**FAT Introduction**

File Allocation Table (FAT) file system. This file system was used on all versions of MS-DOS and PC-DOS, and on early versions of Windows; it is still used on floppy disks formatted by Windows and some other systems. Modified versions are also still supported by Windows on hard disks, if required.

The FAT file system is heavily based on the *file map* model in terms of its on-disk layout; that model was around for many years before Microsoft inherited the initial FAT file system from the original writers of DOS (Seattle Computer Products). It is a reasonably simple, reasonably robust file system.

There are three basic variants of the FAT file system, which differ mainly in the construction of the actual file allocation table. Floppy disks and small hard disks usually use the *12-bit* version, which was superseded by the *16-bit* version as hard disks became bigger. This in turn was superseded by the *32-bit* version as disks became bigger still. We shall concentrate on the 16-bit version, since the 12-bit version can be tricky for beginners, and the 32-bit version is more complex than needed for this tutorial.

**Overview**

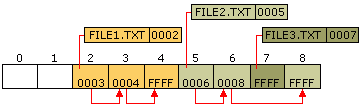
Any disk is made up of *surfaces* (one for each head), *tracks* and *sectors*. However, for simplicity, we can consider a disk as a simple storage area made up just of a number of sectors. Further, these sectors are considered to be numbered consecutively, the first being numbered 0, the second numbered 1, etc.; we will not worry about the physical location of any sector on the actual disk. Because we want to emphasise that the location of a sector is irrelevant to the actual disk structure, and because sectors have their own numbers within each track, we shall call these sectors *blocks* from now on; as previously stated, they form a linear, densely numbered list.

All blocks are the same size, 512 bytes, on practically all FAT file systems. Howewer, large disks can have too many blocks for comfort, so blocks are sometimes grouped together in pairs (or fours, or eights, etc...); each such grouping is called an *allocation unit*. The FAT file system actually works in allocation units, not blocks, but for simplicity we shall assume in the description below that each allocation unit contains exactly one block, which means that we can use the terms interchangeably.

The precise manner in which data is organised on a hard disk drive is determined by the file system used. File systems are generally operating system dependent. However, since it is the most widely used PC operating system, most other operating systems’ file systems are at least read-compatible with Microsoft Windows.

The FAT file system was first introduced in the days of MS-DOS way back in 1981. The purpose of the File Allocation Table is to provide the mapping between clusters – the basic unit of logical storage on a disk at the operating system level – and the physical location of data in terms of cylinders, tracks and sectors – the form of addressing used by the drive’s hardware controller.

The FAT contains an entry for every file stored on the volume that contains the address of the file’s starting cluster. Each cluster contains a pointer to the next cluster in the file, or an end-of-file indicator at (0xFFFF), which indicates that this cluster is the end of the file. The diagram shows three files: File1.txt uses three clusters, File2.txt is a fragmented file that requires three clusters and File3.txt fits in one cluster. In each case, the file allocation table entry points to the first cluster of the file.



The first incarnation of FAT was known as FAT12, which supported a maximum partition size of 8MB. This was superseded in 1984 by FAT16, which increased the maximum partition size to 2GB. FAT16 has undergone a number of minor modifications over the years, for example, enabling it to handle file names longer than the original limitation of 8.3 characters. FAT16’s principal limitation is that it imposes a fixed maximum number of clusters per partition, meaning that the bigger the hard disk, the bigger the cluster size and the more unusable space on the drive. The biggest advantage of FAT16 is that it is compatible across a wide variety of operating systems, including Windows 95/98/Me, OS/2, [Linux](http://www.knoppix.net/) and some versions of UNIX.

Dating from the Windows 95 OEM Service Release 2 (OSR2), Windows has supported both FAT16 and FAT32. The latter is little more than an extension of the original FAT16 file system that provides for a much larger number of clusters per partition. As such, it offers greatly improved disk utilisation over FAT16. However, FAT32 shares all of the other limitations of FAT16 plus the additional one that many non-Windows operating systems that are FAT16-compatible will not work with FAT32. This makes FAT32 inappropriate for dual-boot environments, although while other operating systems such as Windows NT can’t directly read a FAT32 partition, they can read it across the network. It’s no problem, therefore, to share information stored on a FAT32 partition with other computers on a network that are running older versions of Windows.

