Chapter 6

Storage & File Systems

How does your computer persistently store your data?

6.1 Storage Interface

Definition 6.1 (Storage Device, Pages, Address Space)**.** *A **storage device** consists of n **pages** (also known as **sectors** or **blocks** in the literature) of fixed size, e.g. 512 bytes per page. The **address space** of the device is 0 to $n-1$. To write to or read from a storage device, the OS specifies the address(es) of the page(s) it wants to access, and in case of a write, it also specifies the data to be written.*

Remarks:

- There are many different types of storage devices: HDDs, SSDs, tapes, DVDs, etc.
- For an operating system to be able to handle this diversity of devices, there are standardized interfaces for storage media. An interface consists of a standardized hardware connector and a protocol to interact with the device. Examples are IDE, SATA, and SAS.
- Internally, a storage device is free regarding how it manages its space. We discuss HDDs and SSDs as examples.

6.2 HDDs

Definition 6.2 (Magnetic Storage)**.** ***Magnetic storage** is an electronic storage medium that uses magnetized areas on a surface to store information. To set a bit to 1, the area storing the bit is polarized in one way, and for a 0 in the opposite way.*

Definition 6.3 (Hard Disk Drive, HDD)**.** *An HDD is a magnetic storage device that consists of circular **platters** that are put on top of each other on a **spindle** (a rotating cylinder in the middle). Each platter stores data on its **surfaces**, and each surface contains **tracks** that are subdivided into **sectors** (we will refer*

*to them as pages). A **reading/writing head** per surface allows storing and reading data.*

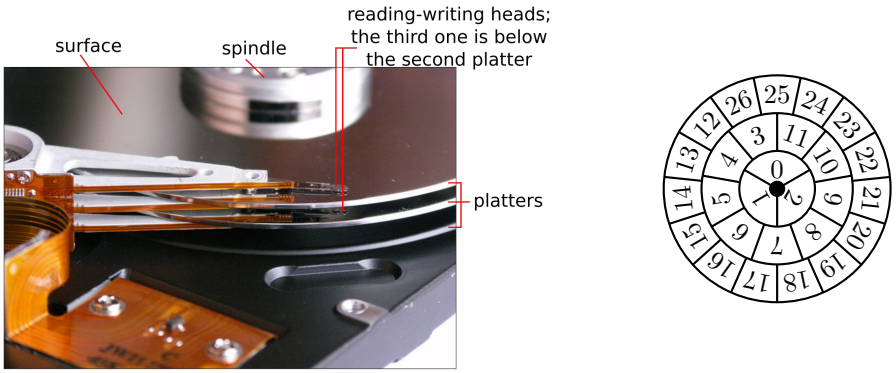

Figure 6.4: Left: schematic of an HDD with 3 platters. Right: geometry of a single platter with 3 tracks and a total of 27 pages.

Remarks:

- One could put multiple read/write heads per surface into the HDD, or give each head its own motor to move independently of the others. Neither of these options is put into practice in modern HDDs.
- Since *inner* tracks closer to the spindle cover less area than *outer* tracks further away, inner tracks have fewer pages per track.
- On real HDDs, the platter is often subdivided into *zones* of tracks where all tracks in a zone have the same number of pages. Tracks are so thin that hundreds of tracks fit within the width of a human hair. In total, a modern HDD contains several billion pages.
- Multi-page operations are possible; indeed, many file systems will read or write 4KB (or more) at a time. If an untimely power loss occurs, only a portion of a larger write may complete (this is called a *torn write*). However, manufactures guarantee that a single page is written *atomically* (the page will either complete fully or not at all).
- A request to read from/write to the disk is called an *access request*. How fast can we service an access request?

Definition 6.5 (I/O time)**.** *The **I/O time** $T_{I/O}$ it takes to transfer data to/from a page S is*

$$T_{I/O} = \underbrace{T_{seek} + T_{rotation}}_{T_{positioning}} + T_{transfer}$$

*where the **seek time** T_{seek} is the time it takes to move the read/write head to the track on which S lies, the **rotational delay** $T_{rotation}$ is the time it takes the platter to rotate to S, and the **transfer time** $T_{transfer}$ is the time it takes to transfer data to/from S. The **positioning time** $T_{positioning}$ is the sum of*

seek time and rotational delay, i.e. the time it takes for the head to move to the beginning of S.

