The Structure of NTFS – Windows NT's File System

CS-384 Design of Operating Systems

Submitted by: Nhat Nguyen
Submitted to: Dr. Barnicki

Table of Contents

The Need for NTFS

In 1998, Microsoft was supporting two different file systems—the FAT file system for MS-DOS and MS Windows as well as the HPFS (high-performance file system) for IBM's OS/2. When designing the basis for Windows NT, their corporate-level operating system, neither of those file systems proved suitable to handle the requirements demanded by Windows NT. The shortcoming of the FAT file system was that it was originally designed for floppy disks, so that as hard drives grew larger than 2GB, FAT could not support them. On the other hand, HPFS addressed the expanding nature of disk drives and greatly improved file access times for large directories and could be used on hard disks up to 4GB in size and later on it was expanded to support disk sizes up to 2TB. Still, neither file system was suitable for mission-critical applications that required *recoverability, fault tolerance and data redundancy, and file security*. Instead, the design group for NT decided on constructing a new file system, NTFS (NT's file system).

Since Windows NT was targeting businesses and corporations, the reliability of the data stored on the system became more of a priority that speed as in the case of home computer users. In a corporate environment, if a system fails and data is lost, speed becomes irrelevant. To support *recoverability*, the new file system, NTFS, provided file system recovery based upon a transaction-processing model as well as an improved write-caching feature.

To support fault tolerance and data redundancy, various kinds of disk volumes and volume sets were implemented, ranging from RAID (redundant array of independent disks) Level 0, 1, 5 to sector sparing. These features focused on either the reliability of the data, improving disk I/O, or having multiple copies of the same data so that the data is always preserved (data redundancy).

For file security, NTFS used the security model that protected everything else that comes in contact with Windows NT, such as devices, named and anonymous pipes, processes, threads, events, mutexes, semaphores, waitable timers, access tokens, network shares, services, and so on. This security could be implemented for a file by establishing a security descriptor as a file record attribute in the MFT—master file table.

NTFS Disk Structure

Windows NT manages the storage space on secondary storage media such as hard drives through *volumes*. Usually when a hard drive is formatted so that data can be placed and received on it, a logical partition is created. Volumes correspond to these logical partitions on a one-to-one relationship or a one-to-many relationship (in this case, the volume is called a volume set). Essentially, a volume is series of files plus any unused unallocated space remaining on the disk partition or partitions (as in the case of a volume set).

Within a volume exist clusters. Clusters are the fundamental unit of disk allocation that Windows NT uses. In turn, within a cluster exist a number of sectors. The number of sectors in a cluster is always a power of two (e.g. 2, 4, 8, 16, 32, etc. sectors in a cluster)

