File Systems

This article's tone or style may not reflect the encyclopedic tone used throughout the wiki.
See Wikipedia's article on tone for suggestions.

File systems are the operating system's method of ordering data on persistent storage devices like disks. They provide an abstracted interface to access data on these devices in such a way that it can be read or modified efficiently. Which file system is convenient depends on the target application of the operating system. For example, Windows uses the FAT32 or NTFS file system. If a disk has a large capacity, FAT32 is inconvenient, because the FAT system was designed considering the smaller disks available at that time. At the same time, an NTFS file system is not convenient on a tiny disk, because it was designed to work with large volumes of data - there would be excessive overhead when using devices such as a 1.44 MB floppy disk.

For details on specific filesystems, browse this list of filesystems.

JFFS2
YAFFS

Contents

File System Theory

This page is a work in progress.
This page may thus be incomplete. Its content may be changed in the near future.

A filesystem provides a generalized structure over persistent storage, allowing the low-level structure of the devices (e.g., disk, tape, flash memory storage) to be abstracted away. Generally speaking, the goal of a filesystem is allowing logical groups of data to be organized into *files*, which can be manipulated as a unit. In order to do this, the filesystem must provide some sort of index of the locations of files in the actual secondary storage. The fundamental operations of any filesystem are:

- Tracking the available storage space
- Tracking which block or blocks of data belong to which files
- Creating new files
- Reading data from existing files into memory
- Updating the data in the files
- Deleting existing files

(Perceptive readers will note that the last four operations - Create, Read, Update, and Delete, or CRUD - are also applicable to many other data structures, and are fundamental to databases as well as filesystems.)

Additionally, there are other features which go along with a practical filesystem:

- Assigning human-readable names to files, and renaming files after creation

- Allowing files to be divided among non-contiguous blocks in storage, and tracking the parts of files even when they are *fragmented* across the medium
- Providing some form of hierarchical structure, allowing the files to be divided into *directories* or *folders*
- Buffering reading and writing to reduce the number of actual operation on the physical medium
- Caching frequently accessed files or parts of files to speed up access
- Allowing files to be marked as 'read-only' to prevent unintentional corruption of critical data
- Providing a mechanism for preventing unauthorized access to a user's files

Additional features may be found on some filesystems as well, such as automatic encryption, or journalling of read/write activity.

Indexing Methods

There are several methods of indexing the contents of files, with the most commonly used being *i-nodes* and *File Allocation Tables*.

inodes

inodes (information nodes) are a crucial design element in most Unix file systems: Each file is made of data blocks (the sectors that contains your raw data bits), index blocks (containing pointers to data blocks so that you know which sector is the nth in the sequence), and one inode block.

The inode is the root of the index blocks, and can also be the sole index block if the file is small enough. Moreover, as Unix file systems support hard links (the same file may appear several times in the directory tree), inodes are a natural place to store Metadata such as file size, owner, creation/access/modification times, locks, etc.

FAT

The File Allocation Table (FAT) is the primary indexing mechanism for MS-DOS and it's descendants. There are several variants on FAT, but the general design is to have a table (actually a pair of tables, one serving as a backup for the first in case it is corrupted) which holds a list of blocks of a given size, which map to the whole capacity of the disk.

Workings of File Systems

There are several common approaches to storing disk information. However, in comparison to memory management, there are some key differences in managing disk media:

- Data can only be written in fixed size chunks.
- Access times are different for different locations on the disk. Seeking is usually a costly operation.
- Data throughput is very small compared to RAM
- Data commonly has to be maintained

Hence some file systems have specialized structures, algorithms, or combinations thereof to improve speed ratings.

Allocation Table

The allocation table is comparable to the bitmap approach. However instead of just having a field free or occupied, it may contain other information. Advantage is that the use of this structure is simple, Disadvantage is that this approach is relatively slow, and a separate set of data is needed to define the which sections are used, and in which order. FAT for example, combines a linked list and an allocation table in the same structure.

Separate file and system areas

Some filesystems keep metadata and actual contents in separate areas on the disk. This makes specific sorts of data easy to find, as well as allowing for a special 'free' area. Downside is that you have to keep track of the boundary or boundaries between these areas, as the usage can differ and the disk has to adapt for that (big files: more data, less metadata; small files: less data, more metadata). This method is used prominently in SFS.

Other methods

This page is a work in progress.
This page may thus be incomplete. Its content may be changed in the near future.

Network File Systems

All these file systems are a way to create a large, distributed storage system from a collection of "back end" systems. That means you cannot (for instance) format a disk in 'NFS' but you instead mount a 'virtual' NFS partition that will reflect what's on another machine. Note that a new generation of file systems is under heavy research, basing on latest P2P, cryptography and error correction techniques (such as the Ocean Store Project or Archival Intermemory.)

