Global Journal of Computer Science & Technology
Volume 11 Issue 6 Version 1.0 April 2011
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

A Quick Review of On-Disk Layout of Some Popular Disk File Systems

By Wasim Ahmad Bhat , S. M. K. Quadri
Kashmir University

Abstract- : Disk file systems are being researched since the inception of first magnetic disk in 1956 by IBM. As such, many good disk file system designs have been drafted and implemented. Every file system design addressed a problem at the time of its development and efficiently mitigated it. The augmented or new designs rectified the flaws in previous designs or provided a new concept in file system design. As such, there are many file systems that have been successfully d in operating systems. Among these designs, some file systems have made an influential impact on the file system design because of their capability to cope up with change in hardware technology and/or user requirements or because of their innovation in file system ign or because time favored them which allowed them to find space in popular operating systems. In this paper, we vide a quick review of on-disk layout of some popular disk file systems across many popular platforms like Windows, Linux & Macintosh. The goal of this paper is to explore the on-disk layout of these file systems to identify the various layout policies and data structures they exploit which made them to be adapted by their native and other operating systems.

Keywords: *File System, On-Disk, Design, Popular, Review.*

Classification: *GJCST Classification: FOR Code: 100699,100604*

Strictly as per the compliance and regulations of:

A Quick Review of On-Disk Layout of Some Popular Disk File Systems

Wasim Ahmad Bhat[α], S. M. K. Quadri[Ω]

Abstract- Disk file systems are being researched since the inception of first magnetic disk in 1956 by IBM. As such, many good disk file system designs have been drafted and implemented. Every file system design addressed a problem at the time of its development and efficiently mitigated it. The augmented or new designs rectified the flaws in previous designs or provided a new concept in file system design. As such, there are many file systems that have been successfully implemented and incorporated in operating systems. Among these designs, some file systems have made an influential impact on the file system design because of their capability to cope up with change in hardware technology and/or user requirements or because of their innovation in file system design or because time favored them which allowed them to find space in popular operating systems. In this paper, we provide a quick review of on-disk layout of some popular disk file systems across many popular platforms like Windows, Linux & Macintosh. The goal of this paper is to explore the on-disk layout of these file systems to identify the various layout policies and data structures they exploit which made them to be adapted by their native and other operating systems.

Keywords- File System, On-Disk, Design, Popular, Review.

I. Introduction

Since the advent of computers a mechanism for persistent storage of data and/or programs was needed. On the time line, magnetic disks are the primitive [1] (introduced in 1956 as data storage for an IBM accounting computer) and still widely used secondary storage device. Magnetic disk drive is the most primitive and cost effective storage device. There has been continuous improvement in its hardware technology to increase its performance and capacity [2]. Although performance has seen less improvement with respect to capacity, but the tremendous drop in cost per unit byte, reliability over solid state storage and increase in capacity have made disk drives every body's choice [3]. And hence, disk file systems have attracted researchers over the globe to exploit its pros and minimize its cons.

About[α]- Research scholar in P. G. Department of Computer Sciences, Kashmir University, India. He did his Bachelor's degree in Computer Applications from Islamia College of Science & Commerce, India and Master's degree in Computer Applications from Kashmir University, India.
E-mail- wasim.ahmed.bhat@gmail.com
About[Ω] - Head, P. G. Department of Computer Sciences, Kashmir University, India. He did his M. Tech. in Computer Applications from Indian School of Mines, India and Ph. D. in Computer Sciences from Kashmir University.
E-mail-quadrismk@hotmail.com

A File System is a way to organize, store, retrieve, and manage information on a permanent storage medium such as a disk [4]. File system is an important part of an operating system as it provides a way by which data can be stored, organized, navigated, accessed and retrieved in form of files and directories from storage sub system. It is generally a kernel module which consists of algorithms to maintain the logical data structures residing on the storage subsystems. The basic key functions that every file system incorporates are basic file operations like copy, move, create, delete and rename, efficient organization of data for quick storage and retrieval and efficient use of disk space. Apart from these basic functions some file systems also provide additional functions such as compression, encryption, file streams and others. Keeping all the hardware parameters and workload constant, the performance of a hard disk will all depend upon the type of file system used. In general file systems were developed in an incremental fashion by individual efforts of researchers and software industry with high cohesion with the hardware limitation and requirements at that time. Later, refinement of existing file systems [5] and new file systems were developed to keep pace with hardware enhancement and off course need.

To understand the file system design in general and on-disk layout specifically, we need to review the history of its invention a bit so that we can get some overview of the environment and situations in which the first file systems were drafted and implemented. Further, this history will give us some idea about the incremental file system design that has been followed since the inception of first file system. In the early days of computers, file systems were simply considered part of the operating system that ran the computer, and in those days operating systems themselves were rather new and fancy. One of the first file systems to have a name was DEC Tape [6], named after the company that made it (Digital Equipment Corporation) and the physical system the files were stored on (reel-to-reel tape recorders). The tapes acted like very slow disk drives. DEC Tape stored an astoundingly small 184 kilobytes of data per tape on the PDP-8 [7], DEC's popular early minicomputer. It was called a minicomputer only because, while the size of a refrigerator, it was still smaller than IBM's mainframes that took up entire rooms. Of course, the invention of the transistor and integrated circuit allowed another

whole round of miniaturization. DEC slowly became extinct while the rest of the world moved to microcomputers.

In 1972, Gary Kildall [8] got interested in working with Microprocessors and got involved with Intel. His research was related to compilers and code optimization. While working as a consultant in Intel, Kildall developed the Programming Language/Microprocessor (PL/M) [9] and the Control Language/Microprocessor (CP/M) [10]. He wrote CP/M to test out PL/M compiler. CP/M allowed him to store files and retrieve them from 8-inch floppy. He was able to run and test programs from it, modify them and check their portability by putting floppy in other machine's drive. CP/M got very popular because it used small amount of memory required to run it, approximately 3 ½ K and had a file system, but it does not have a name. It was very simple, as it stored files in a completely flat hierarchy with no directories. File names were limited to eight characters plus a three-character "extension" that determined the file's type. This was perfectly sensible because it was exactly the same limitation as the computer Kildall was working with. Gary Kildall and the company he founded to sell CP/M, Digital Research, soon became very wealthy and the usage of CP/M was tripling every year. It turned out that a lot of microcomputer companies needed an operating system, and Gary had designed it in a way that separated all the BIOS from the rest of the OS. Unfortunately for Kildall, other people soon got the same idea he had.

A programmer named Tim Patterson [11] wrote his own OS called "QDOS" (Quick and Dirty Operating System) [12] that was a quick and dirty clone of everything CP/M did, because he needed to run an OS on a new 16-bit computer, and Gary hadn't bothered to write a 16-bit version of CP/M yet. QDOS had a slightly different file system than CP/M, although it did basically the same thing and didn't have directories either. Patterson's file system was based on a 1977 Microsoft program called Microsoft Disk Basic, which was basically a version of Basic that could write its files to floppy disks. It used an organization method called the File Allocation Table.

Bill Gates bought Tim Patterson's QDOS for $50,000, and renamed it MS-DOS [13]. He now was able to sell it to IBM and every company making an IBM clone, and Gary found himself quickly escorted from the personal computing stage. As it was originally a quick and dirty clone of a file system designed for 8-bit microcomputers in the 1970s that was itself a quick-and-dirty hack that mimicked the minicomputers of a decade earlier, FAT was not really up for very much. It retained CP/M's "8 and 3" file name limit, and the way it stored files was designed around the physical structure of the floppy disk drive, the primary storage device of the day. The introduction of hard disks soon made FAT-12 obsolete but file systems got attention and every individual researcher and software industry professional recognized its importance and started either enhancing and augmenting the older designs or re-designing some new file systems from scratch.

In this paper, we will look at some most popular file systems' on-disk layout. The popularity of the file systems selected is solely based on the popularity of the operating systems that support them natively. The goal of this paper is to look at the layout policies they exploit and data structures they use to mitigate the challenges for which they were designed. In this paper, we will review the native file systems of Windows, Linux and Macintosh operating systems.

II. FAT File Systems

The design of FAT [14] file system is very simple as it uses simple data structures. This simplicity in design has made FAT file system popular and supported by almost every operating system. In today's world, several digital devices, such as mini mp3 players, smart phones, digital cameras, etc. are becoming part of our life. These devices exchange data frequently with desktop computers. The PC discovers these devices as standard USB mass storage devices and automatically mounts the file system inside them. This is possible only if the file system used in device is supported by the PC's operating system. That is why; conventional FAT file system is a useful format for solid state memory cards as it provides a convenient way to share data by being supported by almost all operating systems [15]. As mentioned before, FAT12 was the first FAT file system but was able to address only limited number of sectors as it was developed for floppy disks. Later, with the introduction of hard disk drive, FAT16 was introduced and with higher capacity drives, FAT32 and now exFAT [16] (unofficially called FAT64). Almost all the flavors of FAT file system follow same design with the exception of pointer width in bits that is used to access the sectors (or Clusters) and which gives the FAT suffix 12, 16, 32 and 64. FAT12 and FAT16 are obsolete now whereas exFAT is not widely used yet, in contrast to FAT32 which is supported by almost every operating system.