Remarks:

- Notice the page labels in Figure 6.4. Page 12 on the outer track is offset by some angle to page 11 on the middle track. This is called *track skew*; when we want to first access page 11 and then page 12, the head has to be repositioned, and during this time, page 12 would have passed under the read/write head if page 12 was next to page 11. Thus we would have to wait almost a full rotation until being able to access page 12. The purpose of the track skew is to minimize $T_{rotation}$ when accessing pages in order.

- HDD manufacturers provide average seek time, number of rotations per minute, and maximum transfer speed in their data sheets. From these values, we can calculate the average I/O time we can expect for a given access request.

- Usually we are interested in the amount of data we can transfer per unit of time.

Definition 6.6 (Rate of I/O)**.** *The* ***rate of I/O*** $R_{I/O}$ *we get for an access request A is*

$$R_{I/O} = \frac{size\ of\ A}{T_{I/O}}$$

Example 6.7. *We are given an HDD with average seek time* $6ms$ *that rotates at a frequency of* $12000RPM$ *(rotations per minute) and has a maximum transfer speed of* $100MB/s$*. Assuming* $1MB = 1000KB$*, what is the I/O rate for an access request of 4KB to a uniformly random position on the disk?*

To solve this, we find that

- $T_{seek} = 6ms$
- A uniformly random position is on average half a rotation away from the current head position. We get 12000 rotations in 60s, thus 1/2 rotation takes $1/2 \cdot 1/200s = 2.5ms = T_{rotation}$.
- With a maximum transfer speed of $100MB/s$, transferring $4KB$ takes $4KB/(100MB/s) = 40\mu s = 0.04ms = T_{transfer}$

We get

$$R_{I/O} = \frac{size\ of\ A}{T_{I/O}} = \frac{4KB}{6ms + 2.5ms + 0.04ms} \approx 0.47MB/s$$

Remarks:

- $T_{transfer}$ is a function of page size, number of pages per track, and rotations per minute.
- We considered a specific access pattern: small, uniformly random accesses. If we want to access data that is stored in sequence, we will get a different I/O rate.
- System designers devise *disk scheduling algorithms* to reorder access requests such that the I/O rate is high. In Linux, the disk scheduler used by the OS can be changed at runtime.

6.3 Disk Scheduling

We list some of the most common disk scheduling algorithms and shortly summarize their strengths and weaknesses in Figure 6.8, and Figure 6.9 shows the head movements those algorithms produce for an example sequence of requests.

Algorithm	Description
First Come First Serve (FCFS)	Process requests in the order they arrived.
Shortest Seek Time First (SSTF)	Pick request on nearest track.
Shortest Positioning Time First (SPTF)	Pick request with shortest positioning time.
Elevator (SCAN)	Move the head like an elevator, inside to outside and back again, and service all pending requests on the way.
C-SCAN	Similar to SCAN; starting from the current head position, service requests in ascending order towards the outermost track, then move head without servicing any requests to the now-innermost request.
F-SCAN	Like SCAN, but service requests in batches; wait with sending a new batch of requests to disk until the last one was fully serviced.

Figure 6.8: Some of the most common disk scheduling algorithms.

Remarks:

- FCFS can be quite inefficient depending on the sequence of requests sent to disk.
- While usually an improvement over FCFS, SSTF/SPTF suffer from *starvation:* if there is always a request on the current track, requests on other tracks will never be served.
- Since knowledge of the disk geometry is necessary to determine accurate seek/positioning time estimates for SSTF/SPTF, and this knowledge is not available to the operating system, these algorithms are best