With the advent of Windows XP in October 2001, support was extended to include the NTFS. NTFS is a completely different file system from FAT that was introduced with first version of Windows NT in 1993. Designed to address many of FAT’s deficiencies, it provides for greatly increased privacy and security. The Home edition of Windows XP allows users to keep their information private to themselves, while the Professional version supports access control and encryption of individual files and folders. The file system is inherently more resilient than FAT, being less likely to suffer damage in the event of a system crash and it being more likely that any damage is recoverable via the chkdsk.exe utility. NTFS also journalises all file changes, so as to allow the system to be rolled back to an earlier, working state in the event of some catastrophic problem rendering the system inoperable.

FAT16, FAT32 and NTFS each use different cluster sizes depending on the size of the volume, and each file system has a maximum number of clusters it can support. The smaller the cluster size, the more efficiently a disk stores information because unused space within a cluster cannot be used by other files; the more clusters supported, the larger the volumes or partitions that can be created.

The table below provides a comparison of volume and default cluster sizes for the different Windows file systems still commonly in use:

|  |  |  |  |
| --- | --- | --- | --- |
| **Volume Size** | **FAT16**  **Cluster Size** | **FAT32**  **Cluster Size** | **NTFS**  **Cluster Size** |
| 7MB – 16MB | 2KB | Not supported | 512 bytes |
| 17MB – 32MB | 512 bytes | Not supported | 512 bytes |
| 33MB – 64MB | 1KB | 512 bytes | 512 bytes |
| 65MB – 128MB | 2KB | 1KB | 512 bytes |
| 129MB – 256MB | 4KB | 2KB | 512 bytes |
| 257MB – 512MB | 8KB | 4KB | 512 bytes |
| 513MB – 1GB | 16KB | 4KB | 1KB |
| 1GB – 2GB | 32KB | 4KB | 2KB |
| 2GB – 4GB | 64KB | 4KB | 4KB |
| 4GB – 8GB | Not supported | 4KB | 4KB |
| 8GB – 16GB | Not supported | 8KB | 4KB |
| 16GB – 32GB | Not supported | 16KB | 4KB |
| 32GB – 2TB | Not supported | Not supported | 4KB |

ReFS

Resilient File System (ReFS), codenamed "Protogon", is a Microsoft proprietary file system introduced with Windows Server 2012 with the intent of becoming the "next generation" file system after NTFS.

## Platform

**Servers** – Windows Server 2012

## Description

Resilient File System (ReFS) is a new local file system. It maximizes data availability, despite errors that would historically cause data loss or downtime. Data integrity ensures that business critical data is protected from errors and available when needed. Its architecture is designed to provide scalability and performance in an era of constantly growing data set sizes and dynamic workloads.

The key features of ReFS are:

* **Integrity**: ReFS stores data so that it is protected from many of the common errors that can cause data loss. File system metadata is always protected. Optionally, user data can be protected on a per-volume, per-directory, or per-file basis. If corruption occurs, ReFS can detect and, when configured with Storage Spaces, automatically correct the corruption. In the event of a system error, ReFS is designed to recover from that error rapidly, with no loss of user data.
* **Availability**: ReFS is designed to prioritize the availability of data. With ReFS, if corruption occurs, and it cannot be repaired automatically, the online salvage process is localized to the area of corruption, requiring no volume down-time. In short, if corruption occurs, ReFS will stay online.
* **Scalability**: ReFS is designed for the data set sizes of today and the data set sizes of tomorrow; it’s optimized for high scalability.
* **App Compatibility**: To maximize AppCompat, ReFS supports a subset of NTFS features plus Win32 APIs that are widely adopted.
* **Proactive Error Identification**: The integrity capabilities of ReFS are leveraged by a data integrity scanner (a “scrubber”) that periodically scans the volume, attempts to identify latent corruption, and then proactively triggers a repair of that corrupt data.

