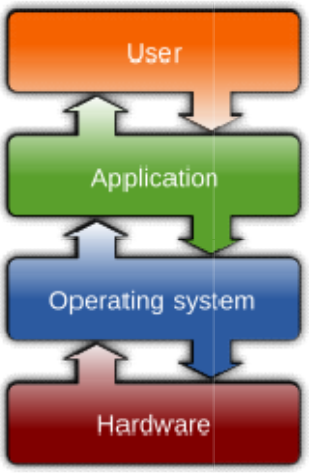
SUBJECT –DATABASE TECHNOLOGY
File system [unit-i]

Operating systems



In computing, a **file system** or **filesystem** (often abbreviated to **FS** or **fs**) governs file organization and access. A *local* file system is a capability of an operating system that services the applications running on the same computer.[1][2] A distributed file system is a protocol that provides file access between networked computers.

A file system provides a data storage service that allows applications to share mass storage. Without a file system, applications could access the storage in incompatible ways that lead to resource contention, data corruption and data loss.

There are many file system designs and implementations – with various structure and features and various resulting characteristics such as speed, flexibility, security, size and more.

Files systems have been developed for many types of storage devices, including hard disk drives (HDDs), solid-state drives (SSDs), magnetic tapes and optical discs.[3]

A portion of the computer main memory can be set up as a RAM disk that serves as a storage device for a file system. File systems such as tmpfs can store files in virtual memory.

A *virtual* file system provides access to files that are either computed on request, called *virtual files* (see procfs and sysfs), or are mapping into another, backing storage.

Etymology

From c. 1900 and before the advent of computers the terms *file system*, *filing system* and *system for filing* were used to describe methods of organizing, storing and retrieving paper documents.[4] By 1961, the term *file system* was being applied to computerized filing alongside the original meaning.[5] By 1964, it was in general use.[6]

Architecture

A local file system's architecture can be described as layers of abstraction even though a particular file system design may not actually separate the concepts.[7]

The *logical file system* layer provides relatively high-level access via an application programming interface (API) for file operations including open, close, read and write – delegating operations to lower layers. This layer manages open file table entries and per-process file descriptors.[8] It provides file access, directory operations, security and protection.[7]

The *virtual file system*, an optional layer, supports multiple concurrent instances of physical file systems, each of which called a file system implementation.[8]

The *physical file system* layer provides relatively low-level access to a storage device (e.g. disk). It reads and writes data blocks, provides buffering and other memory management and controls placement of blocks in specific locations on the storage medium. This layer uses device drivers or channel I/O to drive the storage device.[7]

Attributes

File names

A **file name**, or **filename**, identifies a file to consuming applications and in some cases users.

A file name is unique so that an application can refer to exactly one file for a particular name. If the file system supports directories, then generally file name uniqueness is enforced within the context of each directory. In other words, a storage can contain multiple files with the same name, but not in the same directory.

Most file systems restrict the length of a file name.

Some file systems match file names as case sensitive and others as case insensitive. For example, the names `MYFILE` and `myfile` match the same file for case insensitive, but different files for case sensitive.

Most modern file systems allow a file name to contain a wide range of characters from the Unicode character set. Some restrict characters such as those used to indicate

special attributes such as a device, device type, directory prefix, file path separator, or file type.

Directories

File systems typically support organizing files into **directories**, also called **folders**, which segregate files into groups.

This may be implemented by associating the file name with an index in a table of contents or an inode in a Unix-like file system.

Directory structures may be flat (i.e. linear), or allow hierarchies by allowing a directory to contain directories, called subdirectories.

The first file system to support arbitrary hierarchies of directories was used in the Multics operating system.[9] The native file systems of Unix-like systems also support arbitrary directory hierarchies, as do, Apple's Hierarchical File System and its successor HFS+ in classic Mac OS, the FAT file system in MS-DOS 2.0 and later versions of MS-DOS and in Microsoft Windows, the NTFS file system in the Windows NT family of operating systems, and the ODS-2 (On-Disk Structure-2) and higher levels of the Files-11 file system in OpenVMS.