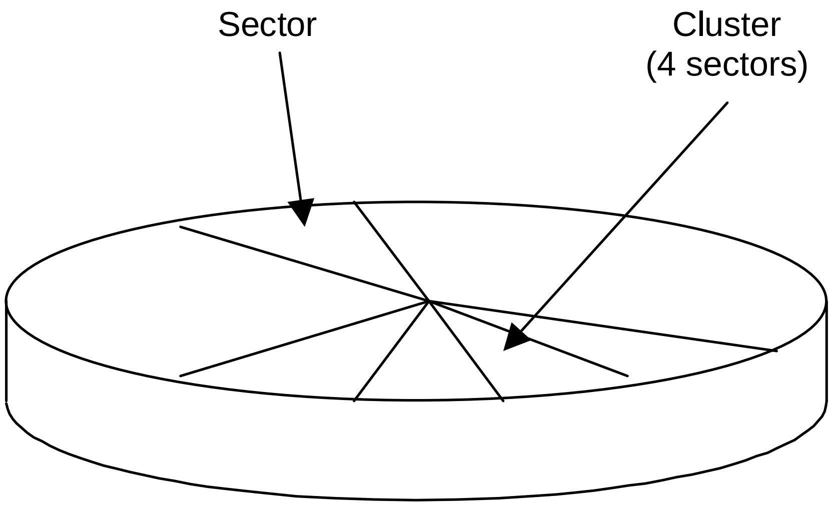

Figure 1: Sectors and a Cluster on Disk

The cluster size, or *cluster factor*, varies with the volume size. By varying the cluster factor, NTFS is able to support very large hard disks efficiently. For example, a small cluster factor would reduce the amount of unused space within a cluster, but increase the amount of file fragmentation (i.e. clusters containing a file exist on noncontiguous parts of the disk). A large cluster factor would reduce the amount of fragmentation, but increase the amount of unused space within each cluster. To access parts of the disk, NTFS uses *logical cluster numbers*, *LCNs*, as disk addresses. LCNs are simply the numbering of all clusters of a volume from beginning to end. With this schema, a physical disk address can be calculated by multiplying the LCN by the cluster factor to get the physical byte offset on the volume (Silberschatz and Galvin, 766). In regard to files, NTFS uses *VCNs—virtual cluster numbers*—to refer to the data within a file. VCNs number the clusters belonging to a particular file from 0 through n - 1. VCNs are not necessarily physically contiguous (especially when the file is fragmented), but they can be mapped to any number of LCNs on the NT volume (Solomon, 407).

The *MFT (Master File Table)* is the vital core of the NTFS structure. All data stored on a volume is contained in the MFT. By storing all of this information within a file, NTFS can easily locate and maintain the data, and a security descriptor, used by NT's security model, can protect each separate file. The MFT is essentially an array of records with each record holding data about a particular file in the volume. In addition, it also includes a file record for itself so that the MFT can be rebuilt in case it becomes damaged. The MFT also includes file records for the NTFS metadata files that help implement the file system structure—i.e. data structures used to locate and retrieve files, the bootstrap data, and the bitmap that records the allocation state of the entire volume. Each of these NTFS metadata files has a name beginning with a dollar sign ($) to differentiate them from other system files and user files:

- ***$MFT*** – the Master File Table
- ***$MFTMirr*** – contains a copy of the first few rows of the MFT, used to locate metadata files in case the MFT file is corrupt for some reason
- ***$LogFile*** – used to record all disk operations that affect the NTFS volume structure such as file creations, file copying, file deletion, etc. After a system failure, NTFS uses the log file to recover the NTFS volume.
- ***$Bitmap*** – records the allocation state of the NTFS volume. Each bit in the bitmap represents a cluster on the volume, identifying whether the cluster is free or has been allocated to a file.
- ***$Boot*** – stores the Windows NT bootstrap code.
- ***$BadClus*** – records any bad clusters on the volume so that Windows NT will not write to that disk region in the future.
- ***$Volume*** – records the NT volume name, version of NTFS on which the volume was written, and a bit that signifies whether or not a disk corruption has occurred.
- ***$AttrDef*** – *attribute definition table* – defines attribute types supported on the volume and indicates whether they can be indexed, recovered during a system recovery operation, etc.

For NTFS to reference these file records, each file record has a unique 64-bit ID value called the *file reference*. This 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, because the first file record is referenced as File Number 0 (the MFT itself). The file sequence number enables NTFS to perform internal consistency checks, since it is incremented every time an MFT file record position is reused.

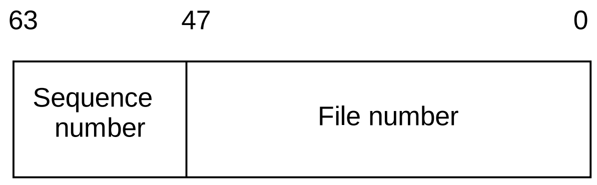

Figure 2: File Reference

Each row in the MFT's array is a file record, which is a collection of attribute/value pairs that refer to a specific file. File records store such attributes as the filename, time stamp, security descriptor, the *unnamed data attribute*—holds the actual data in the file, and so on. Small attributes that can fit in the MFT itself are called *resident* attributes while large attributes that cannot fit in the MFT are called *nonresident* attributes. In the case of nonresident attributes, a pointer is stored in that file record. The pointer points to a 2 or 4 KB area of the disk, called a *run* or *extent*, where the value of that particular attribute is stored (Silberschatz and Galvin, 416).

