Operating Systems: Files

Fabrice BOISSIER <fabrice.boissier@epita.fr>

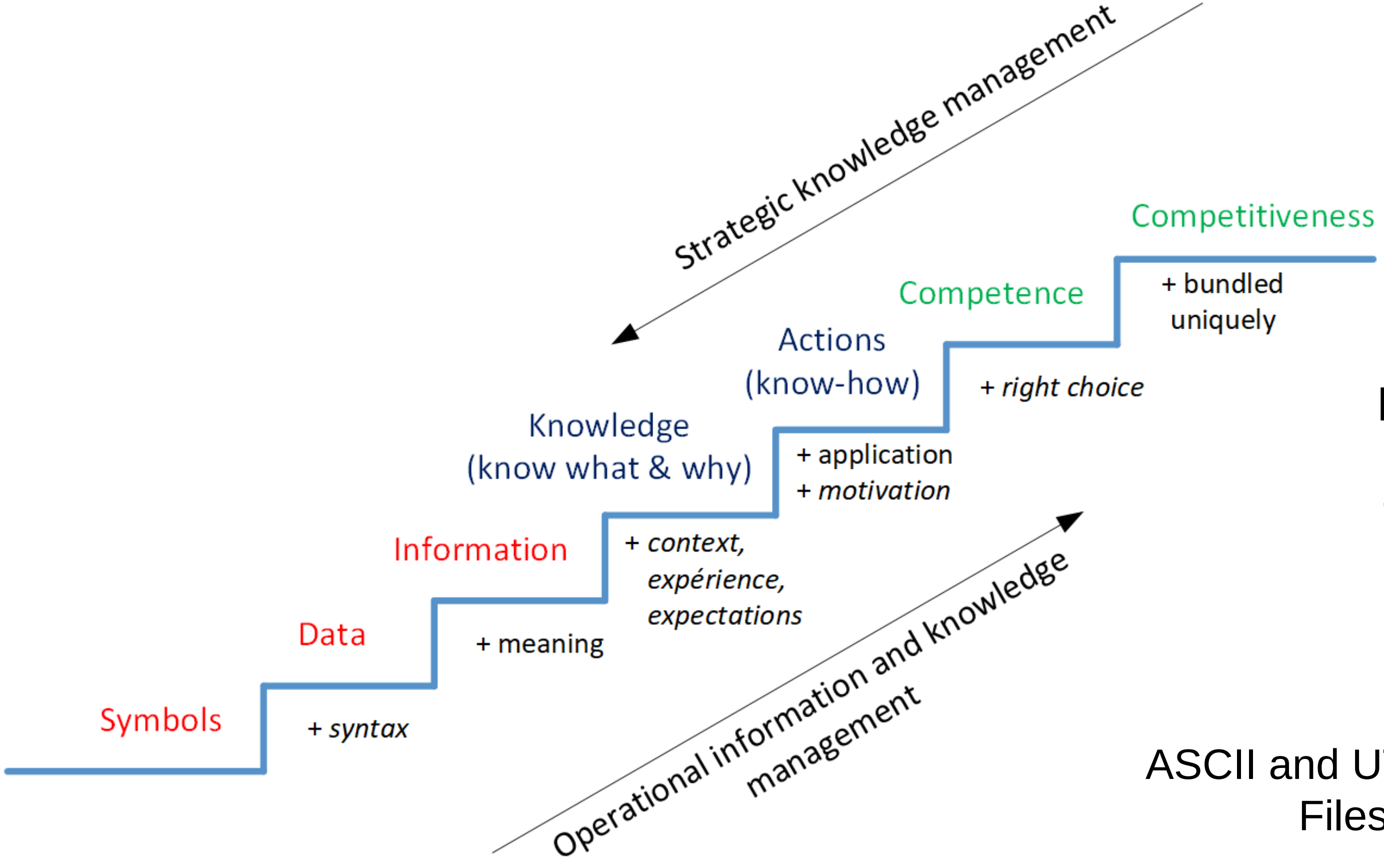
2021-11-02

Why storing data?

Functionally in mankind:

- Persistence of data
    - Keep data for a later use (hours, days, weeks, months, years, ...)
    - Remember the past/facts/history
    - Keep a trace of ownership or transaction
    - Identify peoples
    - ...

Why storing data?


Strategic knowledge management
Competitiveness
+ bundled uniquely
Competence
Actions (know-how)
+ right choice
Knowledge (know what & why)
+ application
+ motivation
Information
+ context, expérience, expectations
Data
+ meaning
Symbols
+ syntax
Operational information and knowledge management

Knowledge ladder

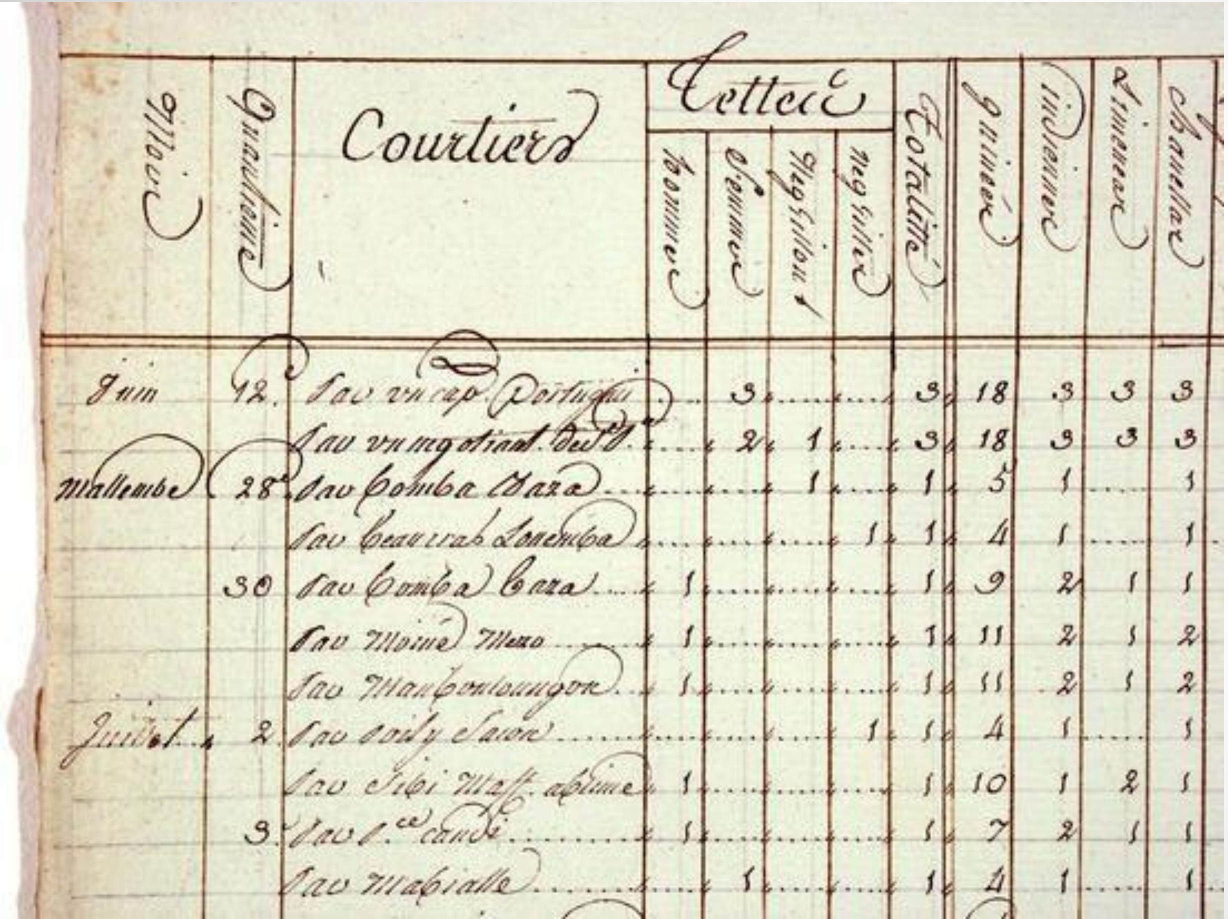
[Klaus North & Gita Kumta 2018 Knowledge management : Value creation through organizational learning]

Data is essential, yes… but not only data. The « knowledge era » requires information and data.

ASCII and UTF-8 are the « symbols ». Files contain « data ».

(I promise there are 4 slides only)

Very old history: how to store data?



Mois	Quantième	Courtiers	Têtes				Totalité	Guinées	indiennes	Limeneas	Barrilles
			hommes	femmes	négrillons	négrittes					
Juin	12.	Par un cap. Portugais		3			3	18	3	3	3
		Par un negotiant. Dudit		2	1		3	18	3	3	3
Mallembe	28.	Par Comba Caza			1		1	5	1		1
		Par Beauirab Lonemba				1	1	4	1		1
	30	Par Comba Caza	1				1	9	2	1	1
		Par Moine Mezo	1				1	11	2	1	2
		Par Manfoutoungon	1				1	11	2	1	2
Juillet	2.	Par Coily Saion				1	1	4	1		1
		Par Tibi Maf. afaime	1				1	10	1	2	1
	3.	Par 1.er Cande	1				1	7	2	1	1
		Par Mafialle		1			1	4	1		1

Obviously, by writing in a structured document…

You should see what is recurrent text on each page, and what is added by the user

Well done, you know what is in the declaration of the « struct », and what is put in each field during execution of the « process » by a human!

(I promise there are 4 slides only)

Very old history: where to store data?

1 card is equivalent to 1 record in a database or in a dataset (or 1 line in a file)

1 drawer is equivalent to 1 table in a database, or 1 file

1 furniture is equivalent to 1 database, or 1 directory

(I promise there are 4 slides only)

Very old history: how to find back data?

Where is the data about John Smith?...

Let’s search in the « J » or « S » closets until we find it…

(This furniture is litterally a « fichier » [file] in french)

(I promise there are 4 slides only)

Old history: how to find back data?

Decades later:

Where is the data about John Smith?...
(and which John Smith are we talking about?)

IBM punch card storage at the NARA
(National Archives and Records Administration)

(1 cardboard box
=
2,000 cards)

Why storing data?

Technically with IT:

- Persistence of data
  - Keep data even after a process has ended
  - *or between each reboot... or in case of a power outage*
- Quantity of data
  - Large amount of data that do not fit in memory
- Data sharing
  - Share the same data between multiple processes

Glossary

- Media or Support:
  - The hardware that contains blocks
  - Can be seen as a huge array with cells of a fixed size

    ...ho ho ho, it looks like memory... (« auxiliary memory » ?...)
- Block:
  - Contains a fixed size of data
  - Minimum part of a file
  - Also, the « cells » contained in the media
- Directory:
  - An abstract container of files
- File:
  - An abstract container of records or raw data

Reminder on measures & prefixes

Value			SI
10^3	1000	k	Kilo
10^6	1000^2	M	Mega
10^9	1000^3	G	Giga
10^12	1000^4	T	Tera
10^15	1000^5	P	Peta
10^18	1000^6	E	Exa
10^21	1000^7	Z	Zetta
10^24	1000^8	Y	Yotta

In formal documentation, you might find the *power of two* version of the units…

Just remember it exists and some conversions are sometimes required

Value			IEC
2^10	1024	Ki	Kibi
2^20	1024^2	Mi	Mebi
2^30	1024^3	Gi	Gibi
2^40	1024^4	Ti	Tebi
2^50	1024^5	Pi	Pebi
2^60	1024^6	Ei	Exbi
2^70	1024^7	Zi	Zebi
2^80	1024^8	Yi	Yobi

1KB = 0,97 KiB *1MB = 0,95 MiB* *1GB = 0,93 GiB* *…*

Physical Storage

Access methods

- Sequential access
    - Needs to read (or at least passes on) each cell before the required data
    - Each new data is written at the end of the media
    - Typical example: magnetic tape

- Random access
    - Direct access to data (or nearly direct) for read or write
    - *Optionnally by using an index updated after each new write/delete*
    - Typical examples: disk drives, flash memory

Access methods

- Access methods concern multiple aspects

- Access to a record « within » a file...
  - *Usually as a developper by organizing/structuring your data*
- Access to a file « within » a media...
  - *Usually as a program by using file system services*

(In a certain way, accessing data within a stack or a fifo is not so far)

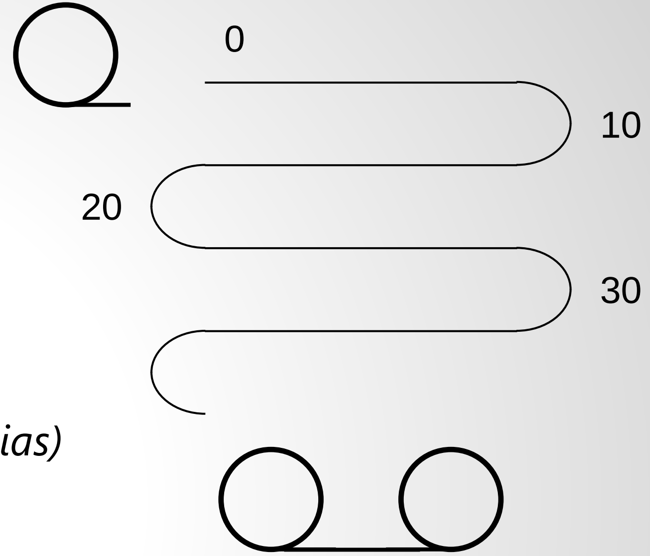
Sequential medias

- Magnetic tapes

- You must unroll/rewind the media until the beginning of your data

- Very VERY cheap
  - LTO: version 8, 12TB raw/30TB compressed, for 75~140€
- Very VERY dense
  - LTO: version 8, 12~30TB on 10x10x2cm for 200g (strip size: 960m)

Sequential medias



- Magnetic tapes
- Current usage:
    - Archiving
    - Backup *(is replaced by disks and regular medias)*

- Might be WORM (Write Once Read Many)

Even if the « sequential medias » tends to disappear, the abstract concept is similar to some structures, like the LIFO.

For example in any stack container (like in C, or the std::stack in C++) you must use « top » to get the element, and « pop » to go deeper.

Random access medias

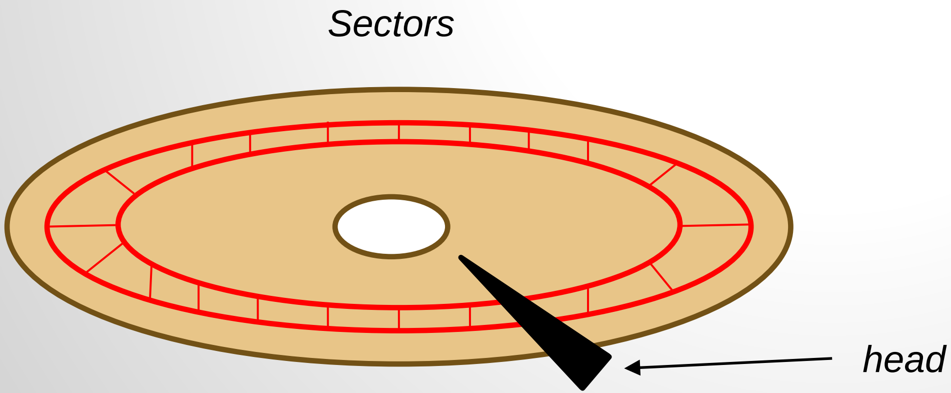
- Disk drives, Flash Memory, …

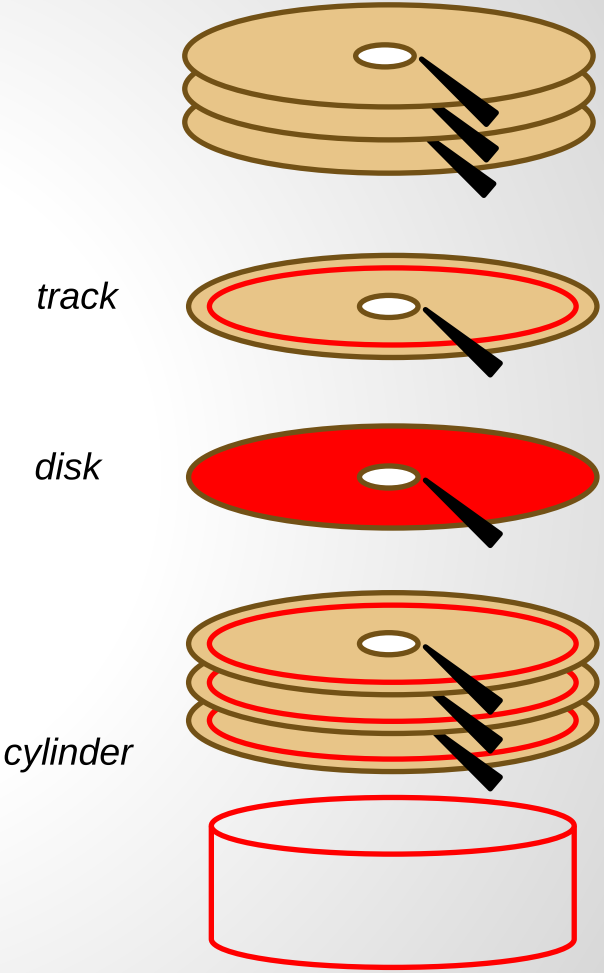
- You directly access data by giving its position on the device
  - Exactly like pointers in memory, except its an address « on » the device

- Cost, density, speed, durability … are variables
  - Choose wisely based on the characteristics you wish

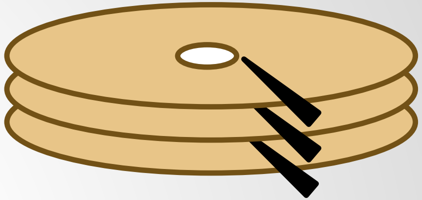
Random access medias

- Disks
- Sector, track, disk, cylinder
    - Cylinder, Head, Sector (CHS)
- Sector: smallest physical unit





Random access medias



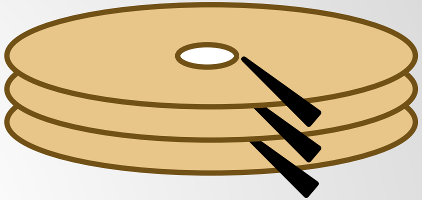
- Disks

- Currently, using sectors only (see LBA)
  - Firmwares abstract the hardware…
  - Your kernel asks for sectors, the firmware gives sectors…
    (The firmware manages the disk's internal buffers and the mapping)
  - *LBA Block = (C × Head per Cylinder + H) × Sectors per Tracks + (S − 1)*

- *See SMR and CMR if you expect to buy a NAS…*
  - *These are the physical methods to store data*
    (North / South in magnetism, size of the cells, …)
  - *The lower level still has an impact on the usage and File System*

Random access medias

- Disks



- Pretty flexible, but still a bit slow and dies easily
  - Capacities are pretty large
  - Access time is slow, but waaaay better than tapes
  - Too much mechanical parts == fragile
- See RAID for redundancy
  - Data is copied multiple times…
  - …or blocks are replicated
  - If one disk dies, the data is still intact

Random access medias

- Flash Memory

- Extreme speed of access, but less capacity of storage
  - *See NAND and NOR memory cells*

- And variable lifetime: each write reduces the lifetime
  - Cells are sensibles… *(see NAND and NOR memory cells)*
  - Again: the firmware manages the physical locations in order to avoid destroying the memory cells too quickly

- Used in SSD (Solid States Disks), Memory Card, …

Physical supports read and store data…

…Kernel manages how to organize & retrieve data for a user without knowing the physical supports

File Systems

File Systems: Objectives

Functionally:

- How to store data efficiently?
- How to retrieve data quickly?

Technically:

- How to organize data physically?...
- ...and expose a nice abstraction?

File Systems

- Abstraction from the physical supports
  - No need to search for sectors
  - Uses « blocks » (or « clusters »)
  - Block: smallest allocation unit managed by the file system
  - The file system manages the relations between sectors and blocks
    *1 block/cluster is 1, 2, 4, 8, 16, … sectors (N * sector size)*
  - *Currently: sector size = 4096 B = 4KiB*

- Presents data within abstracts objects
  - Files, directories, …

File Systems

- Tons of filesystems...
  - FAT32, NTFS, ext2/3/4, Reiser4, ZFS, ...
  - ISO9660, exFAT, ...

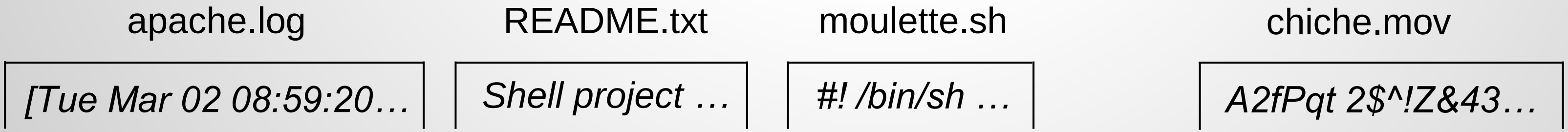
- Some FS have special objectives
  - Network or distributed file system

- *Reminder: FS are dependant of the physical support!*

File Systems

- File

- Contiguous records packed within a container
  - *Even movies or pictures are built with a precise structure and records*
  - *Repetition of images, pixels, …*
  - *Only the right interpretation of that data makes sense*
- Well, it can also be *one* record of raw data
  - *Like raw data from a probe*
  - *Only the right interpretation of these data makes sense*



apache.log

[Tue Mar 02 08:59:20…

README.txt

Shell project …

moulette.sh

#! /bin/sh …

chiche.mov

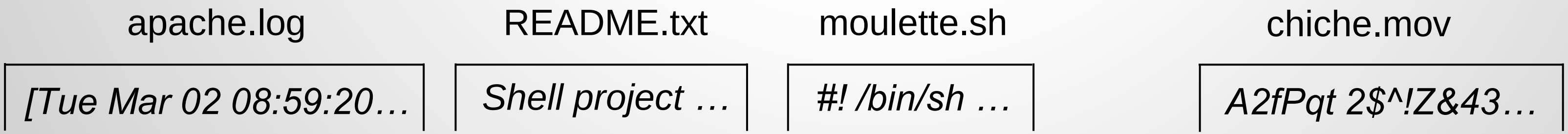
A2fPqt 2$^!Z&43…

File Systems

- File

- Files are identified by their names
  - How to retrieve them in the physical support?