Metadata

In addition to data, the file content, a file system also manages associated metadata which may include but is not limited to:

- name
- size which may be stored as the number of blocks allocated or as a byte count
- when created, last accessed, last backed-up
- owner user and group
- access permissions
- file attributes such as whether the file is read-only, executable, etc.
- device type (e.g. block, character, socket, subdirectory, etc.)

A file system stores associated metadata separate from the content of the file.

Most file systems store the names of all the files in one directory in one place—the directory table for that directory—which is often stored like any other file. Many file systems put only some of the metadata for a file in the directory table, and the rest of the metadata for that file in a completely separate structure, such as the inode.

Most file systems also store metadata not associated with any one particular file. Such metadata includes information about unused regions—free space bitmap, block availability map—and information about bad sectors. Often such information about an allocation group is stored inside the allocation group itself.

Additional attributes can be associated on file systems, such as NTFS, XFS, ext2, ext3, some versions of UFS, and HFS+, using extended file attributes. Some file systems provide for user defined attributes such as the author of the document, the character encoding of a document or the size of an image.

Some file systems allow for different data collections to be associated with one file name. These separate collections may be referred to as *streams* or *forks*. Apple has long used a forked file system on the Macintosh, and Microsoft supports streams in NTFS. Some file systems maintain multiple past revisions of a file under a single file name; the file name by itself retrieves the most recent version, while prior saved version can be accessed using a special naming convention such as "filename;4" or "filename(-4)" to access the version four saves ago.

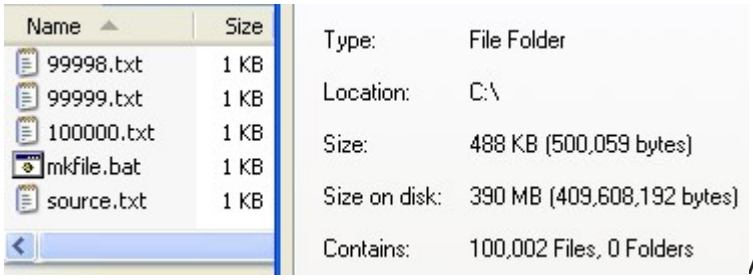
See comparison of file systems#Metadata for details on which file systems support which kinds of metadata.

Storage space organization

A local file system tracks which areas of storage belong to which file and which are not being used.

When a file system creates a file, it allocates space for data. Some file systems permit or require specifying an initial space allocation and subsequent incremental allocations as the file grows.

To delete a file, the file system records that the file's space is free; available to use for another file.



An example of slack space, demonstrated with 4,096-byte NTFS clusters: 100,000 files, each five bytes per file, which equal to 500,000 bytes of actual data but require 409,600,000 bytes of disk space to store

A local file system manages storage space to provide a level of reliability and efficiency. Generally, it allocates storage device space in a granular manner, usually multiple physical units (i.e. bytes). For example, in Apple DOS of the early 1980s, 256-byte sectors on 140 kilobyte floppy disk used a *track/sector map*.[citation needed]

The granular nature results in unused space, sometimes called slack space, for each file except for those that have the rare size that is a multiple of the granular allocation.[10] For a 512-byte allocation, the average unused space is 256 bytes. For 64 KB clusters, the average unused space is 32 KB.

Generally, the allocation unit size is set when the storage is configured. Choosing a relatively small size compared to the files stored, results in excessive access overhead. Choosing a relatively large size results in excessive unused space. Choosing an allocation size based on the average size of files expected to be in the storage tends to minimize unusable space.

Fragmentation

File systems may become fragmented

As a file system creates, modifies and deletes files, the underlying storage representation may become fragmented. Files and the unused space between files will occupy allocation blocks that are not contiguous.

A file becomes fragmented if space needed to store its content cannot be allocated in contiguous blocks. Free space becomes fragmented when files are deleted.[11]

This is invisible to the end user and the system still works correctly. However this can degrade performance on some storage hardware that work better with contiguous blocks such as hard disk drives. Other hardware such as solid-state drives are not affected by fragmentation.

Access control

A file system often supports access control of data that it manages.

The intent of access control is often to prevent certain users from reading or modifying certain files.

