternals.com/winobj.htm>.) You can see the physical disk and partition device objects in the right-hand pane.



In NT 4.0, the partition device objects that the Disk class driver creates have assigned drive letters and are mounted by file systems. Win2K does things differently. In Win2K, the FtDisk driver creates disk device objects that represent volumes on Basic disks. Win2K assigns drive letters to the volumes, which file systems mount. FtDisk is present in NT 4.0 only when you have at least one advanced volume; in Win2K, FtDisk plays an integral role in managing all Basic disk volumes, including volumes that aren't advanced. FtDisk uses the Basic disk configuration information that the Registry value `HKEY_LOCAL_MACHINE\SYSTEM\Disk` stores to determine what Basic volumes, advanced and

Inside Storage Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

otherwise, a system includes. For each volume, FtDisk creates a symbolic link of the form `\Device\HarddiskVolumeX`, where X is a number (starting with 1) that identifies the volume. The link refers to the partition device object that corresponds to the volume or to the first partition device object of a multipartition volume.

An interesting aspect of Win2K's version of FtDisk is that Win2K's FtDisk leverages Win2K's PnP subsystem with the aid of the Partition Manager (`partmgr.sys`) driver to determine what Basic disk partitions exist. Partition Manager registers with the PnP subsystem so that Win2K can inform Partition Manager whenever the Disk class driver creates a partition device object. Partition Manager informs FtDisk about new partition objects through a private interface and creates filter device objects that Partition Manager attaches to the objects. The existence of the filter objects prompts Win2K to inform Partition Manager whenever a partition device object is deleted so that Partition Manager can update FtDisk. The Disk class driver deletes a partition device object when you delete a partition in the Disk Management Microsoft Management Console (MMC) snap-in.

Win2K Basic volume drive-letter assignment, a process I'll describe shortly, creates drive-letter symbolic links under `\??` that point to the volume device objects that FtDisk creates. When the system or an application accesses a volume for the first time, Win2K performs a mount operation that is identical to NT 4.0's mount process. Just as in NT 4.0, FtDisk intercepts I/O request packets (IRPs) that the system aims at multipartition volumes and handles them appropriately. For example, FtDisk splits read requests aimed at mirrored drives and services requests aimed at stripe sets by using derivative IRPs that FtDisk routes to specific partitions of the set. If the system directs an IRP to a nonadvanced volume, FtDisk simply forwards the request to the Disk class driver.