- Some attributes are attached
  - It depends on the file system and which attributes are managed



apache.log

[Tue Mar 02 08:59:20…

README.txt

Shell project …

moulette.sh

#! /bin/sh …

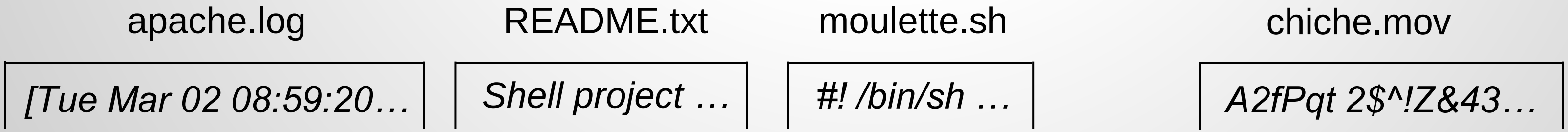
chiche.mov

A2fPqt 2$^!Z&43…

File Systems

- File

- Examples of attributes:
  - Name
  - Size (logical, physical, maximal allowed size, ...)
  - Date of creation, access, ...
  - Permissions (user/group/other, creator, owner, ...)
  - ...



apache.log

[Tue Mar 02 08:59:20…

README.txt

Shell project …

moulette.sh

#! /bin/sh …

chiche.mov

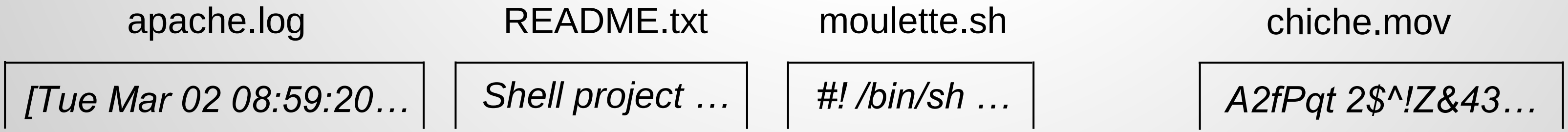
A2fPqt 2$^!Z&43…

File Systems

- File

- In some file systems, each file has a type
  - Typically in the Windows world: the extension (the 3 last letters)
    .exe => executable,   .bmp => bitmap,   .txt => text,   …
  - « Magic numbers »
  - (On some OSes, the type was stored in the file attribute)
  - Currently: everything is just a flat file

Depending on the type, the OS might know what to do with it
(which program to use and/or how to interpret data inside)



apache.log

[Tue Mar 02 08:59:20…

README.txt

Shell project …

moulette.sh

#! /bin/sh …

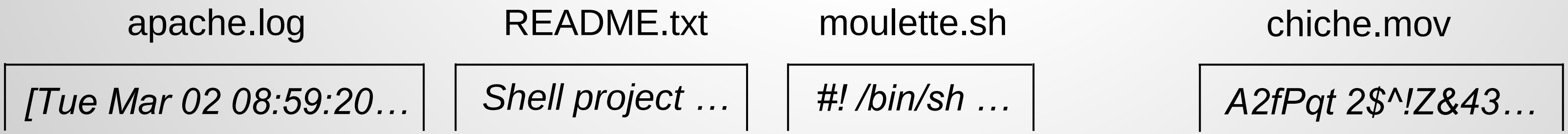
chiche.mov

A2fPqt 2$^!Z&43…

File Systems

- File

- Operations on files:
  - Create
  - Delete
  - Open
  - Close
  - Read
  - Write...



apache.log

[Tue Mar 02 08:59:20…

README.txt

Shell project …

moulette.sh

#! /bin/sh …

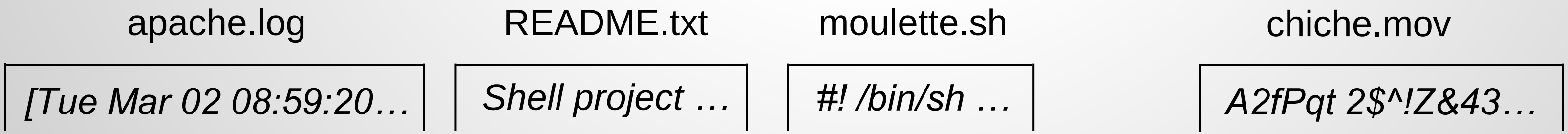
chiche.mov

A2fPqt 2$^!Z&43…

File Systems

- File

- Operations on files:
  - Append *(writes at the end of the file)*
  - Seek *(moves the read/write cursor inside the file)*
  - Get Attributes
  - Set Attributes
  - Rename



apache.log

[Tue Mar 02 08:59:20…

README.txt

Shell project …

moulette.sh

#! /bin/sh …

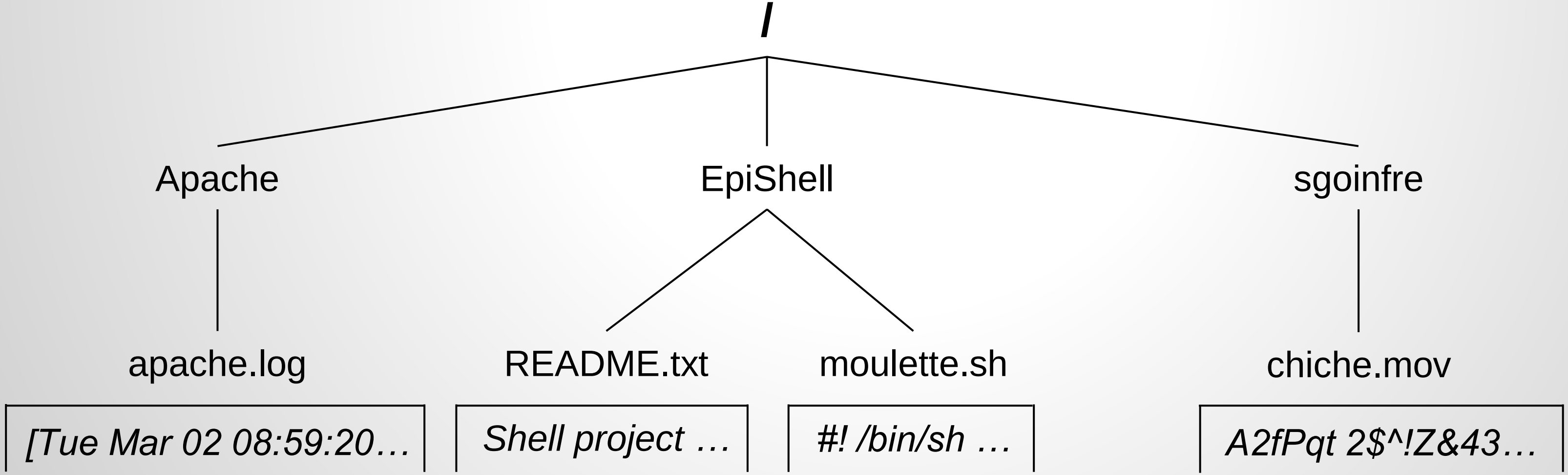
chiche.mov

A2fPqt 2$^!Z&43…

File Systems

- Directory (or folder)

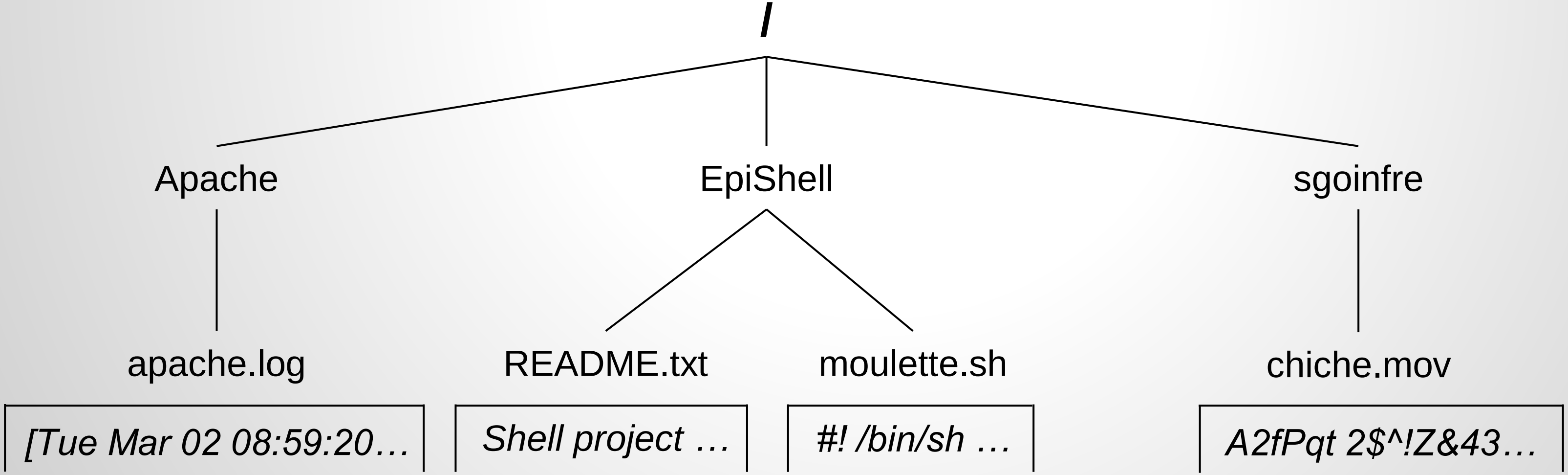
- Abstract container of files
  - *Well, it does have a physical existence and it contains data*



File Systems

- Directory (or folder)

- Directories are also identified by their names
  - And also have attributes (that may vary depending on the file system)



/

Apache

EpiShell

sgoinfre

apache.log

README.txt

moulette.sh

chiche.mov

[Tue Mar 02 08:59:20…

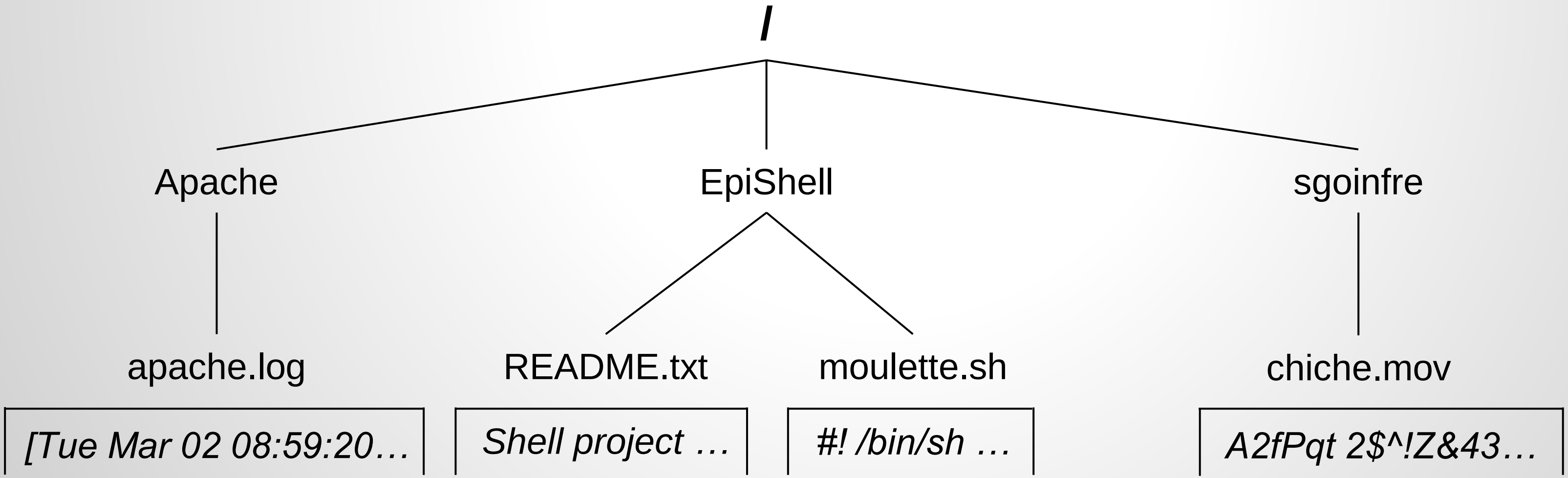
Shell project …

#! /bin/sh …

A2fPqt 2$^!Z&43…

File Systems

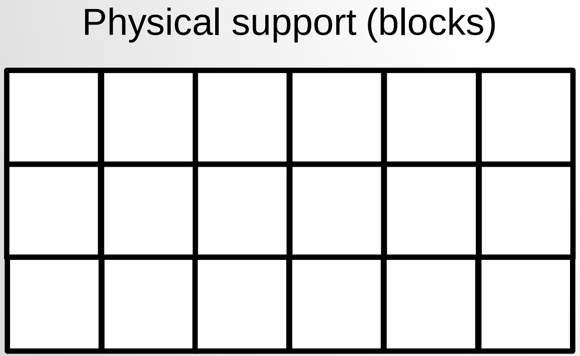
- *Hierarchy of directories and files*
    - ○ *Check hier(7) for UNIX hierarchy and where to store what*

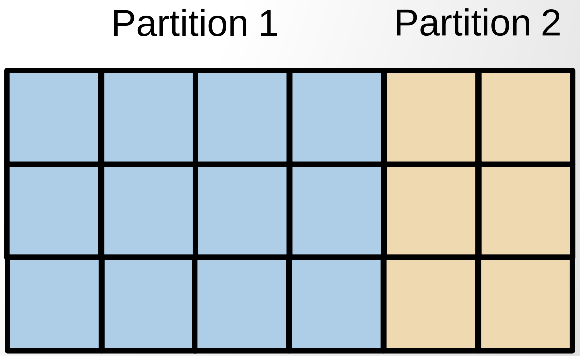


File Systems

- Partition (or old definition of « Volume »)

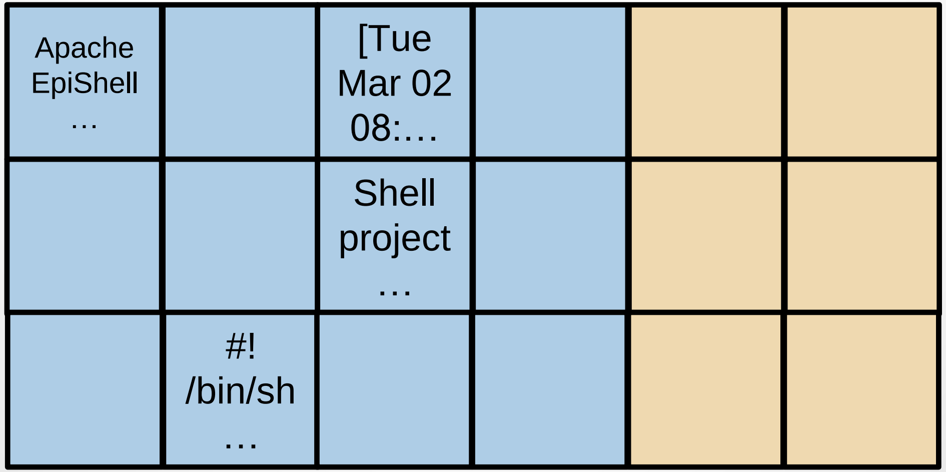
- A section on the physical support
  - *Separates the physical support into smaller storage spaces*
  - *Might be one partition with the full physical support into it*





File Systems

- Organization of files and folders within a partition
  - *Some file systems might be distributed on multiple physical supports…*
  - *…or through a network protocol*
  - *See also NFS, AFS, ZFS, DFS, …*



Apache EpiShell …

[Tue Mar 02 08:…

Shell project …

#! /bin/sh …

File Systems: Other objectives

1. How to keep a list of free blocks? *(Similar to malloc(3))*
   - ○ How to quickly find a free block?
   - ○ How to avoid fragmentation?

2. How to find the blocks that compose a file?

3. How to store attributes of files and directories?
   (Owner, group, rights, last access, date of creation, ...)

FAT example

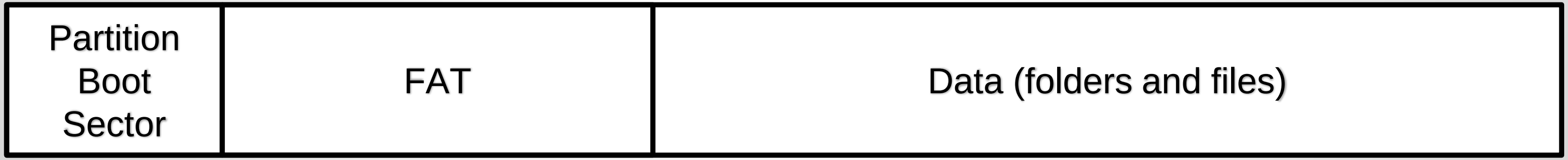
File Systems: FAT example

(http://ntfs.com/fat_systems.htm)
(http://www.c-jump.com/CIS24/Slides/FAT/FAT.html)

- FAT (File Allocation Table)
  - FAT12, FAT16, FAT32: evolutions based on bigger requirements

- One of the most simple file system
  - Very easy to implement

- List of clusters and their states (free/broken/used)
- Content in a linked list of clusters



Partition Boot Sector

FAT

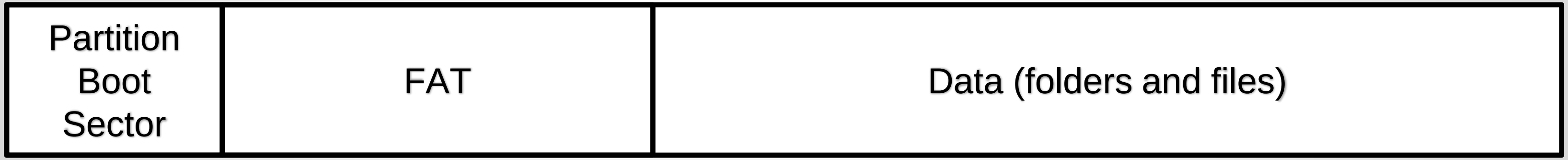
Data (folders and files)

File Systems: FAT example

- FAT (File Allocation Table)

- 1st part contains « Partition Boot Sector »
- 2nd part contains the file allocation table (a structure)

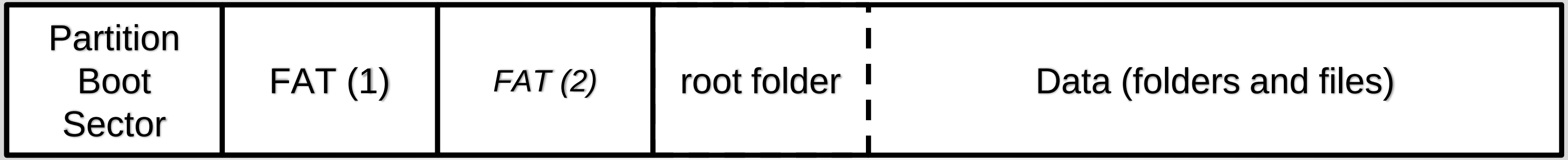
- Last part contains data (contents of files and folders)



File Systems: FAT example

- FAT (File Allocation Table)

- 1st part contains « Partition Boot Sector »
- 2nd part contains the file allocation table (a structure)
  - It is optionnally duplicated in a 3rd part, for corruption check
- Last part contains data (contents of files and folders)
  - It begins with the root folder with a predefined size



Partition Boot Sector

FAT (1)

FAT (2)

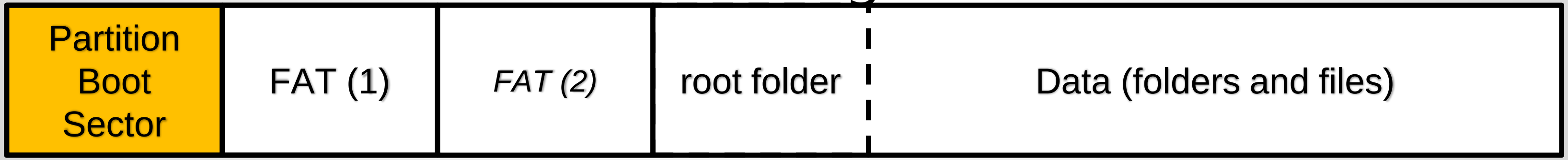
root folder

Data (folders and files)

File Systems: FAT example

- FAT: Partition Boot Sector

- Fixed size
  - Can directly be mapped in memory and read ;)
- Contains informations about the partition
  - Size of sectors, sectors per tracks, volume label, FAT version, ...
  - Size of the FAT (1) and FAT(2) *(number of sectors used)*
  - Size of the root folder *(number of entries inside)*
- Contains the code for booting



Partition Boot Sector

FAT (1)

FAT (2)

root folder

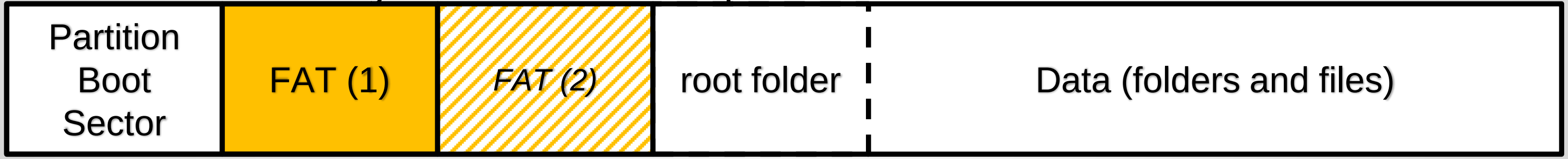
Data (folders and files)

File Systems: FAT example

- FAT: File Allocation Table

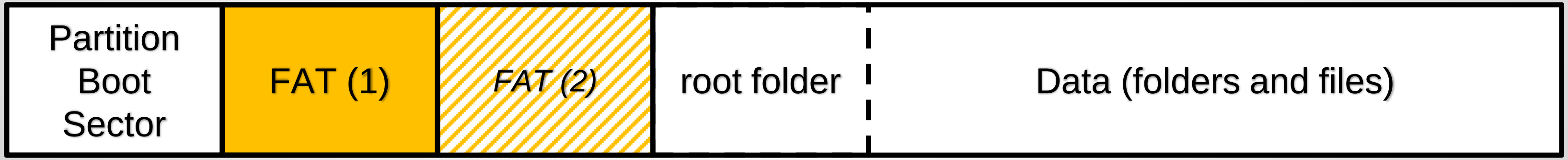
- Contains a list of all the clusters and their states
  - Size is fixed when the physical support was formatted
  - *Clusters 0 and 1 are reserved*
  - *First available cluster for data is the cluster 2*

- Linked list of files
  - A folder is just a file with a specific attribute



File Systems: FAT example

- FAT: File Allocation Table
- File: Linked list of clusters
  - A file is an entry in a « Directory Entry », that points to a 1st cluster
  - Each cluster has the address of the next one, or an « EOF » indicator
- Folder: Linked list of files
  - A folder is just a file with a specific attribute
  - And its content is a « Directory Entry » *(we'll see later what it is)*



Partition Boot Sector

FAT (1)

FAT (2)

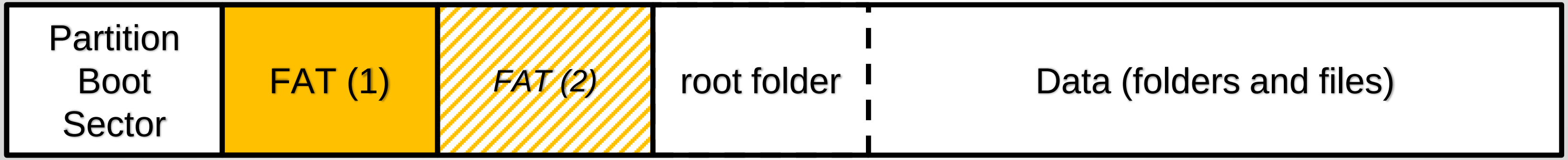
root folder

Data (folders and files)

File Systems: FAT example

- FAT: File Allocation Table

- Huge array containing state of each cluster in data
  - 0x0000 : Free
  - 0xFFF7 : Bad cluster *(it shouldn't be used by the file system)*
  - 0xFFF8 – 0xFFFF : Last cluster of a file *(« EOF »)*
  - (else) : Next cluster for this file



Partition Boot Sector

FAT (1)

FAT (2)