Access control can also restrict access by program in order to ensure that data is modified in a controlled way. Examples include passwords stored in the metadata of the file or elsewhere and file permissions in the form of permission bits, access control lists, or capabilities. The need for file system utilities to be able to access the data at the media level to reorganize the structures and provide efficient backup usually means that these are only effective for polite users but are not effective against intruders.

Methods for encrypting file data are sometimes included in the file system. This is very effective since there is no need for file system utilities to know the encryption seed to effectively manage the data. The risks of relying on encryption include the fact that an attacker can copy the data and use brute force to decrypt the data. Additionally, losing the seed means losing the data.

Storage quota

Some operating systems allow a system administrator to enable disk quotas to limit a user's use of storage space.

Data integrity

A file system typically ensures that stored data remains consistent in both normal operations as well as exceptional situations like:

- accessing program neglects to inform the file system that it has completed file access (to close a file)
- accessing program terminates abnormally (crashes)
- media failure
- loss of connection to remote systems
- operating system failure
- system reset (soft reboot)
- power failure (hard reboot)

Recovery from exceptional situations may include updating metadata, directory entries and handling data that was buffered but not written to storage media.

Recording

A file system might record events to allow analysis of issues such as:

- file or systemic problems and performance
- nefarious access

Data access

Byte stream access

[

Many file systems access data as a stream of bytes. Typically, to read file data, a program provides a memory buffer and the file system retrieves data from the medium and then writes the data to the buffer. A write involves the program providing a buffer of bytes that the file system reads and then stores to the medium.

Record access

Some file systems, or layers on top of a file system, allow a program to define a record so that a program can read and write data as a structure; not an unorganized sequence of bytes.

If a *fixed length* record definition is used, then locating the n^{th} record can be calculated mathematically, which is relatively fast compared to parsing the data for record separators.

An identification for each record, also known as a key, allows a program to read, write and update records without regard to their location in storage. Such storage requires

managing blocks of media, usually separating key blocks and data blocks. Efficient algorithms can be developed with pyramid structures for locating records.[12]

Utilities

Typically, a file system can be managed by the user via various utility programs.

Some utilities allow the user to create, configure and remove an instance of a file system. It may allow extending or truncating the space allocated to the file system.

Directory utilities may be used to create, rename and delete *directory entries*, which are also known as *dentries* (singular: *dentry*),[13] and to alter metadata associated with a directory. Directory utilities may also include capabilities to create additional links to a directory (hard links in Unix), to rename parent links (".." in Unix-like operating systems] and to create bidirectional links to files.

File utilities create, list, copy, move and delete files, and alter metadata. They may be able to truncate data, truncate or extend space allocation, append to, move, and modify files in-place. Depending on the underlying structure of the file system, they may provide a mechanism to prepend to or truncate from the beginning of a file, insert entries into the middle of a file, or delete entries from a file. Utilities to free space for deleted files, if the file system provides an undelete function, also belong to this category.

Some file systems defer operations such as reorganization of free space, secure erasing of free space, and rebuilding of hierarchical structures by providing utilities to perform these functions at times of minimal activity. An example is the file system defragmentation utilities.

Some of the most important features of file system utilities are supervisory activities which may involve bypassing ownership or direct access to the underlying device. These include high-performance backup and recovery, data replication, and reorganization of various data structures and allocation tables within the file system.

File system API

ities, libraries and programs use file system APIs to make requests of the file system. These include data transfer, positioning, updating metadata, managing directories, managing access specifications, and removal.

Multiple file systems within a single system

igured with a single file system occupying the entire storage device.

Another approach is to partition the disk so that several file systems with different attributes can be used. One file system, for use as browser cache or email storage, might be configured with a small allocation size. This keeps the activity of creating and deleting files typical of browser activity in a narrow area of the disk where it will not interfere with other file allocations. Another partition might be created for the storage of

audio or video files with a relatively large block size. Yet another may normally be set *read-only* and only periodically be set writable. Some file systems, such as ZFS and APFS, support multiple file systems sharing a common pool of free blocks, supporting several file systems with different attributes without having to reserved a fixed amount of space for each file system.[14][15]

A third approach, which is mostly used in cloud systems, is to use "disk images" to house additional file systems, with the same attributes or not, within another (host) file system as a file. A common example is virtualization: one user can run an experimental Linux distribution (using the ext4 file system) in a virtual machine under his/her production Windows environment (using NTFS). The ext4 file system resides in a disk image, which is treated as a file (or multiple files, depending on the hypervisor and settings) in the NTFS host file system.