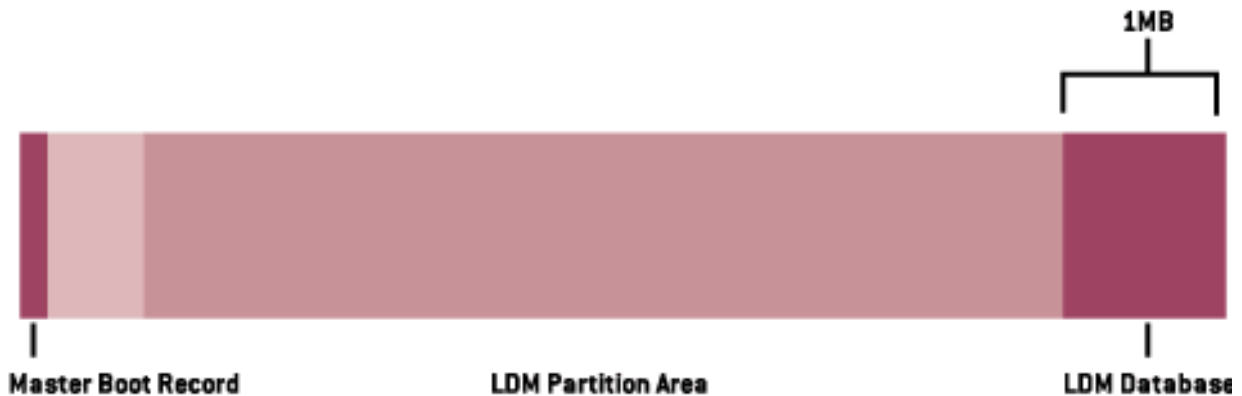
Dynamic Disks

As I've stated, Dynamic disks are Win2K's favored disk format and are necessary for creating new advanced volumes. Win2K's Logical Disk Manager (LDM) subsystem, which consists of user-mode and device-driver components, oversees Dynamic disks. Microsoft licenses LDM from VERITAS Software, which originally developed LDM technology for UNIX systems. Working closely with Microsoft, VERITAS ported its LDM to Win2K to provide Win2K with more robust partitioning and advanced volume capabilities. A major difference between LDM's partitioning and DOS-style partitioning is that LDM maintains one unified database that stores partitioning information for all the Dynamic disks on a system—including advanced volume configuration. LDM's UNIX version incorporates disk groups, in which all the Dynamic disks that the system assigns to a disk group share a common database. VERITAS' commercial volume management software for Win2K also includes disk groups, but Win2K's LDM implementation includes only one disk group.

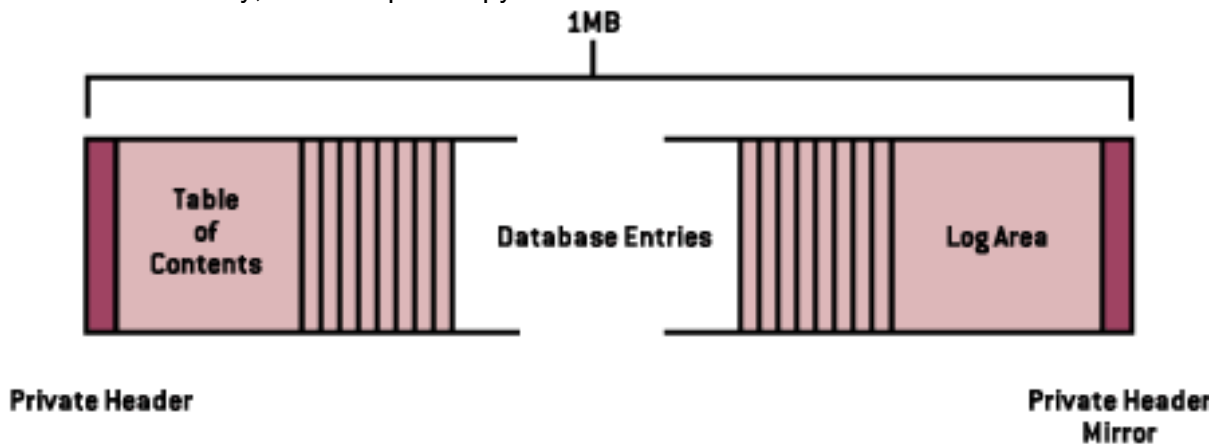
The LDM database resides in a 1MB reserved space at the end of each Dynamic disk. The need for this space is why Win2K requires free space at the end of Basic disks before you can convert them to Dynamic disks. Although Dynamic disks' partitioning data resides in the LDM database, LDM implements a DOS-style partition table so that legacy disk-management utilities, including those that run under Win2K and under other OSs in dual-boot environments, don't mistakenly believe a Dynamic disk is unpartitioned. LDM also creates a DOS-style partition table so that Win2K's boot code can find the system and boot volumes, even if the volumes are on Dynamic disks (NT Loader—NTLDR—for example, knows nothing about LDM partitioning). If a disk contains the system or boot volumes, partitions describe the location of those volumes. Otherwise, one partition begins at the first cylinder of the disk (typically 63 sectors into the disk) and extends to the beginning of the LDM database. In the region this place-holding partition encompasses, LDM creates partitions that the LDM database organizes. The figure below illustrates this Dynamic disk layout.

Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)



The LDM database consists of four regions, which the figure below shows: a header sector that LDM calls the Private Header, a table of contents area, a database records area, and a transactional log area. The Private Header sector resides 1MB before the end of a Dynamic disk and anchors the database. You'll quickly notice as you spend time with Win2K that the OS uses GUIDs to identify just about everything, and disks are no exception. LDM assigns each Dynamic disk a GUID, and the Private Header sector notes the GUID of the Dynamic disk on which the sector resides (hence the Private Header's designation as information that is private to the disk). The Private Header also stores the name of the disk group, which is the name of the computer concatenated with Dg0 (i.e., DesktopDg0 if the computer's name is Desktop), and a pointer to the beginning of the database table of contents. For reliability, LDM keeps a copy of the Private Header in the disk's last sector.



