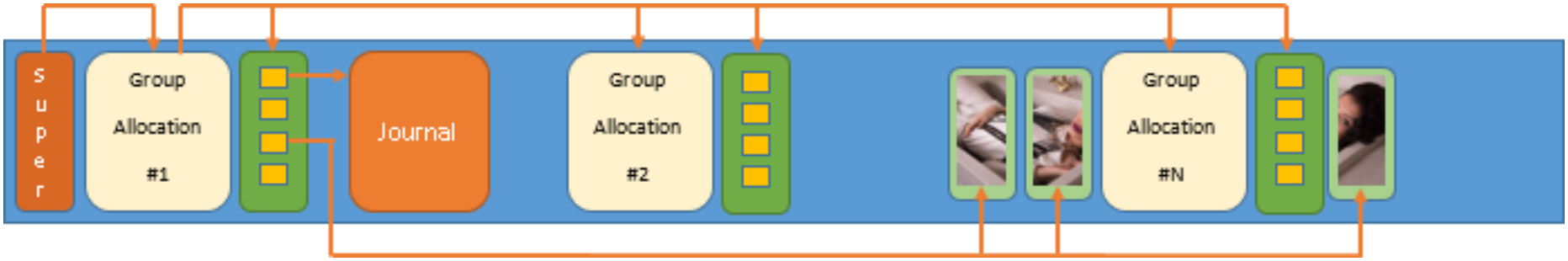
Operating Systems

Eric Lo

12 - File Systems

File System

- A way that lays out how data is organized on a storage device



http://www.raisedr.com/kb/recovery_cha.php

File System

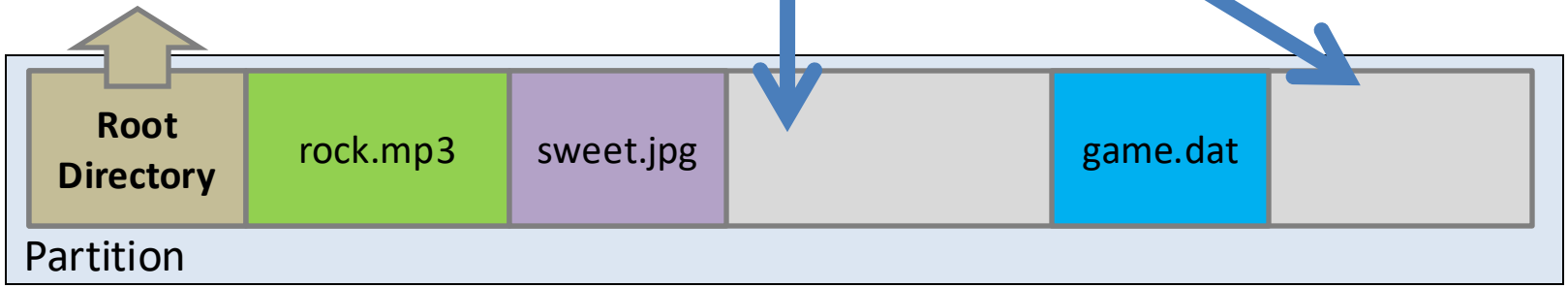
- Layout
  - Contiguous allocation
  - Linked allocation
  - INode allocation

Contiguous allocation – basics

Locate files easily.

Filename	Starting Address	Size
rock.mp3	100	1900
sweet.jpg	2001	1234
game.dat	5000	1000

Free space is here



Contiguous allocation – basics

File deletion is easy! Space de-allocation is the same as updating the root directory!

Yet, how about file creation?