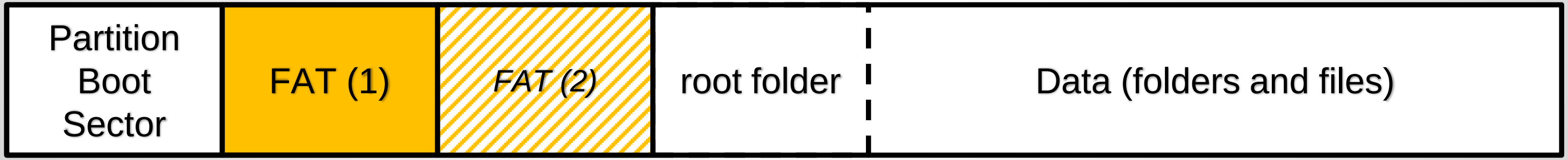
root folder

Data (folders and files)

File Systems: FAT example

- FAT: File Allocation Table

Cluster	Content
[2]	0
[3]	4
[4]	65,535
[5]	0
…	…



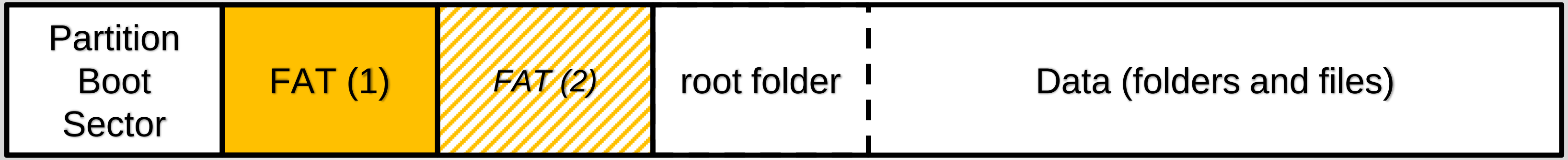
Partition Boot Sector | FAT (1) | *FAT (2)* | root folder | Data (folders and files)

File Systems: FAT example

- FAT: File Allocation Table

Cluster	Content
[2]	0
[3]	4
[4]	65,535
[5]	0
…	…

Cluster	Content
[2]	Free cluster
[3]	*data* [see 4]
[4]	*data* [end]
[5]	Free cluster
…	…



Partition Boot Sector

FAT (1)

FAT (2)

root folder

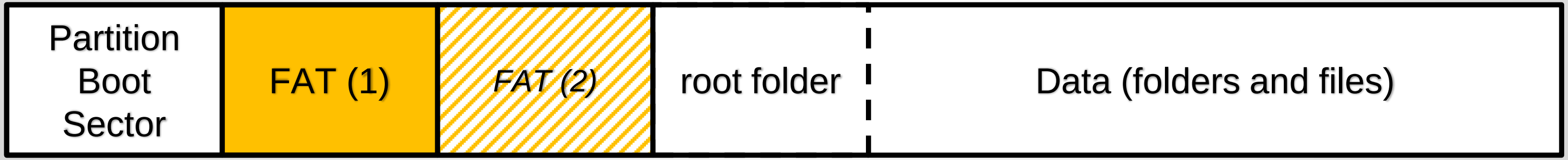
Data (folders and files)

File Systems: FAT example

- FAT: File Allocation Table

Cluster	Content
[2]	0
[3]	4
[4]	65,535
[5]	0
…	…

Cluster	Content
[2]	Free cluster
[3]	*data* [see 4]
[4]	*data* [end]
[5]	Free cluster
…	…



Partition Boot Sector

FAT (1)

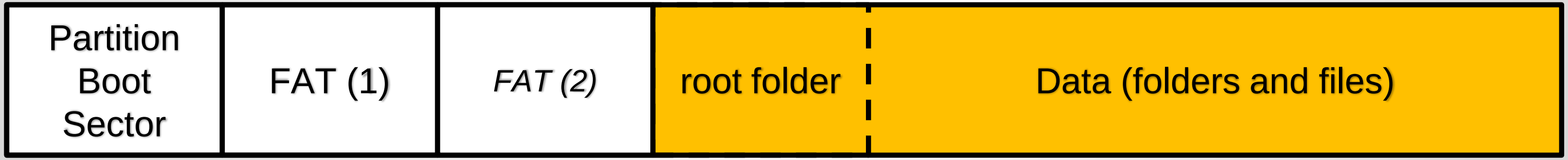
FAT (2)

root folder

Data (folders and files)

File Systems: FAT example

- FAT: Folders and files
- Contains clusters of files and folders
- Files: *pure data within the cluster*
- Folder: *directory entry structure*



Partition Boot Sector

FAT (1)

FAT (2)

root folder

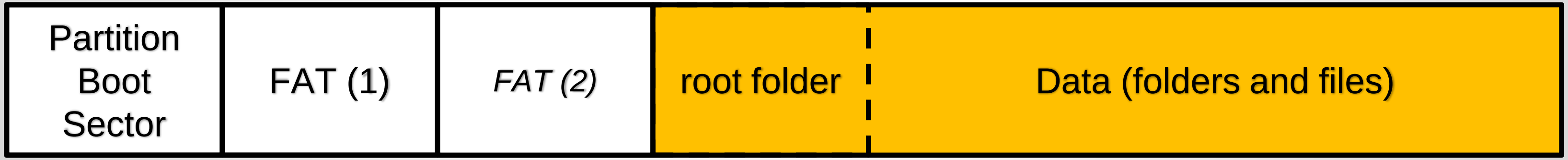
Data (folders and files)

File Systems: FAT example

- FAT: Folders and files

- Directory entry (32 Bytes): one per file or subfolder
  - Name (8 chars + 3 chars) => FILENAME.TXT
  - Attribute byte (8 bits) => archive, system, hidden, read-only
  - Create time (24 bits)
  - …
  - Starting cluster number in the FAT (16 bits)
  - File size (32 bits)

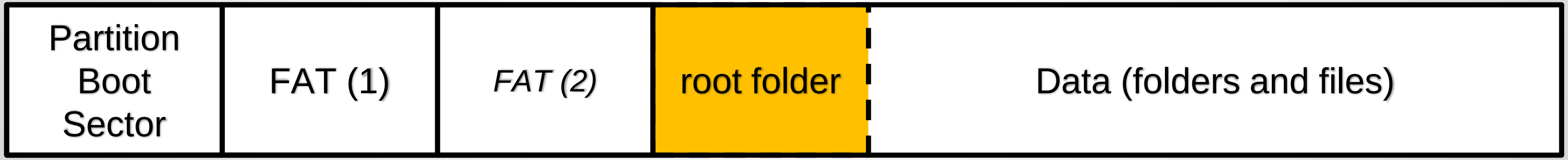
!!! OLD FAT ARE NOT CASE SENSITIVE !!!



Partition Boot Sector | FAT (1) | *FAT (2)* | root folder | Data (folders and files)

File Systems: FAT example

- FAT: Root folder
- Size fixed when formatting
  - 512 entries max for files and folders
- Contains files and folders entries



File Systems: FAT example

- FAT: Files

Root Directory Entry

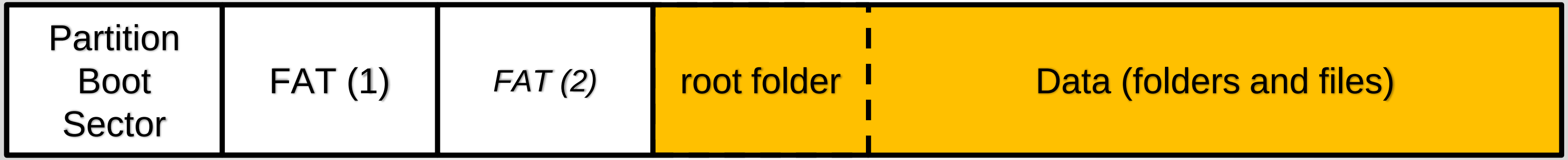
Direntry	Structure
[0]	Name: Mail.txt Attributes: Normal … 1st cluster: 3 File size: 7,550 B
…	…

FAT

Cluster	content
[2]	0
[3]	4
[4]	65,535
[5]	0
…	…

Data

Cluster	Content
2	??????????
3	Hello, this is...
4	…you soon.
5	??????????
…	…



Partition Boot Sector

FAT (1)

FAT (2)

root folder

Data (folders and files)

File Systems: FAT example

- FAT: Files

Root Directory Entry

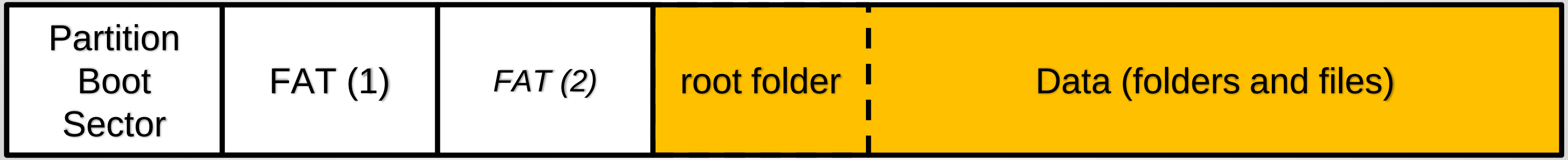
Direntry	Structure
[0]	**Name: Mail.txt** Attributes: Normal … **1st cluster: 3** **File size: 7,550 B**
…	…

FAT

Cluster	content
[2]	0
[3]	**4**
[4]	**65,535**
[5]	0
…	…

Data

Cluster	Content
2	??????????
3	Hello, this is...
4	…you soon.
5	??????????
…	…



Partition Boot Sector

FAT (1)

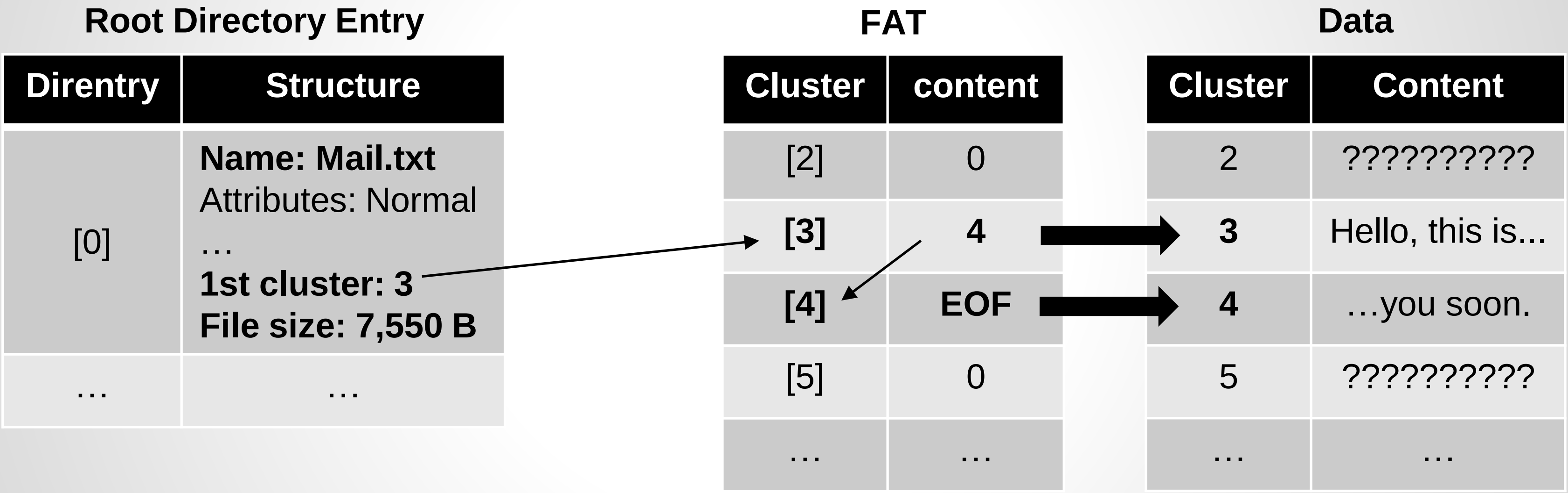
FAT (2)

root folder

Data (folders and files)

File Systems: FAT example

- FAT: Files



Root Directory Entry

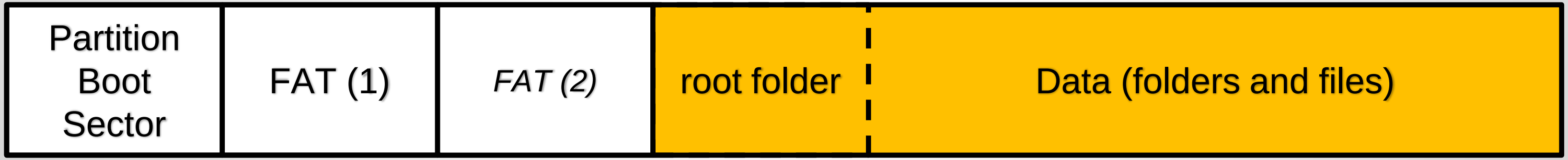
Direntry	Structure
[0]	**Name: Mail.txt** Attributes: Normal … **1st cluster: 3** **File size: 7,550 B**
…	…

FAT

Cluster	content
[2]	0
[3]	**4**
[4]	**EOF**
[5]	0
…	…

Data

Cluster	Content
2	??????????
3	Hello, this is...
4	…you soon.
5	??????????
…	…



Partition Boot Sector	FAT (1)	*FAT (2)*	root folder	Data (folders and files)

File Systems: FAT example

- FAT: Folders

Root Directory Entry

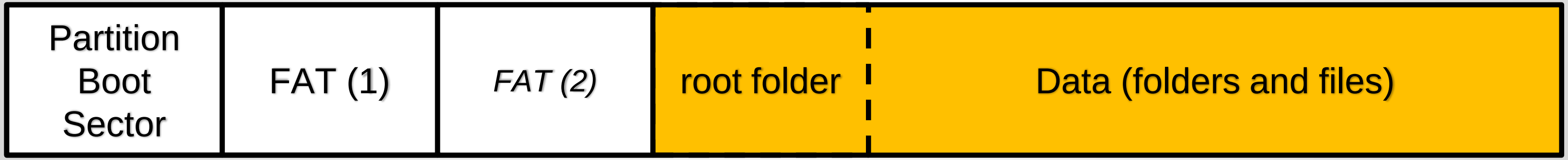
Direntry	Structure
[10]	Name: src Attributes: subfolder … 1st cluster: 56 …
…	…

FAT

Cluster	Content
[55]	4242
[56]	EOF
[57]	1337
[58]	0
…	…

Data

Cluster	Content
55	??????????
56	*[DIRENTRY]*
57	??????????
58	??????????
…	…



File Systems: FAT example

- FAT: Folders

Root Directory Entry

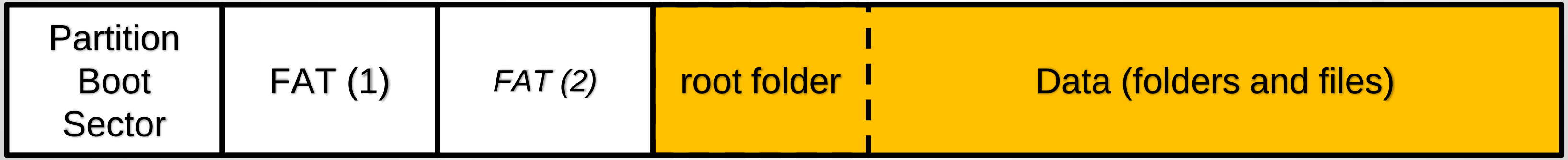
Direntry	Structure
[10]	**Name: src** **Attributes: subfolder** … **1st cluster: 56** …
…	…

FAT

Cluster	Content
[55]	4242
[56]	**EOF**
[57]	1337
[58]	0
…	…

Data

Cluster	Content
55	??????????
56	*[DIRENTRY]*
57	??????????
58	??????????
…	…



File Systems: FAT example

- FAT: Folders

Root Directory Entry

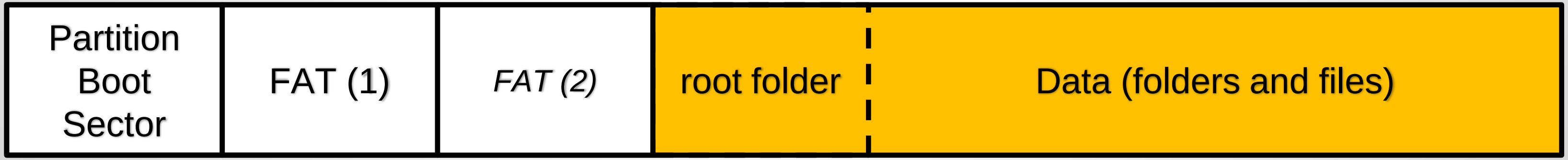
Direntry	Structure
[10]	**Name: src** **Attributes: subfolder** … **1st cluster: 56** …
…	…

FAT

Cluster	Content
[55]	4242
[56]	**EOF**
[57]	1337
[58]	0
…	…

Data

Cluster	Content
55	??????????
56	*[DIRENTRY]*
57	??????????
58	??????????
…	…



Partition Boot Sector

FAT (1)

FAT (2)

root folder

Data (folders and files)

File Systems: FAT example

- FAT: Create a file (or a folder)

1. Use the directory entry of the folder where to create the file
2. Add an entry within the parent directory entry
   - Eventually, add one more cluster to support the directory entry
3. Search for the first available cluster in the FAT array
4. Use it as the 1st cluster, and write data inside
   - Eventually: add more clusters, update directory entry, update FAT for last cluster

File Systems: FAT example

- FAT: Delete a file (or a folder)

1. Use the directory entry of the folder where to delete the file
2. Clear the FAT clusters list until the end
3. Clear the entry within the directory entry
   - Well, precisely, a 0xE5 (‘_’) is put as the first character in the name

File Systems: FAT example

- FAT: Reading a file

1. Use the directory entry of the folder that contains the file in order to obtain the number of the first cluster

2. Get the cluster state in the FAT part
3. Get the cluster content within the Data part
4. Continue from to the next cluster within the FAT
   - Loop to step 2 until you reach an EOF as next cluster

File Systems: FAT example

- Maximal size of 1 file
- Maximal number of files within 1 folder
- Maximal number of clusters within 1 partition
  - « Sectors per Clusters » and « Cluster size » are linked
- Maximal number of files within the root folder
- Maximal size of the partition

- Because of the file table allocation (2nd part), the cluster size has an impact on all of the previous values
  - *And in some cases, it can lose some clusters and fragments space*

ext2 example

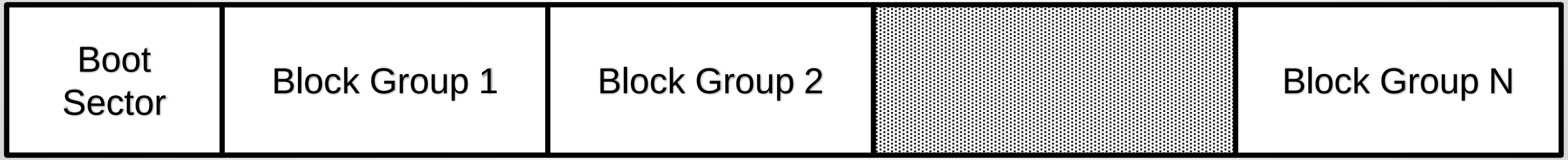
File Systems: ext2 example

(http://web.mit.edu/tytso/www/linux/ext2intro.html)

- ext2 / ext2fs (Second Extended File System)

- Fewer fragmentation than FAT
  - But still a bit fragmented

- More attributes managed than in FAT
  - Manages UNIX attributes and ACL

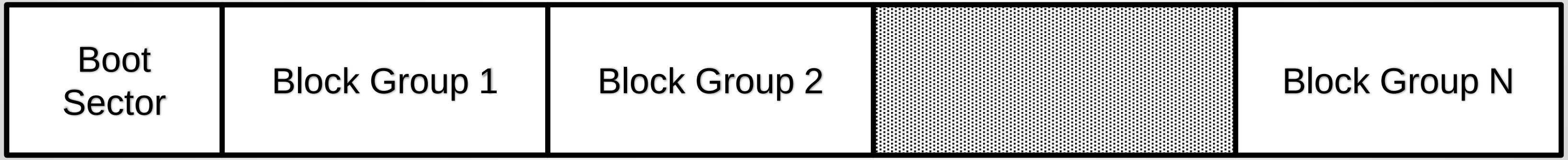


Boot Sector | Block Group 1 | Block Group 2 | | Block Group N

File Systems: ext2 example

(http://web.mit.edu/tytso/www/linux/ext2intro.html)

- ext2 / ext2fs (Second Extended File System)
- 1st part contains « Boot Sector »
- Other parts contain « Block Group »
  - Which contain a lot of structures and data



Boot Sector

Block Group 1

Block Group 2

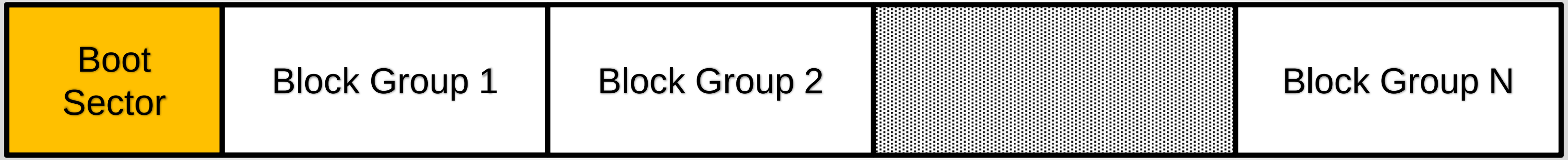
Block Group N

File Systems: ext2 example

- Boot Sector

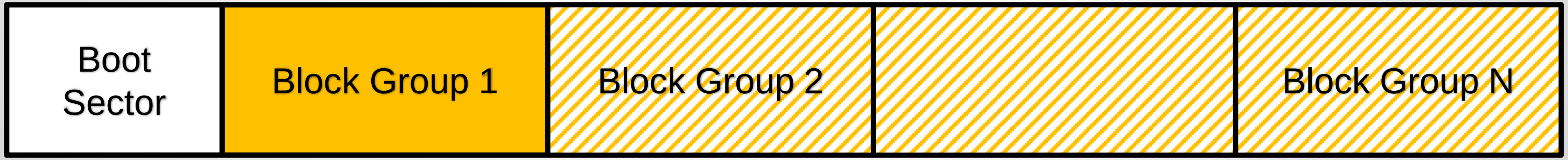
- First 1024kB of the partition

- Unused by ext2/Reserved for booting purpose
  - Code for loading the kernel in memory when the machine is started