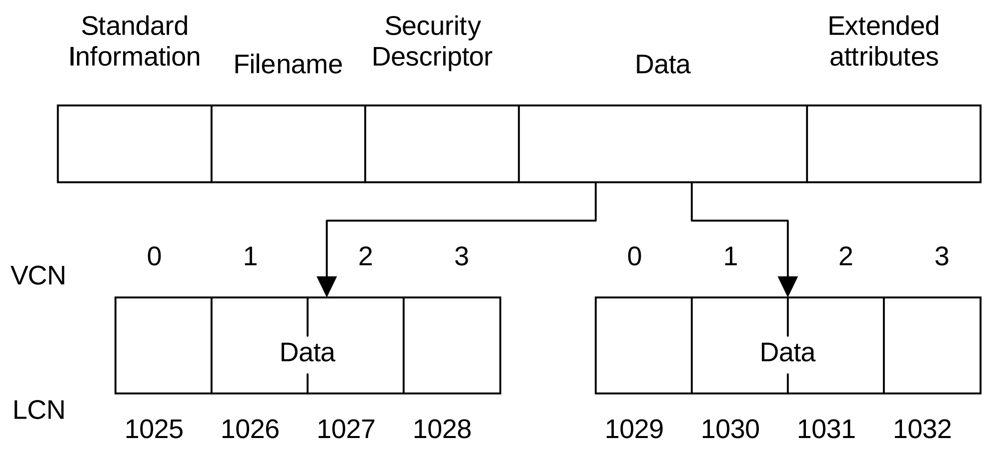

Figure 3: MFT File Record with two Data Runs

When a file's attributes can't fit in an MFT file record and separate disk runs (extents) are required, NTFS keeps track of the runs through use of VCNs. The VCNs reference the clusters constituting the particular file to the LCN numbering system for an NT disk volume.

Each attribute in a file record has an optional name and a value; NTFS identifies an attribute by its name in uppercase preceded by a dollar sign (e.g. $FILENAME or $DATA). However, the attribute names are just labels that correspond to numeric type codes that NTFS uses to order the attributes within a file record. The value of each attribute is the byte stream that makes up the attribute.

For the most part, files only need to be referenced by a single file record (with disk runs if the data attribute is large). Sometimes, though, a file has too many attributes to fit in the MFT file record, so a second MFT file record is used to contain the additional attributes. In this situation, an attribute called the *attribute list* is added to the second file record. This attribute list attribute stores 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. This attribute list attribute is necessary for the instances in which a file grows so large or so fragmented that a single file record cannot contain the collection of VCN-to-LCN mappings required to locate all of its disk runs.

Along with file records stored in the MFT, directory records also are store in the MFT. A file directory is just an index of filenames with their file references organized in a specific fashion for quick and easy access. NTFS creates directories by indexing the filename attributes of the files in the directory.