Having multiple file systems on a single system has the additional benefit that in the event of a corruption of a single file system, the remaining file systems will frequently still be intact. This includes virus destruction of the *system* file system or even a system that will not boot. File system utilities which require dedicated access can be effectively completed piecemeal. In addition, defragmentation may be more effective. Several system maintenance utilities, such as virus scans and backups, can also be processed in segments. For example, it is not necessary to backup the file system containing videos along with all the other files if none have been added since the last backup. As for the image files, one can easily "spin off" differential images which contain only "new" data written to the master (original) image. Differential images can be used for both safety concerns (as a "disposable" system - can be quickly restored if destroyed or contaminated by a virus, as the old image can be removed and a new image can be created in matter of seconds, even without automated procedures) and quick virtual machine deployment (since the differential images can be quickly spawned using a script in batches).

Types

Disk file systems

A *disk file system* takes advantages of the ability of disk storage media to randomly address data in a short amount of time. Additional considerations include the speed of accessing data following that initially requested and the anticipation that the following data may also be requested. This permits multiple users (or processes) access to various data on the disk without regard to the sequential location of the data. Examples include FAT (FAT12, FAT16, FAT32), exFAT, NTFS, ReFS, HFS and HFS+, HPFS, APFS, UFS, ext2, ext3, ext4, XFS, btrfs, Files-11, Veritas File System, VMFS, ZFS, ReiserFS, NSS and ScoutFS. Some disk file systems are journaling file systems or versioning file systems.

ISO 9660 and Universal Disk Format (UDF) are two common formats that target Compact Discs, DVDs and Blu-ray discs. Mount Rainier is an extension to UDF supported since 2.6 series of the Linux kernel and since Windows Vista that facilitates rewriting to DVDs.

Flash file systems

A *flash file system* considers the special abilities, performance and restrictions of flash memory devices. Frequently a disk file system can use a flash memory device as the underlying storage media, but it is much better to use a file system specifically designed for a flash device.[16]

Tape file systems

A *tape file system* is a file system and tape format designed to store files on tape. Magnetic tapes are sequential storage media with significantly longer random data access times than disks, posing challenges to the creation and efficient management of a general-purpose file system.

In a disk file system there is typically a master file directory, and a map of used and free data regions. Any file additions, changes, or removals require updating the directory and the used/free maps. Random access to data regions is measured in milliseconds so this system works well for disks.

Tape requires linear motion to wind and unwind potentially very long reels of media. This tape motion may take several seconds to several minutes to move the read/write head from one end of the tape to the other.

Consequently, a master file directory and usage map can be extremely slow and inefficient with tape. Writing typically involves reading the block usage map to find free blocks for writing, updating the usage map and directory to add the data, and then advancing the tape to write the data in the correct spot. Each additional file write requires updating the map and directory and writing the data, which may take several seconds to occur for each file.

Tape file systems instead typically allow for the file directory to be spread across the tape intermixed with the data, referred to as *streaming*, so that time-consuming and repeated tape motions are not required to write new data.

However, a side effect of this design is that reading the file directory of a tape usually requires scanning the entire tape to read all the scattered directory entries. Most data archiving software that works with tape storage will store a local copy of the tape catalog on a disk file system, so that adding files to a tape can be done quickly without having to rescan the tape media. The local tape catalog copy is usually discarded if not used for a specified period of time, at which point the tape must be re-scanned if it is to be used in the future.

IBM has developed a file system for tape called the Linear Tape File System. The IBM implementation of this file system has been released as the open-source IBM Linear Tape File System — Single Drive Edition (LTFS-SDE) product. The Linear Tape File System uses a separate partition on the tape to record the index meta-data, thereby avoiding the problems associated with scattering directory entries across the entire tape.