The FAT32 file system consists of 4 different data structures to allow semantics of hierarchical file systems to be implemented on volume.

a) *BOOT Sector*

Boot Sector is located at the beginning of the volume. It includes an area called BPB (Bios Parameter Block) at offset 11 of length 49 bytes and contains some basic file system information. The rest of the sector usually contains boot code with boot signature word (0x55AA) at offset 509.

BPB is a one dimensional table that contains variable length entries. Each entry in BPB stores file system layout information except one (BPB_Reserved) which is kept reserved for future extension. Different versions of FAT file systems have size difference in BPB and contain different entries. Table 1 shows the BPB for FAT32 file system. Each entry has been given a name to identify its role along with entry offset and size.

Name	Offset (byte)	Size (bytes)
BS_jmpBoot	0	3
BS_OEMName	3	8
BPB_BytsPerSec	11	2
BPB_SecPerClus	13	1
BPB_RsvdSecCnt	14	2
BPB_NumFATs	16	1
BPB_RootEntCnt	17	2
BPB_TotSec16	19	2
BPB_Media	21	1
BPB_FATSz16	22	2
BPB_SecPerTrk	24	2
BPB_NumHeads	26	2
BPB_HiddSec	28	4
BPB_TotSec32	32	4
BPB_FATSz32	36	4
BPB_ExtFlags	40	2
BPB_FSVer	42	2
BPB_RootClus	44	4
BPB_FSInfo	48	2
BPB_BkBootSec	50	2
BPB_Reserved	52	12
BS_DrvNum	64	1
BS_Reserved1	65	1
BS_BootSig	66	1
BS_VolID	67	4
BS_VolLab	71	11
BS_FilSysType	82	8

Table 1. Description of FAT32 BPB

Reserved Sectors immediately follow Boot Sector. The number of reserved for volume includes Boot Sector and is indicated by BPB at offset 14 of Boot Sector. Typically, reserved sectors include FSInfo sector at sector 1 and BkBoot sector at sector 6 of the volume. FSinfo sector further qualifies the FAT32 volume, while BkBoot is replica of boot sector and is used for recovery purposes.

b) *File Allocation Table (FAT)*

The File Allocation Table (FAT) is an array of n-bit wide entries and spans over a number of sectors indicated by BPB at offset 36 of Boot Sector. FAT32 volume has generally 2 consecutive copies of FAT data structure and is called FAT Mirroring. Mirroring is done for recovering from FAT corruption in case one copy of FAT gets corrupt. In case of solid state storage devices, FAT is not mirrored to prolong the life of solid state device by reducing the write cycles. Bit 7 of BPB offset 40 of boot sector indicates whether FAT is mirrored or not. This data structure of FAT file system gives it the name and is the heart of the file system. The suffixes used by various FAT file systems indicate the bit width of entries in FAT data structure. Thus, in FAT32, the FAT entries are 32-bit wide.

FAT data structure is a table that stores the information about which clusters are free, used or possibly unusable. A cluster is a fixed length group of consecutive data sectors which are located immediately after FAT data structure and occupy rest of the volume. The number of sectors per cluster is indicated by BPB at offset 13 of boot sector. FAT file system always allocates space on storage device in terms of clusters. This is done to increase the performance of the file system by avoiding individual multiple accesses to disk. Thus, the file system may suffer from high internal fragmentation if cluster is too large and there are many small sized files; and may degrade the performance if it is small and the volume has large sized files. Depending upon the type of file system and size of the volume, the cluster size varies but the number of sectors per cluster is restricted to a value that is power of 2 i.e. 1,2,4,8,16,32,64, etc. In addition to keep track of used and unused clusters, FAT data structure also keeps track of chain of clusters allocated to a file. The technique used by FAT32 file system is simple. Every file and directory except the root directory of volume has an entry in its parent directory that contains its name, attributes & 32 bit wide entry that indicates the first cluster number allocated to it. The FAT data structure entries are 32 bit wide and each entry uniquely corresponds to the cluster on the volume sequentially i.e. the first entry corresponds to cluster 0, second entry corresponds to the cluster 1, etc. The formula used to locate the cluster entry in FAT data structure for any valid cluster number N is

$$FATOffset = N * 4$$

$$ThisFATSecNum = BPB\_ReservedSecCnt + \frac{FATOffset}{BPB\_BytsPerSec}$$

$$ThisFATEntOffset = FATOffset \% BPB\_BytsPerSec$$

where ThisFATSecNum is the logical sector number of the volume and ThisFATEntOffset is the offset in the sector where 32-bit FAT entry corresponding to cluster number N exists. The contents of any valid cluster entry in FAT can have values as shown in Table 2.

FAT32 Cluster Entry Values	Description
0x00000000	Is Free Cluster
0x00000001	Reserved value
0x00000002 – 0x0FFFFFEF	Is Used Cluster and value points to next cluster in the chain allocated to file/directory
0x0FFFFFF0 – 0x0FFFFFF6	Reserved values
0x0FFFFFF7	Some Bad sector in Cluster,

	Unusable
0x0FFFFFF8 – 0x0FFFFFFF	Is Last Cluster in file/directory or EOC (End Of Cluster chain) marker

Table 2. Description of Valid FAT Entries

Let's suppose two files, say MYFILE1.TXT and MYFILE2.TXT are currently residing on a FAT32 volume such that the former is fragmented and is 3 clusters long while the latter is not fragmented and is 2 clusters long as shown in figure 1. MYFILE1.TXT has first cluster allocated 0x00000029, FAT contents against that cluster shows another cluster 0x0000002A, then 0x0000002D whose FAT contents show that this cluster is the last cluster in chain. Similarly, for MYFILE2.TXT the first cluster allocated is 0x0000002B whose FAT contents point to next cluster in chain, 0x00000002C, which is the last cluster in chain as pointed by its FAT content.

Each file/directory may occupy one or more clusters depending upon its size. Thus, a file/directory is represented by a chain of these clusters. However, these clusters are not necessarily to be stored adjacent to one another on the disk's surface but are often fragmented throughout the volume as shown in figure 1 where MYFILE1.TXT is fragmented while MYFILE2.TXT is not.

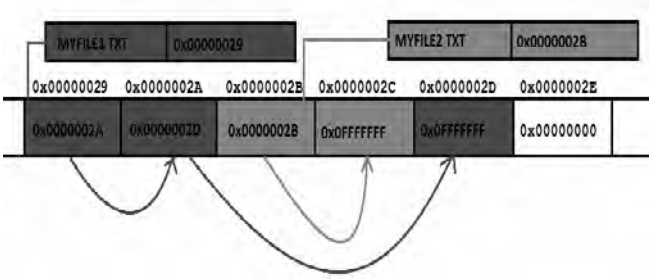

Figure 1. A Snapshot of FAT Data Structure.

As memory cost per unit capacity is dramatically decreasing every year and storage size is increasing, the maximum number of clusters have increased dramatically and also the cluster size. In FAT32, the FAT entry is 32 bit wide which points to next cluster in chain but it only uses lower 28 bits to address clusters. Thus, FAT entry values say 0xA0000000 and 0xB0000000 point to same cluster on volume. As such, 2^{28} clusters can exist on FAT32 volume. As mentioned before, the successive major versions of FAT file systems are named after the number of table entry bits; FAT12, FAT16, FAT32 & FAT64, the goal of every new version is to address large volume and large file size. Although, KFAT [17], TFAT [18] and FATTY [19] versions of FAT file system have also been designed but the goal was reliability. Because the number of bytes per sector as indicated by BPB at offset 11 of Boot sector is always divisible by 4, a FAT32 FAT entry never spans over a sector boundary.

The first two entries in FAT store special values:

- The first entry contains a copy of BPB at offset 21 of Boot Sector which is 8 bit long which indicates the type of storage media. The remaining 20 bits between high 4 and low 8 of this entry are set to 1.
- The second entry stores the EOC marker. The high order two bits of this entry are sometimes, used for dirty volume management: high order bit if set to 1 indicates that last shutdown was clean otherwise abnormal. The next highest bit, if set to 1 indicates that during the previous mount no disk I/O errors were detected else there were.

Because the first two FAT entries store special values, there is no cluster 0 or 1. The first addressable cluster in FAT32 FAT data structure is cluster 2, which is the reason why BPB value at offset 44 of Boot Sector which indicates the Root Directory cluster number cannot be less than 2 and is usually 2, i.e., the Root Directory is at the start of file/directory region.

c) *Directory Structure*

The semantics of hierarchical file system lives on the notion of files and directories. The hierarchical file system is like a tree where every non-leaf node is a subdirectory containing any number of non-leaf nodes (sub-directories) or leaf nodes (files) or both. The tree begins at a root node called root directory. In FAT32, the root directory is of variable size and is assigned the first cluster, whose address is indicated in BPB at offset 44. Among all the files and directories that may reside on FAT32 volume, root directory is the only directory that does not have filename and attributes; more precisely does not have any entry like other files and directories have. In case of FAT12 and FAT16, root directory is located at fixed location after FAT copy and is of fixed size indicated in BPB.

