R5CYBgos - Introduction to computer systems - 2

Samuel Pélissier (samuel.pelissier@centralesupelec.fr)

2025

CentraleSupélec | université PARIS-SACLAY

Reminders

Why do we need an OS?

Problem

Why a file system?

To record and retrieve the information used by a process

- Impossible to save in the address space dedicated to a process in use

We must be able to:

- Manage simultaneous access to data
- Store information over time (after shutdown)
- Store large amounts of information (> RAM)

Solution: files

We store in files on disks or other media

- The stored information is permanent = not affected by the creation or termination of a process

Files are managed by the OS (structure, naming, usage, protection, implementation, ...)

- The part of the **OS that manages files** is called the **file system**

Two distinct concepts:

- User interface related to file access and usage
- Implementation of the file system in memory space

Solution: files

We store in files on disks or other media

- The stored information is permanent = not affected by the creation or termination of a process

Files are managed by the OS (structure, naming, usage, protection, implementation, ...)

- The part of the **OS that manages files** is called the **file system**

Two distinct concepts:

- **User interface related to file access and usage**
- Implementation of the file system in memory space

A file

- The **smallest unit of storage** in secondary memory **from the user's point of view**
- Stored in secondary memory according to a physical representation (descriptor + series of blocks, clusters, ...depending on the OS)
- Accessible to the user by its name (logical representation)
  - name + path = entry (same for directory)
- The link between logical and physical representations is ensured by the OS

File format

- It contains … a sequence of 0s and 1s
  - Text file == composed of bytes interpretable by a text editor
    - Byte > 0x20
    - The 0x7F range: interpretation depends on the region
    - See `ascii(7)` and `charsets(7)`

```
/tmp/my_directory ❯ cat hello.txt | xxd
00000000: 776f 726c 640a                           world.
```

- Binary **formats**: a predefined way to organize data
  - E.g., an executable, jpg, png, mp3…
  - The associated software (the OS, VLC, …) knows the format and can interpret the data

File format

- **Strong typing** (MS-DOS)
  - The extension (`.txt`, `.dat`, `.exe`...) is required to determine the nature of files
  - A file in MS-DOS can only be executed if it has the extension `.exe`, `.com`, or `.bat`
- **Inferred typing** (Unix)
  - The extension is irrelevant
  - The nature of the file is *inferred* by the system through analysis of the *file header*
  - An executable often has no extension under Linux
  - Extensions are used only for user convenience
    - `Image.jpg`
    - `Texte.txt`
    - `README.md`

File format

Illustration/demo

```
/tmp/my_directory ❯ file hello.txt
hello.txt: ASCII text
/tmp/my_directory ❯ cp hello.txt hello
/tmp/my_directory ❯ file hello
hello: ASCII text
```

Files names

- A file name is unique within a given directory
- It is possible to have identical file names in different directories
- File name composition under Linux:
  - Up to 256 characters
  - Avoid characters: `- * ? < > ! / \ <space>`
  - Case sensitivity: lowercase and uppercase are different (e.g., `Makefile` is different from `makefile`)
- There are hidden (configuration) files (starting with `.`)

File types

```
-rw-r--r-- 1 user group    6 19 sept. 11:24 hello.txt
```

File types

```
-rw-r--r-- 1 user group    6 19 sept. 11:24 hello.txt
```

The first letter determines the file type (7 types):

- **`-` ordinary file**
- **`d` directory**
- `l` link to a file or a directory
- `s` socket
- `b` special block file (e.g., `/dev/sda`, hard disks)
- `c` special character file (e.g., serial port, keyboard)
- `p` special FIFO file

Permissions

Permissions (or access rights) are defined by 3 characters for 3 categories of users: file **owner** (u), **group** (g), **others** (o). These 3 roles can be adressed at once via all (a).

```
-rw-r--r-- 1 user group    6 19 sept. 11:24 hello.txt
```

Owner Group Others

Access rights are of 3 types and represented by 4 possible characters:

- **Read** (`r`): access the content
- **Write** (`w`): modify the file
- **Execute** (`x`): execute the file
- None (–): absence of right

Permissions: debug

Will the `./script` command run?

```
cat script
# echo 'Hello !'
ls -l
# -rw-r--r-- 1 s s 18 19 sept. 17:32 script
./script
```

Permissions: debug

Will the `./script` command run? No: execution permission is required

```
cat script
# echo 'Hello !'
ls -l
# -rw-r--r-- 1 s s 18 19 sept. 17:32 script
./script
# bash: ./script: Permission denied
```