Tape formatting

Writing data to a tape, erasing, or formatting a tape is often a significantly time-consuming process and can take several hours on large tapes.[a] With many data tape technologies it is not necessary to format the tape before over-writing new data to the tape. This is due to the inherently destructive nature of overwriting data on sequential media.

Because of the time it can take to format a tape, typically tapes are pre-formatted so that the tape user does not need to spend time preparing each new tape for use. All that is usually necessary is to write an identifying media label to the tape before use, and even this can be automatically written by software when a new tape is used for the first time.

Database file systems

Another concept for file management is the idea of a database-based file system. Instead of, or in addition to, hierarchical structured management, files are identified by their characteristics, like type of file, topic, author, or similar rich metadata.[17]

IBM DB2 for i [18] (formerly known as DB2/400 and DB2 for i5/OS) is a database file system as part of the object based IBM i[19] operating system (formerly known as OS/400 and i5/OS), incorporating a single level store and running on IBM Power Systems (formerly known as AS/400 and iSeries), designed by Frank G. Soltis IBM's former chief scientist for IBM i. Around 1978 to 1988 Frank G. Soltis and his team at IBM Rochester had successfully designed and applied technologies like the database file system where others like Microsoft later failed to accomplish.[20] These technologies are informally known as 'Fortress Rochester'[*citation needed*] and were in few basic aspects extended from early Mainframe technologies but in many ways more advanced from a technological perspective[*citation needed*].

Some other projects that are not "pure" database file systems but that use some aspects of a database file system:

- Many Web content management systems use a relational DBMS to store and retrieve files. For example, XHTML files are stored as XML or text fields, while image files are stored as blob fields; SQL SELECT (with optional XPath) statements retrieve the files, and allow the use of a sophisticated logic and more rich information associations than "usual file systems." Many CMSs also have the option of storing only metadata within the database, with the standard filesystem used to store the content of files.
- Very large file systems, embodied by applications like Apache Hadoop and Google File System, use some *database file system* concepts.

Transactional file systems

Some programs need to either make multiple file system changes, or, if one or more of the changes fail for any reason, make none of the changes. For example, a program which is installing or updating software may write executables, libraries, and/or

configuration files. If some of the writing fails and the software is left partially installed or updated, the software may be broken or unusable. An incomplete update of a key system utility, such as the command shell, may leave the entire system in an unusable state.

Transaction processing introduces the atomicity guarantee, ensuring that operations inside of a transaction are either all committed or the transaction can be aborted and the system discards all of its partial results. This means that if there is a crash or power failure, after recovery, the stored state will be consistent. Either the software will be completely installed or the failed installation will be completely rolled back, but an unusable partial install will not be left on the system. Transactions also provide the isolation guarantee[*clarification needed*], meaning that operations within a transaction are hidden from other threads on the system until the transaction commits, and that interfering operations on the system will be properly serialized with the transaction.

Windows, beginning with Vista, added transaction support to NTFS, in a feature called Transactional NTFS, but its use is now discouraged.[21] There are a number of research prototypes of transactional file systems for UNIX systems, including the Valor file system,[22] Amino,[23] LFS,[24] and a transactional ext3 file system on the TxOS kernel,[25] as well as transactional file systems targeting embedded systems, such as TFFS.[26]

Ensuring consistency across multiple file system operations is difficult, if not impossible, without file system transactions. File locking can be used as a concurrency control mechanism for individual files, but it typically does not protect the directory structure or file metadata. For instance, file locking cannot prevent TOCTTOU race conditions on symbolic links. File locking also cannot automatically roll back a failed operation, such as a software upgrade; this requires atomicity.

Journaling file systems is one technique used to introduce transaction-level consistency to file system structures. Journal transactions are not exposed to programs as part of the OS API; they are only used internally to ensure consistency at the granularity of a single system call.

Data backup systems typically do not provide support for direct backup of data stored in a transactional manner, which makes the recovery of reliable and consistent data sets difficult. Most backup software simply notes what files have changed since a certain time, regardless of the transactional state shared across multiple files in the overall dataset. As a workaround, some database systems simply produce an archived state file containing all data up to that point, and the backup software only backs that up and does not interact directly with the active transactional databases at all. Recovery requires separate recreation of the database from the state file after the file has been restored by the backup software.