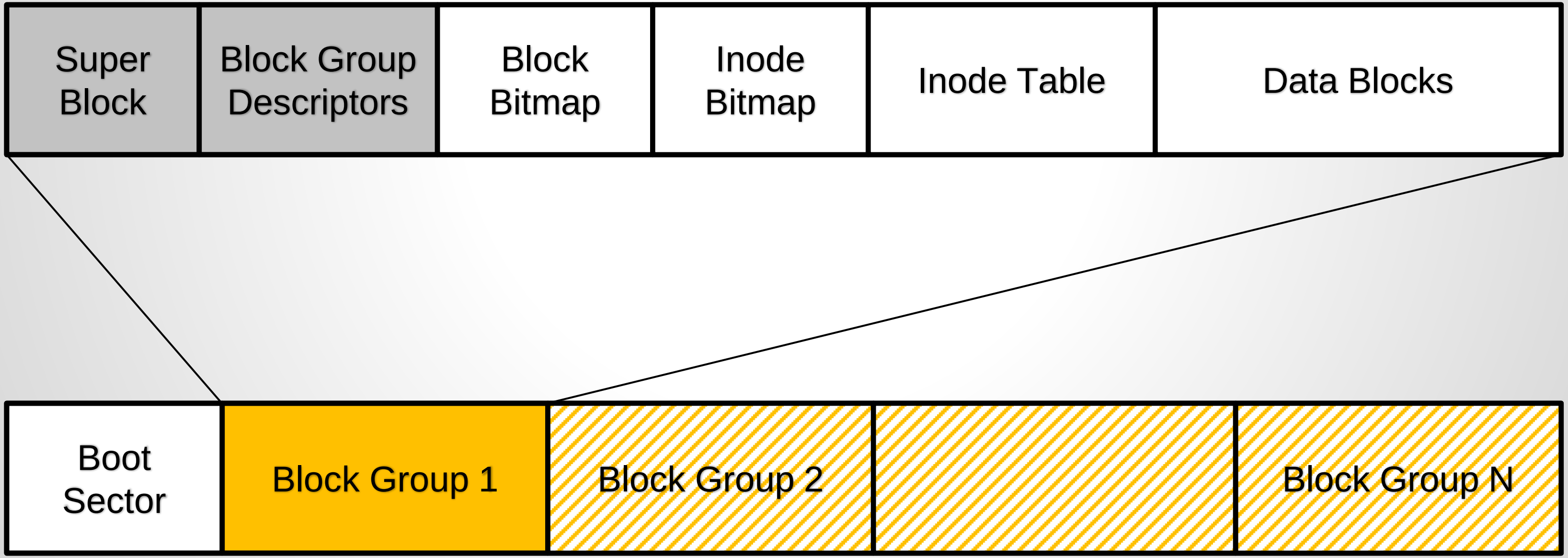
File Systems: ext2 example

- Block Group
- Contains a structure
- Meta-data AND data within each of them
    - Some meta-data are replicated in each Block Group



File Systems: ext2 example

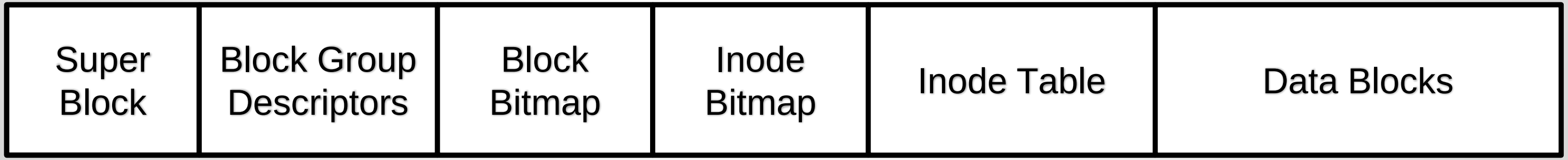
- Block Group



File Systems: ext2 example

- Block Group

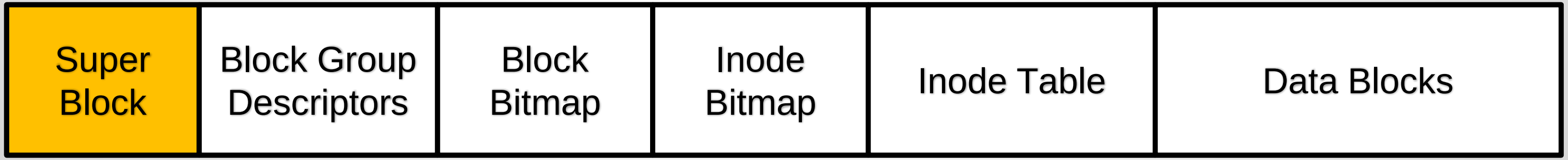
- Super Block: *describes the partition*
- Block Group Descriptors: *content of each Block Group*
- Block Bitmap: *describes state of data blocks*
- i-node Bitmap: *describes state of i-nodes*
- i-node Table: *describes each file, directory, …*



Super Block	Block Group Descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

File Systems: ext2 example

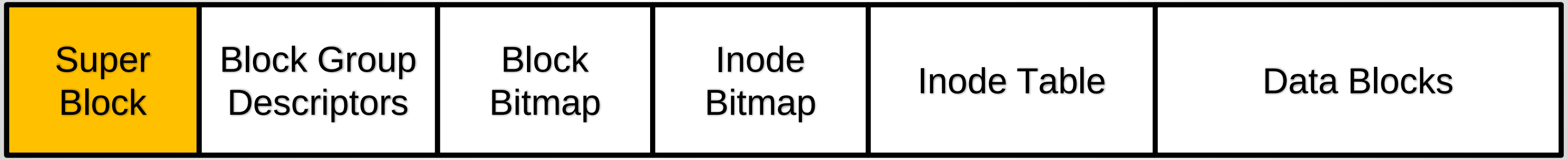
- Super Block
- *Replicated in some (or all) Block Groups*
- Contains essential informations about the partition



Super Block | Block Group Descriptors | Block Bitmap | Inode Bitmap | Inode Table | Data Blocks

File Systems: ext2 example

- Super Block
  - Total number of blocks and i-nodes
  - Block size
  - Free blocks, free i-nodes
  - ID of the first Block Group
  - Blocks per Block Group, i-nodes per BG, Block Bitmap per BG *(allows to find the position on the disk of each Block Group)*
  - ...



Super Block | Block Group Descriptors | Block Bitmap | Inode Bitmap | Inode Table | Data Blocks

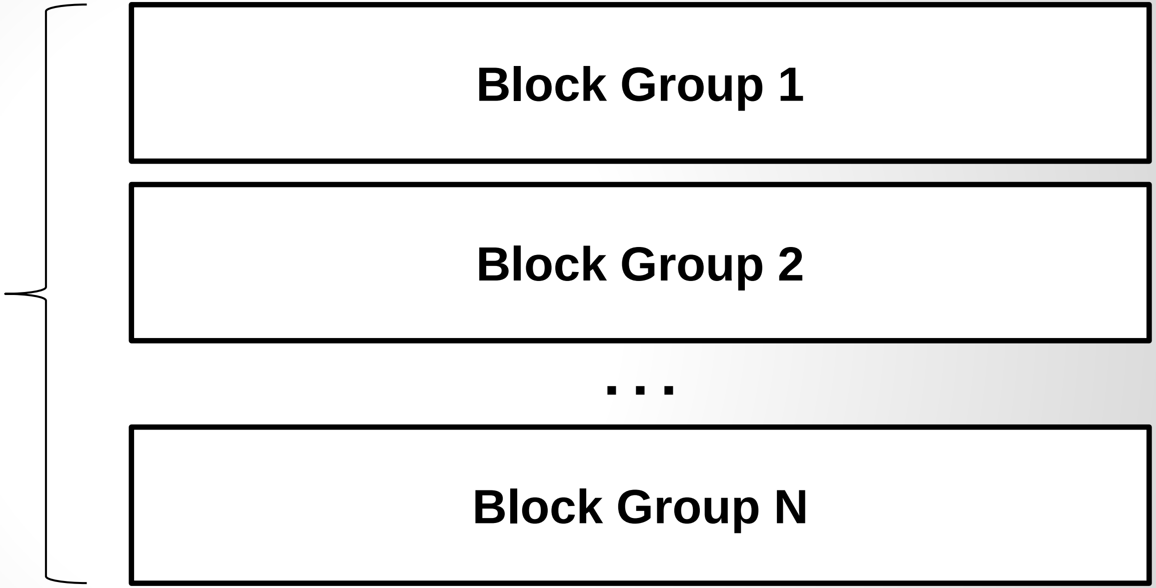
File Systems: ext2 example

- Super Block

WWW blocks in the partition
XXX i-nodes in the partition

YYY free blocks in total
ZZZ free i-nodes in total

…

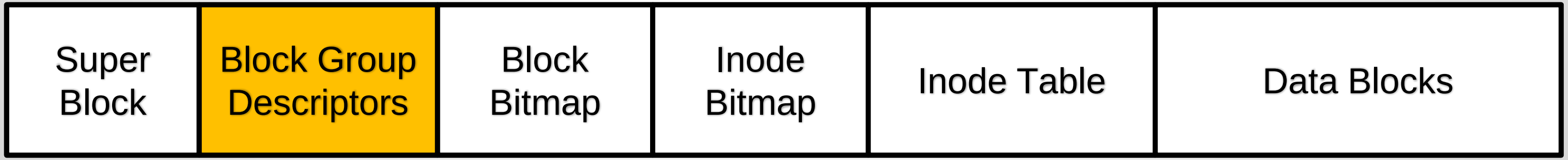


Super Block	Block Group Descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

File Systems: ext2 example

- Block Group Descriptors

- *Replicated in some (or all) Block Groups*
- Array describing « each » Block Group of the partition
  - Address (in block) of Block Bitmap, i-node Bitmap, i-node Table
  - Free blocks & i-nodes count
  - Directories count
  - …

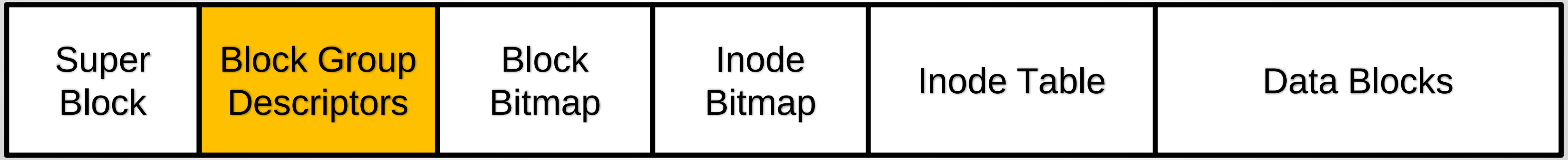


Super Block	Block Group Descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

File Systems: ext2 example

- Block Group Descriptors

- Super Block has a fixed size
  - 1st Block Group Descriptor is easy to find
- Number max of Block Group can be calculated:
  - [Super Block] Block size && Blocks per Block Group
  - (Same with total i-nodes, and i-nodes per Block Group)
- Next Block Group can be found by adding structures size



Super Block

Block Group Descriptors

Block Bitmap

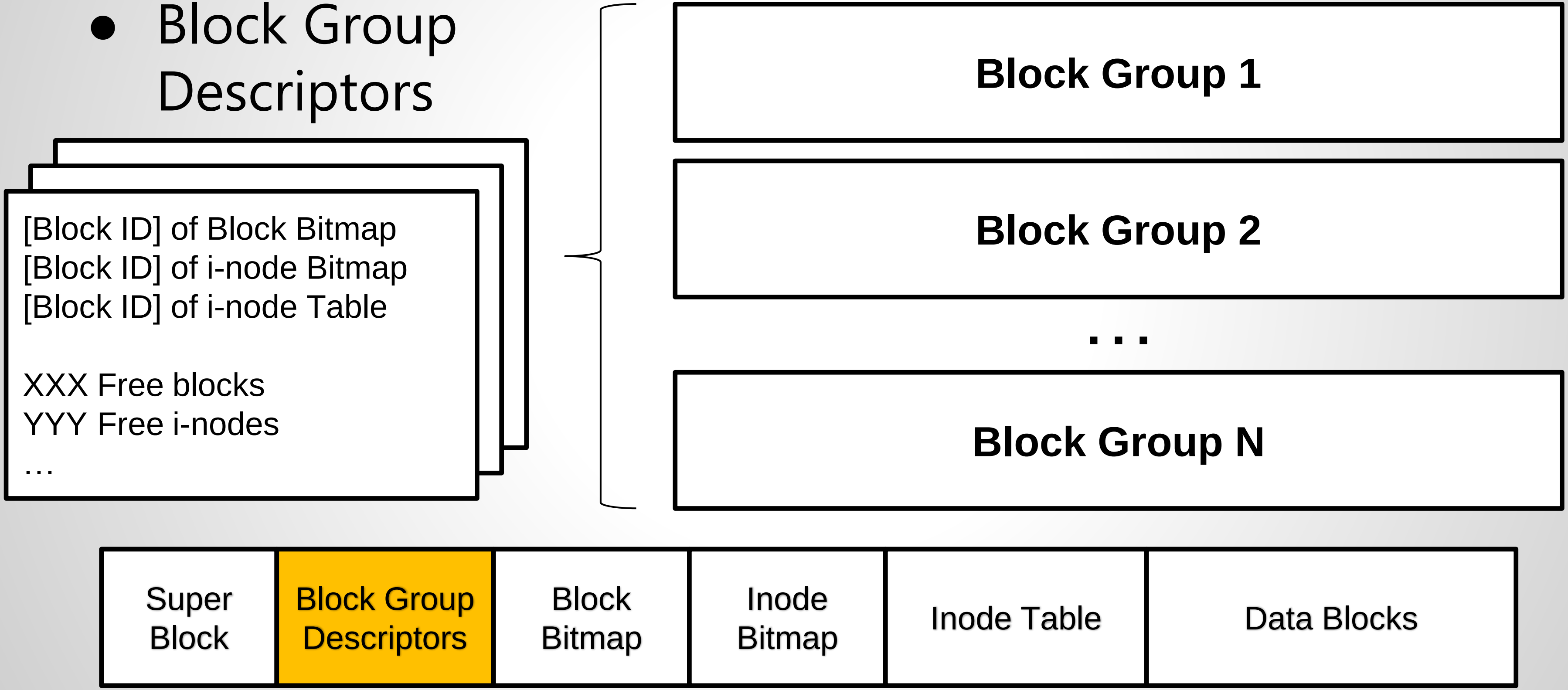
Inode Bitmap

Inode Table

Data Blocks

File Systems: ext2 example

- Block Group Descriptors



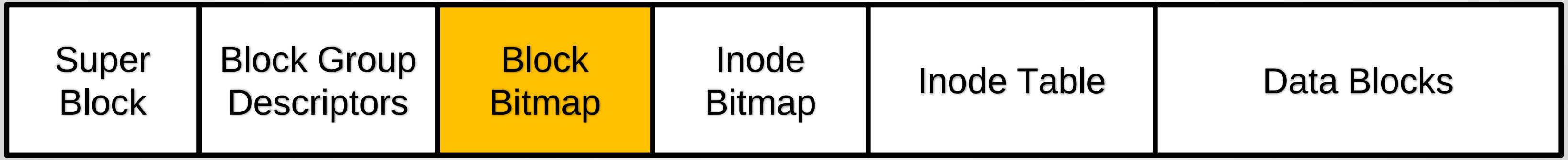
[Block ID] of Block Bitmap
[Block ID] of i-node Bitmap
[Block ID] of i-node Table

XXX Free blocks
YYY Free i-nodes
…

File Systems: ext2 example

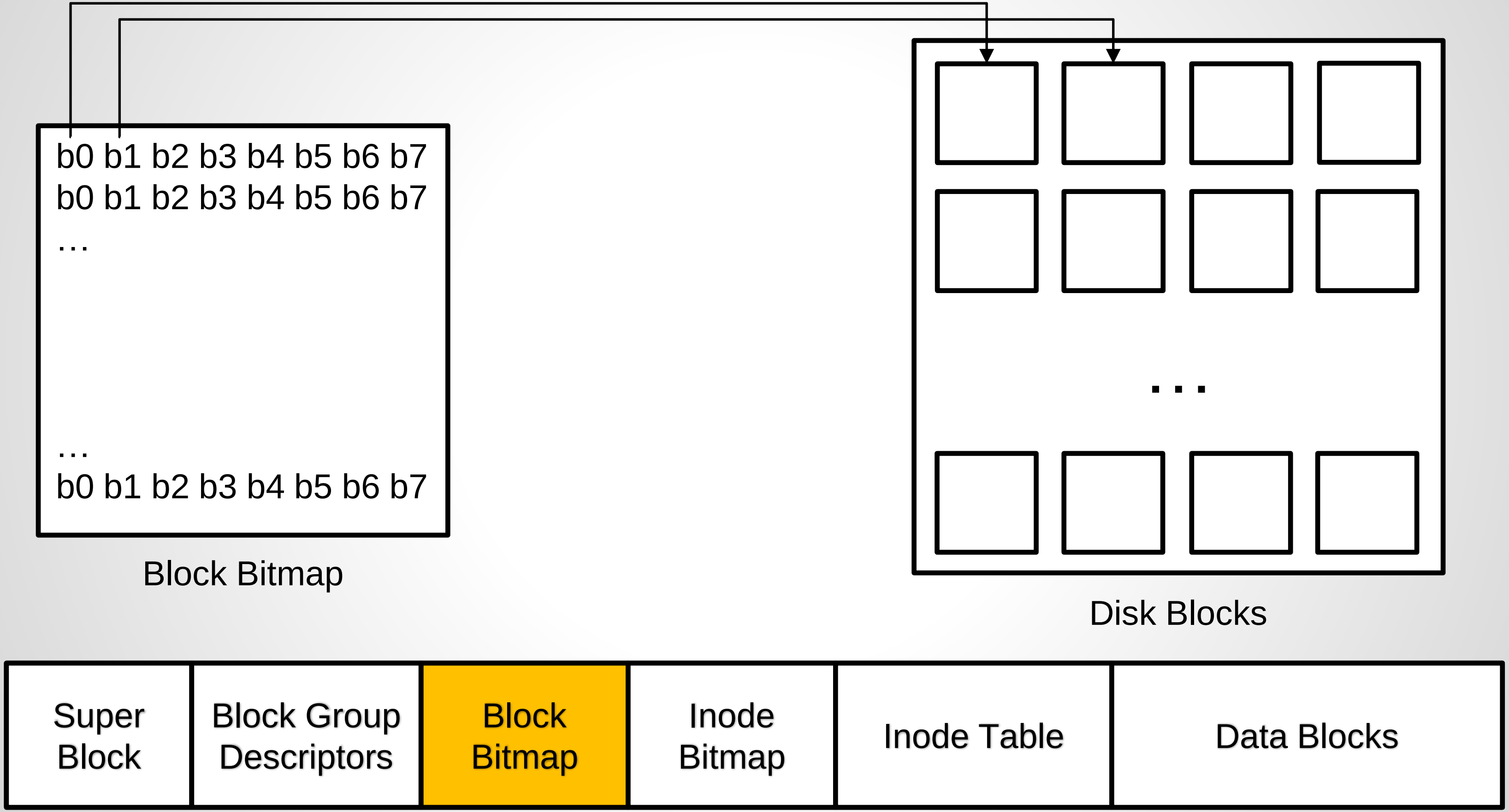
- Block Bitmap

- Describes the state of each data block of the current block group with a bit
  - 1 = « used »
  - 2 = « free »
- Size of bitmap defines the number of Data Blocks
  - *Align the Block Bitmap on block size (1024B = 8 * 1024 data blocks)*



Super Block	Block Group Descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

File Systems: ext2 example



b0 b1 b2 b3 b4 b5 b6 b7
b0 b1 b2 b3 b4 b5 b6 b7
…

…
b0 b1 b2 b3 b4 b5 b6 b7

Block Bitmap

…

Disk Blocks

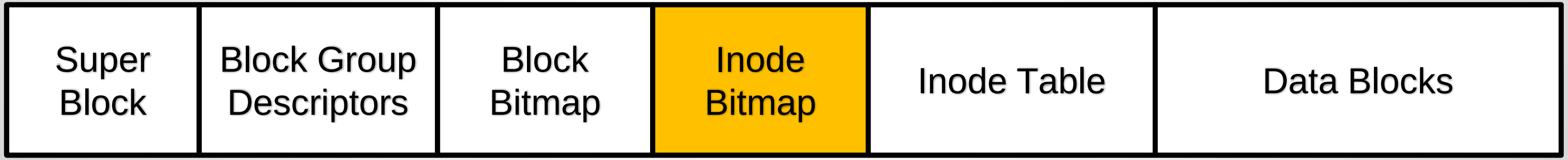
Super Block	Block Group Descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

File Systems: ext2 example

- i-node Bitmap

- Describes the state of each i-node of the i-node table of the current block group with a bit
  - 1 = « used »
  - 2 = « free »

- When the i-node table is created, all i-nodes are marked as « used »



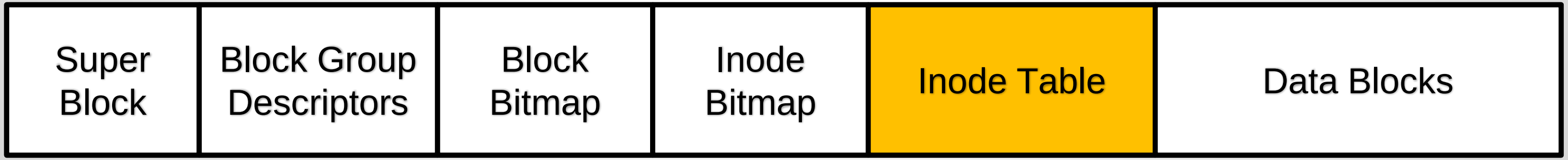
Super Block | Block Group Descriptors | Block Bitmap | Inode Bitmap | Inode Table | Data Blocks

File Systems: ext2 example

- i-node Table: *array of i-nodes*

- Each i-node describes either:
  - a directory
  - a regular file
  - a symbolic link
  - a special file (multiples types)

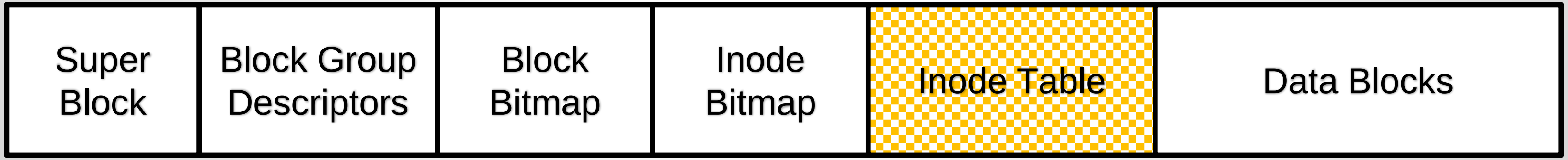
- No filename are stored within an i-node



Super Block | Block Group Descriptors | Block Bitmap | Inode Bitmap | Inode Table | Data Blocks

File Systems: ext2 example

- i-node structure: General case
    - Type of file (Directory, file, symbolic link, special file, …)
    - Access rights
    - Owner UID
    - Group GID
    - Creation, modification, deletion, last access time
    - Links count *(how many links point to this i-node)*
    - Blocks count
    - Blocks *[array – see next slide]*
    - …