Filename	Starting Address	Size
rock.mp3	100	1900
~~sweet.jpg~~	~~2001~~	~~1234~~
game.dat	5000	1000

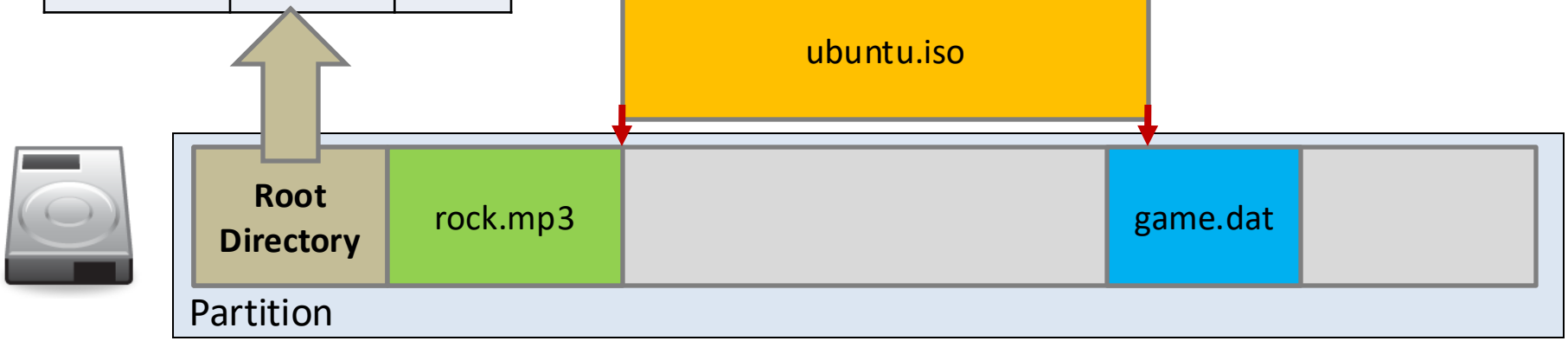
Filename	Starting Address	Size
rock.mp3	100	1900
game.dat	5000	1000

Contiguous allocation – basics

Really BAD! We have enough space, but there is no holes that I can satisfy the request. The name of the problem is called:

External Fragmentation

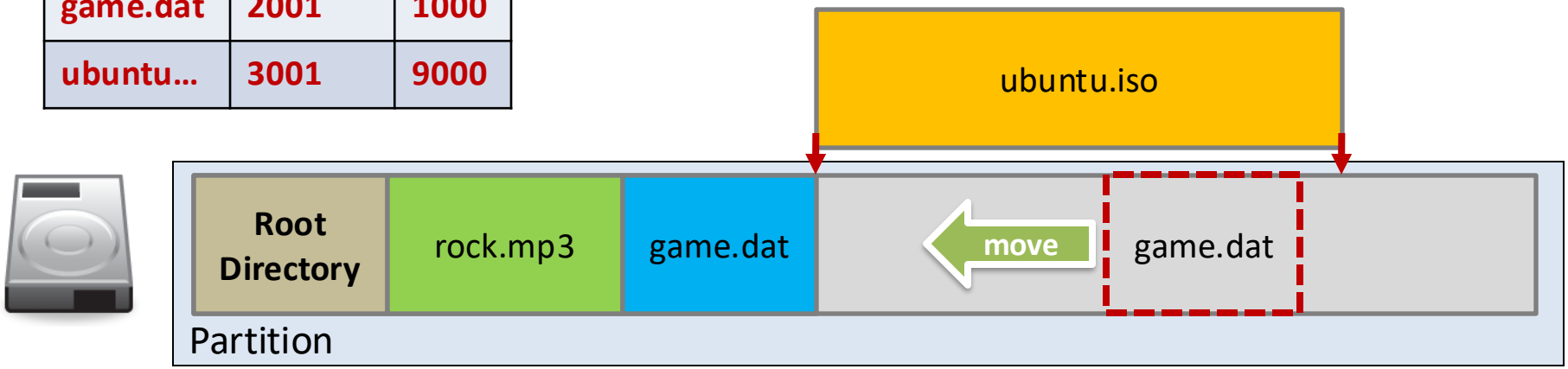
Filename	Starting Address	Size
rock.mp3	100	1900
game.dat	5000	1000



Contiguous allocation – basics

Defragmentation process may help!
You know, this is very expensive as you're working on disks.