The database table of contents is 16 sectors in size and contains information regarding the database's layout. LDM starts the database record area immediately after the table of contents, with a sector that serves as the database record header. This sector stores information about the database record area, including the number of records it contains, the name and GUID of the disk group the database relates to, and a sequence number identifier that LDM uses for the next entry it creates in the database. Sectors following the database record header contain 128-byte fixed-size records that store entries that describe the disk group's partitions and volumes.

A database entry can be one of four types: partition, disk, component, and volume. LDM uses the database entry types to identify three levels that describe volumes. LDM connects entries with internal


Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

object identifiers. At the lowest level, partition entries describe contiguous regions on a disk; identifiers stored in a partition entry link the entry to a component and disk entry. A disk entry represents a Dynamic disk that is part of the disk group and includes the disk's GUID. A component entry serves as a connector between one or more partition entries and the volume entry the partitions are associated with. A volume entry stores the GUID of the volume, the volume's total size, state, and a drive-letter hint. Disk entries that are larger than a database record span multiple records; partition, component, and volume entries rarely span multiple records.

LDM requires three entries to describe a simple volume: a partition entry, a component entry, and a volume entry. Figure 3 depicts the contents of a simple LDM database that defines one 200MB volume that consists of one partition. The partition entry describes the area on a disk that the system assigned to the volume, the component entry connects the partition entry with the volume entry, and the volume entry contains the GUID that Win2K uses internally to identify the volume. Advanced volumes require more than three entries. For example, a stripe set consists of at least two partition entries, a component entry, and a volume entry. The only volume type that has more than one component entry is a mirror; mirrors have two component entries, each of which represents one-half of the mirror. LDM uses two component entries for mirrors so that when you break a mirror, LDM can split it at the component level, creating two volumes with one component entry each. Because a simple volume requires three entries and the 1MB database contains space for approximately 8000 entries, the effective upper limit on the number of volumes you can create on a Win2K system is approximately 2500.

Disk Entry	Volume Entry	Component Entry	Partition Entry
Name: Disk1	Name: Volume1	Name: Volume1-01	Name: Disk1-01
GUID: XXX-XX...	ID: 0x408	ID: 0x409	ID: 0x407
Disk ID: 0x404	State: ACTIVE	Parent ID: 0x408	Parent ID: 0x409
	Size: 200MB		Disk ID: 0x404
	GUID: XXXX-XXX...		Start: 300MB
	Drive Hint: H:		Size: 200MB

 **FIGURE 3:** LDM Database Describing One 200MB Volume

The final area of the LDM database is the transactional log area, which consists of a few sectors for storing backup database information as the information is modified. This setup safeguards the database in case of a crash or power failure because LDM can use the log to bring the database back to a consistent state.

Dynamic Disk Management

The MMC plugin DLL `\winnt\system32\dmconfig.dll` (DMConfig) creates and changes the contents of the LDM database. When you launch the Disk Management MMC snap-in, DMConfig loads into memory and reads the LDM database from each of the disks. If DMConfig detects a database from another computer's disk group, it notes that the volumes on the disk are foreign and lets you import them into the current computer's database if you want to use them. As you change the configuration of Dynamic disks, DMConfig updates an in-memory copy of the database that it passes to DMIO, the `dmio.sys` device driver. DMIO is the Dynamic disk equivalent of FtDisk, so it controls access to the on-disk database and creates device objects that represent the volumes on Dynamic disks.