EXT 2/3/4 JFS, XFS

The original Linux system used a simple file system that mimicked the functionality of the Unix file system. In this tutorial we will discuss basic file system used in Linux.

###### **The ext File system**

The original file system introduced with the Linux operating system is called the extended file system (or just ext for short). It provides a basic Unix- like file system for Linux, using virtual directories to handle physical devices, and storing data in fixed-length blocks on the physical devices.

The ext file-system uses a system called inodes to track information about the files stored in the virtual directory. The inode system creates a separate table on each physical device, called the inode table , to store the file information. Each stored file in the virtual directory has an entry in  the inode table. The extended part of the name comes from the additional data that it tracks on each file, which consists of:

* The file name
* The file size
* The owner of the file
* The group the file belongs to
* Access permissions for the file
* Pointers to each disk block that contains data from the file

Linux references each inode in the inode table using a unique number (called the inode number ), assigned by the file system as data files are created. The file system uses the inode number to identify the file rather than having to use the full file name and path.

###### **The ext2 File system**

The original ext file system had quite a few limitations, such as limiting files to only 2GB in size. Not too long after Linux was first introduced, the ext file system was upgraded to create the second extended file system, called ext2 .

As you can guess, the ext2 file system is an expansion of the basic abilities of the ext file system, but maintains the same structure. The ext2  file system expands the inode table format to track additional information about each file on the system.

The ext2 inode table adds the created, modified, and last accessed time values for files to help system administrators track file access on the  system. The ext2 file system also increases the maximum file size allowed to 2TB (then in later versions of ext2, that was increased to 32TB) to help  accommodate large files commonly found in database servers.

In addition to expanding the inode table, the ext2 file system also changed the way in which files are stored in the data blocks. A common  problem with the ext file system was that as a file is written to the physical device, the blocks used to store the data tend to be scattered throughout the device (called fragmentation ). Fragmentation of data blocks can reduce the performance of the file system, as it takes longer to search the storage device to access all of the blocks for a specific file.

The ext2 file system helps reduce fragmentation by allocating disk blocks in groups when you save a file. By grouping the data blocks for a file, the file system doesn’t have to search all over the physical device for the data blocks to read the file. The ext2 file system was the default file system used in Linux distributions for many years, but it, too, had its limitations. The inode table, while a  nice feature that allows the file system to track additional information about files, can cause problems that can be fatal to the system. Each time the file system stores or updates a file, it has to modify the inode table with the new information. The problem is that this isn’t always a fluid action.

If something should happen to the computer system between the file being stored and the inode table being updated, the two would become out  of sync. The ext2 file system is notorious for easily becoming corrupted due to system crashes and power outages. Even if the file data is stored just  fine on the physical device, if the inode table entry wasn’t completed, the ext2 file system wouldn’t even know that the file existed!  It wasn’t long before developers were exploring a different avenue of Linux file systems.

###### **Journaling File systems**

Journaling file systems provide a new level of safety to the Linux system. Instead of writing data directly to the storage device and then updating the  inode table, journaling file systems write file changes into a temporary file (called the journal) first. After data is successfully written to the storage device and the inode table, the journal entry is deleted.

If the system should crash or suffer a power outage before the data can be written to the storage device, the journaling file system just reads through the journal file and processes any uncommitted data left over.

There are three different methods of journaling commonly used in Linux, each with different levels of protection. These are shown in  below Table.

**Journaling File system Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Data mode | Both inode and file data are journaled. Low risk of losing data, but poor performance. |
| Ordered mode | Only inode data written to the journal, but not removed until file data is successfully written. Good compromise between performance and safety. |
| Writeback mode | Only inode data written to the journal, no control over when the file data is written. Higher risk of losing data, but still better than not using journaling. |

**Limitation:**

The data mode journaling method is by far the safest for protecting data, but it is also the slowest. All of the data written to a storage device must  be written twice, once to the journal, then again to the actual storage device. This can cause poor performance, especially for systems that do a lot  of data writing.  Over the years, a few different journaling file systems have appeared in Linux. The following sections describe the popular Linux journaling  file systems available.