Network file systems

A *network file system* is a file system that acts as a client for a remote file access protocol, providing access to files on a server. Programs using local interfaces can

transparently create, manage and access hierarchical directories and files in remote network-connected computers. Examples of network file systems include clients for the NFS,[27] AFS, SMB protocols, and file-system-like clients for FTP and WebDAV.

Shared disk file systems

: Shared disk file system

A *shared disk file system* is one in which a number of machines (usually servers) all have access to the same external disk subsystem (usually a storage area network). The file system arbitrates access to that subsystem, preventing write collisions.[28] Examples include GFS2 from Red Hat, GPFS, now known as Spectrum Scale, from IBM, SFS from DataPlow, CXFS from SGI, StorNext from Quantum Corporation and ScoutFS from Versity.

Special file systems

Some file systems expose elements of the operating system as files so they can be acted on via the file system API. This is common in Unix-like operating systems, and to a lesser extent in other operating systems. Examples include:

- devfs, udev, TOPS-10 expose I/O devices or pseudo-devices as special files
- configfs and sysfs expose special files that can be used to query and configure Linux kernel information
- procfs exposes process information as special files

Minimal file system / audio-cassette storage

In the 1970s disk and digital tape devices were too expensive for some early microcomputer users. An inexpensive basic data storage system was devised that used common audio cassette tape.

When the system needed to write data, the user was notified to press "RECORD" on the cassette recorder, then press "RETURN" on the keyboard to notify the system that the cassette recorder was recording. The system wrote a sound to provide time synchronization, then modulated sounds that encoded a prefix, the data, a checksum and a suffix. When the system needed to read data, the user was instructed to press "PLAY" on the cassette recorder. The system would *listen* to the sounds on the tape waiting until a burst of sound could be recognized as the synchronization. The system would then interpret subsequent sounds as data. When the data read was complete, the system would notify the user to press "STOP" on the cassette recorder. It was primitive, but it (mostly) worked. Data was stored sequentially, usually in an unnamed format, although some systems (such as the Commodore PET series of computers) did allow the files to be named. Multiple sets of data could be written and located by fast-forwarding the tape and observing at the tape counter to find the approximate start of the next data region on the tape. The user might have to listen to the sounds to find the right spot to begin playing the next data region. Some implementations even included audible sounds interspersed with the data.

Flat file systems

.

In a flat file system, there are no subdirectories; directory entries for all files are stored in a single directory.

When floppy disk media was first available this type of file system was adequate due to the relatively small amount of data space available. CP/M machines featured a flat file system, where files could be assigned to one of 16 *user areas* and generic file operations narrowed to work on one instead of defaulting to work on all of them. These user areas were no more than special attributes associated with the files; that is, it was not necessary to define specific quota for each of these areas and files could be added to groups for as long as there was still free storage space on the disk. The early Apple Macintosh also featured a flat file system, the Macintosh File System. It was unusual in that the file management program (Macintosh Finder) created the illusion of a partially hierarchical filing system on top of EMFS. This structure required every file to have a unique name, even if it appeared to be in a separate folder. IBM DOS/360 and OS/360 store entries for all files on a disk pack (*volume*) in a directory on the pack called a *Volume Table of Contents* (VTOC).

While simple, flat file systems become awkward as the number of files grows and makes it difficult to organize data into related groups of files.

A recent addition to the flat file system family is Amazon's S3, a remote storage service, which is intentionally simplistic to allow users the ability to customize how their data is stored. The only constructs are buckets (imagine a disk drive of unlimited size) and objects (similar, but not identical to the standard concept of a file). Advanced file management is allowed by being able to use nearly any character (including '/') in the object's name, and the ability to select subsets of the bucket's content based on identical prefixes.

Implementations

An operating system (OS) typically supports one or more file systems. Sometimes an OS and its file system are so tightly interwoven that it is difficult to describe them independently.

An OS typically provides file system access to the user. Often an OS provides command line interface, such as Unix shell, Windows Command Prompt and PowerShell, and OpenVMS DCL. An OS often also provides graphical user interface file browsers such as MacOS Finder and Windows File Explorer.