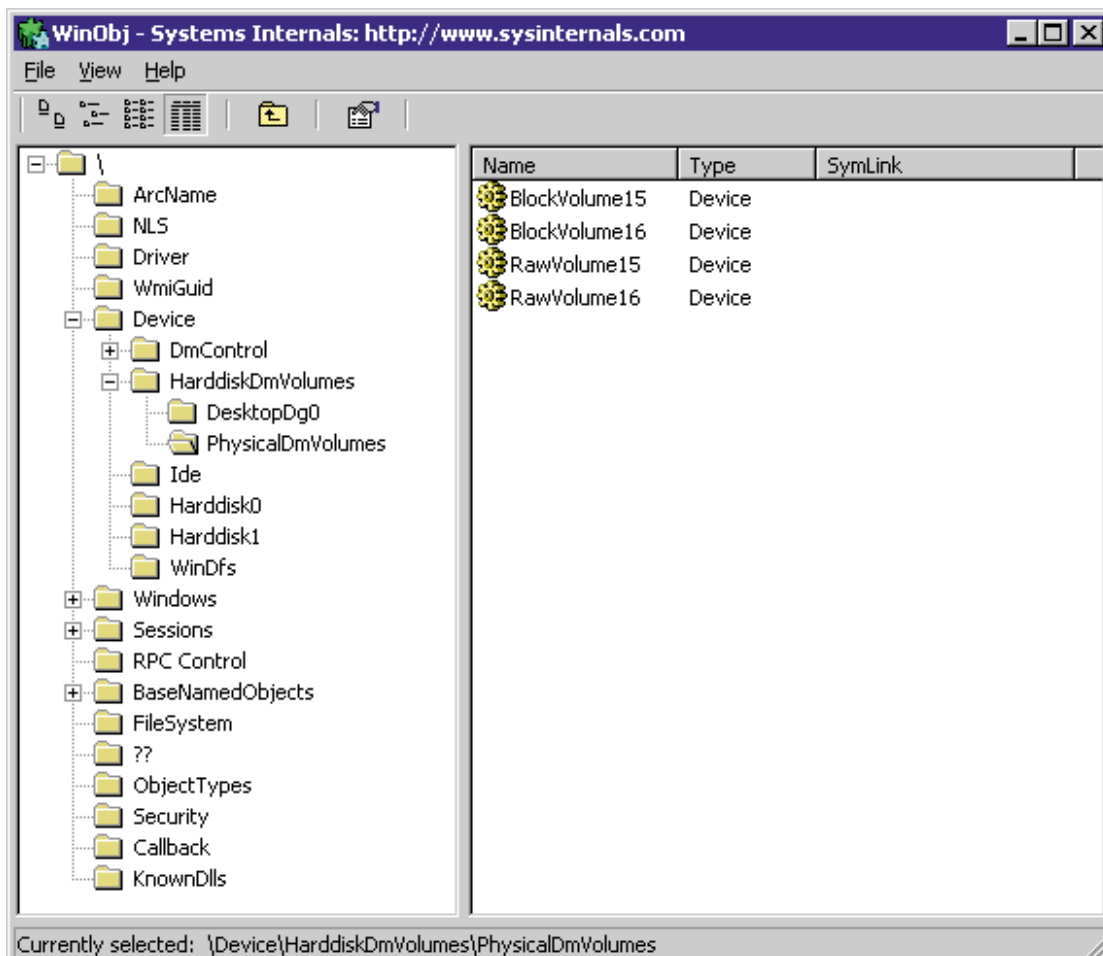
DMIO doesn't know how to interpret the database it oversees. DMConfig and another device driver, `dmboot.sys` (DMBoot), are responsible for interpreting the database. DMConfig knows how to read

Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

and how to update the database; DMBoot knows only how to read the database. DMBoot loads during the boot process if another LDM driver, dmload.sys (DMLoad), determines that at least one Dynamic disk is present on the system. DMLoad makes this determination by asking DMIO, and if at least one Dynamic disk is present, DMLoad starts DMBoot, which scans the LDM database. DMBoot informs DMIO of the composition of each volume it encounters so that DMIO can create device objects to represent the volumes. DMBoot unloads from memory immediately after it finishes its scan. Because DMIO has no database interpretation logic, it is relatively small. Its small size is beneficial because DMIO is always loaded.

DMIO creates a device object for each Dynamic disk volume with a name in the form `\Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolumeX`, where X is an identifier that DMIO assigns to the volume. In addition, DMIO creates another device object that represents raw (unstructured) I/O to a volume named `\Device\HarddiskDmVolumes\PhysicalDmVolumes\RawVolumeX`. The figure below shows the device objects that DMIO created on a system that consists of two Dynamic disk volumes. DMIO also creates numerous symbolic links in the Object Manager namespace for each volume, starting with one link in the form `\Device\HarddiskDmVolumes\ComputerNameDg0\VolumeY` for each volume. DMIO replaces ComputerName with the name of the computer and replaces Y with a volume identifier (different from the internal identifier that DMIO assigns to the device objects). These links refer to the block-device objects under the PhysicalDmVolumes directory.



Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

DMIO's IRP management is virtually identical to FtDisk's because DMIO keeps track of the partitions that constitute a volume and the type of volume a device object represents. However, because the Disk class driver is aware only of DOS-style partitions, DMIO must perform the simple partition-to-disk translations that FtDisk lets the Disk class driver handle. When the system or an application performs a file-system-related operation for a Dynamic volume, the file-system driver managing the data on the volume forwards an IRP to the DMIO device object that represents the volume. If the IRP's target is a simple volume, DMIO adjusts the IRP's partition-relative offset to a disk-relative offset and hands the IRP to the Disk class driver. However, if the IRP is aimed at an advanced volume, DMIO might create additional IRPs or perform more complex offset and length adjustments. For example, if DMIO receives an IRP that designates a write operation to a mirrored volume's device object, DMIO sends an IRP to the Disk class driver's physical disk device objects for the disks on which both halves of the mirror reside.

One type of advanced volume that is present in Win2K is a volume set, which is similar to NT 4.0's volume set but can be extended without rebooting. New support in NTFS for extending the size of a volume, including resizing metadata files, makes this extension possible. DMIO also fully supports the creation of advanced volumes such as mirrors and stripe sets without requiring a reboot.

Reparse Points

I've highlighted Win2K's freedom from drive letters as one of the compelling features of the OS's new storage management architecture. A new mechanism, mount points, lets you link volumes through directories on NTFS volumes, which makes volumes with no drive-letter assignment accessible. For example, an NTFS directory that you've named C:\Projects could mount another volume (NTFS or FAT) that contains your project directories and files. If your project volume had a file you named \CurrentProject\Description.txt, you could access the file through the path C:\Projects\CurrentProject\Description.txt. What makes mount points possible is reparse point technology.

A reparse point is a block of arbitrary data with some fixed header data that Win2K associates with an NTFS file or directory. An application or the system defines the format and behavior of reparse points, including the value of the unique reparse point tag that identifies the application's or system's reparse points and the size and meaning of the data portion of a reparse point (the data portion can be as large as 16KB). Reparse points store their unique tag in a fixed segment. Any application that implements a reparse point must supply a file-system filter driver to watch for reparse-related return codes for file operations that execute on NTFS volumes, and the driver must take appropriate action when it detects the codes. NTFS returns a reparse status code whenever it processes a file operation and encounters a file or directory with an associated reparse point.

The Win2K NTFS file-system driver, the I/O Manager, and the Object Manager all partly implement reparse point functionality. The Object Manager initiates pathname-parsing operations by using the I/O Manager to interface with file-system drivers. Therefore, the Object Manager must retry operations for which the I/O Manager returns a reparse status code. The I/O Manager implements pathname modification that mount points and other reparse points might require, and the NTFS file-system driver must associate and identify reparse point data with files and directories. You can therefore think of the I/O Manager as the reparse point file-system filter driver for many Microsoft-defined reparse points.

An example of a reparse point application is an HSM system that uses reparse points to designate files that an administrator moves to offline tape storage. When a user tries to access an offline file, the HSM filter driver detects the reparse status code that NTFS returns, communicates with a user-mode service to retrieve the file from offline storage, deletes the reparse point from the file, and lets the file

Inside Storage Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

operation retry after the service retrieves the file. This process describes exactly how Win2K's Remote Storage Manager (RSM) filter driver, rsfilter.sys, uses reparse points.

If the I/O Manager receives a reparse status code from NTFS and the file or directory for which NTFS returned the code isn't associated with one of a handful of built-in Win2K reparse points, then no filter driver claimed the reparse point. The I/O Manager then returns an error to the Object Manager that propagates as a File cannot be accessed by the system error to the application making the file or directory access.

Junctions and Mount Points

Microsoft decided not to implement a symbolic link feature for files in NT because many Windows programs won't behave properly when using such a feature. For example, when deleting a file that is a symbolic link, a Windows program would inadvertently delete the target of the link, rather than the link itself.