# The Third Extended Filesystem

|  |  |
| --- | --- |
|  | *Ext3* (the third extended filesystem) is the most commonly used *[filesystem](http://www.linfo.org/filesystem.html)* on [Linux](http://www.linfo.org/linuxdef.html). It is basically an extension of [*ext2*](http://www.linfo.org/ext2.html) to which a *journaling* capability has been added.  A filesystem is a way of organizing [data](http://www.linfo.org/data.html) on a [computer](http://www.linfo.org/computer.html) system. On Linux and other [Unix-like](http://www.linfo.org/unix-like.html) [operating systems](http://www.linfo.org/operating_system.html), the most obvious part of the main filesystem is the hierarchy of [directories](http://www.linfo.org/directory.html) that starts with the [*root directory*](http://www.linfo.org/root_directory.html) (designated by a [forward slash](http://www.linfo.org/forward_slash.html)), which contains a series of subdirectories, each of which, in turn, may contain further subdirectories, etc.  Ext2 is the most basic and the most *portable* of the *native* Linux filesystems. A native Linux filesystem is one that was developed specifically for Linux, or a *foreign*filesystem (i..e., one that was first developed for some other operating system) that was rewritten so that it would have functions and performance on Linux comparable or superior to those of original Linux filesystems. Ext2 is the most portable of the native Linux operating systems because it is easiest to transfer data to and from other filesystems.  Ext2 was developed as an improved version of the *extended file system*, commonly referred to as *ext*, Linux's first native filesystem, and was incorporated into the [kernel](http://www.linfo.org/kernel.html) (i.e., core of the operating system) from January 1993. It remained *the* standard filesystem on Linux for a number of years, and it is still in widespread use.  A *journaling filesystem* is a filesystem that maintains a special [file](http://www.linfo.org/file.html) called a *journal* that is used to repair any inconsistencies in the filesystem that occur as the result of an *unclean shutdown* of a computer and thus always maintains an internal consistency. Such shutdowns are usually due to an interruption of the power supply or to a software [crash](http://www.linfo.org/crash.html) that cannot be resolved without rebooting.  During rebooting from an unclean shutdown with an ext2 filesystem, it is necessary for the *e2fsck* [program](http://www.linfo.org/program.html) to run a consistency check and repair any inconsistencies before the filesystem can be [*mounted*](http://www.linfo.org/mounting.html) (i.e., logically attached to the system) and the [booting](http://www.linfo.org/boot.html) can be completed. The wait can be very long, possibly several hours in the case of filesystems with hundreds of [gigabytes](http://www.linfo.org/gigabyte.html). Ext3, in contrast, usually does not require any checking of the filesystem during rebooting after an improper shutdown because of its guaranteed internal consistency.  Ext3 can also do a substantially better job of protecting data integrity than can ext2 in the event of an unclean system shutdown. In fact, it offers a choice of how carefully to protect data: (1) ensuring that data is consistent with the state of the filesystem (which eliminates corrupted data from appearing in files that were rewritten after an unclean shutdown) or (2) keeping the filesystem consistent but allowing for damage to data (which can result in slightly increased speed in some situations). The former is the default.  In addition, ext3 is often faster than ext2, despite the fact that it writes some data more than once. This is because its journaling optimizes [hard disk drive](http://www.linfo.org/hdd.html) (HDD) [head](http://www.linfo.org/magnetic_head.html) motion. There is also a choice of three journaling modes that provide trade-offs between optimizing speed and maximizing data integrity.  Further contributing to ext3's high degree of reliability is the extremely high degree of reliability of the underlying ext2 technology. An additional feature is the ability for ext2 [partitions](http://www.linfo.org/partition.html) (i.e., logically independent sections of HDDs) to be easily converted to ext3 and vice-versa and without any need for backing up of data and repartitioning. Moreover, because ext3 has the same HDD formating as ext2, it can, if needed, likewise utilize the extensively tested and highly reliable e2fsck program to check filesystem consistency and repair any errors.  