Super Block

Block Group Descriptors

Block Bitmap

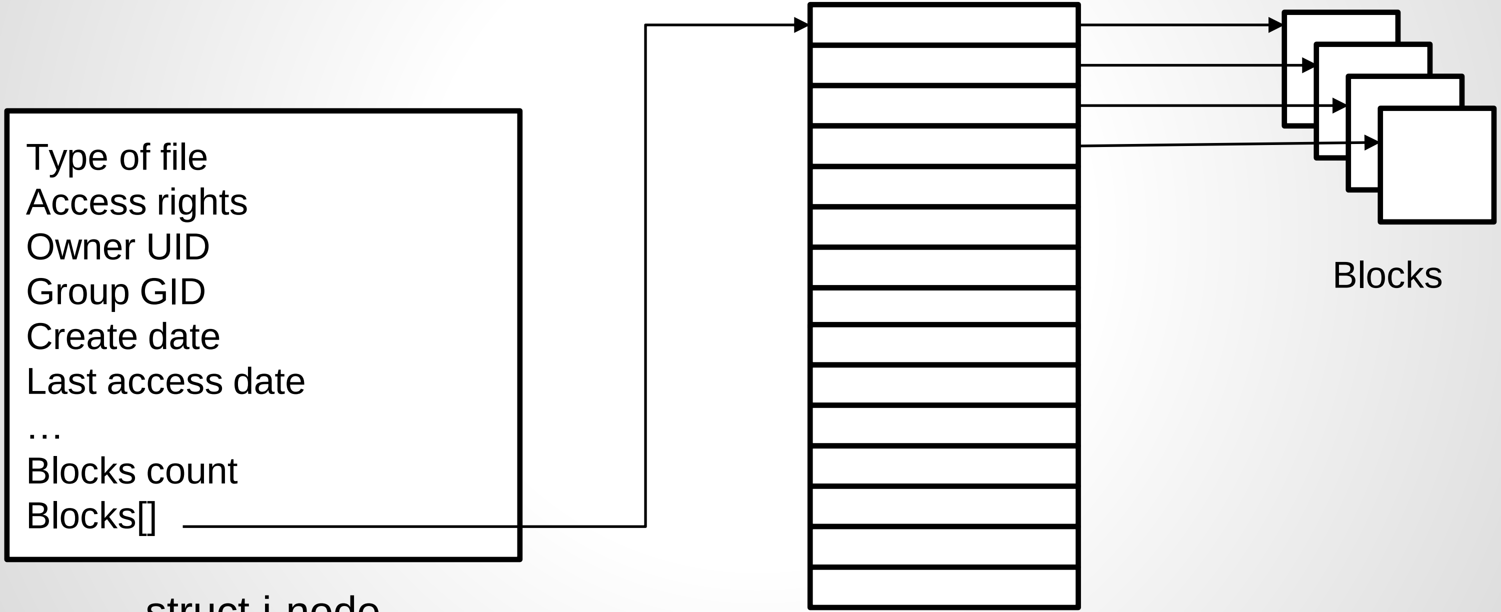
Inode Bitmap

Inode Table

Data Blocks

File Systems: ext2 example

i-node structure: General case



Type of file
Access rights
Owner UID
Group GID
Create date
Last access date
…
Blocks count
Blocks[]

struct i-node

i-node -> Blocks[]

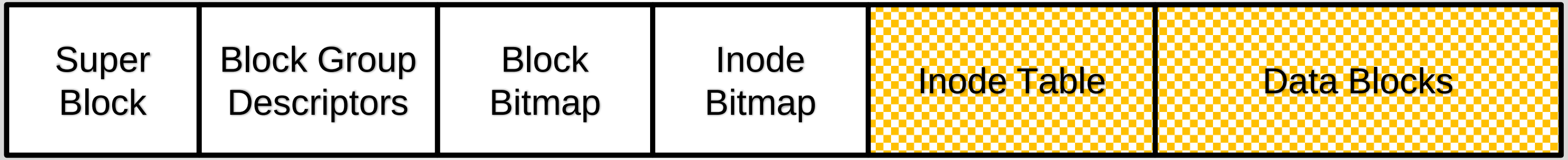
Blocks

File Systems: ext2 example

- i-node structure: Directory case

- If the i-node type is a directory, the Blocks[] is an array of a specific structure: *dir_entry*
  - i-node number of the targeted object (file, or directory, or symlink, ...)
  - Name and name length
  - Type (file, or directory, or symlink, ...)
  - ...

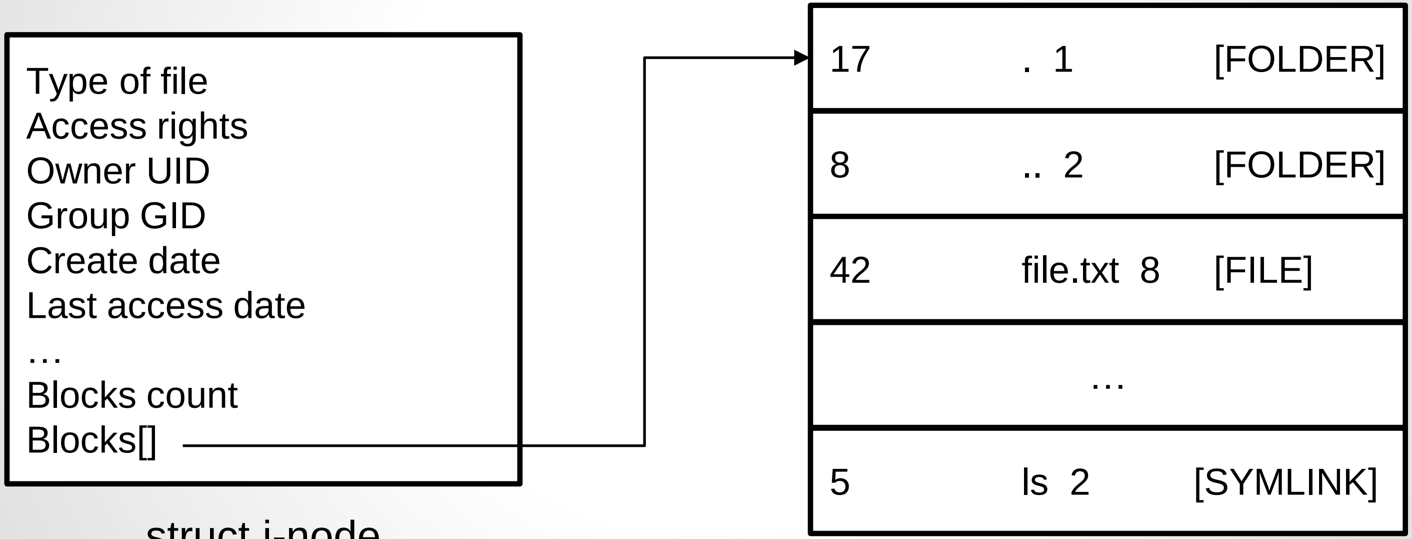
File names are stored in the directory entry!



Super Block | Block Group Descriptors | Block Bitmap | Inode Bitmap | Inode Table | Data Blocks

File Systems: ext2 example

i-node structure: Directory case



Type of file
Access rights
Owner UID
Group GID
Create date
Last access date
…
Blocks count
Blocks[]

struct i-node

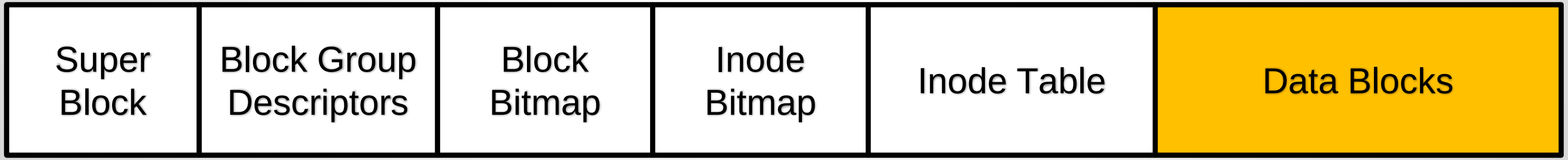
17	. 1	[FOLDER]
8	.. 2	[FOLDER]
42	file.txt 8	[FILE]
	…	
5	ls 2	[SYMLINK]

i-node -> Blocks[]
(struct dir_entry)

File Systems: ext2 example

- Data Blocks

- Content of the file OR of the directory entry



Super Block	Block Group Descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

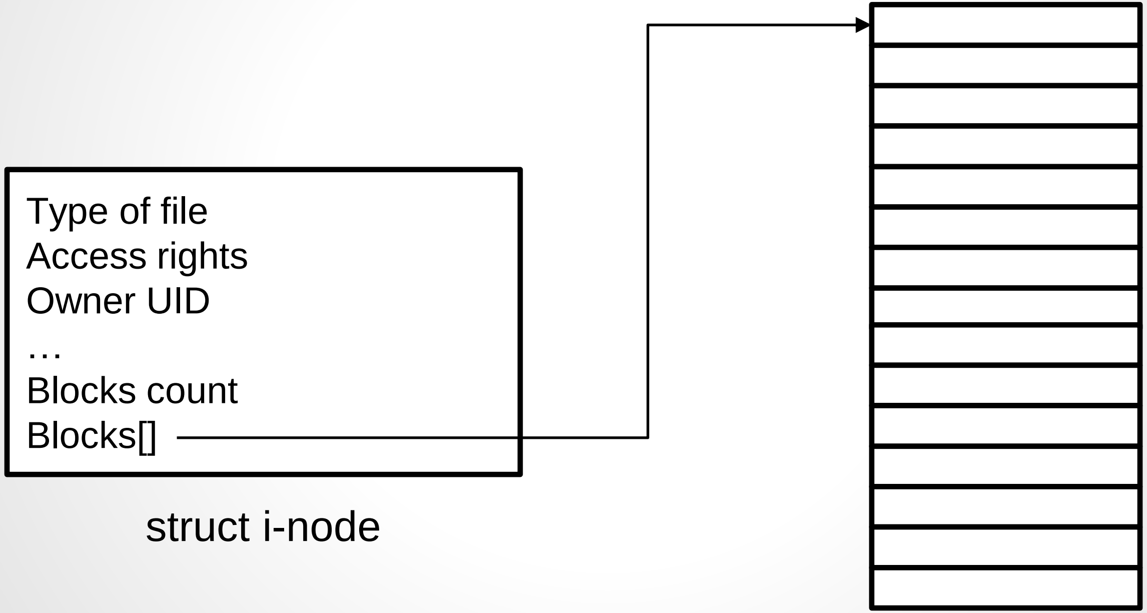
File Systems: ext2 example

How is stored a file in an ext2 file system?

Type of file
Access rights
Owner UID
Group GID
Create date
Last access date
…
Blocks count
Blocks[]

struct i-node

File Systems: ext2 example



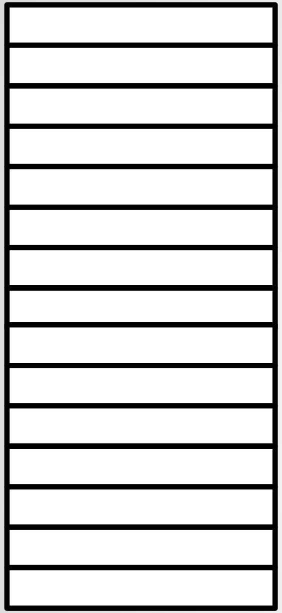
Type of file
Access rights
Owner UID
…
Blocks count
Blocks[]

