Engineer

file system

the structure that turns a disk into named files and folders

What it is

A file system is the part of a computer that turns a disk, which on its own stores only numbered blocks of bytes, into named files arranged in directories. It is also written as one word, filesystem, which is the spelling the Linux kernel documentation mostly uses. POSIX, the standard that Unix-like systems follow, defines it in its 2024 edition as “A collection of files and certain of their attributes.” The same definition adds that file serial numbers belong to a namespace supplied by the file system, the numbers that identify each file uniquely within the file system that holds it.

The rest follows from a few more of the standard’s definitions. A file is an object that can be written to, read from or both, with attributes such as its access permissions and its type: a regular file, a directory, a symbolic link, a FIFO, a socket, or a character or block special file. A directory is a file made of directory entries, no two of which in one directory share a name. A directory entry, which POSIX also calls a hard link, ties a filename to a file, and several entries can name the same file. The name of a file and the file itself are therefore kept apart. On Linux the record of the file itself is the inode, which the GNU df manual describes as holding the file’s owner, permissions, timestamps and location on the file system, while the names live in the directories that point to it.

A Unix-like system shows all of its file systems as a single tree of names. The first provides the root directory, and every other one is attached at a directory of the tree, its mount point, after which the paths below that directory belong to the attached file system. Inside the Linux kernel, the Virtual File System sits between programs and the many file systems the kernel supports. The kernel documentation describes it, also known as the Virtual Filesystem Switch, as the layer that gives programs their filesystem interface and lets different implementations exist side by side. A program calls `open(2)`, `stat(2)`, `read(2)` or `write(2)`, and the VFS resolves the path and hands the work to whichever file

system holds the file: ext4 on a disk partition, say, or a pseudo file system whose inodes live only in memory. The types the running kernel has registered are listed in `/proc/filesystems`.

Use the concept in a decision

Put “What is a file system?” to work in a real decision. We can help relate the concept to the system you are building or buying. Day-to-day care of servers, mail, accounts and updates, so the things everybody relies on stay boring.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Ran 10 PostgreSQL and MS SQL servers on Ubuntu — 10 database servers deployed and kept healthy

Ran 10 PostgreSQL and MS SQL servers on Ubuntu Get in touch →

Why it matters

Almost everything a server keeps past its next reboot is kept in a file system: the data files of its databases, its logs, uploaded documents, its configuration. Whether those survive a power cut, a full disk or a failing drive is decided in large part by choices made when the file system was created and mounted, and some of those choices cannot be taken back later.

Creation fixes capacity in two currencies. Blocks hold the contents of files, and inodes hold the files themselves. mke2fs, the tool that creates ext2, ext3 and ext4 file systems, creates one inode for every so many bytes of space, and its manual warns that this bytes-per-inode ratio cannot be changed once the file system exists. A volume can therefore run out of room for new files while it still has room for their contents, which surprises anyone who watches only the percentage of space in use.

Mounting decides what a crash costs. ext4 keeps a journal, and its documentation lists three data modes. The default, `data=ordered`, forces file data out to the main file system before the metadata that refers to it is committed to the journal. `data=writeback` drops that ordering, and the documentation warns that after an unclean shutdown, files written shortly before it may expose stale data, which it notes can be a security risk in some cases. `data=journal` sends the data through the journal as well, and it is deprecated, with support due to be removed in January 2028. The `errors=` option decides what the file system does when it finds an error in itself: remount read-only, carry on, or panic and halt the machine.

Recovery depends on it too. When a file system is damaged, the contents of its files may still be on the disk while the structure that names them is broken, and that is the ground data recovery works on. The portfolio entry on recovering data from SD cards, hard disks, SSDs and RAID says each kind of medium breaks down differently and has to be coaxed into returning its data differently, and that Mac file systems bring quirks of their own. What counted there was choosing the approach that suited the medium on the bench, more than any single technique.

What it looks like in practice

Take a new partition, `/dev/sdb1`, that is to hold a service's data under `/srv/data`. Creating an ext4 file system on it and mounting it by hand takes three commands:

```
mke2fs -t ext4 -L data /dev/sdb1
mkdir -p /srv/data
mount -t ext4 /dev/sdb1 /srv/data
```

`-t ext4` names the type, as in the kernel's own ext4 documentation, and `-L` gives the file system a label, which mke2fs limits to 16 bytes. Everything else comes from defaults; in the configuration shipped with e2fsprogs, for example, each inode takes 256 bytes. Two `df` commands then show the two kinds of capacity:

```
df -hT /srv/data
df -i /srv/data
```

The first prints the file system's type and its blocks in use and available, in units built on powers of 1024 (`-H` would use powers of 1000). The second lists inodes in place of blocks: how many there are, how many are used and available, and the percentage in use.