A directory is an array of 32 byte wide structures where each structure represents a file or directory either existing or deleted and in case of long name support, the remaining the parts of long name. The structure of 32 byte wide entry of directory is shown in Table 3.

Name	Offset (byte)	Size (bytes)
DIR_Name	0	11
DIR_Attr	11	1
DIR_NTRes	12	1
DIR_CrtTimeTenth	13	1
DIR_CrtTime	14	2
DIR_CrtDate	16	2
DIR_LstAccDate	18	2
DIR_FstClusHI	20	2
DIR_WrtTime	22	2
DIR_WrtDate	24	2
DIR_FstClusLO	26	2
DIR_FileSize	28	4

Table 3. Description of FAT32 Directory Entry Structure

The name and other metadata about a file are all stored in the 32-byte directory entry for file. The list of characters that cannot be used in a file name, ". " / \ [ ] ; : | = or 0x20 is really an operating system issue, not a file system issue. Linux, via its FAT support, can create files with some of these characters in their names. This may cause problems with portability if that disk is later read in a Windows environment. Dating back to the creation of the first FAT12 volumes in the 70's, all files were given a name in the 8.3 naming convention. That is, eight characters for the name and three characters for an extension that identified the type of file; 'dot' is never saved. Long file name support was later introduced but not in any semblance of an elegant way.

Usually, FAT32 places the root directory in the first available cluster, which places it right behind the FAT area. All other directories in all the FAT file systems will be allocated clusters as they need them and can reside anywhere on the disk.

III. NT File System

NTFS was designed to quickly perform standard file operations such as read, write & search. It was developed from scratch although some concepts were borrowed from OS/2's HPFS [20]. The design of NTFS file system is bit complex but very nicely drafted and crafted. It includes many new features of modern file system like transparent compression and encryption, sparse files, multiple data streams, reliability, fast recovery, security features, privileges and permissions, and representation of everything as file and everything belonging to a file as collection of attribute/value pairs from filename attribute to data attribute [21]. The design of NTFS file system is such that every sector of volume belongs to some file unlike FAT. Even the file system metadata that describes the file system is part of some file.

When a volume is formatted with NTFS file system, it leads to the creation of several system files used by file system to store volume metadata and implement the file system. These files are not accessible to user directly. These system files have entry just like other regular volume files and directories have, and have been given some reserved names prefixed by $ sign. The standard configuration of NTFS file system has 16 system files out of which last 4 entries are reserved [22]. Table 4 lists these system files along with their $MFT name, $MFT entry offset (explained later) and purpose of the file.

System File	File Name	MFT Record	Purpose of the File
Master file table	$Mft	0	Contains one base file record for each file and folder on an NTFS volume.
Master file table 2	$MftMirr	1	A duplicate image of the first four records of the $MFT.
Log file	$LogFile	2	Contains a list of transaction steps used by NTFS for recoverability.
Volume	$Volume	3	Contains information about the volume.
Attribute definitions	$AttrDef	4	A table of attribute names, numbers, and descriptions.
Root file name index	$	5	The root folder.
Cluster bitmap	$Bitmap	6	A representation of the volume showing which clusters are in use.
Boot sector	$Boot	7	Includes the BPB used to mount the volume and additional bootstrap loader code used if the volume is bootable.
Bad cluster file	$BadClus	8	Contains bad clusters for the volume.
Security file	$Secure	9	Contains unique security descriptors for all files within a volume.
Upcase table	$Upcase	10	Converts lowercase characters to matching Unicode uppercase characters.
NTFS extension file	$Extend	11	Used for various optional extensions such as quotas, reparse point data, and object identifiers.
		12-15	Reserved for future use.

Table 4. $MFT Entry name, & Offset & Purpose of NTFS System Files

a) *$BOOT*

The location of $BOOT file is fixed and resides on first 16 sectors of NTFS volume. The first sector is called Boot Sector as it contains the boot strap code and following 15 sectors are boot sector's IPL (Initial

Program Loader). The boot sector is duplicated at last sector of the volume. The boot sector of $BOOT file contains two data structures; BPB followed by Extended BPB. Table 5 describes the BPB and Extended BPB of NTFS boot sector (Offset, Length & Field Name).

Byte Offset	Field Length	Field Name
0x0B	WORD	Bytes Per Sector
0x0D	BYTE	Sectors Per Cluster
0x0E	WORD	Reserved Sectors
0x10	3 BYTES	*always 0*
0x13	WORD	*not used by NTFS*
0x15	BYTE	Media Descriptor
0x16	WORD	*always 0*
0x18	WORD	Sectors Per Track
0x1A	WORD	Number Of Heads
0x1C	DWORD	Hidden Sectors
0x20	DWORD	*not used by NTFS*
0x24	DWORD	*not used by NTFS*
0x28	LONGLONG	Total Sectors
0x30	LONGLONG	Logical Cluster Number for the file $MFT
0x38	LONGLONG	Logical Cluster Number for the file $MFTMirr
0x40	DWORD	Clusters Per File Record Segment
0x44	DWORD	Clusters Per Index Block
0x48	LONGLONG	Volume Serial Number
0x50	DWORD	Checksum

Table 5. BPB & Extended BPB of NTFS file system

Among other things, the two data structures contain sectors per cluster, bytes per sector, total sectors, logical cluster number of $MFT file, logical cluster number of $MFTMirr file, clusters per file record segment and clusters per index block.

b) *$MFT*

$MFT file or Master File Table file is an array of fixed records where each record represents uniquely every file or directory of the volume even the system files including the $MFT file. The first 16 records are reserved for system files. Table 4 shows the list of first 16 records ordered as per their position and corresponding system files they represent along with short description. The first entry represents the $MFT file itself while second entry represents the mirrored copy of $MFT file named $MFTMirr whose first record is identical to first record of $MFT. Actually, $MFTMirr duplicates first 4 records of $MFT for recovery purpose. In case the first record of $MFT that defines $MFT, is corrupted the file system code should read the second record of $MFT to locate $MFTMirr and read its first record to build $MFT or should directly read the $MFTMirr file's first record by locating its position from logical cluster number in BPB to build $MFT. As $MFT actually defines the NTFS layout, logical cluster number of $MFT is kept in BPB so that file system driver can locate $MFT at boot time. $MFT is not fixed like FAT and hence can be relocated in case it is damaged; same is true for other system files.

A record in $MFT is a 1 KB structure that stores attributes of file/directory to which it corresponds. NTFS stores everything belonging to file or directory as a collection of attribute/value pairs including filename, security information, time stamps, data, etc [23]. Each $MFT record corresponds to a unique file. If a file has large number of attributes, more than one record is allocated to a file. In this case, the first record that stores the location of others in Attribute List attribute is called Base File Record. Whether a file consumes one or more $MFT records, if the value for any particular attribute is completely stored in record, such an attribute is called Resident Attribute. Several attributes are defined as always being resident so that NTFS can locate non-resident attributes for e.g. $STANDARD_INFORMATION, $INDEX_ROOT, $ATTRIBUTE_LIST, etc. A non-resident attribute is one whose value cannot be completely stored in an $MFT record. In such case, NTFS allocates clusters for the attribute's data separate from $MFT. This area is called a *run* or technically an *extent*. If resident attribute's value grows, it is converted to non-resident attribute and allocated a run. $DATA attribute for files greater than 1 KB, $BOOT, $MFTMirr and $LogFile is always non-resident. Table 6 shows the standard attribute names and their description [24]. Actually attributes correspond to numeric codes which NTFS uses to order (in ascending order) the attributes within an $MFT record with same attribute types appearing more than once in case a file has multiple values for that attribute. Most attributes never have names, though Index related attributes and $DATA attribute often does. Names distinguish among multiple attributes of same type that a file can include. The value of an attribute is a byte stream and is stored as a separate stream in a file. NTFS does not read and write files instead attribute streams. The read and write APIs exported by file system driver normally operate on file's unnamed $DATA attribute.

Attribute Type	Description
Standard Information	Includes information such as timestamp and link count.
Attribute List	Lists the location of all attribute records that do not fit in the base MFT record.
File Name	A repeatable attribute for both long and short file names. The long name of the file can be up to 255 Unicode characters. The short name is the 8.3, case-insensitive name for the file. Additional names, or hard links, required by POSIX can be included as additional file name attributes.
Security Descriptor	Describes who owns the file and who can access it.