Permissions: changing access rights

chmod to the rescue

2 representations:

- Textual: who's the target (ugoa), operation (+-=), access right (rwx)

```
chmod ugo+rw hello.txt
chmod g-x script

chmod a+rwx every_permission_for_everyone.sh
chmod a=rwx every_permission_for_everyone.sh

chmod a-rwx no_permission.png
chmod a=--- no_permission.png
```

Permissions: changing access rights

- Numeric: sum the numbers corresponding to each node

Number	Meaning	Ref
0 (000)	No right	-
1 (001)	Execute	x
2 (010)	Write	w
4 (100)	Read	

When using `chmod`, we specify in order:
owner (user), group, and others:

```
chmod 644 hello.txt
chmod 744 script

chmod 777 every_permission_for_everyone.sh
```

File user and group

The ownership information is also shown in the `ls` output:

```
-rw-r--r-- 1 user group    6 19 sept. 11:24 hello.txt
```

We can also update this (if allowed):

```
chown anotheruser hello.txt
chgrp anothergroup hello.txt
```

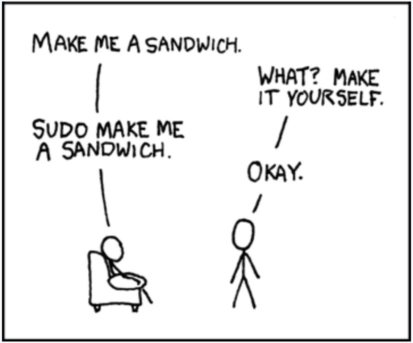
Quiz

2 users on the system: Alice & Bob. Can Alice change the permissions of Bob's file?

Interlude: sudo

SuperUserDO:

- Generally used to run a command as `root`
- Privilege escalation by design



Not the perfect solution when there is an error message!

Interlude: sudo

Can also run a command as another user (if allowed by the sudoers configuration):

```
sudo -u anotheruser chmod 644 anotheruserfile.txt
```

- This works only if your account has permission to run commands as anotheruser.
- By default, only users listed in /etc/sudoers can do this.

```
[username] ALL=(ALL:ALL) ALL      # add a specific user
%groupname ALL=(ALL:ALL) ALL      # add a group
```

File metadata TL;DR

```
-rw-r--r-- 1 user group    6 19 sept. 11:24 hello.txt
```

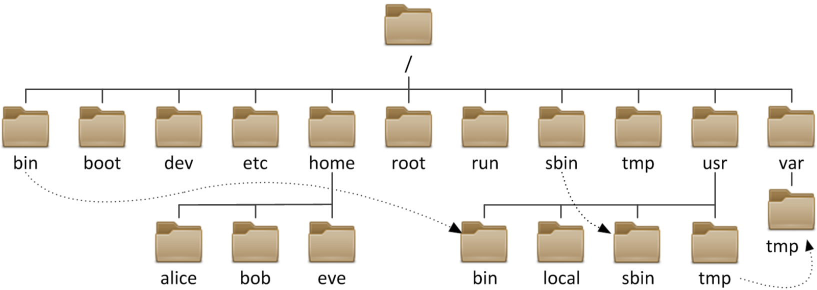
File metadata TL;DR

```
-rw-r--r-- 1 user group    6 19 sept. 11:24 hello.txt
```

- Type
- Permissions
- Number of hard links
- User and group
- Size
- Timestamp
- Name

Filesystem hierarchy

Files are organized in a tree (hierarchy)



Filesystem hierarchy

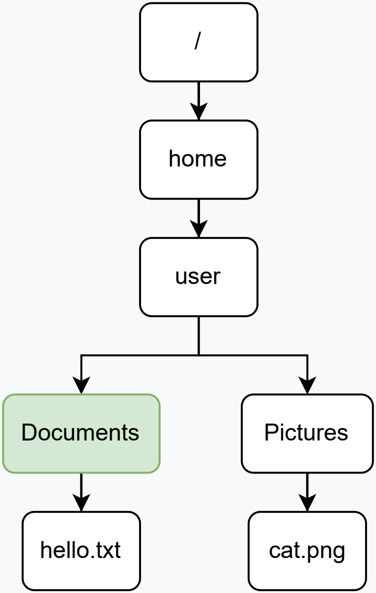
On Linux, partial filesystem hierarchy (FHS):