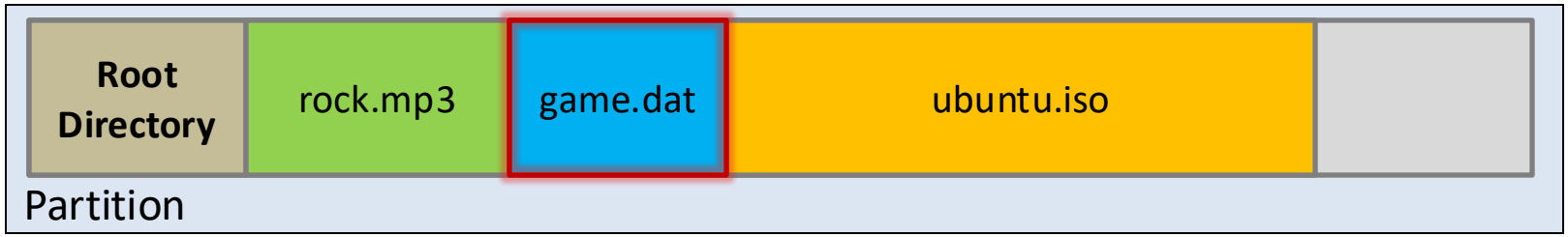
Filename	Starting Address	Size
rock.mp3	100	1900
game.dat	**2001**	**1000**
ubuntu…	**3001**	**9000**



Contiguous allocation – basics

Filename	Starting Address	Size
rock.mp3	100	1900
game.dat	2001	1000
ubuntu…	3001	9000

Growth problem!

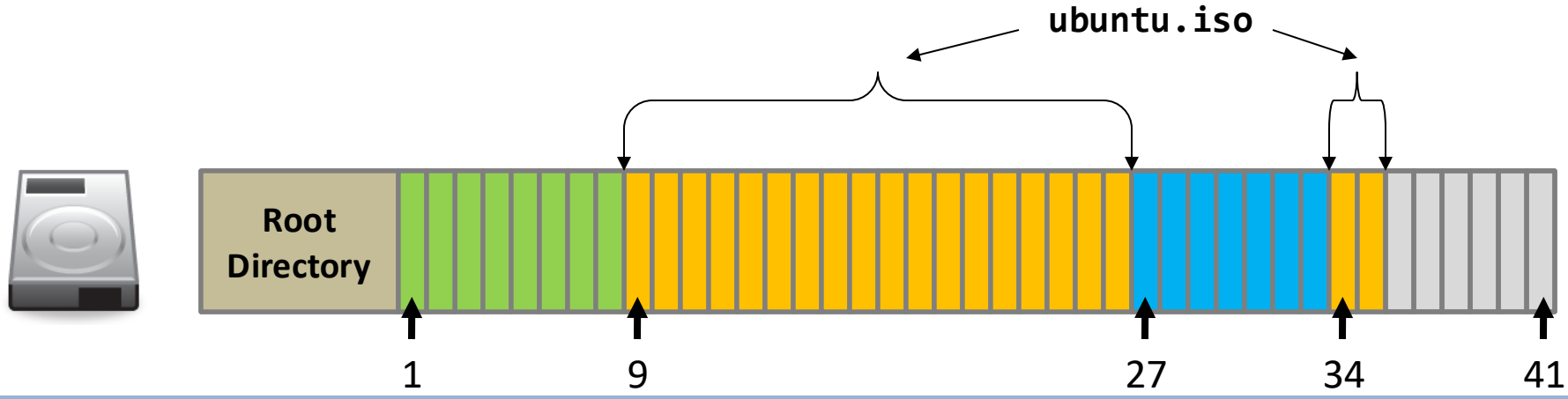


Contiguous allocation – application?