Data	Contains file data. NTFS allows multiple data attributes per file. Each file typically has one unnamed data attribute. A file can also have one or more named data attributes, each using a particular syntax.
Object ID	A volume-unique file identifier. Used by the distributed link tracking service. Not all files have object identifiers.
Logged Utility Stream	Similar to a data stream, but operations are logged to the NTFS log file just like NTFS metadata changes. This is used by EFS.
Reparse Point	Used for volume mount points. They are also used by Installable File System (IFS) filter drivers to mark certain files as special to that driver.
Index Root	Used to implement folders and other indexes.
Index Allocation	Used to implement folders and other indexes.
Bitmap	Used to implement folders and other indexes.
Volume Information	Used only in the $Volume system file. Contains the volume version.
Volume Name	Used only in the $Volume system file. Contains the volume label.

Table 6. Standard Attribute Types & their Description

Each $MFT record begins with an entry header which is 42 bytes long. This standard header contains a magic number "FILE", number of entries in fix up array, $Log File sequence number, Sequence number, Hard Link count, offset to first attribute, flags that indicate whether record is in use or not, used and allocated size of MFT entry, file reference to base file record in case it is not base record, attributes and fix up values. Each attribute begins with a standard header containing information about the attribute like type and length of attribute, length of name and offset to name, non-resident flag, etc. The header of every attribute is always resident and records whether the value is resident or non-resident.

For resident attributes, the header also contains the offset from the header to attribute's value and length of attribute's value. Figure 2 shows the typical structure of a $MFT entry record [25].

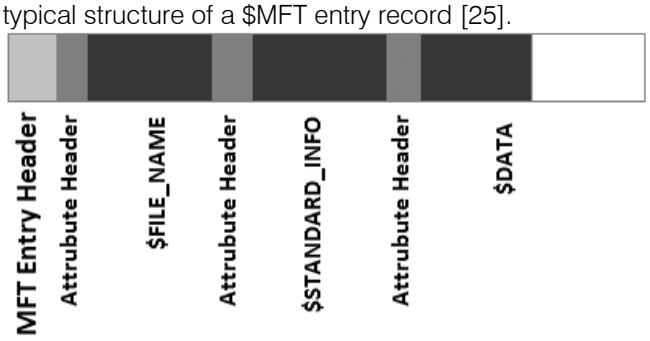

Figure 2. Typical MFT Entry Record

NTFS refers to physical locations on a disk by means of logical cluster numbers (LCNs). LCNs are simply the numbering of all clusters from the beginning of the volume to the end. To convert an LCN to a physical disk address, NTFS multiplies the LCN by the cluster factor (i.e. number of sectors per cluster) to get the physical byte offset on the volume. NTFS refers to the data within a file by means of virtual cluster numbers (VCNs). VCNs number the clusters belonging to a particular file from 0 through m. VCNs aren't necessarily physically contiguous, however; they can be mapped to any number of LCNs on the volume. When an attribute is nonresident, as the data attribute for a large file might be, its header contains the information NTFS needs to locate the attribute's value on the disk. This information is typically the VCN-to-LCN mapping pairs. Figure 3 shows the data attribute header containing VCN-to-LCN mappings for the two runs, which allows NTFS to easily find the allocations on the disk. Other attributes can be stored in runs if there isn't enough room in the $MFT file record to contain them.

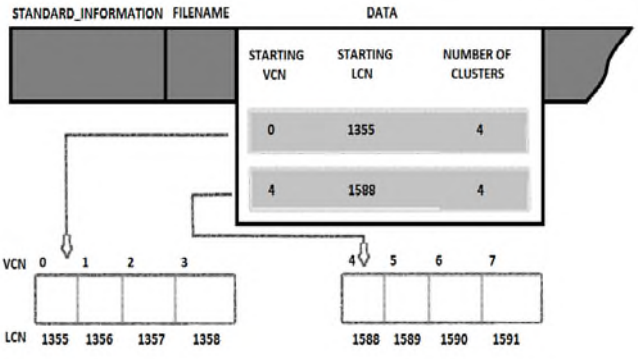

Figure 3. Non-Resident $DATA attribute of File.

A file on an NTFS volume is identified by a 64-bit value called a *File Reference*. The file reference consists of a file number and a sequence number. The file number corresponds to the file's $MFT record entry offset (or to that of base file record if the file has more than one file record entries). The file reference sequence number, which is incremented each time an $MFT file record position is reused, enables NTFS to perform internal consistency checks. If a particular file has too many attributes to fit in the $MFT record, a second $MFT record is used to contain the additional attributes (or attribute headers for nonresident attributes). In this case, an attribute called the *Attribute List* is added to file in base record. The attribute list attribute contains the name and type code of each of the file's attributes and the file reference of the $MFT record where the attribute is located. The attribute list attribute is also provided for those cases in which a file grows so large or so fragmented that a single $MFT record can't contain the multitude of VCN-to-LCN

mappings needed to find all its runs. Files with more than 200 runs typically require an attribute list.

In NTFS, a file directory is simply an index of filenames, i.e., a collection of filenames along with their file references organized in a particular way (B-tree) for quick access [26]. To create a directory, NTFS indexes the filename attributes of the files in the directory. Conceptually, an $MFT entry for a directory contains in its Index Root attribute a sorted list of the files and/or directories in the directory. It also contains the file reference in the MFT where the file/directory is described and time stamp and size information for the file/directory. A large directory can also have nonresident attributes (or parts of attributes), as Figure 4 shows.

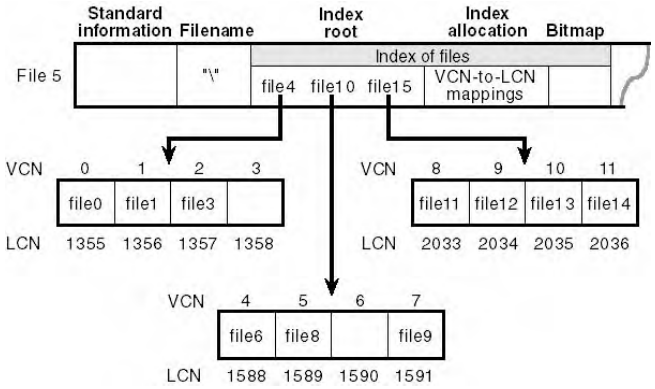

Figure 4. Root Directory [21]

In this example, the $MFT file record doesn't have enough room to store the index of files that make up this large directory. A part of the index is stored in the Index Root attribute, and the rest of the index is stored in non-resident runs called Index Buffers. For large directories, however, the filenames are actually stored in 4-KB fixed-size index buffers that contain and organize the filenames. Index Buffers implement a B+ tree data structure, which minimizes the number of disk accesses needed to find a particular file, especially for large directories. The index root attribute contains the first level of the B+ tree (root subdirectories) and points to index buffers containing the next level (more subdirectories, perhaps, or files). Figure 4 shows only filenames in the index root attribute and the index buffers (file6, for example), but each entry in an index also contains the file reference in the $MFT where the file is described and time stamp and file size information for the file. NTFS duplicates the time stamp and file size information from the file's $MFT record. This technique, which is used by FAT and NTFS, requires updated information to be written in two places. Even so, it's a significant speed optimization for directory browsing because it enables the file system to display each file's time stamps and size without opening every file in the directory.

The index allocation attribute maps the VCNs of the index buffer runs to the LCNs that indicate where the index buffers reside on the disk, and the bitmap attribute keeps track of which VCNs in the index buffers are in use and which are free. Figure 4 shows one file entry per VCN (that is, per cluster), but filename entries are actually packed into each cluster. Each 4-KB index buffer can contain about 20 to 30 filename entries. The B+ tree data structure is a type of balanced tree that is ideal for organizing sorted data stored on a disk because it minimizes the number of disk accesses needed to find an entry. In the $MFT, a directory's index root attribute contains several filenames that act as indexes into the second level of the B+ tree. Each filename in the index root attribute has an optional pointer associated with it that points to an index buffer. The index buffer contains filenames with lexicographic values less than its own. In Figure 4, for example, file4 is a first-level entry in the B+ tree. It points to an index buffer containing filenames that are (lexicographically) less than itself—the filenames file0, file1, and file3. Note that the names file1, file2, and so on that are used in this example are not literal filenames but names intended to show the relative placement of files that are lexicographically ordered according to the displayed sequence.

c) *$LogFile*

The internal structure of the $LogFile is not well understood. Once the log is full, the first entry is overwritten with the next new entry. What get logged are the individual transactions that make up each file access or file write or whatever. For instance, when modifying a file the following steps might occur:

- read $MFT entry for directory entry file is in
- read directory entry file is in
- read $MFT record for file
- write file
- update Atime in file's MFT record
- update Mtime in file's MFT record
- update Atime in directory entry for that file
- update Mtime in directory entry for that file

This list gets considerably longer if the file is encrypted or compressed. If the command fails before the entire string of transactions are completed, due to system crash or whatever other reason, the file system has to have a way to change each of the transactions involved back to their previous values in order to maintain consistency of the file system. The file system provides a reliable, crash-resilient environment.