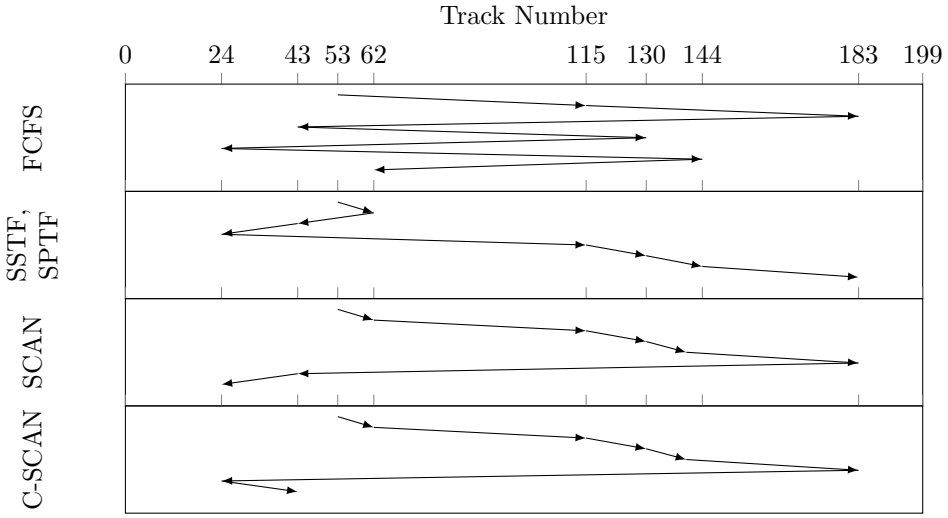

Figure 6.9: Head movements for different scheduling algorithms. The head starts at track 53 in each example run, and the sequence of requests sent to the disk is for pages on tracks 115, 183, 43, 130, 24, 144, 62.

employed by the HDD controller. The disk looks like an array of blocks to the operating system, thus it can implement the SSTF/SPTF-analogue *nearest block first (NBF)*, where the next request issued to disk is the one with the block number closest to the last request sent to disk.

- With SCAN, the pages on outer and inner tracks wait longer between separate accesses than pages on middle tracks; C-SCAN provides a more uniform $T_{I/O}$ for pages independent of their position on the disk.
- F-SCAN solves the starvation problem by not accepting new requests during a pass over the platter. Requests issued by the OS during a pass will be buffered until the next pass.
- The operating system also schedules its requests: e.g., the OS may want to allocate certain I/O rates for some processes, or it may want to make sure some I/O requests are serviced before a deadline.

6.4 SSDs

In the past few years, random-access persistent storage in the form of *solid-state drives (SSDs)* has been continuously gaining on HDDs in market share.

Definition 6.10 (Flash Memory)**.** ***Flash memory*** *is an electronic storage medium without moving parts. To set a bit to 1, electrons are trapped within a transistor via a momentary surge of power, and can only be released via another surge of power.*

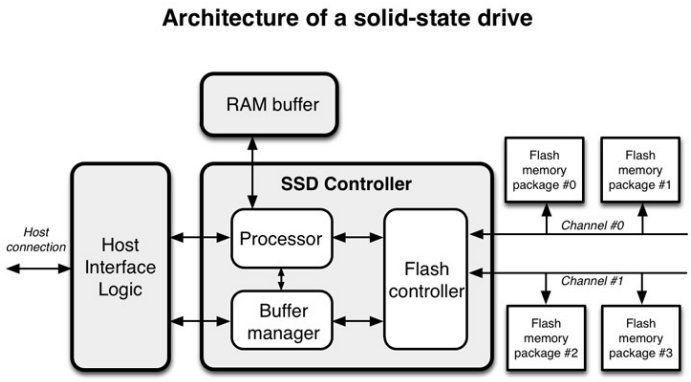

Figure 6.12: The architecture of an SSD.

- We will be discussing *NAND-based flash drives*, the most common technology used in storage devices. While NOR flash has some advantages over NAND flash, its write/erase times are significantly higher, making it less attractive as a technology for storage media.

- Unlike HDDs, flash-based SSDs have no mechanical components, and in fact are in many ways more similar to DRAM as they are "random access" devices. Their access times are less dependent on which pages are accessed.

Definition 6.11 (Blocks, Pages, Page States, Programming, Erasing)**.** *An SSD consists of equal-sized* ***pages*** *that are grouped into equal-sized* ***blocks****. A page can be in one of three states: it contains* ***valid*** *data,* ***invalid*** *data, or it is* ***erased****. Only an erased page can have new data written to it (can be* ***programmed****). In order to erase a page, the whole block containing the page has to be* ***erased****. Erasing a block deletes all data from all pages in the block. Programming a page sets its state to valid. If the data in a valid page p is updated by writing it to a different page p', then p becomes invalid.*

Block	0				1			
Page	0	1	2	3	4	5	6	7
Content	a'			a				
State	v	e	e	i	e	e	e	e

Figure 6.13: A very small SSD containing 8 pages grouped into 2 blocks of 4 pages each; invalid pages are marked "i", valid pages "v", and erased pages "e".