- `/` : root of the filesystem
- `/bin` : essential user binaries (executables needed in single-user mode, e.g. `ls`, `cp`)
- `/dev` : device files (interfaces to peripherals and kernel devices)
- `/etc` : host-specific system configuration
- `/lib` : essential shared libraries and kernel modules (e.g. `libc.so`)
- `/lost+found` : recovered files after filesystem corruption
- `/mnt` : temporary mount point for filesystems

Filesystem hierarchy

- `/usr` : secondary hierarchy, read-only user data:
  - `/usr/bin` : non-essential user binaries
  - `/usr/lib` : non-essential libraries
  - `/usr/share` : architecture-independent data (docs, icons, locales, etc.)
- `/tmp` : temporary files (may be cleared on reboot)
- `/home` : user home directories (`/home/user/`)
- `/var` : variable data (logs, spool files, caches, databases, etc.)

Navigating the tree

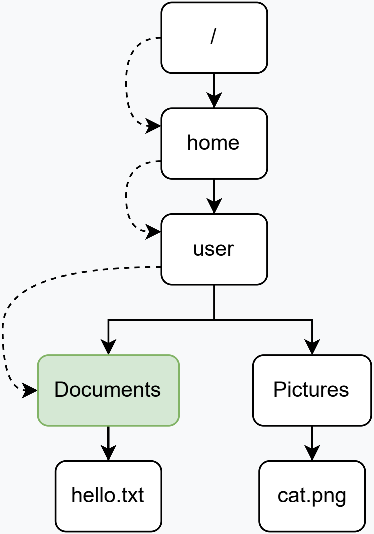


- Relative path: relative to the current working directory
- Absolute path: starting at the root (/)

Current directory: `Documents`

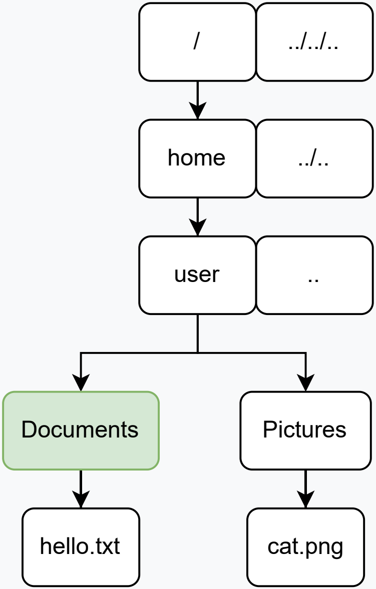
What is the relative path of `hello.txt`? Absolute path?

Navigating the tree



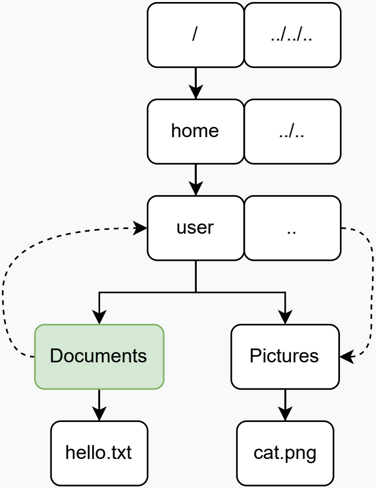

```
ls hello.txt
ls /home/user/Documents/hello.
```

Navigating the tree



How can we access the cat picture?
(Absolute path, relative path?)

Navigating the tree




```
ls ../Pictures/cat.png
```

Files TL;DR

Files...

- Allow to store information on disk
- Files are contained in directories, organized in a tree structure
- Are managed by the OS
- Are associated with metadata (e.g., permissions, users, group, create date...)

File System

On the OS side of things:

- Goal of the file management system
  - Manage information (organization on disk, protection, sharing...)
  - Provide an interface to access it:
    - Creation, reading, deletion independently of the physical structure
- Functions
  - Allocate and reclaim (secondary) memory
  - Keep track of free areas in (secondary) memory
  - Optimize access time and protect information

Partitions

A disk is divided into *partitions*

- A partition is a logically defined section of a disk
- It is represented by a special block device file (e.g., `/dev/sda1`)
- A partition can be formatted with a filesystem (e.g., `ext4`, `ntfs`, `btrfs`, `vfat`...)
- At the lowest level, a filesystem is just a way of interpreting the partition's raw binary data

Volumes

A *volume*:

- The unit of storage the OS makes available after formatting
- Typically a partition *with a filesystem*, but:
  - It may span multiple partitions or disks (e.g., LVM, RAID, ZFS pools)
  - It may be a whole disk with no partition table
- In Windows: usually shown as a drive letter (`C:`, `D:`, ...)
- In Linux/Unix: shown as a device (`/dev/sda1`, `/dev/mapper/vg0-home`, ...)
- **Partition = slice of a disk, volume = usable storage entity the OS mounts**

Mount Points

A volume is *mounted* on a mount point

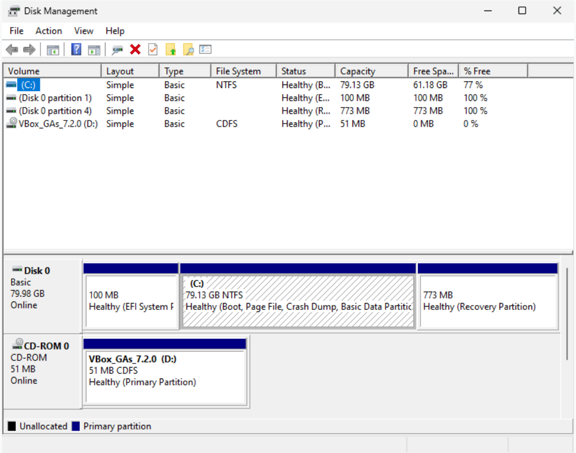
- A *mount point* is a directory (preferably empty)
  - Linux tools: `mount(8)`, `umount(8)`
  - Option `-o loop` allows mounting a regular file as if it were a block device (e.g., .iso)
- This makes the filesystem content available under that directory
- Multiple filesystems can coexist in a single directory tree
- On Linux:
  - The root filesystem (/) is mounted at boot
  - Other volumes (e.g., `/home`, `/var`) can be mounted later
  - Configuration is usually stored in `/etc/fstab`

Note on Terminology

- People often say "*mount a partition*"
- Technically:
  - A *partition* is just a slice of a disk
  - A partition becomes a *volume* once it has a filesystem
  - The OS actually mounts the *filesystem (volume)* on a mount point
- Example:
  - Command: `mount /dev/sda1 /mnt`
  - What really happens: the filesystem stored in `/dev/sda1` is mounted at `/mnt`

Partitions/volumes

Illustration/demo



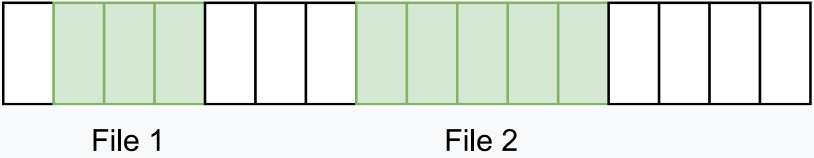

```
~ ❯ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/dm-0       937G  653G  237G  74% /
devtmpfs         32G     0   32G   0% /dev
```

Storing files

- A file is stored in memory as blocks of bytes
- The allocation of a file's blocks in memory affects filesystem performance
- Several allocation strategies:
  - Contiguous allocation
  - Non-contiguous allocation
  - Pros/cons: similar to static vs dynamic allocation in C

Storing files: contiguous allocation

Allocation of a number of consecutive (fixed-size) blocks according to the file size



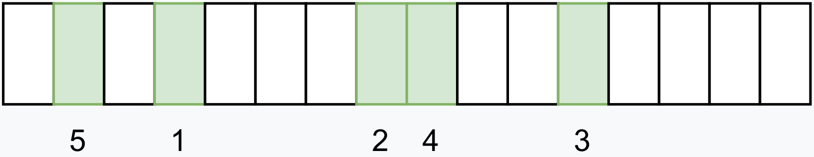
Simple implementation

- Fast access
- Memory waste (unused fragments)
- Expensive file compaction
- File relocation required in case of extension (memory reallocation)

Storing files: non-contiguous allocation

Allocation of a number of file blocks not necessarily consecutive

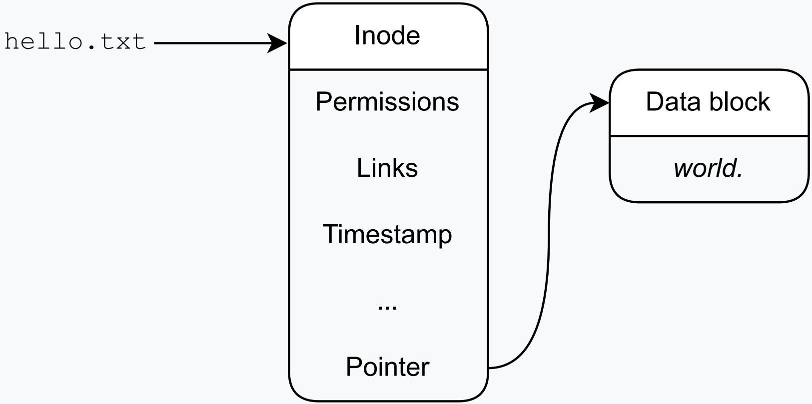
Example for a 6-block file:



- No memory waste
- File extension without reallocation
- Three methods to locate the blocks:
  - Linked list of blocks
  - Indexed linked list (of block numbers): `FAT12`, `FAT16`, `FAT32`, `VFAT`
  - Inode or information node: `ext2`, `ext3`, ...

Storing files and metadata: inodes

Inodes store a file's metadata (permissions, timestamps, etc.) along with pointers to the data blocks on disk



Storing files and metadata: inodes

Inodes are used on Linux: ext2, ext3, ext4

```
 * Structure of an inode on the disk
 */
struct ext2_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Creation time */
        __le32  i_mtime;        /* Modification time */
        __le32  i_dtime;        /* Deletion Time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_flags;        /* File flags */
```

Source: Linux kernel 6.17

(Similar concept on Windows: `FileId`)

Storing files and metadata: inodes

Illustration/demo

```
~/Documents/work/cours/CS/ordi/demo/my_directory main !1 ?5 ❯ ls -li
total 8
53761173 ---------- 1 s s  6 19 sept. 11:49 hello
53767430 lrwxrwxrwx 1 s s  9 19 sept. 18:11 hello2 -> hello.txt
53761174 -rw-r--r-- 1 s s 18 19 sept. 17:32 script
```

Storing files and metadata: inodes

What is the maximum size of file?

Storing files and metadata: inodes

What is the maximum size of file?

number of addressable blocks * block size

Storing files and metadata: inodes

What is the maximum size of file?

number of addressable blocks * block size

There is a maximum number of pointers per inode.

What happens if the file is too large for this maximum number?

Storing files and metadata: inodes

What is the maximum size of file?

number of addressable blocks * block size

There is a maximum number of pointers per inode.

What happens if the file is too large for this maximum number?

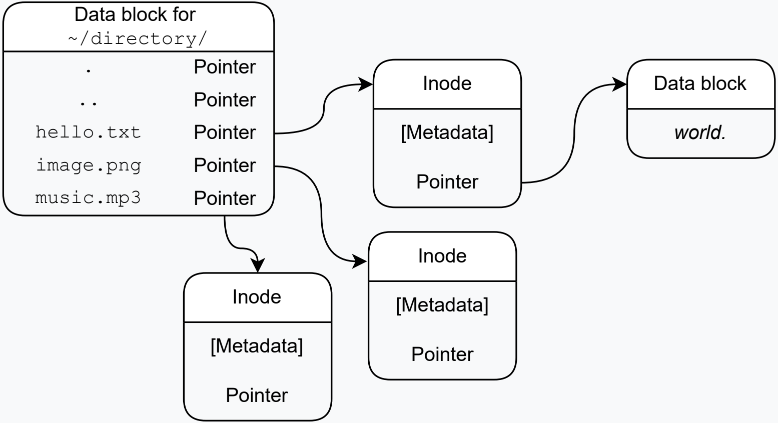
- The filesystem uses **indirect blocks**
- These are blocks that store additional pointers
- Can be single, double, or triple indirection (pointers to blocks of pointers)

Storing files and metadata: inodes

What is a directory then?

Storing files and metadata: inodes

What is a directory then?



A special file that contains entries mapping filenames to inode pointers

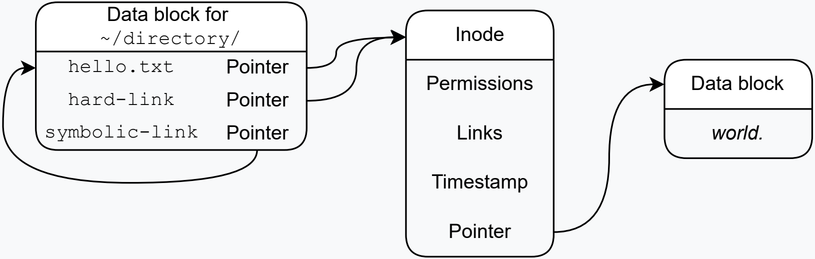
File links

This number is the number of *hard links* on the file

```
-rw-r--r-- 1 user group    6 19 sept. 11:24 hello.txt
```