A mount made by hand lasts until the next reboot. To make it permanent, the file system gets a line in `/etc/fstab`, the file that mount, umount and fsck read in order:

```
LABEL=data  /srv/data  ext4  defaults,nofail  0  2
```

The first field names the file system by its label rather than as `/dev/sdb1`. The fstab manual recommends `LABEL=` or `UUID=` there, because device names often follow the order in which hardware happened to be detected and can change when disks are added or removed. `defaults` usually means read-write plus the kernel's other standard options, and `nofail` tells mount not to report an error if the device does not exist. The fifth field is read by dump(8), and 0 leaves the file system out. The sixth gives fsck the order of checks at boot: 1 for the root file system, 2 for the others and 0 for none. On a system run by systemd, the manual recommends `systemctl daemon-reload` after an edit. `mount -a` then mounts every entry not marked `noauto`, which makes a mistake in the new line show up while someone is still at the keyboard.

Watching the space is part of running one. The portfolio entry on dead-man's-switch host monitoring names running out of disk as one of the two realistic ways a 464 MB machine with a forge and a web server on it can die, beside running out of memory, and points out that neither failure announces itself. Its answer is a small script that a timer starts four times an hour, at a slightly shifted moment each time, to measure free disk and available memory and to ping an outside service only while both stay above their thresholds, so the alarm comes from the pings stopping. A check of that shape can read the inode column of `df -i` as well as the space column, since either one can reach its limit first.

Where it goes wrong

Most file system trouble arrives as a full disk, a machine that will not mount what it mounted yesterday, or data missing after a crash, and much of it traces back to a setting nobody looked at when the file system was made.

The disk is full with space to spare. Every file needs an inode, and mke2fs makes a fixed number of them when it creates the file system. A volume that holds millions of small files can use up its inodes while `df -h` still shows free space, and from then on the volume refuses every file it is asked to create. The fix is to watch `df -i` beside `df -h`, and for a volume meant for many small files to set the ratio with `-i`, or the count with `-N`, when running mke2fs, since the ratio cannot be changed afterwards. Resizing the file system keeps the ratio and adds or removes inodes to match.

A check runs on a mounted file system. The e2fsck manual says that running it on a mounted file system is in general not safe. The one exception is `-n`, which opens the file system read-only and answers no to every question, and even then

the results it prints are not valid while the file system is mounted. When e2fsck asks whether to go ahead on a mounted file system, the manual's answer is no. The fix is to unmount first, or, for the root file system, to let the check run at boot, which is what the sixth field of fstab arranges.

"Read-only" writes to the disk. Mounting ext4 with `ro` does not stop it writing: the documentation notes that it replays the journal, and so writes to the partition, although the mount is read-only. That matters most on a disk being rescued, where any write may land on something still needed. The options `ro,noload` prevent the writes, at a price the same page spells out: after an unclean unmount, skipping the replay leaves the file system with inconsistencies. The fix for a disk whose contents matter more than its uptime is to mount it `ro,noload`, copy off what is needed, and treat what is read as possibly inconsistent.

A disk changes its name. A line that mounts `/dev/sdb1` mounts whatever the kernel calls `/dev/sdb1` today, and the fstab manual warns that this can change when disks are added or removed. A new disk can then be mounted where the old one belongs, or the old one can be missing at boot. The fix is `LABEL=` or `UUID=` in the first field, with labels set by `mke2fs -L` or later by a tool such as `e2label`.

Power fails between the write and the disk. By default ext4 commits its journal every five seconds, so a power cut can lose the last five seconds of metadata changes, while the journal keeps the file system itself undamaged. File data can go further back: with delayed allocation, writing it out begins only after the interval set in `/proc/sys/vm/dirty_expire_centisecs`. The documentation calls applications that replace files without `fsync()` broken, and ext4's default `auto_da_alloc` option detects the two usual patterns, writing a new file and renaming it over the old one or truncating and rewriting in place, to avoid files that come back empty after a crash. The fix is in the program that writes the file: call `fsync()` before telling anyone the data is saved, rather than leaning on that safety net.

Timestamps stop in 2038. The mke2fs manual says that file systems with 128-byte inodes cannot hold timestamps later than January 19, 2038, while inodes of 256 bytes or more support extended timestamps. New file systems get 256 bytes by default, but an old one, or one made with `-I 128`, may still carry the

small size. The fix is to find those file systems before the date does; the ext4 documentation shows `tune2fs -I 256` converting a file system created with 128-byte inodes.

Further reading

- Definitions, POSIX.1-2024 (The Open Group Base Specifications Issue 8) — the definitions of file system, file, directory, hard link and mount point.
- Overview of the Linux Virtual File System — how the kernel's VFS resolves a path and hands the work to a file system.
- ext4 General Information — creating and mounting ext4, its data modes, `commit=` and `ro` with the journal.
- mke2fs(8) — the bytes-per-inode ratio, inode size, the 2038 limit and volume labels.
- e2fsck(8) — checking ext2, ext3 and ext4, and why never on a mounted file system.
- fstab(5) — the six fields, `LABEL=` and `UUID=`, `nofail` and the order of boot checks.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a file system?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/file-system/