- ISO 9660
- CD-ROM
  - .iso image

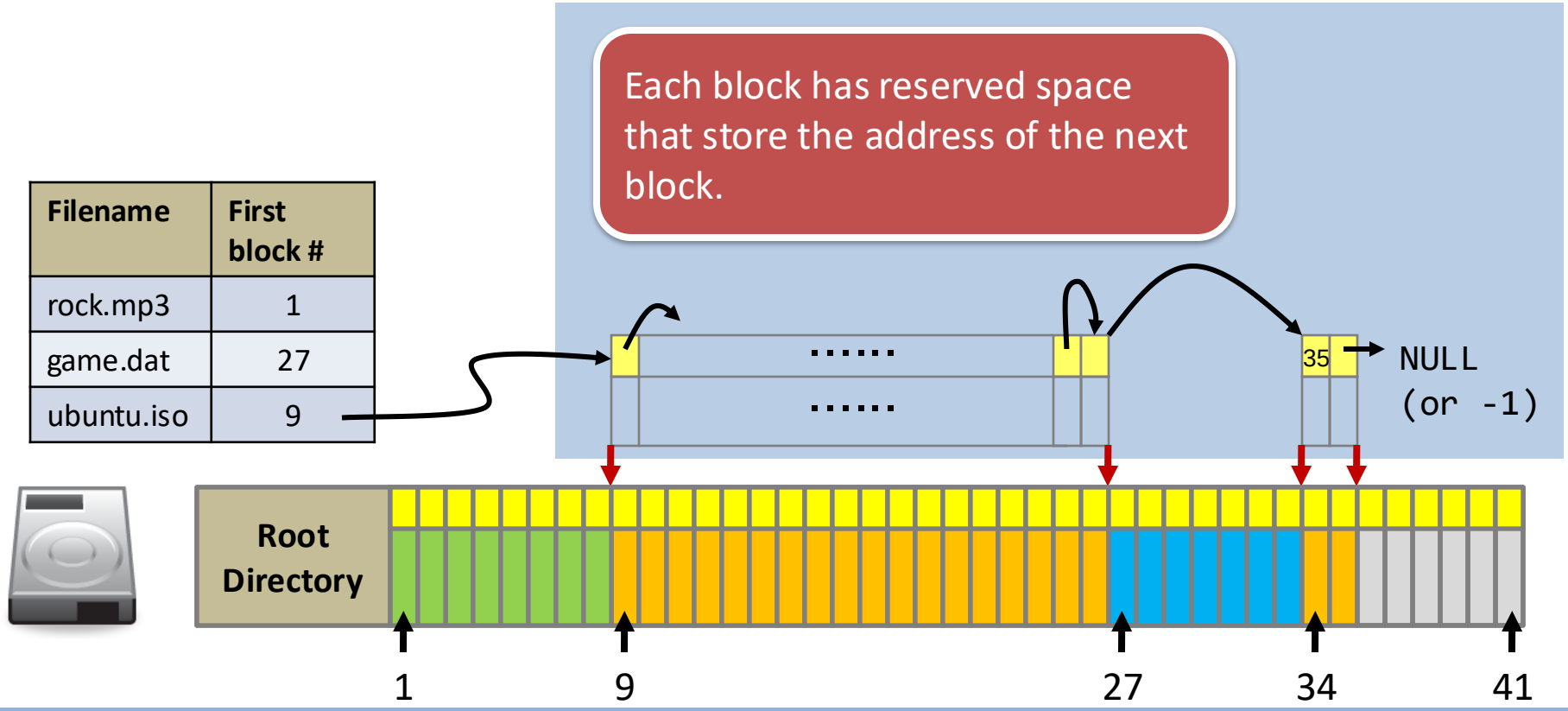
Linked allocation

- Let's borrow the idea from the linked list …
  - Step (1) Chop the storage device into **equal-sized blocks**.
  - Step (2) Fill the empty space in a **block-by-block** manner.



Linked allocation

- Leave **4 bytes from each block** as the "pointer"
  - To write the block # of the next block into the first 4 bytes of each block.