d) *$Volume*

The file $Volume contains the name of the volume. That is its most important function. There is also volume information data in this file that contains a version number and a set of flags. The version number

will be broken into two pieces, a major and a minor version number.

e) *$AttrDef*

This file contains the list of attributes available to the file system in this version of NTFS. It is because of this file that we know the catchy names for the attributes that we are using. The entry for the attribute also contains some information about the allowable sizes and location (resident or not) of the attribute can be.

f) *$Bitmap*

The $BitMap is a special file within the NTFS file system. This file keeps track of all of the used and unused clusters on an NTFS volume. When a file takes up space on the NTFS volume the location is uses is marked out in the $BitMap. The method of keeping track of cluster allocation is relatively simple. Each bit in the Bitmap represents 1 cluster; if that bit is "1" then the cluster is in use.

g) *$BadClus*

This file is the size of the NTFS volume, but is a sparse file of all zeros. Since zeros in sparse files are counted instead of saved, this file takes up no space on the disk. If a cluster is ever deemed 'bad', data will be written to this file at the same offset into this file as the offset the bad cluster is into the volume. This will causes this file to allocate clusters in the $bitmap file, which in turn prevents other files from trying to use the bad cluster in the future.

h) *$Secure*

In Windows NT, every file had a $Security_Descriptor attribute that did this job. Since many files had the same values in that attribute it was moved to this file so that data wasn't repeated.

i) *$UpCase*

Case in the file name is preserved, but is converted to all uppercase for sorting as the directory entry is created. This file contains the uppercase characters of 'every' UNICODE alphabet so that NTFS knows the proper alphabetical order of each code page of UNICODE without having to inherently knows every code page of UNICODE.

j) *$Extend*

$Extend is a directory that contains other system files. This allows for more system files to be added but without pushing the limit of the 16 I-nodes reserved for system files.

IV. Extended File Systems

In response to these problems, two new file systems were developed "Xia" and "Second Extended File System" [31]. Xia file system was based on Minix file system and provided long filenames, support for large volume size and 3 timestamps; while Ext2 file system was based on Ext file system with many reorganizations and improvements. It was designed with evolution in mind and contained space for future extension. Due to minimal design, Xia was more stable than Ext2 file system. Later, bugs were fixed in Ext2 file system and lots of improvements and new features were integrated. Ext2 file system became stable and de facto standard Linux file system. Ext2 uses VFS to extend the maximum volume from 2 GB to 4 TB. It allows root user to recover from incidents where other users overfill the file system. It uses variable length directory entries while filename length could be extended to 1012. Ext2 file system may use synchronous updates like BSD FFS [32]. This is the maximum reliability support provided by Ext2 file system. In synchronous updates, any modification to file system metadata like I-node, bitmap blocks, indirect blocks and directory blocks are synchronously written to the disk. Although this mechanism provides bit reliability, it leads to poor performance. Ext2 file system allows administrator to choose logical block size when creating file system. Block sizes can typically be 1024, 2048 and 4096 bytes. Ext2 implements fast symbolic links which does not use any data block on file system by not storing the target name in a data block but in I-node itself.

Andrew S. Tanenbaum wrote the Minix operating system in 1987 [27]. Tanenbaum created it for teaching purpose. Later, he published a textbook that included source code of Minix. This code was taken and published on Usenet where thousands of readers were able to examine and further develop Minix. As Minix was simple and bug free, Torvalds decided to incorporate its architecture into the operating system he was developing. Torvalds named his operating system Linux. One shortcoming of Torvalds first Linux kernel was that it only supported Minix file system. Minix file system was an efficient and relatively bug free piece of software. However, the restrictions in design of Minix file system were too limiting, so people started thinking and working on the implementation of new file system in Linux [28]. In order to add more file systems to Linux operating system, Torvalds modified a VFS written by Chris Provenzano and integrated it into the kernel [29]. After integration, a new file system called "Extended File System" was implemented which removed two big Minix limitations; maximum volume size and maximum filename length, but still there were some problems; no support for separate access, I-node and data modification timestamps. This file system used linked lists to keep track of free blocks and I-nodes and thus

resulted in bad performance with aging [30].

Ext3 file system was designed to eliminate enormously long file system recovery times after the crash. Ext3 is a journaling file system [33]. A journaling file system differs from a traditional file system in that it keeps transient data in new location, independent of the permanent data and metadata on disk. Because of this, such a file system does not dictate that the permanent data has to be stored in any particular way. As such, it is quite possible for Ext2 file system on disk structure influenced by the layout of the BSD file system to be used in this file system. The layout of journaled Ext2 (or Ext3) file system on disk is entirely compatible with existing Ext2 file system. Ext2 file system design already includes a number of reserved I-node numbers; one among them is used for the file system journal. The features that separate Ext3 from being a valid ext2 system are journaling, h-tree indexing, and file system growth while the system is online. Ext4 [34] is the most recent version of the extended file system. This latest release hosts many new features such as a maximum volume size of one Exabyte, backwards compatibility with ext2 and ext3, online defragmentation, and nanosecond timestamps. The nanosecond timestamp is unique to Ext4 and allows applications that utilize file creation and modification times to track their timing in nanoseconds rather than seconds.

As there has been a large drift in the on-disk layout of Linux file systems from Extended file system to Extended 2 file system while later versions have support Ext2 on-disk layout, we will review only Ext and Ext2 on disk layout in detail.

Extended File System is based on the concepts derived from UNIX operating system. In Extended File Systems, every file is represented by an I-node (Index Node), everything is a file, directory which is a special file contains list of entries pertaining to files it contains along with corresponding I-node. When a volume is formatted with Extended File System, 4 data structures are created as shown in figure 5.

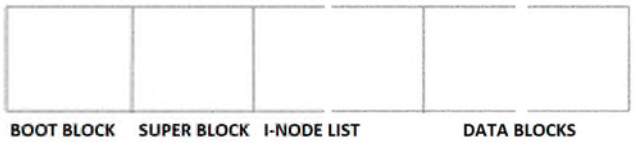

Figure 5. Data Structures of Extended File System

a) *Data Blocks*

Data Blocks immediately follow the I-node list and occupy rest of the volume. A data block is a set of consecutive sectors which is allocated to a file in its entirety. They are internally represented by numbers corresponding to their position in the volume. A file may be allocated one or more data blocks, consecutive or fragmented over the volume.

b) *I-node List*

I-node list structure immediately follows the Super block. The size of I-node list depends upon the volume size and is calculated at initial format and punched in Super block. I-node is the basic building block; every file and directory in the file system is described by one and only one I-node. Each I-node contains the description of the file it represents; file type, access permissions, owner, access times, link count, file size and table of pointers to data blocks. I-nodes are internally represented by I-node number enumerated by their position in the I-node list. The numbering begins from 1, I-node 0 does not exist on newly formatted volume. An I-node of Type=0 and number of links=0, is free otherwise represents a file.

The table of pointers to data blocks is an array of entries where first 9 direct entries contain the address (index number) of data blocks containing data of the file while the next single indirect entry contains the address of data block that contains the direct entries for data blocks containing the data of the file. The next entry in table is a double indirect entry that points to a data block which contains single indirect entries. Similarly, a triple indirect entry in table points to a data block that contains double indirect entries. This level of indirection is used to allow the structure of I-node to be small but at the same time allows large file size to be addressed. This scheme is shown in figure 6.

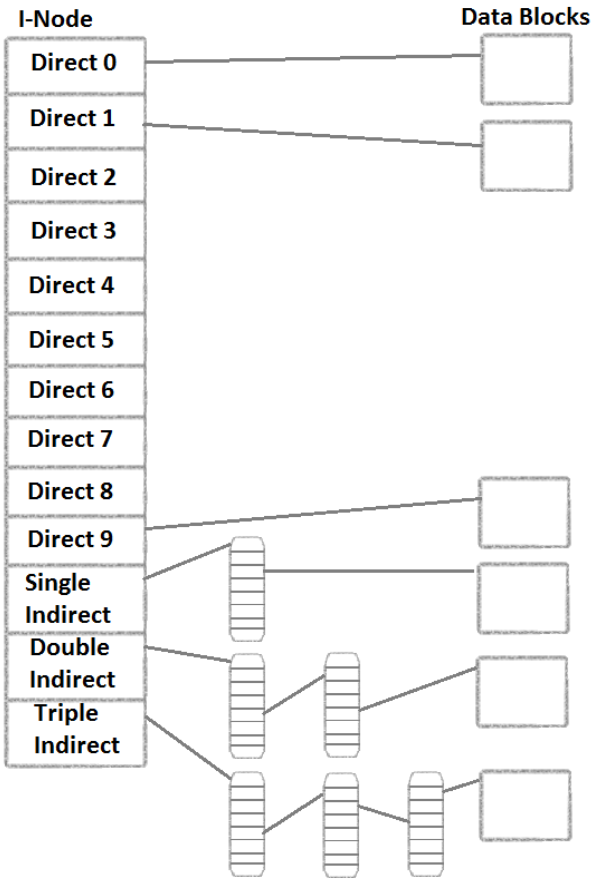

Figure 6. Levels of indirection to address data blocks.