Remarks:

- Each page of an SSD contains two parts: the *data area* where the actual data is stored, and the *out of band area* where metadata about the page is stored. Blocks also have an out of band area associated with them. We omitted the out of band areas in Figure 6.13.

- Every time we erase a block that still has some valid data, the valid data has to be rescued to the SSD. This is called an *internal write request*. Write requests sent by the OS are called *external write requests*. Internal write requests cause *write overhead*.
- Flash cells can only be written a certain number of times before they *wear out* from the power spikes required to program and erase. Once a cell has worn out, it can no longer be erased (it can still be read). We want to balance the write accesses to wear all pages out evenly. This is called *wear leveling*.
- SSDs contain more space than they tell the operating system. The extra space (called *space overhead*) gives engineers the possibility to optimize write overhead and wear leveling to an extent.
- To be able to make use of the space overhead, SSDs map the addresses accessed by the OS to actual physical locations on the flash chips.

6.5 Flash Translation Layer

Definition 6.14 (Logical Address, Physical Address, Flash Translation Layer)**.** *The addresses in access requests sent by the OS to the SSD are called* ***logical addresses****. When a write request for a logical address arrives at the SSD, the SSD chooses a* ***physical address*** *where the data will be written. The SSD stores the mapping from logical to physical addresses, and this mapping is called the* ***flash translation layer (FTL)****.*

Remarks:

- The information whether a physical page is valid, invalid, or erased, is maintained in the FTL.
- If no FTL is used and instead every logical page is stored in the equivalent physical page, we speak of *direct mapping*. This is quite inefficient: Whenever a page is overwritten, we have to erase the entire block and issue internal requests for all valid pages in the block, increasing write overhead enormously. Blocks that have frequently-written pages also wear out fast.
- While the system is running, the FTL is stored in volatile memory. But the FTL has to be available after a power-down. There are two general approaches: Use the out of band area of pages and blocks to store mapping information (as well as which version of a logical page is the most recent one), and reconstruct the FTL after a reboot, or use dedicated space on flash that persistently stores the FTL during power-down.
- For the approach where the FTL is persisted in a dedicated area, we need to make sure that an unexpected power loss is handled correctly. For this purpose, enterprise-grade SSDs usually contain large capacitors or batteries that have enough energy to write the FTL from volatile memory to flash in case of a power failure.

Example 6.15 (Page Level FTL)**.** *We give an example of a page level FTL. Initially, the mapping is empty, and all pages are erased. For simplicity, every write goes to the next available erased page, and we erase round-robin. The rows "Table" and "State" describe the mapping and validity information stored in the FTL, respectively.*

Table												
Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
Content	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
State	e	e	e	e	e	e	e	e	e	e	e	e

First we write logical pages 0 to 4.

Table	$0 \to 0, \ldots, 4 \to 4$											
Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
Content	0	1	2	3	4	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
State	v	v	v	v	v	e	e	e	e	e	e	e

Next we write (update) logical pages 2 to 8.

Table	$0 \to 0, 1 \to 1, 2 \to 5, \ldots, 8 \to 11$											
Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
Content	0	1	2	3	4	2	3	4	5	6	7	8
State	v	v	i	i	i	v	v	v	v	v	v	v

Now we write logical page 6 (erase block 0, write back 0 and 1).

Table	$0 \to 1, 1 \to 2, 2 \to 5, 3 \to 6, 4 \to 7, 5 \to 8, 6 \to 0, 7 \to 10, 8 \to 11$											
Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
Content	6	0	1	$\perp$	4	2	3	4	5	6	7	8
State	v	v	v	e	i	v	v	v	v	i	v	v

Remarks:

- In Example 6.15, we assumed that the SSD always erases the oldest block when it needs to free up space for new writes. While this approach is perfectly fair with regard to wear leveling, there are many other more intricate algorithms that generate less write overhead for typical workloads.
- One simple approach is to always erase the block with the least valid pages; while this reduces write overhead, it may impact wear leveling negatively.