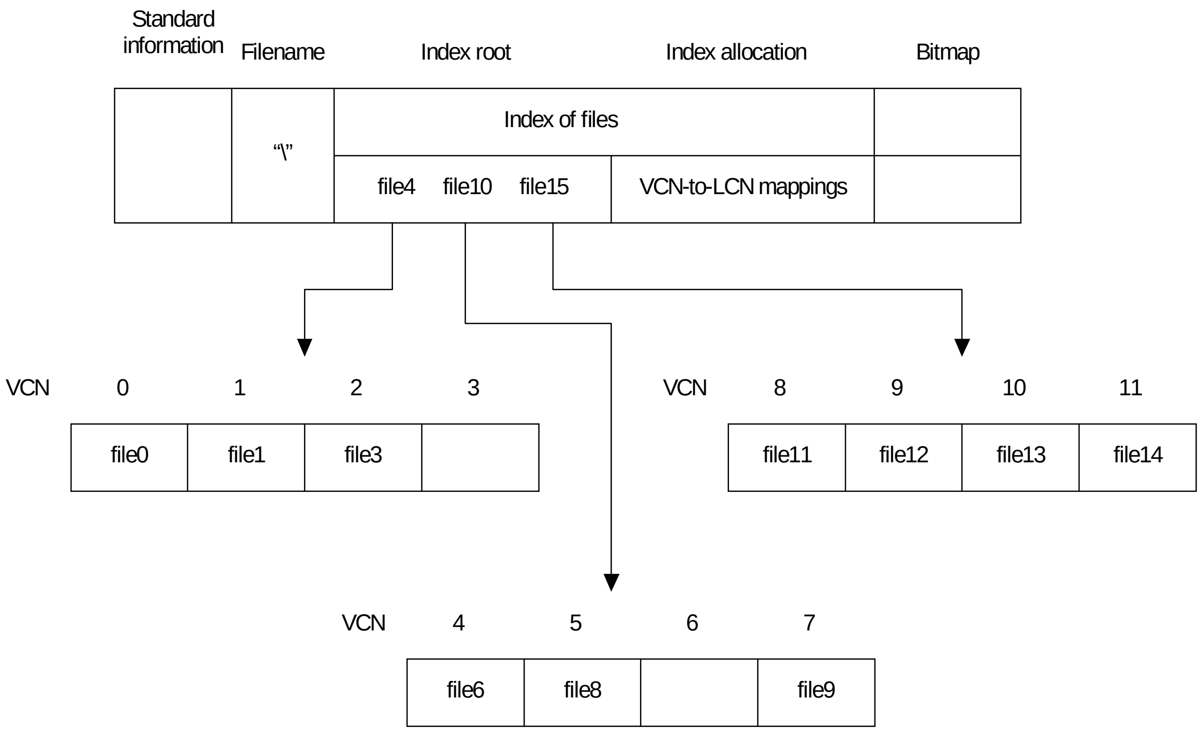

Figure 4: Filename Index for a Root Directory

Each directory record in the MFT contains an index root attribute that in turn stores a sorted list of the files in the directory. For large directories, the filenames are actually stored in 4KB fixed-size index buffers that contain and organize the filenames. These index buffers use a *b+ tree* data structure that minimizes the number of disk accesses required to find a particular file. The index root attribute stores the first level (root subdirectories) of the b+ tree and points to index buffers containing the next level. The index allocation attribute maps the VCNs of the index buffer disk runs to the LCNs that show where the index buffers are located on the disk. The bitmap attribute keeps track of which VCNs in the index buffers are in use and which are free.

The b+ tree data structure is a kind of balanced tree that is efficient at organizing sorted data to minimize disk accesses to search for a file. A directory's index root attribute contains several filenames that behave 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. For example, in the previous figure, *file4* is a first-level entry in the b+ tree, and it points to an index buffer containing filenames that are lexicographically less that itself, *file0*, *file1*, and *file3*. Storing filenames in b+ trees provides three major benefits: faster directory lookups since the filenames are stored in a sorted order; when higher-level software enumerates the files in a directory, NTFS returns names that have already been sorted; because b+ trees have a tendency to grow wide rather than deep, NTFS's fast lookup times do not degenerate as directories get large (Solomon, 405-11). Knowing about NTFS's disk structure allows for a better appreciation of how its mission-critical features operate.