Linked allocation

- Also keep the file size in the root directory table
  - To facilitate "ls –l" that lists the file size of each file
  - (otherwise needs to live counting how many blocks each file has)

Filename	First block #	Size
rock.mp3	1	1900
game.dat	27	1000
ubuntu.iso	9	9000

Linked allocation

- So, how would you grade this file system?
  - External fragmentation?
  - File growth?

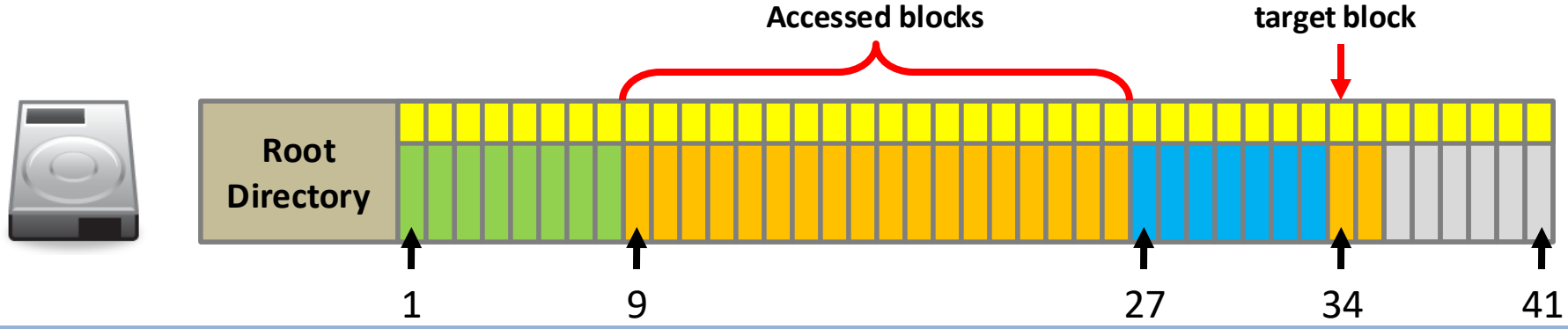
Filename	First block #	Size
rock.mp3	1	1900
game.dat	27	1000
ubuntu.iso	9	9000

Linked allocation

- **Internal Fragmentation**.
  - A file is not always a multiple of the block size.
    - The last block of a file may not be **fully filled**.
    - E.g., a file of size 1 byte still occupies one block.
  - The remaining space will be wasted since no other files can be allowed to fill such space.

Linked allocation

- **Poor random access performance**.
  - What if I want to access the 19-th block of ubuntu.iso?
  - **You have to access blocks 1 – 18 of ubuntu.iso until the 19-th block ➔ pointer chasing**



FAT

- Centralize all the block links as File Allocation Table

FAT

Task: read "ubuntu.iso" sequentially.

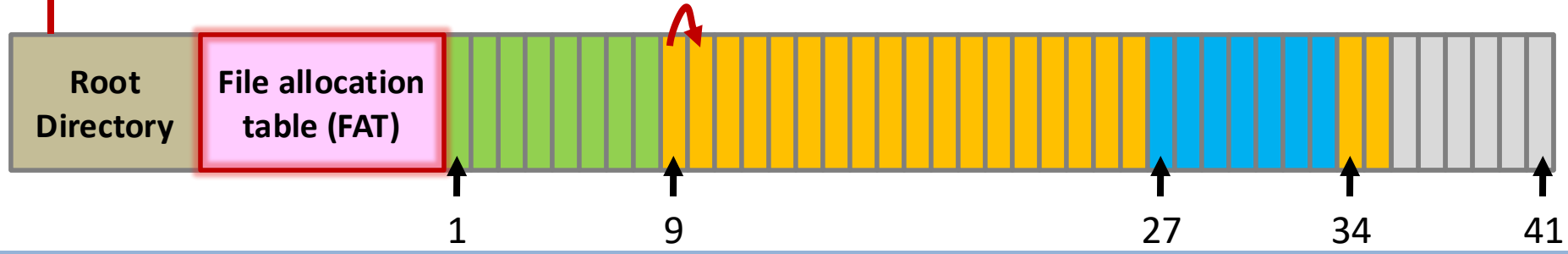
Filename	First block #	Size
rock.mp3	1	1900
game.dat	27	1000
ubuntu.iso	9	9000

Step 1. Read the root directory and retrieve the **first block number**.