- Another approach often used in the literature is to separate logical pages into "hot pages" that are written often and "cold pages" that are written rarely. The hot and the cold pages can then be grouped together into different areas on the SSD.

- The FTL algorithms that do well in simulations take into account how long a block has not been written to, how many valid and invalid pages are in it, and how often it has been erased before.

- In a block-level FTL logical blocks are mapped to physical blocks. If the SSD has 4 pages per block, logical pages 0,1,2,3 are grouped together to form logical block 0. Whenever any page of a logical block is written, the whole logical block has to be written, producing more write overhead.

- Hybrid FTLs often do block-level mapping with a small page-level part for the most updated logical pages. Conversely, some hybrid FTLs work on page-level in general and group together cold pages into blocks.

6.6 Logical File System

To organize the data we want to permanently store, we need to arrange it in a retrievable way on a storage device, and at runtime also in memory. Most file systems follow the scheme we present here.

Definition 6.16 (Logical File System, Physical File System, Virtual File System)**.** *The data structures maintained in memory by the OS to manage stored data is the* ***logical file system****. The arrangement of the data on a storage device is the* ***physical file system*** *(or* ***file system implementation****). The software connecting those two is called the* ***virtual file system*** *(****VFS*** *on Linux, or* ***installable file systems*** *on Windows).*

Remarks:

- The logical file system exposes an API to the user to create, read, or modify files.

- The virtual file system makes it possible for the OS to have a unified view of the (logical) file system while having different implementations of (physical) file systems existing in the background.

Definition 6.18 (Directory Tree)**.** *The* ***directory tree*** *is a tree rooted at the* ***directory "/"****. Every object in the logical file system can be found via its* ***absolute path*** *starting from the root directory, or via a* ***relative path*** *starting from a directory one is currently in.*

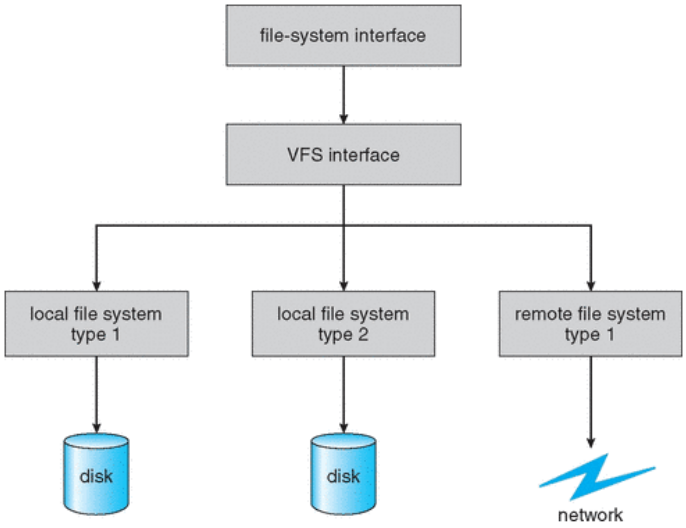

Figure 6.17: The "file-system interface" is the logical file system, the "VFS interface" is the virtual file system, and the "local file systems" are physical file systems.

Remarks:

- The directory tree is the logical file system view used in UNIX variants. Windows can have multiple drive letters, creating a "directory forest".
- The absolute path to file `vlc` is `/app/vlc`; if we have already navigated to directory `/app/`, the relative path is just `vlc`.
- One could allow cycles in the directory "graph", and some file systems do. The UNIX ecosystem has a lot of tools that will break if they run into a cycle. For this reason, it is easier to keep the file system cycle-free.
- Note that file `/film/AI.mp4` has the *file extension* "`.mp4`"; file extensions are not necessary, but merely a useful convention that allows the OS to associate files with applications.
- Android and iOS also have directory trees, but they are hidden.
- How can a logical file system like this be represented on physical storage? First, we need a way to store and find the data that belongs to a file.

6.7 Physical File System

A storage device can be subdivided into multiple *partitions* to hold multiple physical file systems. The information which parts of the storage device belong to which partition is recorded in a *partition table* (different names for data structures that serve this function in different operating systems exist). We describe how a single physical file system can be implemented.