Ext3 was originally written by Dr Stephen C. Tweedie for the 2.2 kernel. It was added to the kernel beginning with version 2.4.15, which was released in November 2001, and it has remained the default filesystem on Red Hat and some other [distributions](http://www.linfo.org/distributions_list.html) despite the development of other native Linux journaling filesystems (the most popular of which is reiserfs). The lack of a journaling filesystem had often been cited as a major factor holding back the widespread use of Linux at the enterprise level. The Ext4 File System The ext4 file system is a scalable extension of the ext3 file system. Ext4 uses extents (as opposed to the traditional block mapping scheme used by ext2 and ext3), which improves performance when using large files and reduces metadata overhead for large files. In addition, ext4 also labels unallocated block groups and inode table sections accordingly, which allows them to be skipped during a file system check. This makes for quicker file system checks, which becomes more beneficial as the file system grows in size.  The ext4 file system features the following allocation schemes:   * Persistent pre-allocation * Delayed allocation * Multi-block allocation * Stripe-aware allocation   Because of delayed allocation and other performance optimizations, ext4's behavior of writing files to disk is different from ext3. In ext4, when a program writes to the file system, it is not guaranteed to be on-disk unless the program issues an **fsync()** call afterwards.  By default, ext3 automatically forces newly created files to disk almost immediately even without **fsync()**. This behavior hid bugs in programs that did not use **fsync()** to ensure that written data was on-disk. The ext4 file system, on the other hand, often waits several seconds to write out changes to disk, allowing it to combine and reorder writes for better disk performance than ext3.  **ReiserFS**  **ReiserFS** is a general-purpose, journaled computer file system formerly designed and implemented by a team at Namesys led by Hans Reiser. **ReiserFS** is currently supported on Linux.  **Strengths and Weaknesses**  ReiserFS has a number of strengths:   Generally higher performance for all file sizes.   Wastes less space. There is no static inode space allocation; small files are packed together.   Much higher performance for large directories, even compared to other balanced tree filesystems.   Uses B\* balanced trees, whereas other balanced tree filesystems use obsolete B+ trees.   Partitions can be resized while in use.   Extremely fast recovery in the event of unplanned machine shutdown (loss of power). Rather than taking many seconds or even minutes to check the filesystem as e2fsck does, ReiserFSck takes only seconds.  Of course, it also has its weaknesses:   * The software is still relatively new. You may want to hold off implementing in production systems, although results to date have been good. * No quota support (yet). * Dump does not work (yet). * Kernel-based NFS is not very stable yet. * Certain programs (qmail, for one) have issues with ReiserFS. Patches are available in many cases. * It is not possible to change live filesystems back and forth from ext2 to ReiserFS. To make the change, you need to back up your data, create the file system, and restore. * Occasionally, you may experience a stall condition (*read starves*). When large writes are scheduled all at once, reads can starve. A fix for this is in the works, and the later your ReiserFS patch, the better this situation is handled.  The XFS File System XFS is a highly scalable, high-performance file system which was originally designed at Silicon Graphics, Inc. It was created to support extremely large filesystems (up to 16 exabytes), files (8 exabytes) and directory structures (tens of millions of entries).  **Main Features**  XFS supports *metadata journaling*, which facilitates quicker crash recovery. The XFS file system can also be defragmented and enlarged while mounted and active. In addition, Red Hat Enterprise Linux 6 supports backup and restore utilities specific to XFS.  **Allocation Features**  XFS features the following allocation schemes:   * Extent-based allocation * Stripe-aware allocation policies * Delayed allocation * Space pre-allocation   Delayed allocation and other performance optimizations affect XFS the same way that they do ext4. Namely, a program's writes to an XFS file system are not guaranteed to be on-disk unless the program issues an **fsync()** call afterwards. |