struct i-node

i-node -> blocks

File Systems: ext2 example

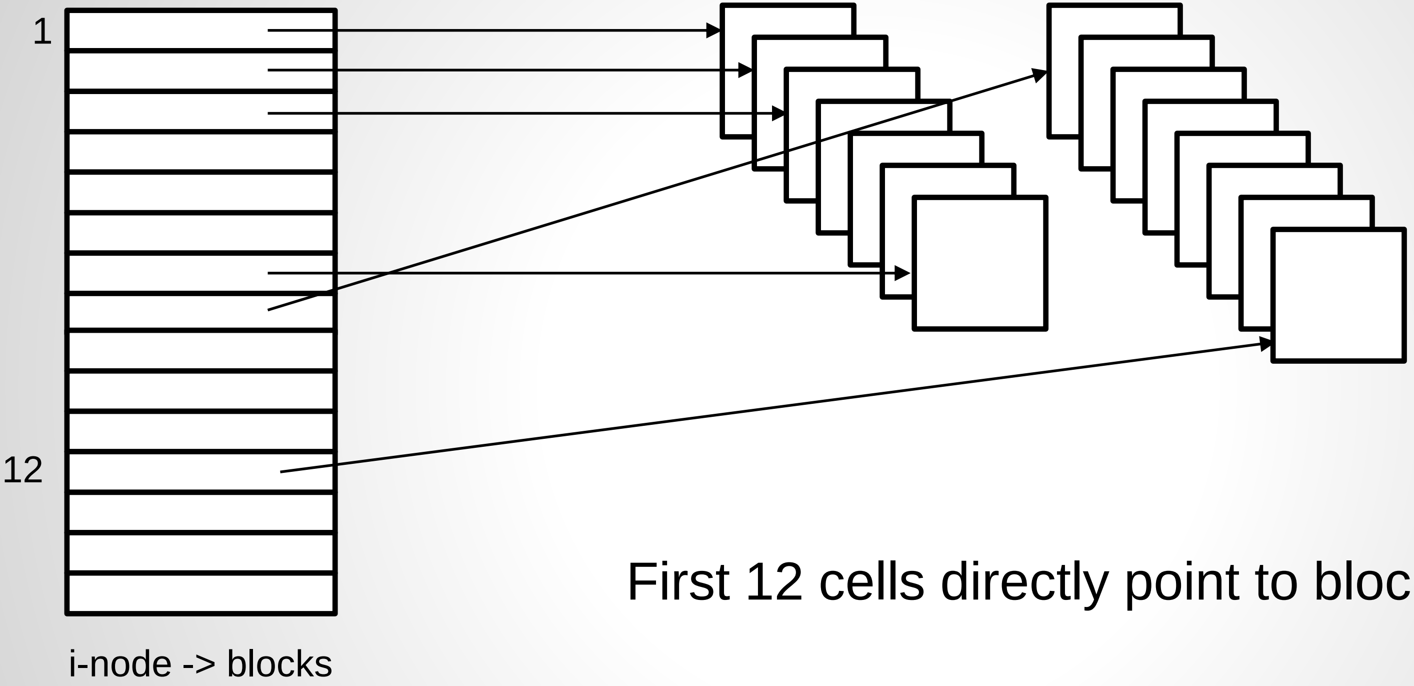
Multi-level indexing



i-node -> blocks

File Systems: ext2 example

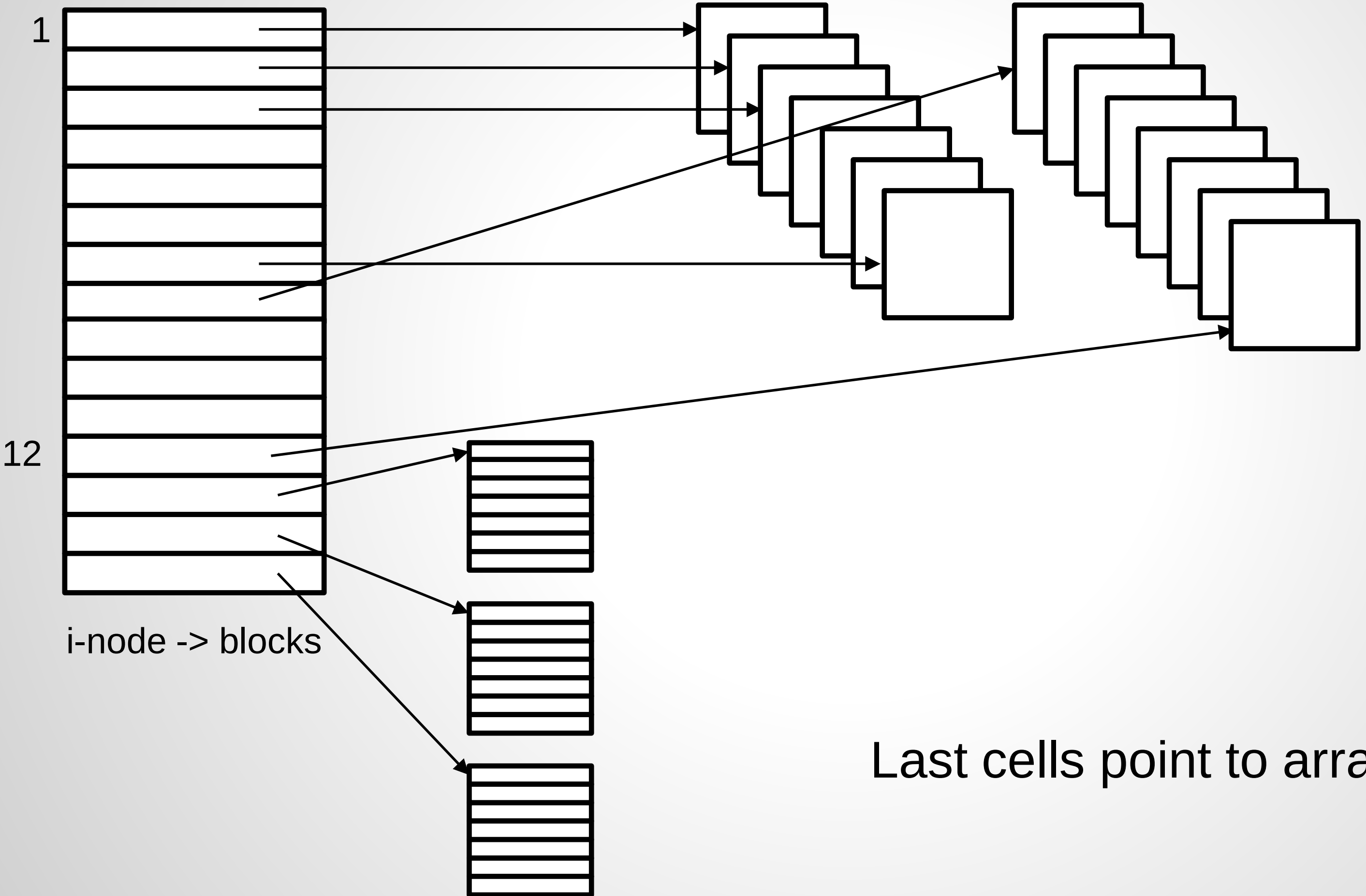
Multi-level indexing



First 12 cells directly point to blocks

File Systems: ext2 example

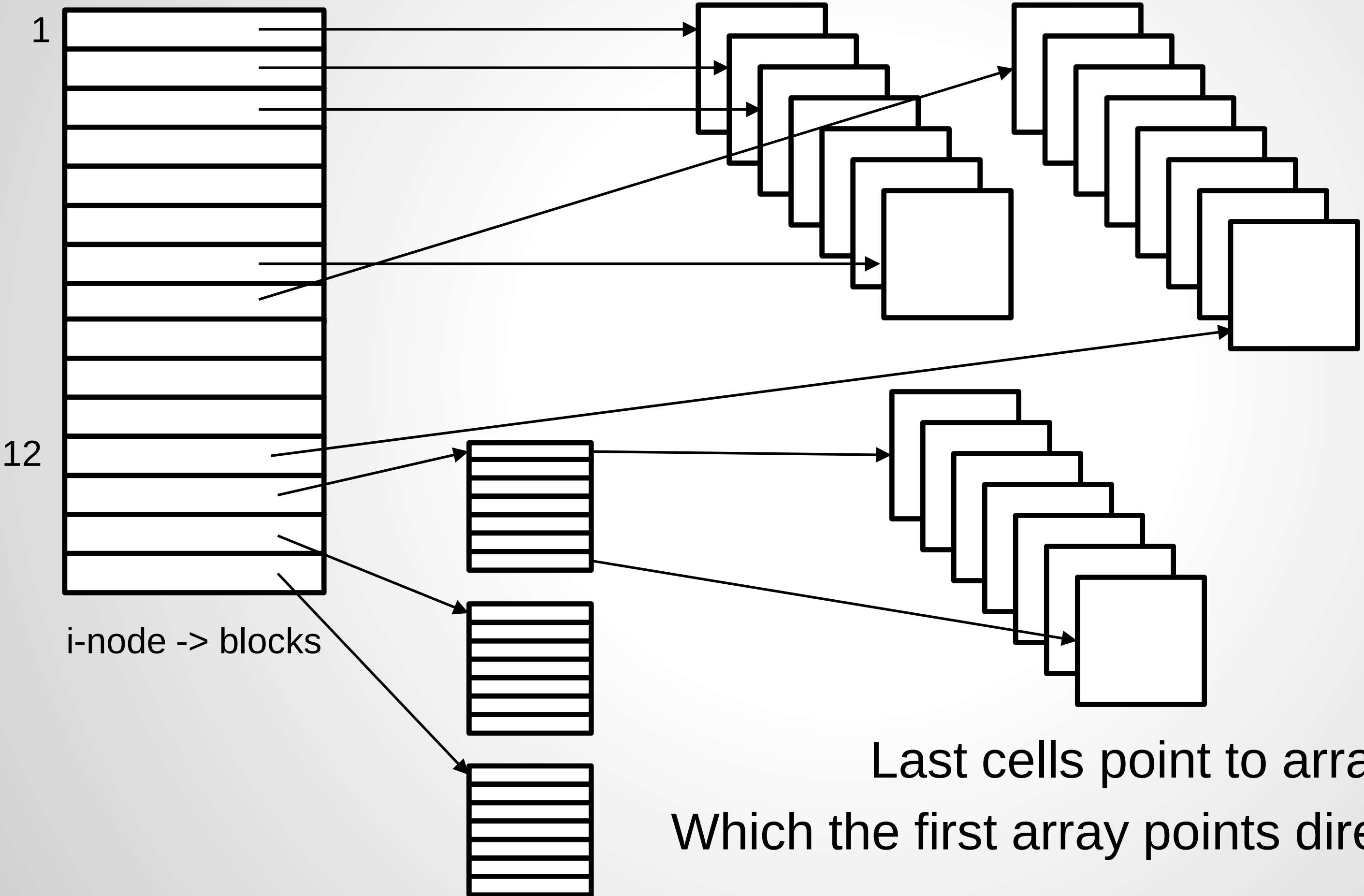
Multi-level indexing



Last cells point to arrays…

File Systems: ext2 example

Multi-level indexing

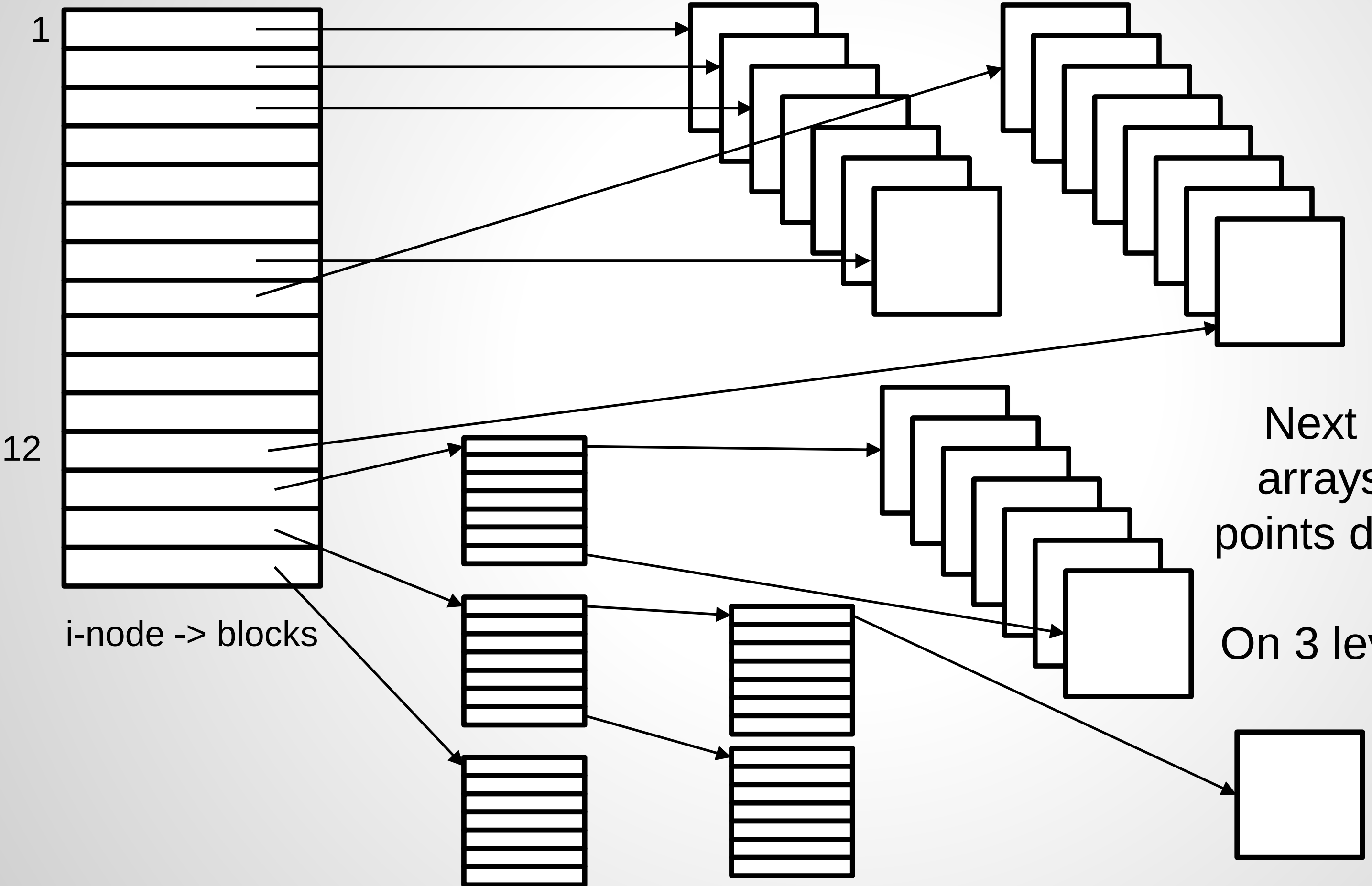


Last cells point to arrays…

Which the first array points directly to blocks

File Systems: ext2 example

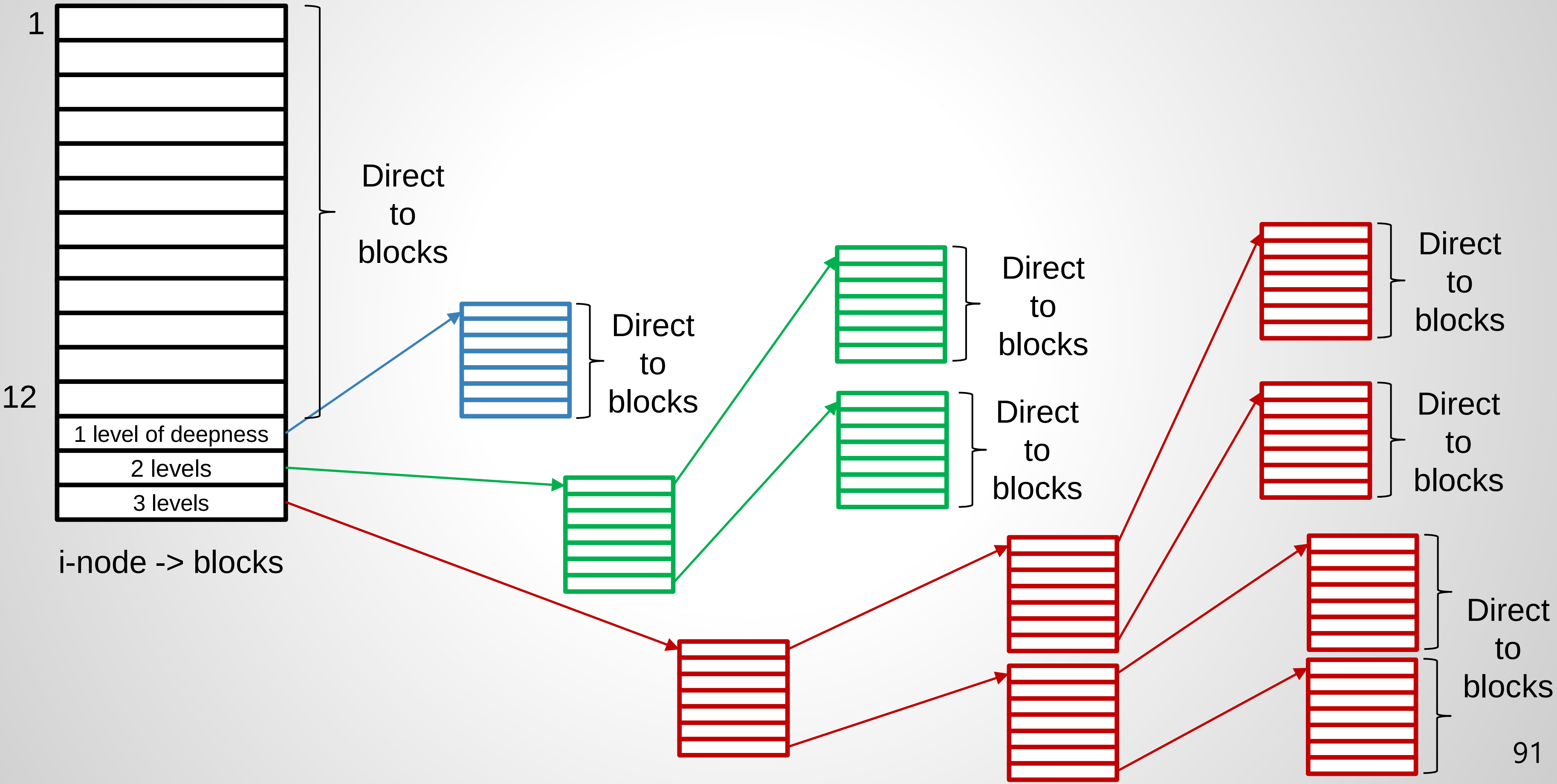
Multi-level indexing



Next arrays points to arrays, which the first points directly to blocks…
…etc…
On 3 levels of depth max

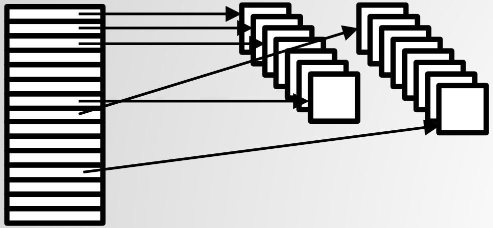
File Systems: ext2 example

Multi-level indexing



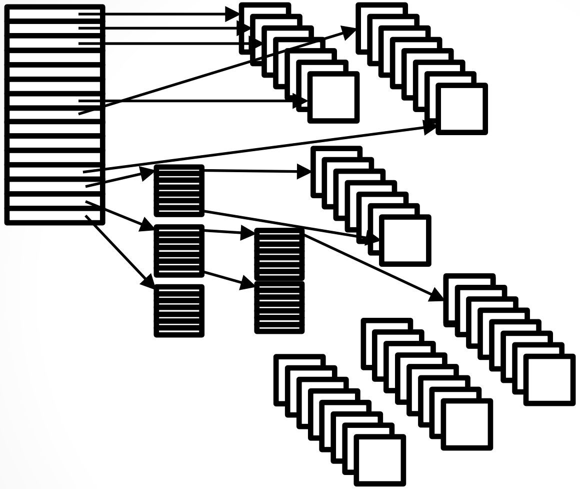
File Systems: ext2 example

Multi-level indexing



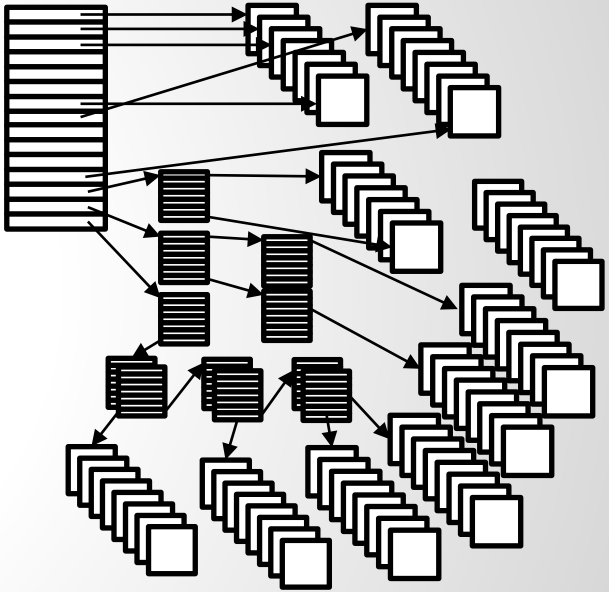
One small file (B~kB):

- 1 array of blocks
- Few blocks (<12)



One medium file (kB~MB):

- Some sub-arrays of blocks
- Some blocks



One large file (MB~GB):

- Lot of sub-arrays of blocks
- Lot of blocks

File Systems: ext2 example

How to find a precise i-node?

- Each i-node in the i-node table is counted
  - No need to « tag » with a number:
    The 1st i-node in the 1st block group is the i-node 1
- As block groups, i-node tables, and block tables are fixed at format time, we know where are each i-node is
  - Just make a calculation

File Systems: ext2 example

How to find a precise i-node?

1. Block Group = (i-node - 1) / i-nodes per group

2. Local i-node index = (i-node - 1) % i-nodes per group

(i-nodes per group can be found in the Super Block)

File Systems: ext2 example

How to find the content of a file?

1. Read the directory entry where the file is, find the entry with the same filename, and get its i-node number

2. Calculate in which Block Group the i-node is, and get the Blocks[]

File Systems: ext2 example

How to find a file from its pathname? « /usr/bin/ls »

1. Read the 1st Block Group and the 1st i-node (it is « / »)
2. Get the Blocks[], read the direntry, and search for the next folder or file
3. Find the i-node pointed by the direntry, and read its Blocks[]
4. Repeat from step 2 until you find your file and its content

« usr » in the 1st loop, « bin » in the 2nd loop, and « ls » in the 3rd loop

Symbolic links

- A symbolic link is an i-node with 1 data block containing the pathname to resolve
  - And it is also indicated in a directory entry as a SYMLINK

- Type of object : « Symbolic Link »
  - The kernel knows it must resolve it, because it is of type « symlink »

- Removing a symlink implies to remove its Directory Entry + the i-node + the data block

Hard links

- Hard links are just entries within Directory Entry

- As the i-node does not have any name…
  …only the counter of references to the i-node is used!

- When the counter reaches 0
  [the i-node is unused/absent of the FS]
  - ○ If the i-node is not used by any file descriptor, it is freed

Folders cannot be hardlinked, because it would break the hierarchy (always symlinks in their case)

Hard links

- Removing a hard link, is just removing the Directory entry and reducing the links counter on the i-node

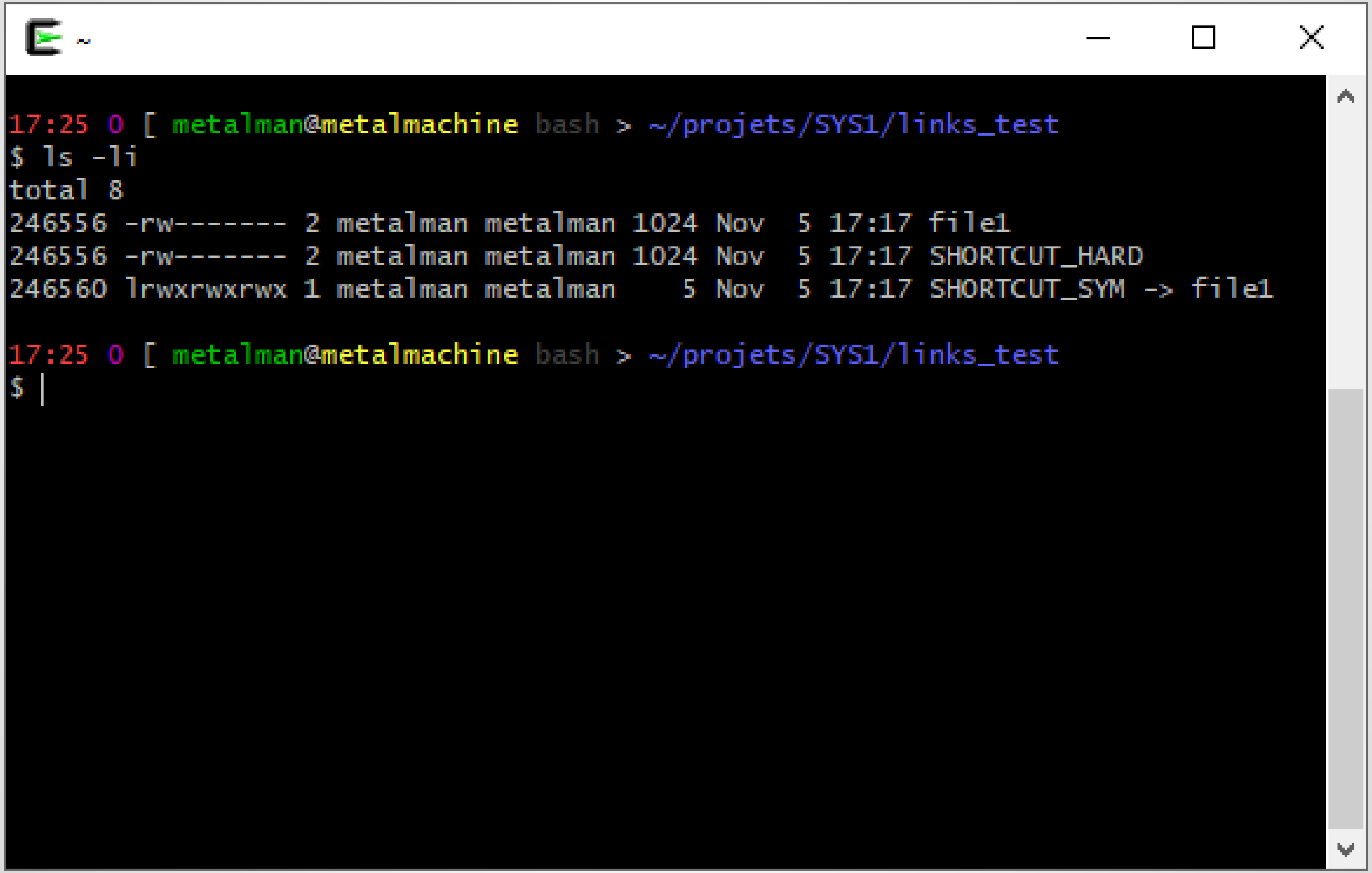
Hard links can be destroyed in your shell without risk to lose data
(as long as there is at least 1 hard link after the deletion…)

```
# Create an empty file
touch file1
# Fill it with 1024 bytes
head -c 1024 < /dev/urandom > file1

# Create a hard link named "SHORTCUT_HARD" pointing to "file1"
ln file1 SHORTCUT_HARD
# Create a symbolic link named "SHORTCUT_SYM" pointing to "file1"
ln –s file1 SHORTCUT_SYM

# Print the i-node behind each directory entry
ls –li
```

Links



```
17:25 0 [ metalman@metalmachine bash > ~/projets/SYS1/links_test
$ ls -li
total 8
246556 -rw------- 2 metalman metalman 1024 Nov  5 17:17 file1
246556 -rw------- 2 metalman metalman 1024 Nov  5 17:17 SHORTCUT_HARD
246560 lrwxrwxrwx 1 metalman metalman    5 Nov  5 17:17 SHORTCUT_SYM -> file1

17:25 0 [ metalman@metalmachine bash > ~/projets/SYS1/links_test
$ |
```

Links & FS in UNIX

How to remove a file within the UNIX world of i-nodes ?

Links & FS in UNIX

- A file is an i-node and a directory entry
- An i-node has a counter of links
- Just remove the link... unlink(2)

(How to create a file?... Create a link, and a directory entry)

Your kernel runs in memory, and processes manipulate data

The physical support is the mean to achieve persistence and it stores huge data (too big for memory)

In this point of view, the physical supports helps you to save your full context of work (applications and kernel states) between each reboot or power failure

The file system helps to organize data within a tree of files and directories, in order to avoid to browse within blocks

As the file system is a precise organization of data, it needs specific algorithms that do not work on other file systems.

There are 2 notions to distinguish:

- *The data stored following the « file system » organization*
- *The program/code/driver to manipulate data organized with this « file system »*

If you write some data in the FAT32 format, you cannot read/write it with a NTFS program or an ext2 program!

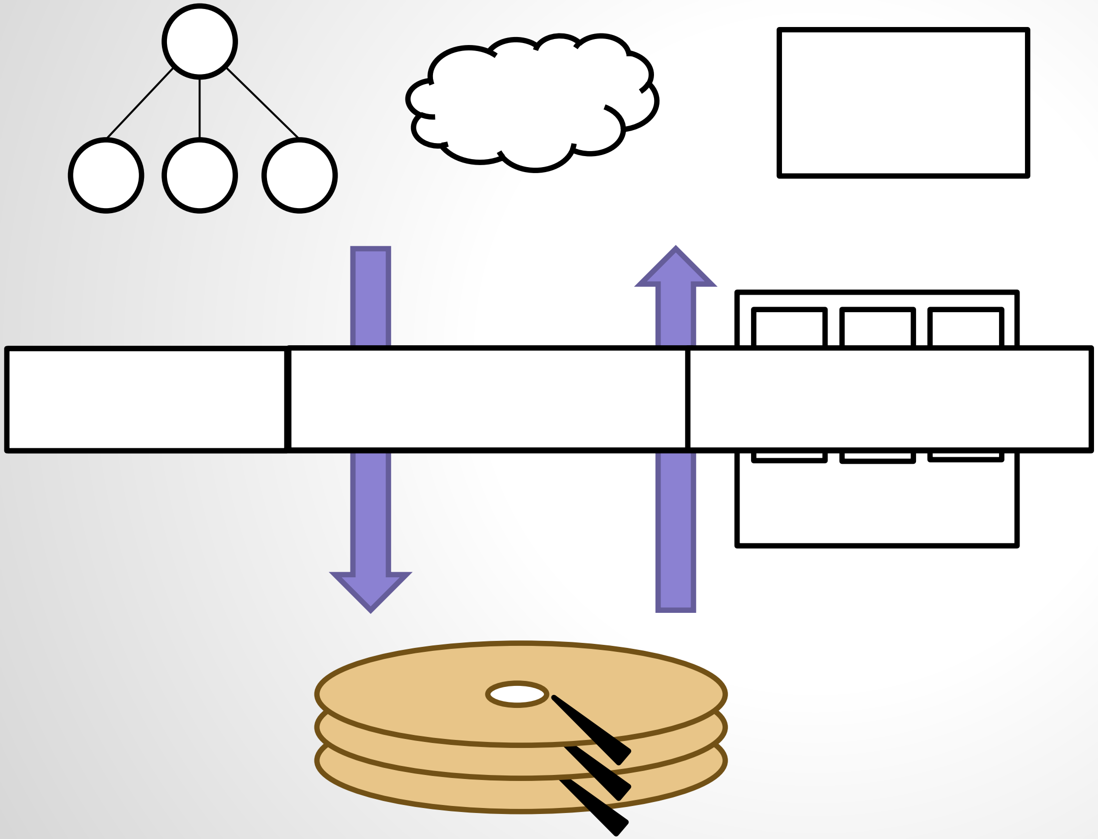
Virtual File System

VFS

(https://www.kernel.org/doc/html/latest/filesystems/vfs.html)
(http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/toc.html)

- Abstraction above concrete file systems
  - A software layer between the file system driver and the processes
  - Regular syscalls can be used above all of the file systems
- Currently, tightly linked with UNIX-likes
  - It has been ported on nearly all of the UNIX-likes
  - Inspired by existing concrete file systems
- Ease the management of the files tree
  - Everything is a file in a tree (even devices)
  - Allows multiple other file systems to be « mounted » on the tree
  - No need to select a « volume » anymore: everything is in the tree