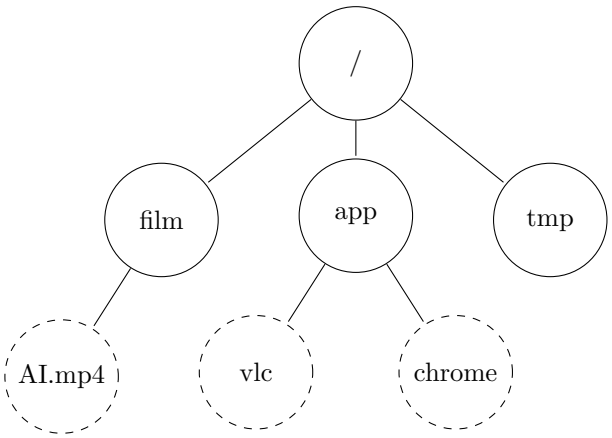

Figure 6.19: An example of a directory tree of a logical file system. The dashed nodes are files, the solid nodes are directories.

Definition 6.20 (Blocks, Inodes, Pointers)**.** *The physical file system groups multiple storage device pages into a* ***block*** *(somtimes also* ***cluster*** *or* ***allocation unit****). Every file in the physical file system is represented uniquely by an* ***inode****. The data of the file is referenced via direct, single indirect, double indirect, or triple indirect* ***pointers*** *that encode on which blocks the data is.*

Remarks:

- Both files and directories are represented by inodes; as such, "directories are files".
- An inode contains some metadata about the file, such as its type (file, directory, or one of 5 other types in UNIX), its owner, access limitations (we will see those later), file size, etc.
- The content of small files can be referenced via direct pointers, which point to data blocks holding file data. Larger files also use indirect pointers. Very small files can be stored in the inode directly.
- File sizes are limited by how large data blocks are in two regards: with larger blocks each block can store more data, and indirect pointers can point to more blocks.
- How are inodes managed on the storage device?

Definition 6.22 (Superblock, Bitmaps, Regions)**.** *At a specified location on the storage device, we find the* ***volume control block*** *(****superblock*** *in the UNIX file system,* ***master file table*** *in NTFS) that gives us the necessary information about the file system - the type of the file system (FAT, NTFS, ext4, . . .), where to find the inode of the root directory, etc. The* ***inode region*** *and the* ***data region*** *contain inodes and data blocks, respectively. The* ***inode bitmap*** *has as many bits as there are inodes in the inode region, and an inode in the inode region is in use if and only if the corresponding bit in the inode bitmap is set to* 1*. The* ***data bitmap*** *fulfills the same role for the data region.*

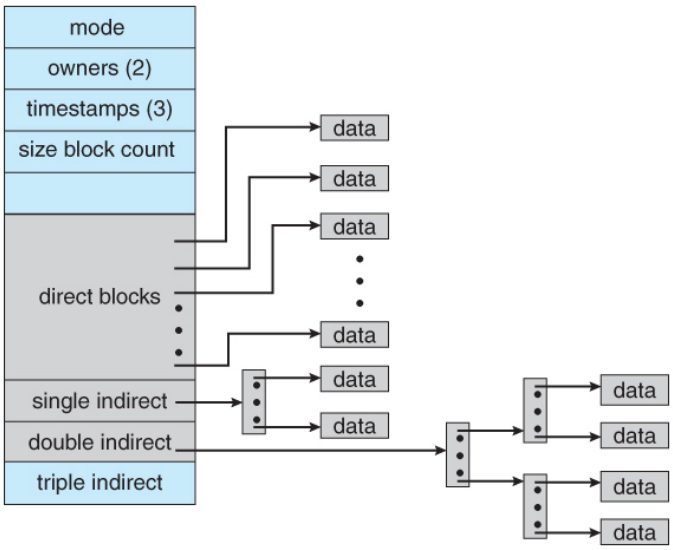

Figure 6.21: The structure of an inode. Direct pointers to blocks are referred to as "direct blocks" here. The inode in this example does not have a triple indirect pointer.