Several block entries in I-node can be 0 meaning that logical block entries contain no data. This happens if no process ever wrote data into the file offsets corresponding to those blocks and hence block numbers remain at their initial value 0. This way Extended File System supports Sparse files.

Byte Offset in Directory	I-Node Number	File Names
0	83	.
16	2	..
32	1798	init
48	1276	fsck
64	85	mount
80	1268	passwd

Figure 7. A typical directory file content.

Directories are implemented as special files containing a list of fixed sized entries. Each entry contains I-node number and fixed length filename it represents. Any entry that contains 0 in I-node but has some valid filename represents a deleted file that existed previously on the volume. Every directory file has first 2 entries containing '.' and '..' entry representing its I-node number and parent directory's I-node number respectively. For root '/' directory both entries have same value. A typical directory file content is shown in figure 7.

c) *Boot Block & Super Block*

The Boot block is located at first sector of volume and contains the boot strap code. The Super block immediately follows the Boot block and contains the information that describes the state of a file system. The information contained in Super block includes:

- Size of the file system,
- Number of free blocks in the file system,
- A list of free blocks in the file system,
- Index of next free block in the free block list,
- Size of I-node list,
- Number of free I-nodes in file system,
- List of free I-nodes in file system,
- Index of next free I-node in free I-node list,
- Lock fields for free block and free I-node list, and
- Flag indicating that Super block has been modified.

Extended file system stores in Super block information that is needed to maintain I-nodes and data blocks. When the volume is created Super block list of free I-nodes is empty and kernel searches the I-node list structure for those I-nodes where the Type=0 and populates the list to its full capacity remembering the highest numbered I-node it finds. The next time the kernel searches the disk for free I-nodes, it uses this remembered I-node as its starting I-node. Keeping track of I-nodes is easy but the list is used to avoid the I-node list search every time an I-node is needed as free I-nodes can be located in I-node list any time by searching for type field. The data blocks are necessarily to be maintained in their entirety because there is no way for kernel to know on the basis of the content they contain that whether the data block is free or allocated.

The Super block contains the list of free blocks populated at the time of volume creation. The data blocks are organized in a linked list fashion. The Super block list contains the list of free blocks to its capacity. One entry in the list points to a block that contains such kind of a list to its capacity. During volume creation, the kernel tries to organize the list in such a manner such that block numbers allocated to a file are nearby but later on no such effort is made. The structure of metadata about the free data blocks is shown in figure 8.

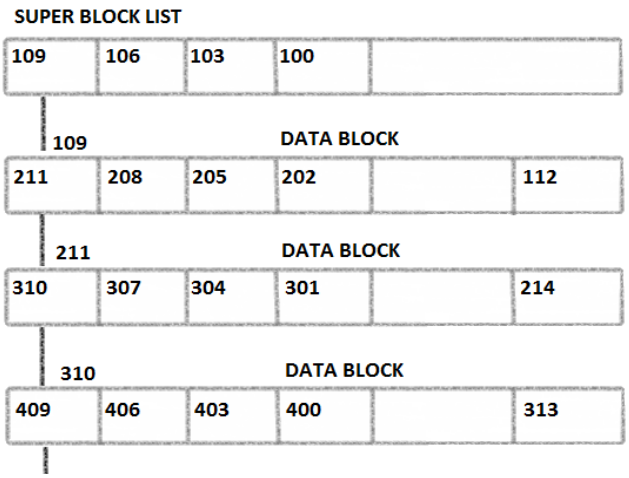

Figure 8. Free data block management.

Extended 2 file system on-disk layout is strongly influenced by BSD file system and is almost similar to Extended file system. Ext2 file system is divided into block groups, which contain a fixed number of blocks where blocks are fixed sized number of sectors. Block groups immediately follow the boot sector and are numbered from 0 onwards. Every block group contains a Super block (1 block in size), Group descriptors (n blocks in size), Data block bitmap (1 block in size), I-node bitmap (1 block in size), I-node table (n blocks in size) and data blocks (n blocks in size). The typical structure of Ext2 file system is shown in figure 9.

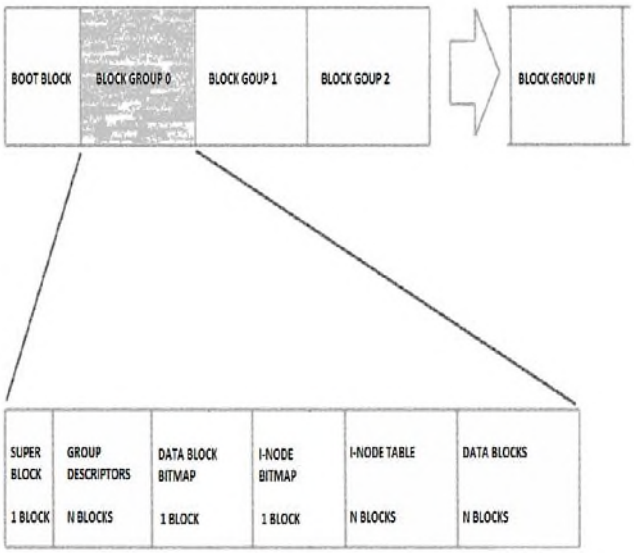

Figure 9. Ext2 data structures.

Using block groups has 3 advantages:

- Each block group contains a redundant copy of Super block and block group descriptors that actually define the file system. As such, it is easy to recover if any Super block gets corrupted.
- This arrangement gives good performance by reducing the distance between the I-node table and the data blocks which reduces the head seeks during file I/O.
- It reduces fragmentation by keeping the data blocks belonging to a file in same block group.

d) *Super Block*

The Super block of Ext2 contains following information:

- Magic number which validates whether the block is Super block or not.
- Revision level which indicates features it supports.
- Mount count and maximum mount count.
- Block group number that holds this copy of Super block.
- Block size fixed at volume creation.
- Blocks per group fixed at volume creation.
- Free blocks which indicates number of free blocks.
- Free I-nodes which indicates number of free I-nodes.
- First I-node which indicates the root '/' I-node.

Ext2 Super block does not contain information regarding the list of free data blocks and I-nodes. This information is individually maintained by Block bitmap and I-node bitmap of block group.

e) *Block Group Descriptor*

Block group descriptor consumes one block and contains following information:

- The block number of block allocation bitmap for this block group used during block allocation and de allocation.
- The block number of I-node allocation bitmap for this block group used during I-node allocation and de allocation.
- I-node table which contains the starting block number of I-node table for this block group.
- Number of free blocks in group.
- Number of free I-nodes in group.
- Number of directories in group.

Only the first copy of Super block and group descriptors is updated by Ext2 file system while for other block groups it is left untouched. When a consistency check is executed, the information is copied on other block groups.

f) *Block & I-node Bitmaps*

Both of these bitmaps occupy one block each and number of blocks they address depends upon fixed number of blocks per group. In these bitmaps, each bit corresponds to a block (or I-node) of group and its state indicates whether it is allocated or not.

g) *I-node Table*

I-node table is an array of fixed sized I-nodes and occupy many blocks depending upon the size of I-node, total number of I-nodes in a group and block size all indicated by Super block.

Ext2 I-node is almost same as that of Extended I-node in that it uses multiple levels of indirection but Ext2 directories contain variable length entries unlike Ext file system directory. Each directory entry contains I-node number, name length and name of file.

Ext3 on-disk data structures are identical to those of an Ext2 file system. As a matter of fact, if an Ext3 file system has been cleanly un-mounted, it can be remounted as an Ext2 file system, conversely, creating a journal of Ext2 file system and remounting it as Ext3 is simple and fast operation.

V. Hierarchical File Systems

Macintosh File System (MFS) was introduced around 1983 with first Mac computer. MFS was optimized to be used on very small and slow media [35]. With the introduction of larger media, the time taken to display the contents of a folder was a concern as MFS used a single flat file to store all of the file and directory listing information. As such, the system had to do a complete search of this file in order to build a list of files stored in a particular folder.