For details on various network file systems look here

File systems for OSDevers

While you could pick <insert favorite filesystem here> as your OS main filesystem, you might want to consider all options. Commonly, you'll want to have a filesystem that is operational very quickly so that you can concentrate on the rest before implementing a 'real filesystem'

"Beginners" filesystems

There are only five filesystems that are both relatively easy to implement and worth to consider. There is no general recommendation as the choice depends largely on style and OS design. Instead you can read the comparison and make your own educated decision.

USTAR

- + Of these beginner "filesystems", this is the simplest by far to implement
- + Incredibly simple: a sector with metadata followed by data sectors
- + Widely used: utilities to create tar images are available for every mainstream OS and many minor ones
- + Supports special files (like devices and symlinks)
- + Supports Unix permissions

- - Not a filesystem in the common understanding of the term
- - Generally read-only, was never designed for in-place modifications
- - No support for fragmentation
- - No standard partition type for it (not that you should even consider using USTAR as a disk partition format)
- - Not actually the format used for ramdisks by things like Linux - that's CPIO

RAMFS/TMPFS

- + High flexibility of implementation
- + Fast
- + Will allow you to test out your VFS API without having to rely on filesystem specifics
- + *Highly* recommended as a starter filesystem to avoid morphing your VFS interface around a specific filesystem
- + Ideal to unpack a USTAR or CPIO initrd image into
- - Changes are, obviously, not persistent, and only in memory, to be wiped after a reboot

FAT

- + Can be read and written by virtually all OSes
- + The 'standard' for floppies
- + Relatively easy to implement
- - Part of it involving long filenames and compatibility is patented by Microsoft (http://en.swpat.org/wiki/Microsoft_FAT_patents)
- - Large overhead
- - No support for large (>4 GB) files
- - No support for Unix permissions

Ext2

- + Supports large files (with an extension)
- + Supports Unix permissions
- + Can be put on floppies
- + Can be read and written from Linux
- - Can not natively be read and written from Windows (but drivers (http://www.fs-driver.org/) are available)
- - Very large overhead
- - Of these beginner filesystems, this is the most complex

BMFS

- + Supports large files
- + Implementation Available as static library
- + Comes with utility program for creating disk images on Linux and Windows
- + Comes with FUSE bindings, allowing it to be mounted on Linux systems
- + Contains source code documentation
- - Does not support fragmentation
- - Less control over the source code

ISO 9660 The defined standard for CDs. If you boot from CD then this is the way to go. If not, don't make it your first filesystem.

Rolling your own

There are many different kinds of filesystems around, from the well-known to the more obscure ones. The most unfortunate thing about filesystems is that every hobbyist OS programmer thinks that the filesystem they design is the ultimate technology, when in reality it's usually just a copy of FAT with a change here and there, perhaps because it is one of the easiest to implement. The world doesn't need another FAT-like filesystem. Investigate all the possibilities before you decide to roll your own.

If despite of this warning you decide to create your own file system, then you should start that by implementing a FUSE driver for it. This gives the advantage that you can mount your file system image as any other storage device, and you can lists its contents, create new files and directories etc. with standard tools. FUSE is available for Linux, MacOSX and Windows as well.

Guidelines if you do decide to roll your own

- Consider carefully what it will be used for.
- Use a program to figure out the layout (e.g. a spreadsheet). The basic areas needed are:
  - Bootsector. This is essential for booting on some systems such as BIOS-x86 and Atari ST, unnecessary for others such as UEFI and OpenFirmware. Even if you don't intend to boot on systems which require it, reserving the first sector will allow your OS to be ported to them at a later time. Note that reserving space for a MBR-like partition table is needed to allow the filesystem to work in "logical partitions".
  - Partition metadata. This could fit into the first sector with the boot code, or be a separate group of sectors at a specific location. (FAT puts it in the first sector, calling it the FAT parameter block. ext* use a separate location, calling it the superblock.) At a minimum, this should contain the filesystem size, location of the file table, and a version number. Leave plenty of reserved space for features you don't think of. If you put it in the first sector, don't forget to leave space for a jmp instruction, the boot code, and a partition table!
  - File table. Don't think of this as just a simple table containing a list of files and their locations. One idea is, instead of storing files, the system would store file parts, and the file table would list the parts in each file. This would be useful for saving space if many files on the disk are the same or similar (for example, license agreements).
  - Data area. Files will be stored here.
- Create a program to read and write disk images with your filesystem. Parts of this will be portable into the fs driver.
- It is strongly recommended to create a FUSE driver for your file system.
- Implement the fs into your OS.

Expert filesystems

Once you have a beginner's file system under your belt you might want support for more advanced ones. Here are some:

- NTFS - (Windows) New Technologies File System. It's hard to find documentation. Try Apple NTFS (open source) (http://www.opensource.apple.com/source/ntfs/).

- Btrfs - B-tree file system. It's a Linux file system with features such as copy-on-write and transparent compression.

See Also

- I use a Custom Filesystem - What Bootloader Solution is right for me?

External links

- 50 years in filesystems (https://blog.koehntopp.info/2023/05/05/50-years-in-filesystems-1974.html) -- an approachable account of the history of file systems. Includes, among other information, some real-world examples of heuristics one can use to avoid fragmentation.