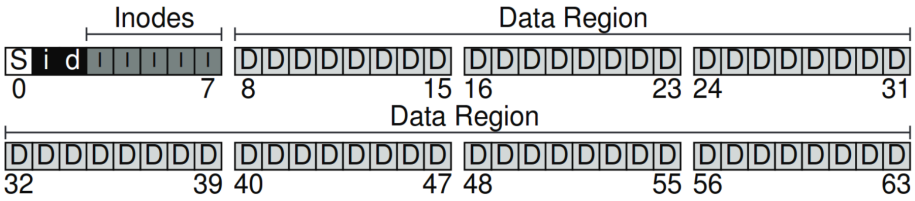

Figure 6.23: A high level view of how the file system is arranged on the storage device. **S** is the superblock, **i** the inode bitmap, **d** the data bitmap, **I** the inode region, and **D** the data region. The numbers refer to blocks. **i** is usually smaller than **d**.

Remarks:

- Figure 6.23 gives a schematic of an inode-based file system. Inode and data bitmap are quite small since we only need a single bit per inode/data block to determine whether the inode/data block is in use or not. The inode region also quite small – multiple inodes fit into a block of the storage device – the data region takes up most of the space.
- When we want to create a new file, we need to find space for two things: a free inode in the inode region we can use to represent the file, and enough space in the data region to store the data of the file. The two bitmaps will tell where to find this space.
- When we edit a file, we have to write the new data to the data region

and update the pointers in the file's inode, thus we do not need to change the inode bitmap.

- Inodes do not store file data or even a name for the file, but only metadata about the file and pointers to the file data. Per design, the information what a directory contains is stored in the data blocks that belong to the directory. Specifically, the data blocks of a directory store key-value pairs `(name: inode number)`.

Definition 6.24 (Hard Link, Soft Link)**.** *A **hard link** is an entry in a directory that refers to an inode, i.e. a key-value pair `(name: inode number)` in the directory data. Thus, the same inode (the same file!) can be accessed via multiple hard links. A **soft link** (or **symbolic link** or **symlink**) is a file with its own inode whose only content is the absolute path of the file it points to, the **target**.*

Remarks:

- Neither hard nor soft links create copies of file data, they are only different ways of pointing to a file!
- Hard links and soft links are not opposing concepts: there can be hard links to a soft link, and there can be soft links to a hard link.
- There is no way to tell which of two hard links to the same inode is the "real" one.
- Hard links face certain restrictions: they can only point to files inside the current file system, and creating additional hard links to directories is not allowed (because this could introduce cycles to the directory tree). Sometimes it might be convenient or necessary to have links to files in other file systems, or shorthands for directories, and soft links can help.
- Since hard links to directories are not allowed by many UNIX variants, there is a *canonical* (i.e., unique) path to every directory, following hard links.
- Soft links can point to directories, which could break tools that require the directory graph to be a tree. However, soft links are recognisable as soft links (they have a flag in their inode's metadata to indicate this), and so those tools can ignore soft links by simply not following them. With hard links, this is not possible.
- Since every hard link is conceptually intended to be its own link to the file, we can only delete a file when no more hard links to it exist. The metadata of an inode contains a field to store how many hard links to the file exist. Only when this counter reaches 0 can the inode and associated data be freed.
- A soft link is a file of its own, and when the file it points to is moved/renamed/deleted, the soft link points to a now-inexistent file – this is called a *broken*, *dead*, *orphaned*, or *dangling* link.

6.8 Virtual File System

Definition 6.25 (Mounting a File System)**.** *To access a file system, the operating system has to* ***mount*** *it: it has to associate it to a particular directory (the* ***mountpoint****) in the directory tree.*

Remarks:

- When you boot your computer, your operating system will mount the partition holding your boot file system to `/boot`. The file `/etc/fstab` on UNIX variants contains a list of all file systems that will be mounted during the boot process.
- UNIX has a file system mounted to the mountpoint `/dev` that contains mostly *device files* which provide raw access to devices; for example, the file `/dev/cdrom` contains the raw bytes of the CD in your drive. Similarly, `/proc` is a filesystem with a file for each process that's currently running, and `/sys` contains configuration files for the kernel.
- To mount a storage device, the virtual file system needs to know how to interact with the physical file system on the storage device. For this, a driver for the physical file system needs to be installed. For the Linux VFS, drivers for around 40 different physical file systems exist.
- You can also mount non-storage file systems. You can mount a program that generates its answers to file system requests from the operating system on the fly. The file `/dev/zero` just returns 0-bytes (as many as you ask for). It doesn't exist anywhere physically and is generated on the fly.
- You can also mount a device that is not in the same physical location as your computer; a remote file system via a network.