Virtually every UNIX OS uses symbolic links, in which accessing a file or directory symbolic link resolves to another file or directory. NT has always supported symbolic links in the Registry, and NT makes extensive use of symbolic links in the Object Manager's namespace. However, until Win2K, no NT file system has supported symbolic links. Win2K introduces directory symbolic links, which Microsoft calls NTFS junctions. A junction is a Microsoft-defined reparse point that you can associate with an empty NTFS directory. The data that the reparse point stores is the name of another directory somewhere on the system. When you access a path that crosses a junction, NTFS returns a reparse status code to the I/O Manager for the directory associated with the junction, and the I/O Manager recognizes the reparse point as a junction. The I/O Manager retrieves the directory name that the junction's reparse data specifies and invokes an internal function, `IoDoNameTransmogrify`. This function alters the pathname that the original request specified and returns a reparse status code to the Object Manager. Upon seeing the reparse status code, the Object Manager reissues the request with the redirected pathname, and NTFS performs the new lookup.

Win2K doesn't include any tools for making junctions. You can use `linkd`, a Windows 2000 Resource Kit program, to create junctions, or you can download `Junction`, a `linkd` clone that I wrote, from <http://sysinternals.com/misc.htm>.

Mount points are similar to junctions—they even share the same reparse tag—but the data that mount points store is a volume name (i.e., `\??\Volume{X}`) instead of a directory. When you use the Disk Manager MMC snap-in to assign or remove path assignments for volumes, you're creating mount points. You can also use the built-in command-line tool `mountvol` to create and display mount points.

The Mount Manager

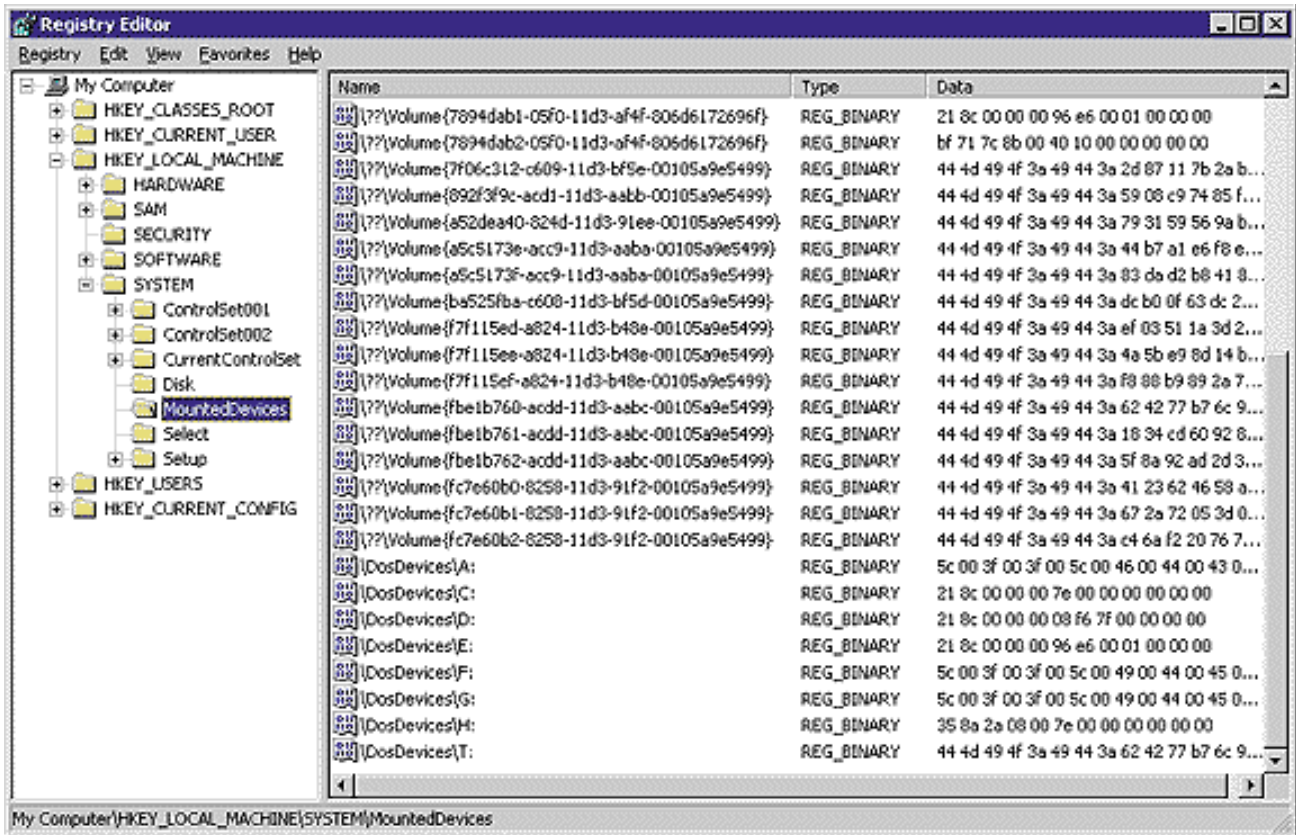
Drive-letter assignment is another aspect of storage management that changed in Win2K. NT 4.0 stores drive-letter assignments in `HKEY_LOCAL_MACHINE\SYSTEM\Disk`, and the NT I/O Manager executes the `IoAssignDriveLetters` function during the boot. `IoAssignDriveLetters` initiates an assignment process that creates drive-letter symbolic links in the `\??` Object Manager directory and honors any assignments you've made from Disk Administrator.