Recoverability

When a disastrous system failure or power failure occurs, NTFS's recovery support makes sure no file system operations will be left incomplete and the volume structure will keep integrity without the need to run a disk repair utility. To implement this ability, NTFS uses transactions to keep track of modified system data. Transactions are a collection of file system operations that are executed *atomically*—where all the file system operations must successfully execute, otherwise, the modified data is *rolled back* to its previous state. Transactions work by writing redo and undo information to a log file before any data is modified. After the data is modified, a commit record is written to the log file to indicate that the transaction succeeded. If the transaction cannot be completed for some reason (system crash, premature interruption) Windows NT can restore the file system data to a consistent state by processing the log records. This entails first redoing the operations for committed transactions, then undoing the operations for transactions that did not commit successfully before the crash. The log file that these transactions use may seem to grow without bound considering the large number of file system operations that occur during a typical user session on the computer (e.g. opening applications, changing resolutions, installing software, and so on), but this is not the case. Periodically (typically every 5 seconds), a checkpoint record is written to the log file. The purpose of the checkpoint record is to mark where records existing before the checkpoint record do not have to be redone or undone for Windows NT to recover from a crash. Therefore, these unneeded records can be discarded to limit the size of the log file. This seems like enormous overhead to maintain volume integrity, but the transaction-logging system applies only to file system data (NTFS's metadata files) in which case transaction execution and recovery are fairly quick (Silberschatz and Galvin, 768). With this said, user file data such as MS Word documents, assembly code files, and JPEG-format picture files are not guaranteed to be in a stable condition after a major system crash. This decision not to implement user data recovery in the file system represents a trade-off between a fully fault tolerant file system and one that provides optimum performance for all file operations.

An exploration into the evolution of file system design will give insight into how NTFS further implements recoverability for the Windows NT operating system. In the past, two techniques were common for constructing a file system's input/output and caching support: *careful write* and *lazy write*. Each technique has its own trade-offs between safety and performance.

When an operating system loses power or crashes, pending I/O operations are often prematurely interrupted. Depending on what I/O operation(s) were in progress at the time of the crash or loss of power, such an abrupt halt can produce inconsistencies in a file system. An inconsistency is some sort of corruption within the file system. For example, a filename may appear in a directory listing but is nonexistent as far as the file system is concerned.

A careful write file system does not try to prevent file system inconsistencies, but rather it orders its write operations so that the worst thing that can happen is a system crash will produce predictable, noncritical inconsistencies, that the file system can fix later without consequence. When the careful write file system receives a request to update the disk, it must perform several suboperations before the update will be complete. These suboperations are always serially written (written one at a time) to disk. For example, when allocating disk space for a file, the careful write file system first sets some bits in its bitmap and then allocates the space to the file. If the power to the computer is cut off just after the bits are set, the careful write file system loses access to some disk space (the space represented by the bitmap bits), but existing data is not corrupted. Through this serialization of write operations, I/O requests are filled in the order in which they are received. For example, if one process allocates disk space and then another process creates a file, the careful write file system completes the disk allocation before it starts to create the file, which is preferable to creating the file, *and* then the allocation is processed.

The major advantage of a careful write file system is that in the situation of a failure, the disk volume stays consistent and usable without the need to immediately run a slow volume repair utility. The repair utility is absolutely necessary to correct the nondestructive disk inconsistencies that result from premature interruption, but the utility can be executed at a convenient time since the file system is in no danger of losing integrity.

The careful write file system compromises speed for the safety it provides because it processes one I/O operation at a time. A lazy write improves performance by using a write-back caching method: it writes file modifications to the cache and flushes the contents of the cache to disk in an optimized way, usually as a background activity. The performance improvements provided by the lazy write caching technique involve three features. First, the overall number of disk writes is reduced. The contents of a buffer can be modified several times before they are written to disk, since serialized, immediate disk writes are not required. Second, the speed of servicing application requests is greatly increased because the file system can return control to the caller without waiting for disk writes to be completed. Finally, the lazy write strategy ignores the inconsistent intermediate states on a disk volume that can result when the suboperations of two or more I/O requests are interleaved. This makes it easier for the file system to be multithreaded—allowing more than one I/O operation to be in progress at a time.

The disadvantage of the lazy-write method is that it creates intervals during which a disk volume is in too inconsistent a state to be corrected by the file system. As a result, lazy write file systems must keep track of the volume’s state at all times. Lazy write file systems have a performance advantage over careful write systems, at the expense of greater risk and user inconvenience if the system fails.