Step 2. Read the FAT to determine the location of next block.

File allocation table (FAT)

Block #	1	…	7	8	9	…	26	27	…	32	33	34	35	…	41
Next Block #	2	…	8	-1	10	…	34	28	…	33	-1	35	-1	…	0



FAT

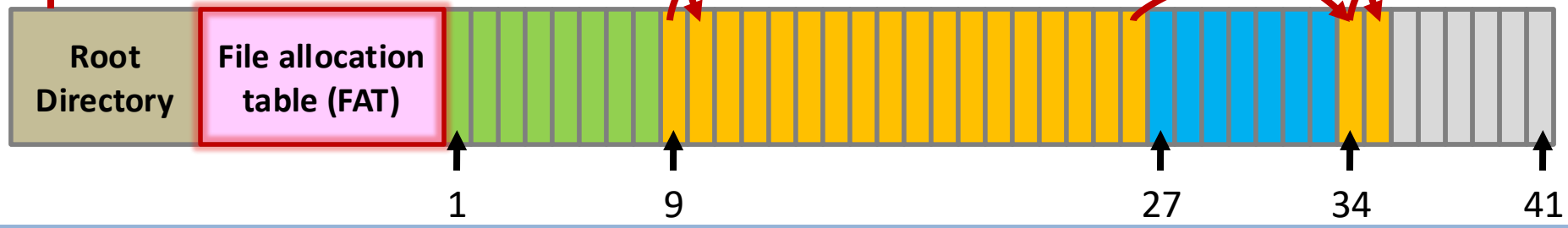
Task: read "ubuntu.iso" sequentially.

Filename	First block #	Size
rock.mp3	1	1900
game.dat	27	1000
ubuntu.iso	9	9000

Step 3. After reading the 2nd block, the process continues. Note that the blocks **may not be contiguously allocated**.

Step 4. The process stops until the FAT says the next block # is -1.

Block #	1	…	7	8	9	…	26	27	…	32	33	34	35	…	41
Next Block #	2	…	8	-1	10	…	34	28	…	33	-1	35	-1	…	0