Definition 6.26 (Portable Operating System Interface, POSIX)**.** *It is a family of standards that defines an operating system interface and environment to make applications portable across operating systems.*

Remarks:

- The standard specifies programming interfaces, command interpreter and common utility programs among other services.
- POSIX is supported by various Operating Systems including macOS, mostly supported by Linux; also Windows has some POSIX-compliant parts.

Definition 6.27 (File Permissions)**.** *In POSIX, access to a file is managed via* ***permissions*** *(or* ***privileges****) assigned to a* ***user****, a* ***group*** *of users, and everyone (****other****). Every file is assigned a user as its* ***owner****, and a group as its* ***group owner****.*

Remarks:

- Permissions can be managed by restricting the owner's/group owner's/ everyone's permissions to *read*, *write*, or *execute* a file. Permissions are attributes of a file. They are represented in the *permissions string* (or *mode word* or *mode*).
- Mode, owner, and group owner are stored in the inode of a file; see the corresponding entries in Figure 6.21.
- If you own a file, you need *user* permissions to access it. If you are only in the owner group of the file, you need *group* permissions. If you are neither owner nor in the owner group of the file, then you need *other* permissions.
- To access (read, write, or execute) a file, one needs execute permissions for every directory in the file's canonical path. The execute permission bits of a directory are also called the *search bits*. Read permissions for a directory allow to read the names in the directory, but not follow them.
- Programs generally inherit their permissions from the user that starts them. If the *suid* bit of the file is set, the program inherits user permissions from the owner of the executable file instead; the *sgid* bit does the same for group permissions.
- Only the owner of a file can change its permissions, and he can do so without having any permissions for the file.
- The program `sudo` can allow a user to run a program with the permissions of another user, by default the *superuser* (also called *root*) who has unrestricted permissions for every file; these permissions of the superuser are hardcoded.

Chapter Notes

Hard disk drives were introduced in the 1950s. As all computing equipment, they were originally huge machines; the IBM 350 drive introduced in 1956 was the size of two tall refrigerators and offered space for just 3.75MB of data. They only became commonplace equipment in household PCs by the second half of the 1980s. See [6] and [1] to get an overview over the technological details regarding how HDDs operate. Regarding disk scheduling algorithms, [7, 3] are good starting points.

While still not as common as HDDs, SSDs are expected to become the default technology in the coming years by many experts. The tradeoffs between space overhead and write overhead are not understood very well yet; there is very little theory on the matter, and the known algorithms are heuristics without good upper bounds; see [2] regarding the worst-case tradeoff between space overhead and write overhead. [8] is a decent starting point for algorithms designed to reduce write overhead while providing a measure of wear leveling. For a recent survey of FTL techniques, see [5]. There are also file systems that try to minimize the impact of how files are managed on the performance SSDs

can provide. [4] is one of many attempts at building a file system specifically for SSDs.

This chapter was written in collaboration with Georg Bachmeier.

Bibliography

[1] Dave Anderson, Jim Dykes, and Erik Riedel. More than an interface-scsi vs. ata. In *FAST*, volume 2, page 3, 2003.

[2] Philipp Brandes and Roger Wattenhofer. Space and write overhead are inversely proportional in flash memory. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 9. ACM, 2015.

[3] David M Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position*. Palo Alto, CA: Hewlett-Packard Laboratories, 1991.

[4] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *FAST*, pages 273–286, 2015.

[5] Dongzhe Ma, Jianhua Feng, and Guoliang Li. A survey of address translation technologies for flash memories. *ACM Computing Surveys (CSUR)*, 46(3):36, 2014.

[6] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[7] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Technical Conference*, pages 313–323. Washington, DC, 1990.

[8] Benny Van Houdt. Performance of garbage collection algorithms for flash-based solid state drives with hot/cold data. *Performance Evaluation*, 70(10):692–703, 2013.