NTFS uses a recoverable file system which is essentially a mixture of both careful write and lazy write techniques: it tries to exceed the safety of a careful write file system while achieving the performance of a lazy write file system. It uses the transaction-logging technique discussed earlier to ensure volume consistency, and since all of the file system's disk write have been logged to a file, the recovery procedure takes a few seconds, regardless of the size of the volume. This recoverable file system does have some drawbacks for the safety it provides. Every transaction that alters the volume structure needs a record to be written to the log file for each of the transaction's suboperations. This logging overhead is compounded by the file system's batching of log records, where many records are written the log file in a single I/O operation. On the positive side, it can increase the interval length between cache flushes because the file system can be recovered if the system crashes before the cache changes have been flushed to disk. This advantage over the caching performance of lazy write file systems exceeds the overhead of the recoverable file system's logging activity (Solomon, 426-29). With the transaction-based recovery system and the intuition of the write caching technique, NTFS can protect itself from most system failures, but it still offers no support for the recovery of user files.

Fault Tolerance/Data Redundancy

In addition to the recoverability features of NTFS, another method NTFS uses to protect the integrity and increase the performance of the disk volume is fault tolerance and data redundancy. Fault tolerance a is a technique of managing the disk volumes and physical hard drives to improve performance, capacity, or reliability. Data redundancy is a technique of keeping multiple copies of data so that if one copy becomes corrupt, the data is still preserved in the other copy. These two schemes allow the protection of all files within a disk volume, so that NTFS has the capability to protect even user files which was discarded in the transaction-based recovery system because of the high overhead. FtDisk is the fault tolerant driver that allows different configurations of NT's volumes. There are several schemes for NTFS volumes that improve performance or reliability.

A volume set is essentially a single logical volume composed of a maximum of 32 areas of free space on one or more disks. Volume sets are useful for consolidating small areas of free space to create a larger volume or for creating a single, large volume out of two or more small disks. An NTFS volume has the capability of extending additional free areas or additional disks without affecting the existing data on the volume. It can do this because the bitmap that records the allocation status of the volume is just another file—the bitmap file.

A stripe set, also known as RAID Level 0 (redundant array of independent disks), is a series of partitions, one partition per disk, that NT's Disk Administrator combines into a single logical volume. NTFS distributes files in a striped set by using a round-robin manner where data is placed in each of the partitions in 64KB stripes at a time. The advantage of stripe sets is that the data is distributed fairly evenly among the disk partitions which in turn increases the probability that multiple read and write operations will be bound for different disks. And because data on all three disks can be accessed simultaneously, latency time for disk input/output is reduced.

Volume sets make managing disk volumes more convenient, and stripe sets spread the disk I/O load over multiple disks, yet neither provide the ability to recover data if a disk fails. For data recovery, FtDisk implements three redundant storage schemes: mirror sets, stripe sets with parity, and sector sparing.

In the mirror set (RAID Level 1) scheme, data is duplicated in two partitions. If the first disk or any data stored on it becomes unreadable because of a hardware or software failure, FtDisk automatically accesses the data from the mirror partition. Mirror sets ca help in I/O throughput on heavily loaded systems. For example, when I/O activity is high, FtDisk balances its read operations between the primary and mirror partition. Two read operations can proceed simultaneously and thus theoretically finish in half the time. When a file is modified, both partitions of the mirror set must be written, but disk writes are completed asynchronously (does not half to be at the same time), so the performance of user programs is generally not affected by the extra disk update.

The second fault tolerant scheme, stripe sets with parity, is similar to the stripe set discussed earlier except that fault tolerance is achieved by reserving about one disk for storing parity for each stripe. The parity stripe contains a byte-for-byte logical sum (XOR) of the other stripes in the other partitions with the same stripe number. For example, on a three-disk stripe set with parity, stripe 1 on disk 1 would contain the parity information for stripe 1 of disks 2 and 3, stripe 2 of disk 2 would contain parity information for stripe 2 of disks 1 and 3, and so on. Recovering from a failed disk in this set up relies on an arithmetic principle: in an equation with *n* variables, if you know the value of *n – 1* of the variables, you can determine the value of the missing variable by subtraction. For example, in the equation $x + y = z$, where z represents the parity stripe, FtDisk computes $z–y$ to determine the contents of x; to find y, it computes $z – x$. FtDisk uses similar logic to recover lost data. If a disk in a stripe set with parity fails or if data on one disk becomes unreadable, FtDisk reconstructs the missing data by using the XOR operation.