VFS

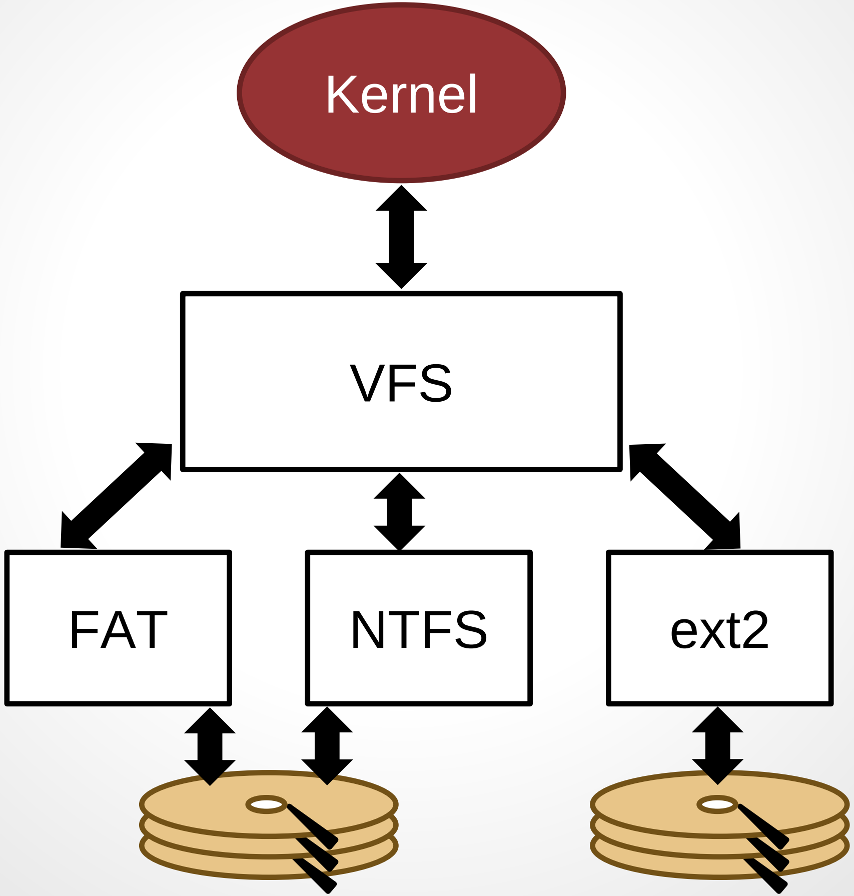


VFS:
Abstract structures and methods in memory

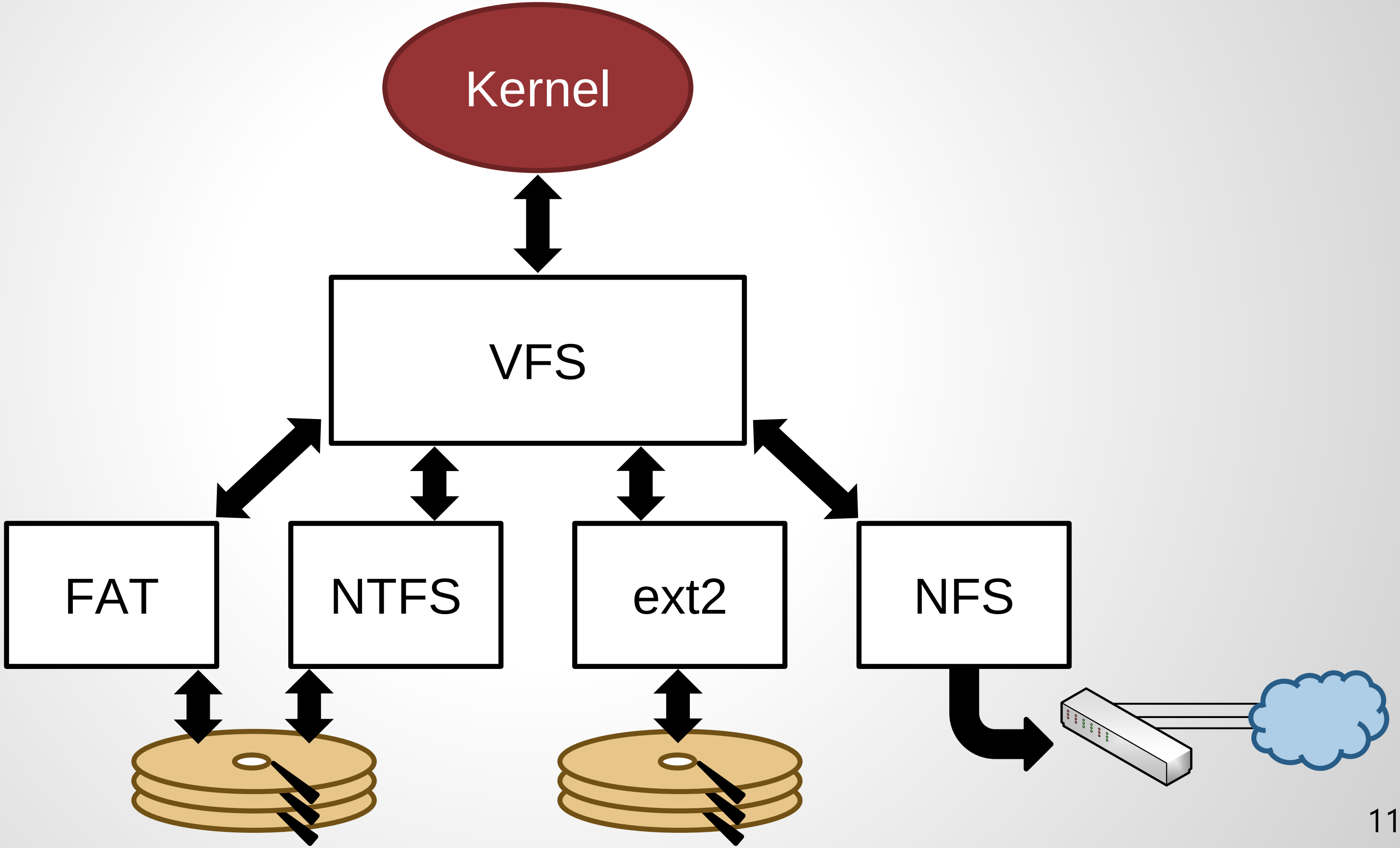
File System:
blocks/clusters and structures (Superblock, FAT, i-nodes, …) on physical support

Physical Support:
sectors

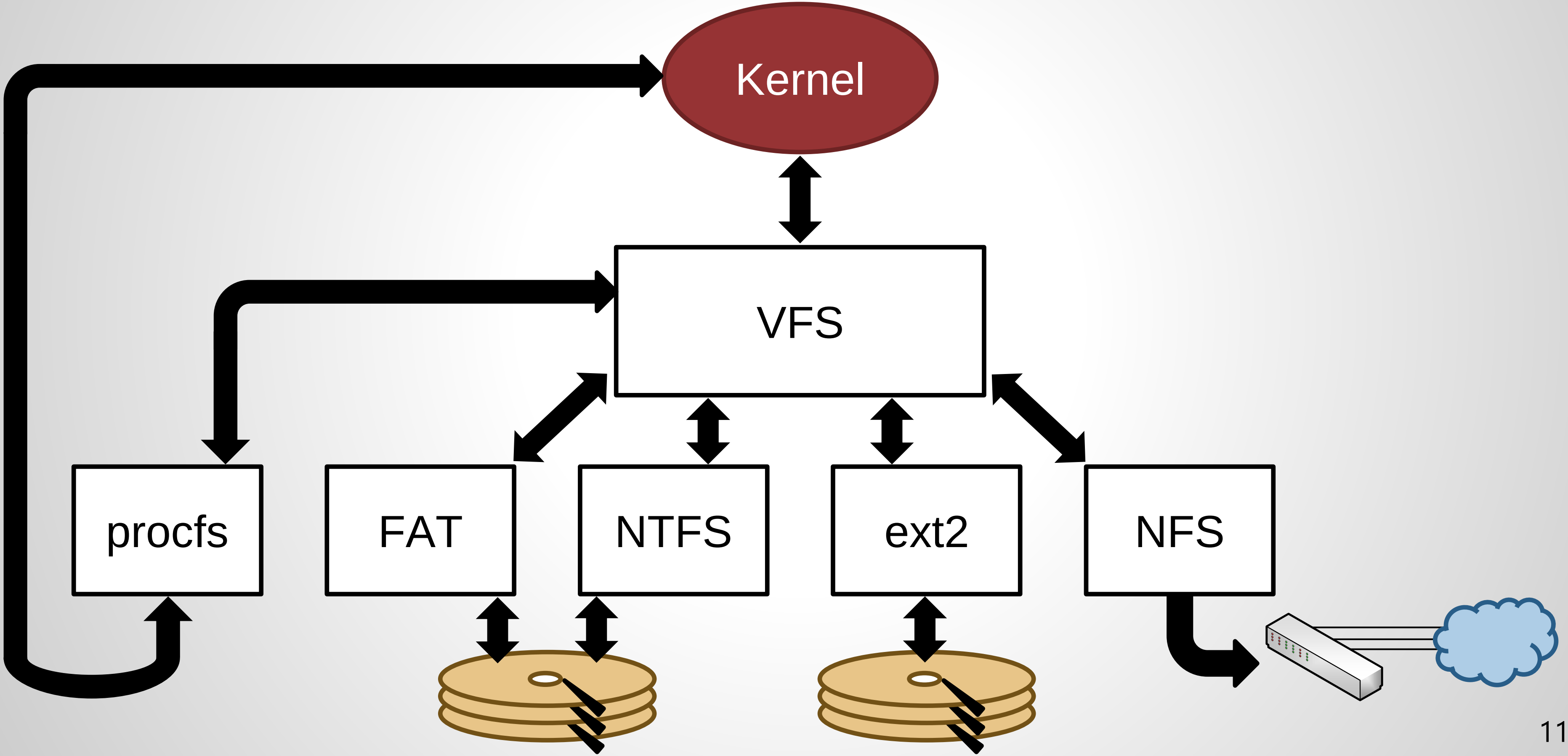
VFS



VFS



VFS



VFS: main concepts

- Superblock
  - Informations about the physical support or the partition
- Inode / i-node
  - index node: a structure describing each file and folder
- dentry (and dcache)
  - *Directory Entry*: Pathname manipulated by the OS (stays in memory)
- file
  - Representation of a file in memory

VFS: main concepts

- « Superblock » and « i-node » in their VFS version are tighly linked to their equivalent in the concrete FS
  - Depending on the underlying FS, they might require an adjustment *e.g.: FAT is not built upon « superblock » and « i-nodes »...*

- « dentry » and « file » are completly dissociated: they only exist as a structure in memory
  - They even are very different compared to their equivalent in ext2
  - They are more linked to the UNIX philosophy: « Everything is a file »

VFS: main concepts

- Each of these structures exposes functions to the kernel
  - open/read/write/close/… use these functions

- All the functions are not « implemented », in the sense that the underlying file system do not manage them
  - *e.g.: changing the owner of a file on a FAT12 volume has no sense*
  - *Or even: llseek() on a pipe or a socket neither has any sense*

VFS: Superblock

- Describes the file system mounted
  - If the object underlying the file system is not a physical support, the superblock is generated on-the-fly and kept in memory
    e.g.: the list of devices (/dev), procfs (/proc), sysfs, … (see mknod(2))
- Contains the attributes to manage the layers under
  - And even some methods/routines/procedures to manage the concrete underlying file system

VFS: Superblock

- Attributes:
  - Device ID *(ex: /dev/sda1)*
  - Mount point *(ex: /mnt or /media)*
  - File system type
  - Max file size
  - Size of the blocks managed
  - Address of the 1st i-node on the device/partition,
    and the address of the i-node of the directory where the device is mounted on (except for « / »)
    ex: « mount /dev/sda1 /mnt » => the @ of the i-node of /mnt is also kept
  - Lock
  - …
  - Superblock operations *(routines specific to the underlying FS)*
  - …

VFS: Superblock

- Superblock operations:
    - alloc_inode / destroy_inode
        - Create/Deallocates an i-node (in memory)
    - read_inode / write_inode
        - Read/Write the i-node on the physical support
    - drop_inode
        - Called when the last reference on an i-node is dropped (might call delete_inode)
    - delete_inode
        - Deletes the i-node from the physical support
    - ...
    - write_super
        - Writes the Superblock on the physical support
    - statfs
        - Get statistics about the underlying file system
    - ...

VFS: Superblock

- VFS Superblock is mapped on concrete FS equivalent

- In ext2: obviously, it is the ext2 Superblock
- In FAT: « Partition Boot Sector », and a bit of the « FAT »

VFS: i-node

- Structure containing attributes of the file
    - But still not his name

- Attributes:
    - device ID
    - i-node number
    - mode *(access rights)*
    - owner ID
    - links counter
    - …
    - i-node operations *(routines specific for the underlying FS)*
    - …
    - *[List of i-nodes, and list of dentries]*

VFS: i-node

- « *Device ID* + *i-node number* » are unique

- 2 ways for browsing through i-nodes:
  - Circular list of all the i-nodes
  - Hashmap using Superblock & i-node number as key

- As it’s a « VFS » i-node, it is manipulated in memory!
  - Concrete modifications are written on the FS with the concrete routines given by the FS driver/program/…

- Each file has an i-node
  - *ls –i myfile.txt*

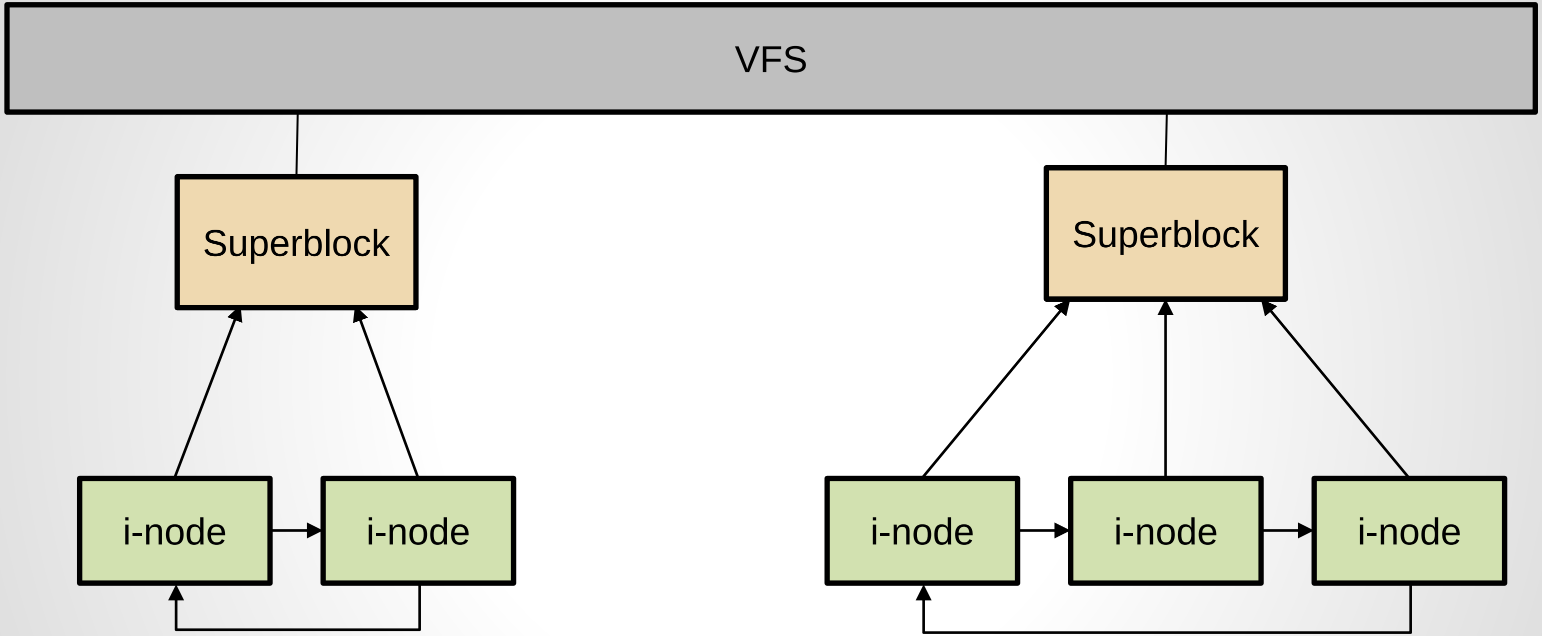
VFS: i-node

- i-node operations:
  - create(i-node, dentry, mode)
    - Creates a new i-node from a dentry
    - Called from a « creat » or « open » syscall
  - lookup(inode, dentry)
    - Searches the i-node corresponding to a filename (given in a dentry struct)
  - link/unlink/symlink
    - Creates/Removes a link (hard or symbolic) of a file in a dentry
    - Called from a « link », « unlink », or « symlink » syscall
  - mkdir/rmdir
    - Creates/Removes a directory from a dentry
    - Called from a « mkdir » or « rmdir » syscall
  - mknod
    - Creates a special file
    - Called from a « mknod » syscall
  - …

VFS: i-node

- rename(i-node, dentry, i-node, dentry)
  - Moves an old file (dentry) from a directory (i-node), into another directory (i-node)
- follow_link
  - Translates a symbolic link into the i-node it points to
- truncate(i-node)
  - Modifies the size of the file pointed by an i-node
  - (Update the new size in the struct, then call this method)
- permission(i-node, mask)
  - Checks if the rights given in mask are allowed on the i-node
- …

VFS: i-node



VFS: dentry

- Directory Entry

- Structure for validating a pathname
  - Contains pointers to parents and childs dentries
  - Makes the link between the name and the i-node associated
- Each part of the pathname is a dentry
  e.g.: /usr/bin/ls => 3 dentries (« usr », « bin », « ls »)

- VFS dentries do not correspond to any structure stored on the concrete file systems
  - But they are used in a more or less similar manner than « direntries »

VFS: dentry

- Attributes:
  - Dentry name
  - i-node of the folder which contains the dentry
  - Child dentries *[list]*
  - Subdirectories *[list]*
  - Parent dentry
  - Usage Count
  - Cache flag
  - Lock
  - Mount point *(is this dentry a mount point for another FS?)*
  - ...
  - dentry operations *(routines specific to VFS dentry)*
  - ...

VFS: dentry

- Dentry state

- When a dentry is tested (by a lookup), it is put in cache (in memory) in order to quickly access it again

- 3 states : used, unused, negative
  - Used: the dentry is associated with a valid i-node and is in use by at least one user (« usage count » not at 0)
  - Unused: the dentry is associated with a valid i-node, but nobody is using it (« usage count » at 0)
  - Negative: the pathname is incorrect, or the i-node pointed was deleted, but it is still in memory

VFS: dentry

We won’t see the cache in detail…

…but keep in mind that the VFS stores multiple lists of dentries in order to quickly resolve pathname…

VFS: dentry

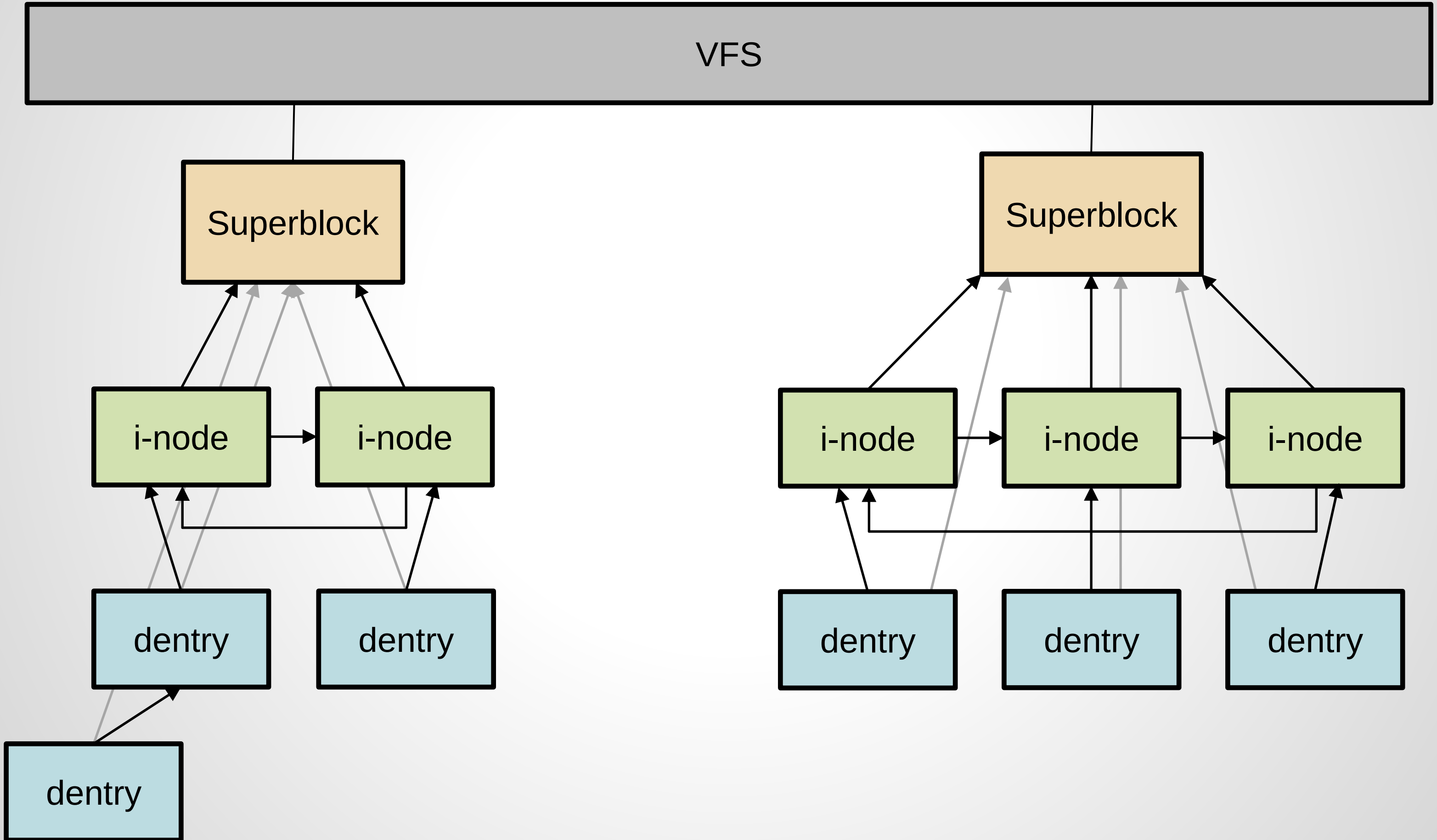
…and keep also in mind that « any » pathname asked to the VFS (even those that do not exist/are invalid) are stored in these lists in memory.

This mechanism avoids to access too often the physical support: overused pathnames are already in memory.