File links

Two types of links:

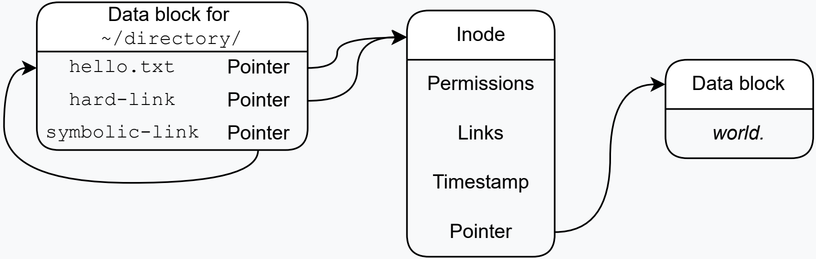


Symbolic links:

- symbolic-link is an alias pointing towards hello.txt
- If we update hello.txt, symbolic-link appears modified (and vice versa)

File links

Two types of links:

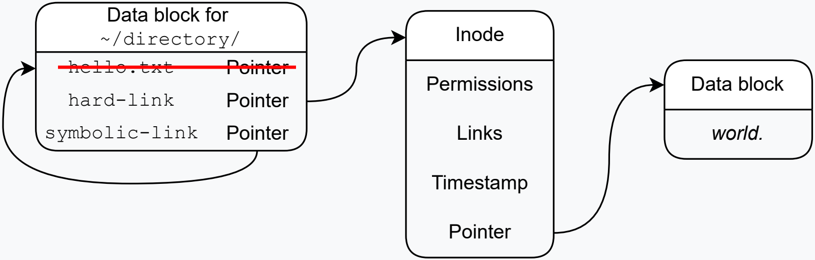


Hard links:

- `hard-link` and `hello.txt` point to the same inode
- If we update `hello.txt`, `hard-link` appears modified (and vice versa)

File links

If we delete `hello.txt`:



- The symbolic link is broken
- The hard link still works

Quiz

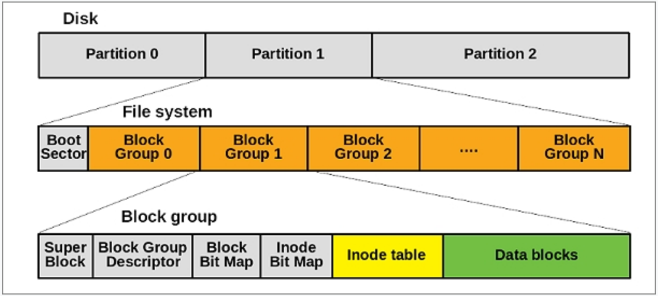
What is the fastest: copying or moving a file? Why?

File system: TL;DR

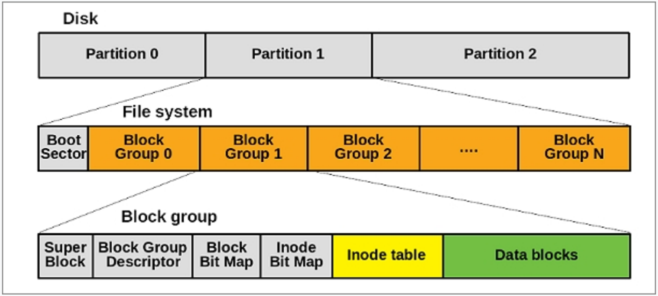
- Disks are divided into *partitions*
- Each partition can contain a filesystem that organizes the files
- A volume (partition+filesystem) is *mounted* into the directory tree to make it available to users
- Files and directories are managed through *inodes*
- Inodes store metadata (permissions, creation date) *and pointers to the file's data blocks*

ext2 file system

- Historically used on Linux, now replaced with ext4
- Starts with a **superblock** containing global filesystem information
- The filesystem is divided into **block groups** (typically hundreds of MB in size)



ext2 file system



Each block group contains:

- A copy of the superblock and group descriptor (for redundancy)
- A bitmap of blocks (which blocks are free/used)
- A bitmap of inodes (which inodes are free/used)
- An inode table (dedicated space for inodes of this group)
- Data blocks

ext2, ext3, ext4

- ext3: adds journaling to the filesystem
  - Allows better recovery from errors
- ext4: widely used on Linux currently
  - Supports largerfiles
  - Reduces file fragmentation
  - Improves memory management of files
  - Backward compatible with ext2 and ext3
  - Still based on inodes with persistent information: mode, owner, etc.
  - Addressing fields from ext2/ext3 replaced by extents
  - Cluster size: 4 KiB to 64 KiB
  - Maximum file size: 16 TiB
  - Maximum volume size: theoretically 1 EiB (kernel code not yet compatible); practical max 16 TiB

btrfs

ext4 is cool, but we have even cooler toys now.