Unix and Unix-like operating systems

Unix-like operating systems create a virtual file system, which makes all the files on all the devices appear to exist in a single hierarchy. This means, in those systems, there is one root directory, and every file existing on the system is located under it somewhere. Unix-like systems can use a RAM disk or network shared resource as its root directory.

Unix-like systems assign a device name to each device, but this is not how the files on that device are accessed. Instead, to gain access to files on another device, the operating system must first be informed where in the directory tree those files should appear. This process is called mounting a file system. For example, to access the files on a CD-ROM, one must tell the operating system "Take the file system from this CD-ROM and make it appear under such-and-such directory." The directory given to the operating system is called the ***mount point*** – it might, for example, be `/media`. The `/media` directory exists on many Unix systems (as specified in the Filesystem Hierarchy Standard) and is intended specifically for use as a mount point for removable media such as CDs, DVDs, USB drives or floppy disks. It may be empty, or it may contain subdirectories for mounting individual devices. Generally, only the administrator (i.e. root user) may authorize the mounting of file systems.

Unix-like operating systems often include software and tools that assist in the mounting process and provide it new functionality. Some of these strategies have been coined "auto-mounting" as a reflection of their purpose.

- In many situations, file systems other than the root need to be available as soon as the operating system has booted. All Unix-like systems therefore provide a facility for mounting file systems at boot time. System administrators define these file systems in the configuration file fstab (*vfstab* in Solaris), which also indicates options and mount points.
- In some situations, there is no need to mount certain file systems at boot time, although their use may be desired thereafter. There are some utilities for Unix-like systems that allow the mounting of predefined file systems upon demand.
- Removable media allow programs and data to be transferred between machines without a physical connection. Common examples include USB flash drives, CD-ROMs, and DVDs. Utilities have therefore been developed to detect the presence and availability of a medium and then mount that medium without any user intervention.
- Progressive Unix-like systems have also introduced a concept called **supermounting**; see, for example, the Linux supermount-ng project. For example, a floppy disk that has been supermounted can be physically removed from the system. Under normal circumstances, the disk should have been synchronized and then unmounted before its removal. Provided synchronization has occurred, a different disk can be inserted into the drive. The system automatically notices that the disk has changed and updates the mount point contents to reflect the new medium.
- An automounter will automatically mount a file system when a reference is made to the directory atop which it should be mounted. This is usually used for file systems on network servers, rather than relying on events such as the insertion of media, as would be appropriate for removable media.

Linux

Linux supports numerous file systems, but common choices for the system disk on a block device include the ext* family (ext2, ext3 and ext4), XFS, JFS, and btrfs. For raw flash without a flash translation layer (FTL) or Memory Technology Device (MTD), there

are UBIFS, JFFS2 and YAFFS, among others. SquashFS is a common compressed read-only file system.

Solaris

]

Solaris in earlier releases defaulted to (non-journaled or non-logging) UFS for bootable and supplementary file systems. Solaris defaulted to, supported, and extended UFS.

Support for other file systems and significant enhancements were added over time, including Veritas Software Corp. (journaling) VxFS, Sun Microsystems (clustering) QFS, Sun Microsystems (journaling) UFS, and Sun Microsystems (open source, poolable, 128 bit compressible, and error-correcting) ZFS.

Kernel extensions were added to Solaris to allow for bootable Veritas VxFS operation. Logging or journaling was added to UFS in Sun's Solaris 7. Releases of Solaris 10, Solaris Express, OpenSolaris, and other open source variants of the Solaris operating system later supported bootable ZFS.

Logical Volume Management allows for spanning a file system across multiple devices for the purpose of adding redundancy, capacity, and/or throughput. Legacy environments in Solaris may use Solaris Volume Manager (formerly known as Solstice DiskSuite). Multiple operating systems (including Solaris) may use Veritas Volume Manager. Modern Solaris based operating systems eclipse the need for volume management through leveraging virtual storage pools in ZFS.

macOS

macOS (formerly Mac OS X) uses the Apple File System (APFS), which in 2017 replaced a file system inherited from classic Mac OS called HFS Plus (HFS+). Apple also uses the term "Mac OS Extended" for HFS+.[29] HFS Plus is a metadata-rich and case-preserving but (usually) case-insensitive file system. Due to the Unix roots of macOS, Unix permissions were added to HFS Plus. Later versions of HFS Plus added journaling to prevent corruption of the file system structure and introduced a number of optimizations to the allocation algorithms in an attempt to defragment files automatically without requiring an external defragmenter.