VFS: dentry

- Dentry operations:
    - revalidate
        - Checks if a dentry is valid within the cache
    - compare(pathname1, pathname2)
        - Compares two filenames
        - Might be implemented as a simple strcmp…
        - …or by a full check on the physical support « if » the concrete FS cannot assure that the pathname is different.
          Example: « /dir/file » and « /DIR/FILE »
          On UNIX, they are two different pathnames (strcmp is enough), but because FAT is not case sensitive (every name is stored in uppercase), it must be compared within the direntry on the physical support
    - …

VFS: dentry



VFS
Superblock
Superblock
i-node
i-node
i-node
i-node
i-node
dentry
dentry
dentry
dentry
dentry
dentry

VFS: file

- Structure describing an opened file
    - It is a « file descriptor »
    - Keeps the state of the read/write cursor for a file descriptor

- VFS files do not correspond to any structure stored on the concrete file systems
    - VFS file has a pointer to its VFS dentry, which points to its VFS i-node… which has a physical equivalent

- VFS file also have attributes and specific operations

VFS: file

- Attributes:
  - dentry associated
  - Mounted FS
  - Usage count
  - Access mode *(permissions)*
  - Offset within the file
  - Error code
  - List of event poll links *(alarm for events on the file)*
  - Owner UID
  - …
  - file operations *(routines specific to VFS file)*
  - …

VFS: file

- File operations:
  - open
    - Creates a new file object and links it to the corresponding i-node
    - Called by the open syscall
  - mmap
    - Maps the file into the given address space
    - Called by mmap syscall
  - flush
    - Called whenever the « usage count » decreases (depends on the underlying FS)
  - release
    - Called when the last reference on the file is destroyed
    - Called when the last process using the file makes a close syscall
  - check_flags
    - Checks the flags given to fcntl syscall
    - Used on some FS (like NFS on network)

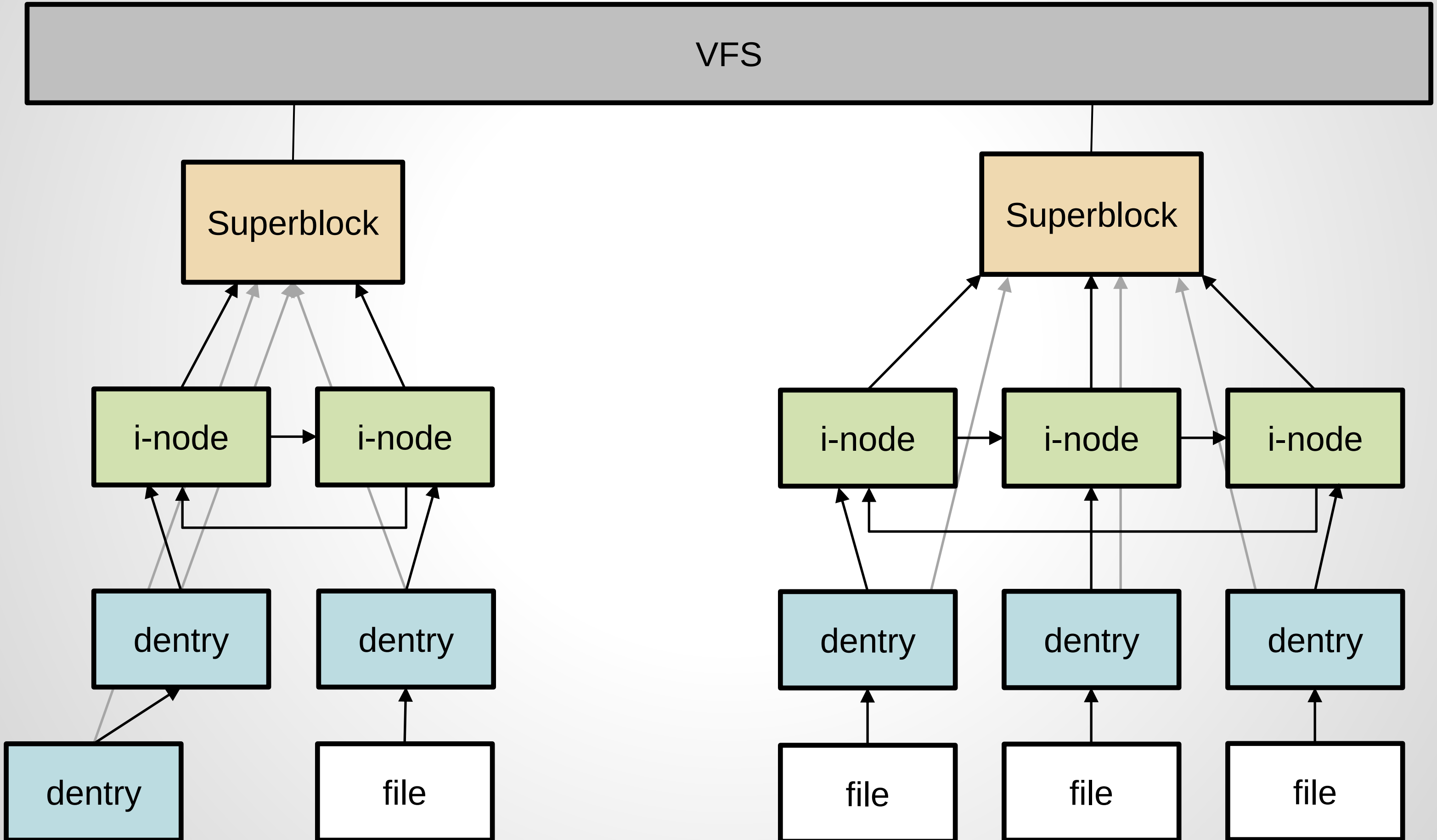
VFS: file

- File operations:
  - llseek
    - Update the offset pointer
    - Called by a llseek syscall
  - read / write
    - Read/Write data into/from a buffer into a file from an offset, and update the struct
    - Called by a read/write syscall
  - aio_read / aio_write
    - Same as read/write, but asynchronously *(we’ll see this later)*
  - readdir
    - Return the next directory pointer
    - Called by readdir syscall
  - sendfile / sendpage
    - Send one VFS « file » to another one (usually: sending a physical file to a socket)
    - Stay in kerneland, do not copy data to userland! (the kernel buffer is transferred)

VFS: file

- File operations:
  - fsync / aio_fsync
    - Writes all cache data into the file *(synchronous and asynchronous versions)*
    - Called by the fsync/aio_fsync syscall
  - poll
    - Waits for an event on the file (as any event on a file must pass through the VFS)
    - Called by a poll syscall
  - ioctl
    - Sends a command and an argument to a device
    - Called by an ioctl syscall
  - lock
    - Manipulates the lock on a file
  - …

VFS: dentry



VFS
Superblock
Superblock
i-node
i-node
i-node
i-node
i-node
dentry
dentry
dentry
dentry
dentry
dentry
file
file
file
file

VFS

- VFS dentries and VFS files are just structures manipulated in memory

- VFS Superblocks and i-nodes are structures in memory, but they also have an existence on the concrete FS
  - When a modification is done in the memory structure, they are marked as « dirty », because they must be written on the physical support before being really stored

- A modification in a VFS file is repercuted in the i-node
  - And the i-node becomes « dirty », awaiting to be written on disk

How are all of those concepts working together in UNIX?

VFS & Processes: open(2) example

- What's happening when an open(2) is made?

1. The process calls the open(2) routine with parameters (filename and opening mode)

2. It passes in syscall mode (higher ring, supervisor, ...) and calls a kernel function « sys_open() » with the filename and the opening mode

VFS & Processes: open(2) example

- What’s happening when an open(2) is made?

3. A new file descriptor entry is made
   -> a VFS « file » is created

4. The filename is converted into a device ID + a VFS « i-node » number
   -> a VFS « dentry » is created

VFS & Processes: open(2) example

- What’s happening when an open(2) is made?

5. The VFS « i-node » of the file is obtained with the « lookup() » method
   -> a VFS « i-node » is created

6. The VFS « i-node » of the file is used to fill various structures (particularly the VFS « file » created at the step 3)

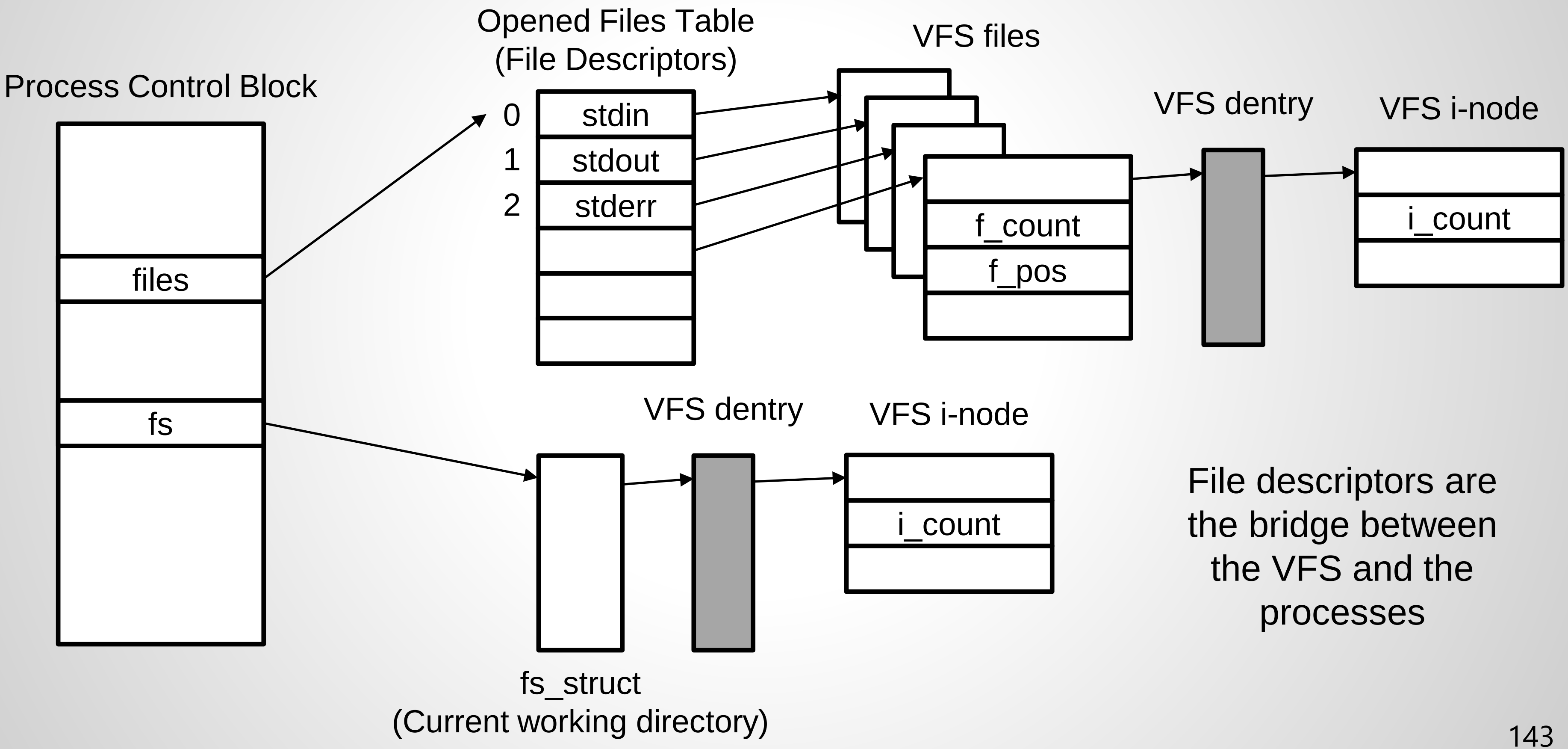
VFS & Processes: open(2) example

- What’s happening when an open(2) is made?

7. The « open() » method of the VFS « file » is called in order to access physically to the concrete file
   -> an « open » on the concrete file system is made

8. The file descriptor table is updated with a reference to the VFS « file »
   -> the file descriptor number is returned to the user

VFS & Processes



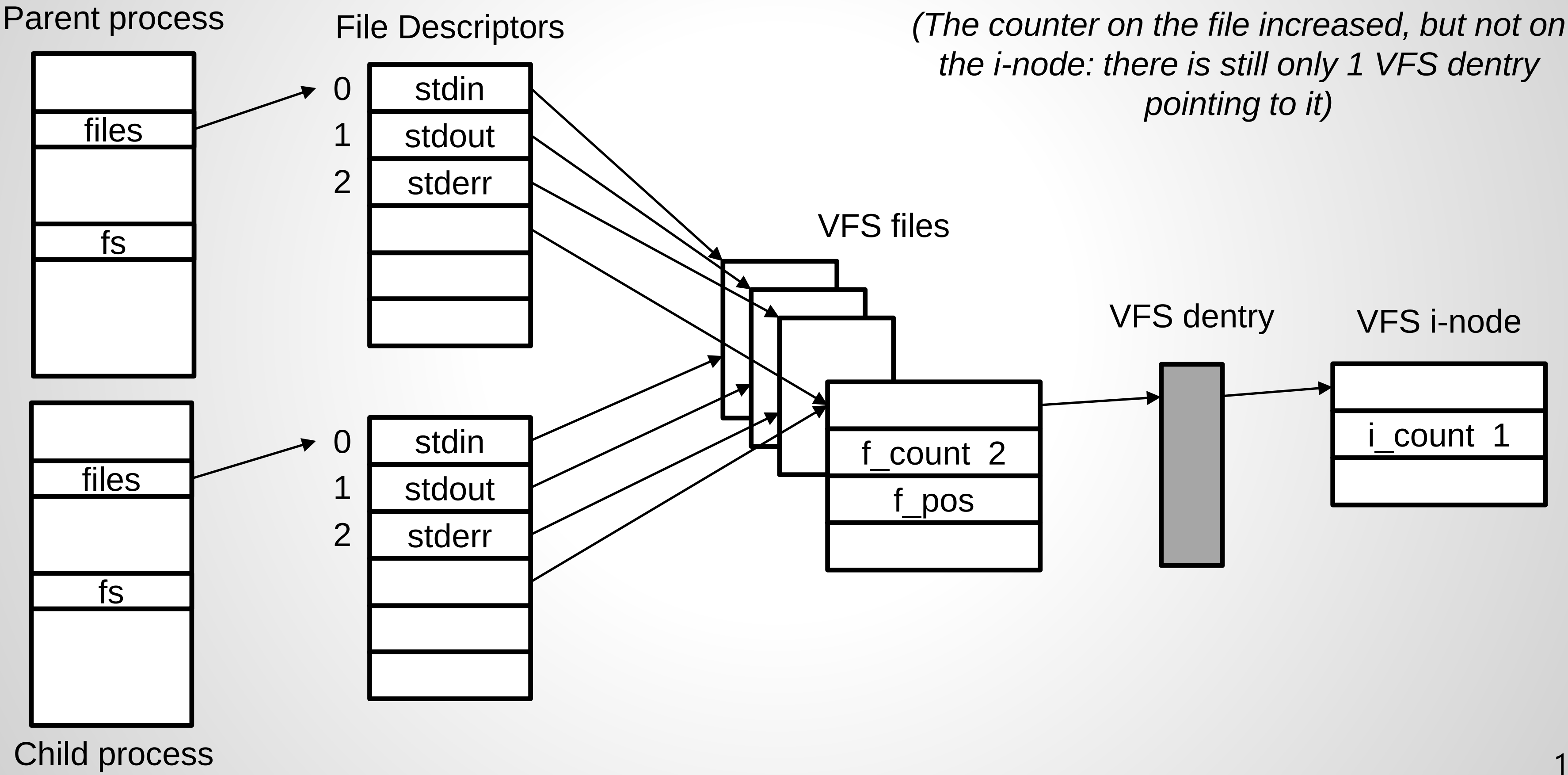
File descriptors are the bridge between the VFS and the processes

What’s happening if there is a fork(2)?

VFS & Processes

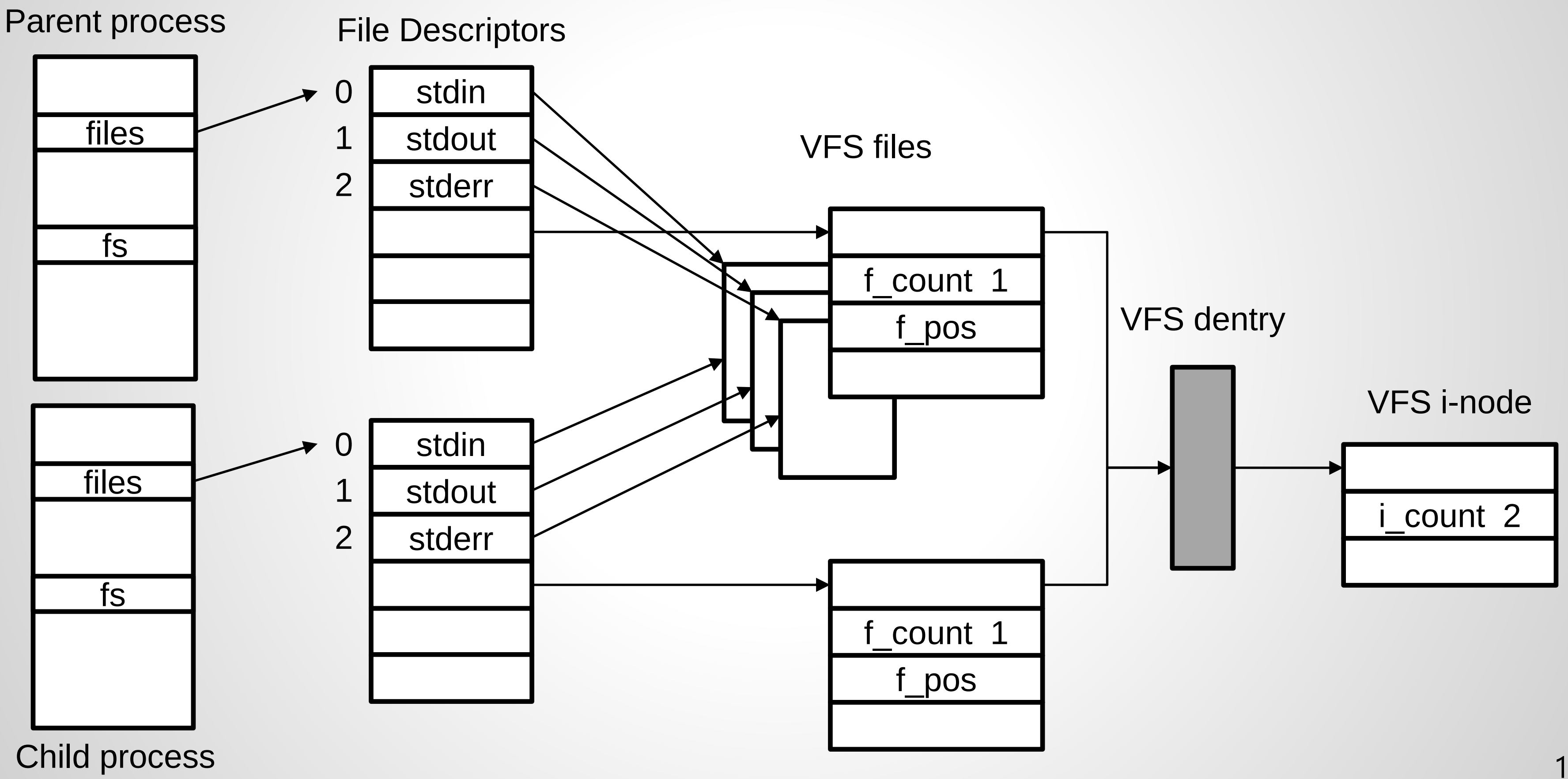
When opening **before** a fork, the PCB and file descriptors are duplicated… but the VFS structures are still the same

(The counter on the file increased, but not on the i-node: there is still only 1 VFS dentry pointing to it)



VFS & Processes

When opening **after** a fork, the VFS file structure is created two times, and the i-node count is increased



Abstraction seen from the userland

File System Syscalls

- open, read, write, close, lseek, …
    - Access to the file
- mkdir, rmdir, opendir, readdir, closedir, chdir, unlink, …
    - Access to the directory
- symlink, readlink
    - Access to a symbolic link
- stat, access, chmod, chown, …
    - Access to the informations about a file or a directory
- fstat, fcntl, …
    - Access to the file descriptor

File System Syscalls: open

- open(char *pathname, int flags, [mode_t mode])
- Opens a file for reading or writing

- Pathname: the pathname to the file
- Flags: the options explaining what the program expects
- [Mode: permissions in case of a file creation]

Don’t forget you can make a binary OR « | » on the flags and mode

File System Syscalls: open

- open(char *pathname, int flags, [mode_t mode])
- Flags:
  - O_RDONLY, O_WRONLY, ORDWR
    read only, write only, read and write
  - O_CREAT
    create a file if it does not exist
    (if with O_EXCL, a failure will happen if the file already exists)
  - O_TRUNC
    empty the file if it exists
  - O_APPEND
    place the cursor at the end of the file (useful for writing…)
  - O_SYNC
    Activates the synchronization mode, each write will be made on the disk immediatly

File System Syscalls: open

- open(char *pathname, int flags, [mode_t mode])
- Mode:
  - S_ISUID, S_ISGID, S_ISVTX
    setuid, setgid, sticky bit *(see chmod(2) and sticky(7))*

  - S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU
    Owner: read, write, execute, Read/Write/eXecute

  - S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXG
    Group: read, write, execute, Read/Write/eXecute

  - S_IROTH, S_IWOTH, S_IXOTH, S_IRWXO
    Others: read, write, execute, Read/Write/eXecute

File System Syscalls: close

- close(int fd)
- Closes a file descriptor

- fd: The file descriptor to close

- The file descriptor entry is removed, therefore, the VFS « file » counter is reduced
    - If the VFS « file » counter is at 0, it is removed, and the VFS « i-node » counter is therefore also reduced

File System Syscalls: lseek

- lseek(int fd, off_t dep, int option)
- Moves the read/write cursor within the file
    - Returns the new position from the beginning of the file

- fd: The file descriptor pointing to the file
- dep: The length (in bytes) to shift
- option: From which point the shhift must be done
    - SEEK_CUR   from current cursor position
    - SEEK_SET   from the beginning of the file
    - SEEK_END   from the end of the file

File System Syscalls

- About permissions:

- Adds the rights you want with a simple bitfield

- 4 / R = Read
- 2 / W = Write
- 1 / X = eXecute

• Owner	X00	XXX------
• Group	0X0	---XXX---
• Others	00X	------XXX

7 = 4 + 2 + 1 => 111

File System Syscalls

- About special permissions on **executable programs**:

- SETUID (or SUID, or *Set User ID*)
  - The process created will run with the UID of the owner of the file

- SETGID (or SGID, or *Set Group ID*)
  - The process created will run with the GID of the group of the file

- *Sticky Bit (obsolete)*
  - *The text segment of the program is kept in memory after the execution*

File System Syscalls

- About special permissions on **directories**:

- ~~SETUID (or SUID, or *Set User ID*)~~ *[No use for directories]*

- SETGID (or SGID, or *Set Group ID*)
  - The files/folders created within a parent folder with SETGID activated will be created with the GID of the parent folder (and the new subfolders will also get the SETGID permission)

- Sticky Bit
  - A user cannot delete or rename files owned by others within a folder with the sticky bit activated

File System Syscalls: umask

- About permissions: umask

- Blocks the selected bits within the calls to open(2), mkdir(2), mkfifo(2)
    - Default value: 022   (Write for owner, but not for group and others)

- Works exactly like a « mask »
    - Just inverse values…
    - umask 777 = Cancels all the rights to everyone
    - umask 000 = Allows any right to anyone

File System Syscalls: mkdir

- mkdir(char *pathname, mode_t *mode)
- Creates a directory
  - UID and GID are used, and the umask is applied
  - Beware, mkdir(2) is not automatically recursive

- Pathname: the pathname where to create a directory
- Mode: permissions to write in case of a file creation

File System Syscalls: stat

- stat(char *pathname, struct stat *buf)
- Get informations about the object pointed by pathname

- Pathname: the pathname to the file
- Buf: a buffer in memory that will be filled by stat(2)

(if it succeeds)

File System Syscalls: stat

- struct stat *(see <sys/stat.h>)*

○ st_dev	ID of the device containing the object
○ st_ino	i-node number of the object
○ st_mode	Status of the object *(permissions & more)*
○ st_uid, st_gid	UID and GID of the owner of the object
○ st_nlink	Number of links pointing to the object
○ …	
○ st_size	Size of the object in bytes
○ st_blocks	Number of blocks used *(512 Bytes « blocks »)*
○ st_blksize	Size of the blocks for the I/O with the device
○ …	
○ st_[a/c/m]time	Timestamp for last access/status change/modification

File System Syscalls: stat

- struct stat *(see <sys/stat.h>)*
  - st_size Size of the object in bytes
  - st_blocks Number of blocks used ***(512 Bytes « blocks »)***
  - st_blksize Size of the blocks for the I/O with the device

```
touch TEST


Stat output:
 File size (B):             0
 Nb blocks:                 0
 Block size:                4096
```

```
touch TEST2
echo "a" > TEST2

Stat output:
 File size (B):             2
 Nb blocks:                 8
 Block size:                4096
```

File System Syscalls: stat

- struct stat *(see <sys/stat.h>)*
  - st_mode Status of the object ***(permissions & more)***

- Permissions can be tested with the S_IRWXU, ...
  - Just test a lot of cases as previously seen

- Other special cases concerns the type of file

File System Syscalls: stat

S_ISREG(m)	*is it a regular file?*
S_ISSOCK(m)	*socket?*
S_ISFIFO(m)	*named pipe?*
S_ISCHR(m)	*character device?*
S_ISDIR(m)	*directory?*
S_ISLNK(m)	*symbolic link?*
S_ISBLK(m)	*block device?*

File System Syscalls

- About character and block devices:

- Character devices are devices that work with characters
    - e.g.: Keyboard, terminal, …
    - *Keyboards and terminal manages characters*

- Block devices are devices that read/write blocks
    - e.g.: Hard drive, network, …
    - *Hard drives manipulate sectors or blocks*
    - *Network cards manipulate IP packets or frames*

The VFS contributes actively to abstract more the management of files either they are stored on « real » devices or just a kernel extension.

File descriptors, thanks to their structure and fork(2) behavior, allow to share access to the same files…

…however, the reading/writing cursor is also share in some cases!

How to manage concurrency challenges?