Hierarchical File System (HFS), also called Mac OS Standard, was introduced in 1985 to mitigate this problem. HFS replaced the flat file of MFS with Catalog File which uses B-tree structure that could be searched very quickly regardless of size. HFS was introduced with 20 MB hard disk drive and was hard coded into 128 KB ROM. HFS file system divides the volume in 512 bytes long sectors and allocates to files allocation blocks which contain one or more consecutive sectors. HFS contains 5 data structures that make up the volume:

- Boot blocks occupy sector 0 and 1 of system and contain system startup information.
- Master Directory Block (MDB) occupies sector 2 and defines the volume layout and other information like location and size of other structures. MDB is duplicated at opposite end of the volume in second to last sector. This is used to recover the volume in case of corruption and is only updated only when either Catalog file or Extent Overflow file size increases.
- Volume Bitmap starts at sector 3 and keeps track of which allocation blocks are free. The size of Volume bitmap depends upon the size of the volume.
- Catalog file is a B-tree that contains records for all files and folders which exist on the volume. Files and folders are uniquely identified in Catalog file by Catalog Node ID (CNID). Each node represents a file or folder and may contain any 2 types of records among the 4 possible types. For a file node, a File Thread Record stores filename and CNID of its parent directory and a File Record stores 16 byte attributes used by Finder, timestamps, its CNID, first 3 extents of file for both data and resource fork, and pointer to first data and resource fork extent records in Extent Overflow file (in case it has any). For a directory node, a Directory Thread Record stores name of directory and CNID of parent directory and a Directory Record stores 16 byte attributes used by Finder, timestamps, its CNID and number of files stored in it.
- Extent Overflow file is a B-tree structure file that contains extra extents pertaining to any file if the initial 3 extents of that file record in Catalog file are used up. Later versions allowed bad blocks to be recorded as extents.

An extent is a contiguous range of allocation blocks allocated to some fork, represented by a pair of numbers; the first allocation block number and number of allocation blocks.

The general on disk layout of HFS file system is shown in figure 10.

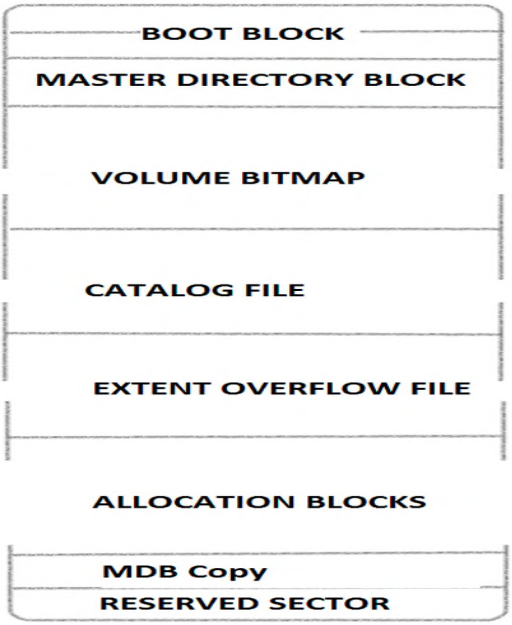

Figure 10. HFS On-Disk Layout.

Under HFS (also in HFS+) files are not monolithic and do not consist of one single element [36]. They may be composed of two or more pieces, called Forks. NTFS also supports this concept by supporting multiple data streams in general and multiple values for same attribute types identified by names. HFS files have 2 named forks (Data & Resource) and can have logically any number of un-named forks. A Data fork contains the actual data pertaining to the file like text for word processor, etc. A Resource fork contains metadata pertaining to the file like icon, preview picture, etc. In other words, Data fork is used to store the unstructured data while Resource fork is used to store the structured data. The Resource fork was designed to store metadata that would be used by GUI. HFS+ supports any arbitrary number of custom named forks in addition to data & resource forks.

As the Catalog file stores all the file and directory records in single data structure, only one program can write to this structure at a time, forcing other programs to wait in a queue to get their turn. This raises both a performance and reliability issue. Also, due to 16 bit pointers used to address allocation blocks, HFS is able to address only 65535 allocation

blocks. This means, a minimum size of allocation block can be 1/65535th of volume size. This means only 65535 files are possible and high internal fragmentation on large volumes.

Hierarchical File System Plus (HFS+), also called Mac OS Extended, was introduced in 1998 to overcome problems of HFS and has become the primary file system used in Mac computers [37]. HFS+ is an improved version of HFS supporting larger files and volumes by using 32-bit allocation block addresses and Unicode for filenames. It also supports multiple named forks for files, Journaling, inline attribute data records, access control list based file security and compatibility with file permission models on other platforms such as Windows.

Like HFS, HFS+ divides volume into 512 byte sectors and groups them into allocation blocks (usually 8) to be allocated to a file. Allocation blocks are addressed by 32-bit pointers [38]. In HFS+ volume everything is a part of one or more allocation blocks with possible exception Alternate Volume Header, unlike HFS were Boot blocks, Master Directory Block and Volume Bitmap are not part of any allocation block. To reduce file fragmentation, contiguous allocation blocks called Clumps are allocated to files. The number of allocation blocks per Clump is fixed and is specified in Volume Header. The first 1024 bytes and last 512 bytes of volume are reserved. The Volume Header is located immediately after first 1024 bytes and is fixed. The Alternate Volume Header which is replica of Volume Header is located at 1024 bytes before the end of volume and is also fixed. The on-disk layout of HFS+ volume is shown in figure 11.

Volume Header is equivalent of Master Directory Block of HFS. It stores timestamps, number of files on volume, location of other structures on volume, size of allocation blocks, size of clumps, etc. When a volume is formatted with HFS+ file system, it leads to the creation of 5 special files in addition to reserved allocation blocks, Volume Header and Alternate Volume Header.

a) *Allocation File*

Allocation file keeps track of which allocation locks are free and which are in use by representing every block by bit. It is equivalent to Volume Bitmap of HFS. The main difference between Volume Bitmap and Allocation File is that Allocation file is a regular file which can exist anywhere on volume, shrink or grow in size and need not to be contiguous while Volume Bitmap always resides in reserved area and its size is fixed. The location of first extent of Allocation file is stored in Volume Header. This architecture of Allocation file induces flexibility in HFS+ file system not found in HFS.

b) *Catalog File*

Catalog file describes every file and folder of the volume including the special files and the hierarchy in the volume. It is similar to Catalog file of HFS. The Catalog file is organized as a B-tree to allow quick and efficient searches through a large hierarchy. This file contains vital information about every file and folder along with the catalog information. The main difference between the records in HFS and HFS+ Catalog file is that in HFS+ the nodes of B-tree pertaining to files and folders contain more information and can have varying size unlike HFS. The location of first extent of Catalog file is stored in Volume Header. Catalog file contains Header node, Index nodes, Leaf nodes and if necessary Map nodes. Each file or folder in Catalog file is given a unique Catalog Node ID (CNID). For folder, CNID is called FolderID and for files FileID. Like HFS Catalog nodes, HFS+ Catalog nodes also store File Record and File Thread Record for files and Folder Record and Folder Thread Record for folders in addition to some more additional information. The main difference between HFS File Record and Directory Record and HFS+ File Record and Folder Record is that in HFS the records contain information about first 3 extents belonging to the file or folder while in HFS+ it is 8.

c) *Extent Overflow File*

Special files only have one fork i.e. Data fork. The Catalog file does not store any extent for special files rather first 8 extents of special files are stored in Volume Header. User files can have both data and resource fork and if necessary other named forks. The first 8 extents of both data and resource forks for user files are stored in Catalog file. In both types of files, if there is need for additional extents for data and resource fork and/or for named forks, the extents are recorded in Extent Overflow file. It is a B-tree structured file that stores standard additional forks' extents and named forks' extents for user files. It does not store for itself any additional data fork extent.

d) *Bad Block File*

Bad Block file is used to mark and record the areas of the volume that contain bad blocks. The Extent Overflow file is used to hold information about the Bad Block file extents.

e) *Attributes File*

An Attributes file is a special file which does not have an entry in Catalog file. An Attributes file is a complex file. A volume can have no Attributes file in which case its description in Volume Header for allocation blocks is 0. Attributes file is a B-tree structured file where nodes can contain records known as Attributes. An Attributes file can have 3 types of attributes:

- Inline Data Attributes which contain small attributes.
- Fork Data Attributes which contain references to a maximum of 8 extents.

- Extended Attributes which contain references to 8 more extents for data attributes.

f) *Startup File*

Startup file is a special file used to hold information needed when booting a system that does not have built-in ROM support for HFS plus. The boot loader can find the location of Startup file from Volume Header which contains the first 8 extents of Startup file. Startup file should not have any additional extents for data fork as it will complicate things for boot loader.