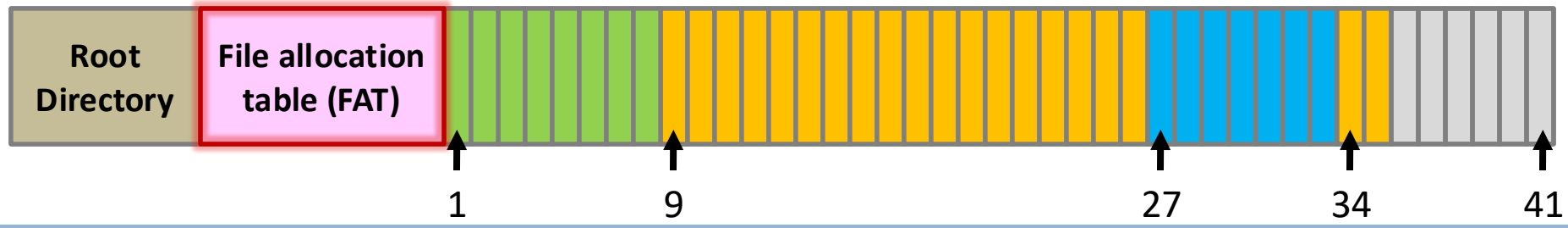


FAT

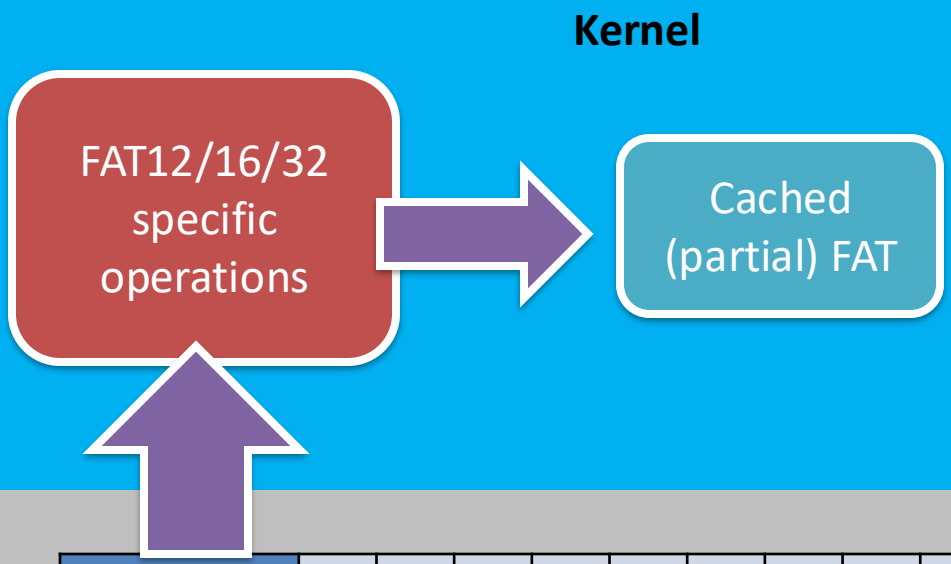
Resulting layout & file allocation.

Filename	First block #	Size
rock.mp3	1	1900
game.dat	27	1000
ubuntu.iso	9	9000

Block #	1	…	7	8	9	…	26	27	…	32	33	34	35	…	41
Next Block #	2	…	8	-1	10	…	34	28	…	33	-1	35	-1	…	0



FAT – keeping a (partial) FAT in kernel cache



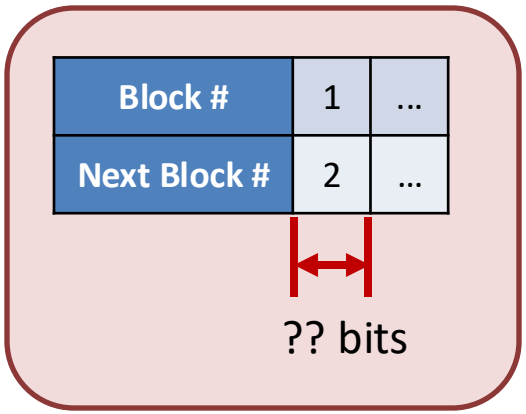
Block #	1	…	7	8	9	…	26	27	…	32	33	34	35	…	41
Next Block #	2	…	8	-1	10	…	34	28	…	33	-1	35	-1	…	0

Disk

File Allocation Table (FAT)

FAT

- Start from floppy disk and DOS
- On DOS, a block is called as a ‘cluster’
- E.g., FAT12
  - 12-bit cluster address
  - Can point up to 2^{12} = 4096 blocks



	FAT12	FAT16	FAT32
Cluster address length	12 bits	16 bits	28 bits
Number of clusters	2^{12} (4,096)	2^{16} (65,536)	2^{28}

MS reserves 4 bits (but nobody eventually used those)

FAT

- Size of a block (cluster):