Data is easily duplicated (e.g., multiple similar VMs). How can we optimize storage usage?

Copy-on-Write (COW) principle:

```
// (In practice, modern C++ strings may not use copy-
// but the idea is similar)
std::string x("Hello");
std::string y = x;
// x and y use the same buffer!
y += ", World!";
// Now, y uses a different buffer;
// x still uses the same old buffer
```

btrfs

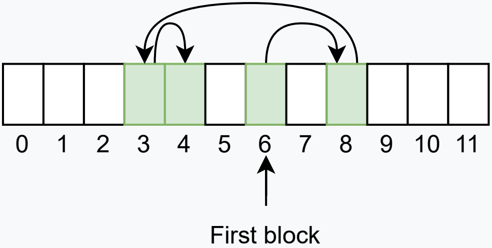
btrfs:

- Modern copy-on-write filesystem
  - Multiple files can reference the same on-disk blocks
  - When modified, data is copied (COW) instead of overwritten
- Supports efficient incremental snapshots

Non-contiguous memory allocation: linked lists

A linked list of blocks:

- A block contains data and a pointer to the memory address of the next block



- The pointers are stored on disk

Non-contiguous memory allocation: linked lists

What if we want to read the last part of a file?

Non-contiguous memory allocation: linked lists

What if we want to read the last part of a file?

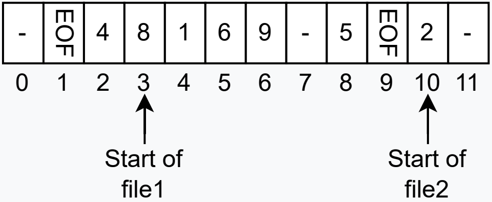
Slow access: all preceding blocks must be traversed to reach a given block.

FAT: linked list with a central index

FAT = File Allocation Table

- Indexed linked list of clusters
  - One entry per cluster: next cluster index/number, EOF, or free (0000)
- The disk is divided into clusters (allocation units, $\sim$ ext4 blocks)
- A cluster is a fixed number (2^n) of contiguous 512-byte sectors
- **The FAT contains one entry for every cluster on the disk**
- Files occupy an integer number of clusters; unused space in the last cluster is wasted
- On average, *half a cluster is wasted per file*

FAT: linked list with a central index

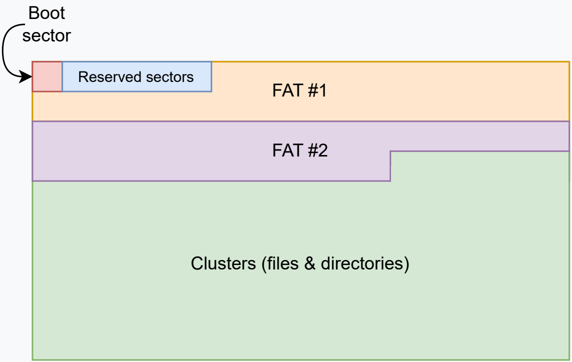


- File 1: starts in cluster 3
  - Clusters involved: 3 - 8 - 5 - 6 - 9
  - Cluster 9 is not necessarily fully used
- File 2: starts in cluster 10
  - Clusters involved: 10 - 2 - 4 - 1
  - Cluster 1 is not necessarily fully used

FAT: short history

- Designed by/for Microsoft (1977)
- FATX: X is the number of bits needed to encode the indexes
  - FAT12: maximum of $2^{12} = 4096$ clusters of fixed size (between 512 B and 4 KiB)
  - Maximum file/volume size: 4096×4 KiB $= 16$ MiB
  - FAT16: maximum file/volume size: 4 GiB
  - FAT32: in theory, maximum file/volume size: 16 TiB
  - In practice, maximum file size cannot exceed 4 GiB (versus the theoretical 16 TiB): see directory descriptor limitation — file size encoded on 32 bits
- VFAT: improvement of FAT to extend file names since Windows 95
  - **Still widely used for its simplicity on USB drives and EFI partitions**
  - Compatible with most operating systems
- exFAT: Extended FAT; supports larger files
- Windows: replaced by NTFS