File names can be up to 255 characters. HFS Plus uses Unicode to store file names. On macOS, the filetype can come from the type code, stored in file's metadata, or the filename extension.

HFS Plus has three kinds of links: Unix-style hard links, Unix-style symbolic links, and aliases. Aliases are designed to maintain a link to their original file even if they are moved or renamed; they are not interpreted by the file system itself, but by the File Manager code in userland.

macOS 10.13 High Sierra, which was announced on June 5, 2017, at Apple's WWDC event, uses the Apple File System on solid-state drives.

macOS also supported the UFS file system, derived from the BSD Unix Fast File System via NeXTSTEP. However, as of Mac OS X Leopard, macOS could no longer be installed on a UFS volume, nor can a pre-Leopard system installed on a UFS volume be upgraded to Leopard.[30] As of Mac OS X Lion UFS support was completely dropped.

Newer versions of macOS are capable of reading and writing to the legacy FAT file systems (16 and 32) common on Windows. They are also capable of *reading* the newer NTFS file systems for Windows. In order to *write* to NTFS file systems on macOS versions prior to Mac OS X Snow Leopard third-party software is necessary. Mac OS X 10.6 (Snow Leopard) and later allow writing to NTFS file systems, but only after a non-trivial system setting change (third-party software exists that automates this).[31]

Finally, macOS supports reading and writing of the exFAT file system since Mac OS X Snow Leopard, starting from version 10.6.5.[32]

OS/2

OS/2 1.2 introduced the High Performance File System (HPFS). HPFS supports mixed case file names in different code pages, long file names (255 characters), more efficient use of disk space, an architecture that keeps related items close to each other on the disk volume, less fragmentation of data, extent-based space allocation, a B+ tree structure for directories, and the root directory located at the midpoint of the disk, for faster average access. A journaled filesystem (JFS) was shipped in 1999.

PC-BSD

PC-BSD is a desktop version of FreeBSD, which inherits FreeBSD's ZFS support, similarly to FreeNAS. The new graphical installer of PC-BSD can handle */ (root) on ZFS* and RAID-Z pool installs and disk encryption using Geli right from the start in an easy convenient (GUI) way. The current PC-BSD 9.0+ 'Isotope Edition' has ZFS filesystem version 5 and ZFS storage pool version 28.

Plan 9

be accessed (i.e., there is no ioctl or mmap): networking, graphics, debugging, authentication, capabilities, encryption, and other services are accessed via I/O operations on file descriptors. The 9P protocol removes the difference between local and remote files. File systems in Plan 9 are organized with the help of private, per-process namespaces, allowing each process to have a different view of the many file systems that provide resources in a distributed system.

The Inferno operating system shares these concepts with Plan 9.

Microsoft Windows

Directory listing in a Windows command shell

Windows makes use of the FAT, NTFS, exFAT, Live File System and ReFS file systems (the last of these is only supported and usable in Windows Server 2012, Windows Server 2016, Windows 8, Windows 8.1, and Windows 10; Windows cannot boot from it).

Windows uses a *drive letter* abstraction at the user level to distinguish one disk or partition from another. For example, the path `C:\WINDOWS` represents a directory `WINDOWS` on the partition represented by the letter C. Drive C: is most commonly used for the primary hard disk drive partition, on which Windows is usually installed and from which it boots. This "tradition" has become so firmly ingrained that bugs exist in many applications which make assumptions that the drive that the operating system is installed on is C. The use of drive letters, and the tradition of using "C" as the drive letter for the primary hard disk drive partition, can be traced to MS-DOS, where the letters A and B were reserved for up to two floppy disk drives. This in turn derived from CP/M in the 1970s, and ultimately from IBM's CP/CMS of 1967