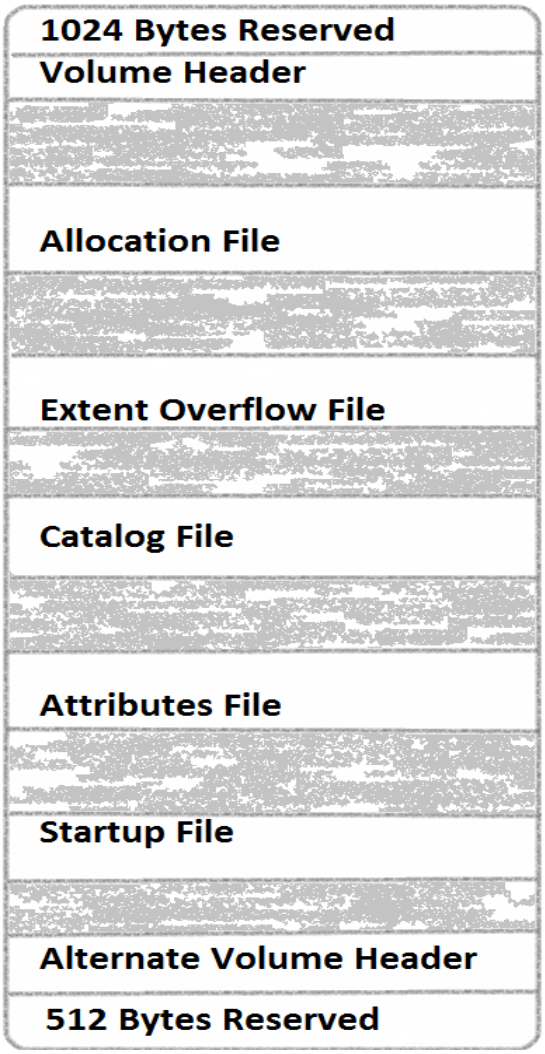

Figure 11. HFS+ On-Disk Layout.

VI. Discussion and Conclusion

We observed that the on-disk layout of file systems reviewed in this paper were objective specific. In case of FAT file systems, the new versions were developed to address the issue of large file size and large volume size support. Similarly in case of Hierarchical file systems; the augmented versions addressed Unicode support in filenames, relocatable system metadata structures and large file and volume size. In both cases, the actual design remained the same. We also observed, in case of NTFS that the design was drafted from scratch which yielded into an elegant file system having almost all features which a modern file system should have. Further, in case of Extended file systems, we observed large drift in on-disk layout from Extended file system to Extended 2 file system to increase performance and reliability. Again, the design of Extended 3 file system which is mount compatible with Extended 2 file system is an excellent example of flexibility in design of Extended 2 file system. We also observed some similarity in heterogeneous file systems. The concept of treating everything residing on the volume as a file is the basic building block of both NTFS and Hierarchical file systems.

References Références Referencias

1. Grochowski, E. (1998), "Emerging trends in data storage on magnetic hard disk drives", Datatech, pages 11–16, Sep 1998.
2. Dahlin, M.D. (1996), "The Impact of Trends in Technology on File System Design", University of California, Berkeley.
3. Gibson, G.A. (1992), "Redundant Disk Arrays: Reliable, Parallel Secondary Storage", ACM Distinguished Dissertations. MIT Press, Cambridge, Massachusetts.
4. Giampaolo, D., "Practical File System Design with the Be File System", Be, Inc.
5. Zadok, E., Iyer, R., Joukov, N., Sivathanu, G. and Wright, C.P. (2006), "On Incremental FileSystem Development", ACM Transactions onStorage (TOS),2(2):161–196, May 2006
6. DEC Tape, http://en.wikipedia.org/wiki/DECtape, Accessed on November 2010.
7. http://www.pdp8.net, Accessed on November 2010
8. http://en.wikipedia.org/wiki/Gary_Kildall, Accessed on November 2010.
9. http://en.wikipedia.org/wiki/PL/M, Accessed on November 2010.
10. http://www.digitalresearch.biz/cpm.htm, Accessed on November 2010.
11. http://en.wikipedia.org/wiki/Tim_Paterson, Accessed on November 2010.
12. http://en.wikipedia.org/wiki/86-DOS, Accessed on November 2010.
13. "The Man Who Could Have Been Bill Gates", http://www.businessweek.com/magazine/content/04_43/b3905109_mz063.htm, Accessed on November 2010.
14. FAT32 File System Specification, http://microsoft.com/whdc/system/platform/firmware/fatgen.mspx, Accessed in 2009.
15. Bhat, W.A., Quadri, S.M.K., (2010), "Review of FAT Data Structure of FAT32 file system",

Oriental Journal of Computer Science & Technology, Volume 3, No 1.
16. Extended FAT File System, http://msdn.microsoft.com/enus/library/aa914353.aspx, Accessed on November 2010.
17. Kwon, M.S., Bae, S.H., Jung, S.S., Seo, D.Y. and Kim, C.K., (2005), "KFAT: Log-based Transactional FAT File system for Embedded Mobile Systems", In Proceedings of 2005 US-Korea Conference, ZCTS-142, 2005.
18. Alei, L., Kejia, L., Xiaoyong, L., (2007), "FATTY : A reliable FAT File System", In Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, Pages: 390-395, 2007.
19. Microsoft Corporation, "Transaction-Safe FAT File System",http://msdn2.microsoft.c0m/en-us/library/aa911939.aspx, Accessed in 2010.
20. Duncan, R., (1989), "Design goals and implementation of the new High Performance File System", Microsoft Systems Journal, September 1989 v4 n5 p1 (13).
21. Russinovich, M., Solomon, D.A. and Ionescu, A. (2009), "File Systems", Windows Internals (5th edition), Microsoft Press. ISBN 0735625301.
22. Nagar, R., (1997), "Windows NT File System Internals : A Developer's Guide", O'Reilly. ISBN 9781565922495.
23. NTFS Concepts http://www.priscilla.com/Courses/ComputerForensics/pdfslides/03-NTFSConcepts.pdf, Accessed on November 2010
24. NTFS Documentation, http://www.scribd.com/doc/2187280/NTFS-Documentation, Accessed on November 2010
25. http://www.ntfs.com, Accessed on November 2010.
26. Probert, D.B., "Windows Kernel Development", Microsoft Corporation, http://i-web.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures/08-NTFS/NTFS.ppt, Accessed on November 2010
27. http://www.minix3.org/, Accessed on November 2010.
28. http://en.wikipedia.org/wiki/MINIX_file_system, Accessed on November 2010.
29. The Virtual File System in Linux, http://www.linux.it/~rubini/docs/vfs/vfs.html, Accessed on November 2010
30. Bach, M.J. (1986), "The Design of the UNIX Operating System", Prentice Hall, 1986.
31. Card, R., Ts'o, T. and Tweedie, S., (1994), "Design and Implementation of the Second Extended Filesystem", In Proceedings of the First Dutch International Symposium on Linux, Amsterdam, Holland, 1994.
32. McKusick, M.K., Joy, W.N., Leffler, S.J. and Fabry, R.S., (1984), "A Fast File System for UNIX", In ACM Transactions of computer Systems, Vol 2, No. 3, 1984.
33. Tweedie, S. (1998), "Journaling the Linux ext2fs filesystem", In LinuxExpo '98,1998.
34. Ts'o, T. (2006), "Proposal and plan for ext2/3 future development work". Linux kernel mailing list. http://lkml.org/lkml/2006/6/28/454
35. HFS,http://en.wikipedia.org/wiki/Hierarchical_File_System, Accessed on November 2010.
36. HFS Plus, http://en.wikipedia.org/wiki/HFS_Plus, Accessed on November 2010.
37. TN1150, "HFS Plus Volume Format", http://developer.apple.com/library/mac/#technotes/tn/tn1150.html, Accessed on November 2010
38. Mac OS X: Mac OS Extended format (HFS Plus) volume and file limits, http://support.apple.com/kb/HT2422, Accessed on November 2010
39. Overview of FAT, HPFS, and NTFS File Systems, http://support.microsoft.com/kb/100108, , Accessed on November 2010.
40. Daily, S. (1996), "NTFS vs. FAT." Windows NT Magazine October 1996: 95.
41. NTFS Directories and Files, http://www.pcguide.com/ref/hdd/file/ntfs/files.htm, Accessed on November 2010.
42. Janes, M., "Progression of Linux File Systems", http://mjanes.public.iastate.edu/Engl314/indiv_doc.pdf
43. Anjoy, R.G., Chakraborty, S.K., (2009), "Feature Based Comparison of Modern File Systems", http://www.idt.mdh.se/kurser/ct3340/ht09/ADMINISTRATION/IRCSE09_submissions/ircse09_submission_16.pdf
44. Mitchell, S. (1997), "Inside the Windows 95 File System", O'Reilly. ISBN 156592200X.
45. Tanenbaum, A.S., Woodhull, A.S., (2006). "File Systems", Operating Systems: Design and Implementation (3rd edition.). Prentice Hall. ISBN 0131429388.
46. Pate, S.D. (2003). "UNIX Filesystems: Evolution, Design, and Implementation", Wiley. ISBN 0471164836.
47. Leffler, S.J. and McKusick, M.K., "The design and implementation of the 4.3BSD UNIX operating System Answer Book", Addison-Wesley, ISBN 0201546299
48. Bar, M., "Linux File Systems", McGraw-Hill, ISBN 0072129557

49. Bovet, D.P. and Cesati, M. (2005), "Understanding the Linux Kernel",O´Reilly Media, 3rd edition, 2005. ISBN 0596005652.
50. Silberschatz, A., Galvin, P.B. and Gagne, G., (2004), "Storage Management", Operating System Concepts , 7th Edition, Wiley. ISBN 0471694665.

This page is intentionally left blank