FAT filesystem structure



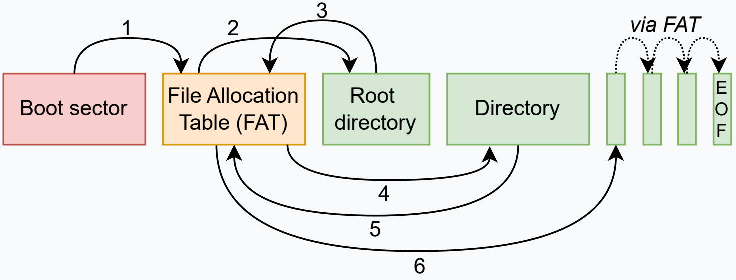
The boot sector (1^{st} sector of the volume) contains:

- Cluster size
- Starting cluster of the root directory (index in FAT)
- Size of the FAT

FAT filesystem structure

What's the plan?

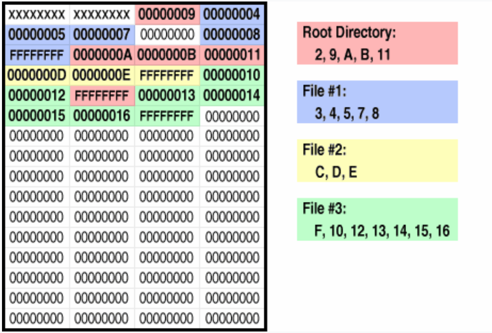
- Boot sector → root directory starting cluster (index in FAT)
- Root directory cluster → entries with file/subdirectory names and first cluster numbers in FAT



Structure of the FAT itself

The FAT = an array of 32-bit (or 16-bit or 12-bit) integers

- value == 0: the cluster is available for allocation
- value != 0 && != 0xFFFFFFFF: cluster used, the entry value points to the next cluster
- value == 0xFFFFFFFF, cluster used, and it is the last cluster of the file or directory



xxxxxxxx	xxxxxxxx	00000009	00000004
00000005	00000007	00000000	00000008
FFFFFFFF	0000000A	0000000B	00000011
0000000D	0000000E	FFFFFFFF	00000010
00000012	FFFFFFFF	00000013	00000014
00000015	00000016	FFFFFFFF	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

FAT pros & cons

- **Pros:**
  - Fast access if the table is *entirely present in memory*
- **Cons:**
  - The entire FAT is needed even if only one file is open
  - The FAT can be large if the disk is large (even if few blocks are used)
    FAT size = number of clusters $\times$ size of FAT entry ($\sim$4 B)
    For a 40 GiB disk with 4 KiB clusters: $\sim$80 MiB for the 2 FATs
  - The linked list in the FAT requires traversing all entries to access the end of a file — access time increases, especially since there is no pointer to the parent directory
  - File size is recorded in the directory — access time between this field and the data increases
  - 32-bit field: limits file size to 4 GiB
    Maximum file size is also determined by the maximum number of clusters and cluster size

New Technology File System (NTFS)

- Microsoft's take on a modern filesystem
- Replaces the FAT by a Master File Table (MFT) + a bitmap tracking free clusters
  - Each entry = information about a file (size, date, permissions)
  - Similar to an inode (see slide 38)
  - Also contains the list of clusters containing the file
  - MFT size increases with each file creation
  - If the file is small, it is stored directly in the MFT (in the block list space)

New Technology File System (NTFS)

Compared to FAT32:

- **Pros:**
  - Faster
  - More secure (addition of Access Control Lists (ACLs))
  - File system journaling
- **Cons:**
  - Not backward compatible with FAT32
  - Documentation limited: Linux drivers (ntfs-3g package), included in recent distributions. Can have strange effects if not properly unmounted

Resources

🎁 Btrfs for mere mortals: inode allocation, Marcos Paulo de Souza, https://mpdesouza.com/blog/btrfs-for-mere-mortals-inode-allocation/

Why did Windows use the FAT structure instead of a conventional linked list with a next pointer for each data block of a file?, @NonNumeric, https://stackoverflow.com/a/22424829

Understanding FAT32 Filesystems, Paul J Stoffregen, https://www.pjrc.com/tech/8051/ide/fat32.html