`IoAssignDriveLetters` in Win2K works much as it does in NT 4.0, but the function assigns drive letters only for volumes on Basic disks because only those volumes rely on the DOS-style partitioning that NT 4.0 uses. A new driver in Win2K, the Mount Manager (`mountmgr.sys`), assigns drive letters for Dynamic disk volumes and for Basic disk volumes you create after the system has started. Win2K stores all drive-letter assignments under `HKEY_LOCAL_MACHINE\SYSTEM\MountedDevices`. If you look under that key, you'll see values with names such as `\??\Volume{X}` (where X is a GUID) and

Inside Storage Management

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

values such as \\?\C:. Every volume has a volume name entry, but a volume need not have an assigned drive letter. The Figure below shows the contents of an example Mount Manager Registry key.



The data that the Registry stores in values for Basic disk volume drive letters and volume names is the NT 4.0-style disk signature and the starting offset of the first partition associated with the volume. The data that the Registry stores in values for Dynamic disk volumes includes the volume's DMIO internal GUID. When the Mount Manager initializes during the boot process, it registers with the Win2K PnP subsystem so that it receives notification whenever either FtDisk or DMIO creates a volume. When the Mount Manager receives such a notification, it determines the new volume's GUID or disk signature, then asks either FtDisk or DMIO (whichever created the volume) for a suggested drive-letter assignment. FtDisk queries the NT 4.0 HKEY_LOCAL_MACHINE\SYSTEM\Disk key (in case the system is an NT 4.0 upgrade that had previous drive-letter assignments), and DMIO looks at the drive-letter hint in the volume's database entry. If no suggested drive-letter assignment exists for the volume, the Mount Manager uses the volume GUID or signature as a guide and looks in its internal database, which reflects the contents of the Registry key. Then, the Mount Manager determines whether its internal database contains the drive-letter assignment. If not, the Mount Manager uses the first unassigned drive letter (if one exists), defines a new assignment, creates a symbolic link for the assignment (e.g., \\?\D), and updates the MountedDevices Registry key. At the same time, the Mount Manager creates a volume symbolic link (i.e., \\?\Volume{X}) that defines a new volume GUID, if the volume doesn't already have one. This GUID is different from the volume GUIDs that DMIO uses internally.

The Mount Manager also maintains the Mount Manager Remote Database on every NTFS volume, in which the Mount Manager records any mount points defined for that volume. The database

Inside Storage Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

file, :\$MountMgrRemoteDatabase, resides in the NTFS root directory. Mount points move when a disk moves from one system to another and in dual-boot environments (i.e., when booting between multiple Win2K installations) because of the Mount Manager Remote Database's existence. NTFS also keeps track of mount points in the NTFS metadata file \Extend\$Reparse. NTFS stores mount-point information in the metadata file so that Win2K can easily enumerate the mount points defined for a volume when a Win32 application, such as the Disk Manager, requests mount-point definitions.

Dynamic Disks

Mount points, HSM, on-disk storage of disk configuration, and junctions are powerful features that make Win2K a compelling upgrade from NT 4.0. The LDM and Dynamic disks, with their ability to support the creation of advanced volumes and dynamic growth of existing volumes without reboots, bring Win2K on par with advanced UNIX systems for enterprise storage management.