In sector sparing, FtDisk uses its redundant data storage to dynamically replace lost data when a disk sector becomes unreadable. The sector sparing technique exploits a feature of some hard disks, which provide a set of physical sectors reserved as “spare.” If FtDisk receives a data error from the hard disk, it obtains a spare sector from the disk driver to replace the bad sector that caused the data error. FtDisk recovers the data that was on the bad sector (by either reading the data from a disk mirror or recalculating the data from a stripe set with parity) and copies it to the spare sector.

If a bad-sector error occurs and the hard disk does not provide spares, runs out of them, or is a nonSCSI-based disk, FtDisk can still recover the data. I recalculates the unreadable data by accessing a stripe set with parity, or it reads a copy of the data from a disk mirror. It then passes the data to the file system along with a warning status that only one copy of the data is left in a disk mirror or that one stripe is inaccessible in a stripe set with parity and that data redundancy is therefore no longer in effect for that sector. It is the file system's decision to respond or ignore the warning. FtDisk will re-recover the data each time the file system tries to read from the bad sector (Solomon, 440-45). With the data fairly secure from system crashes and failures, The only thing NTFS needs to do is to protect files from unauthorized users.

File Security

Files in NTFS are protected by the security model in Windows NT. An exhaustive discussion of NT's security model will not be done here, since this model can warrant its own research paper. Instead, only the security features pertinent to the actual NTFS files will be explored. With that in mind, looking back at the file records in the MFT, the master file table, a security descriptor can be found as an attribute of all file records. This security descriptor controls who has what access to the file. It contains the following:

- ***Owner SID*** – The owner's security ID
- ***Group SID*** – the security ID of the primary group for the file
- ***Discretionary access control list (DACL)*** – specifies who has what access to the file
- ***System access control list (SACL)*** – specifies which operations by which users should be logged in the security audit log

An access control list consists of an ACL header and zero or more *access control entry* (ACE) structures. An ACL with zero ACEs is called a null ACL and indicates that no user has access to the file. In a DACL, each ACE contains a security ID and an access mask. Two types of ACEs can appear in a DACL: access allowed and access denied. Of course, the access-allowed ACE grants access to a user, and the access-denied ACE denies the access rights specified in the access mask. The accumulation of access rights granted by individual

ACEs forms the set of access rights granted by an ACL. If no DACL is present in a security descriptor, everyone has full access to the file. On the other hand, if the DACL is NULL (has 0 ACEs), no user has access to the file. An SACL contains only one type of ACE, called a system audit ACE, which specifies which operations performed on the object by specific users or groups should be audited. The audit information is stored in the system audit log. Both successful and unsuccessful attempts can be audited. If the SACL is NULL, no auditing occurs on the particular file (Solomon, 310-11).

Conclusion

Windows NT's file system, NTFS, is a robust file system that expands upon the earlier file systems such as the FAT (file allocation table) file system and the HPFS (high-performance file system). NTFS added the ability for recoverability, fault tolerance and data redundancy, and file security, so it could support mission-critical applications used by businesses and corporations that demand data integrity and high performance. The outcome of Microsoft's efforts was a file system that could provide preservation and protection of any data that is place in its disk volume.

Works Cited

Calinger, Peter. Operating System Elements: A User Perspective. Englewood Cliffs: Prentice-Hall, 1982.

Galvin, Peter Baer and Abraham Silberschatz. Operating System Concepts. Reading: Addison Wesley Longman, 1998.

Haberman, A. N. Introduction to Operating System Design. Chicago: Science Research Associates, 1976.

Kaisler, Stephen H. The Design of Operating Systems for Small Computer Systems. New York: John Wiley & Sons, 1983.

Solomon, David A. Inside Windows NT: Second Edition. Redmond: Microsoft Press, 1998