Available cluster sizes (bytes)								
512	1K	2K	8K	16K	32K	64K	128K	256K

Cluster size: 32KB

Cluster address: 28 bits

E.g.,

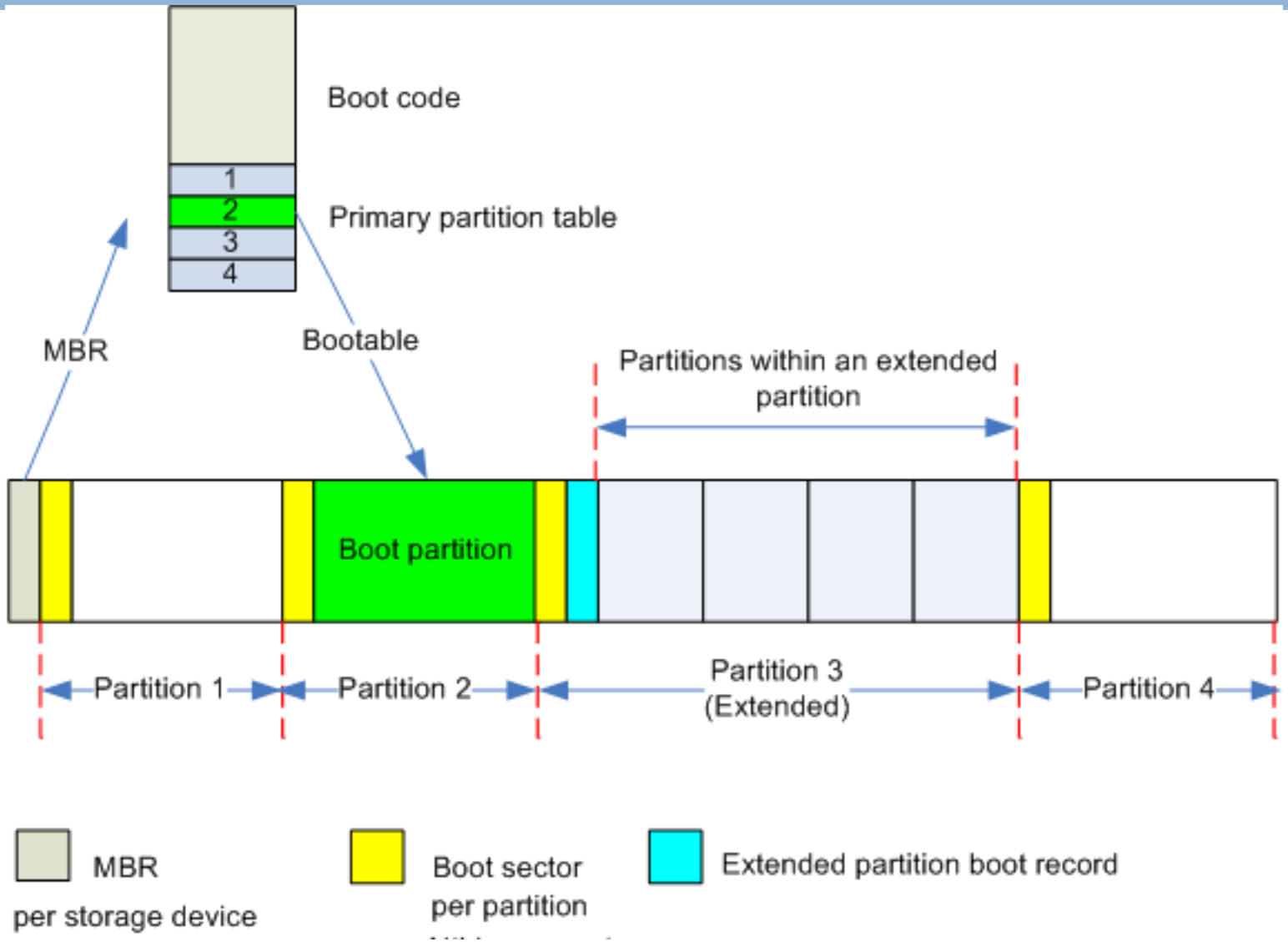
File system size.

$$(32 \times 2^{10}) \times 2^{28} = 2^5 \times 2^{10} \times 2^{28}$$
$$= 2^{43} \ (8\ TB)$$

*** but MS deliberately set its formatting tool to format it up to 32GB only to lure you to use NTFS**

File system size vs. Maximum file size

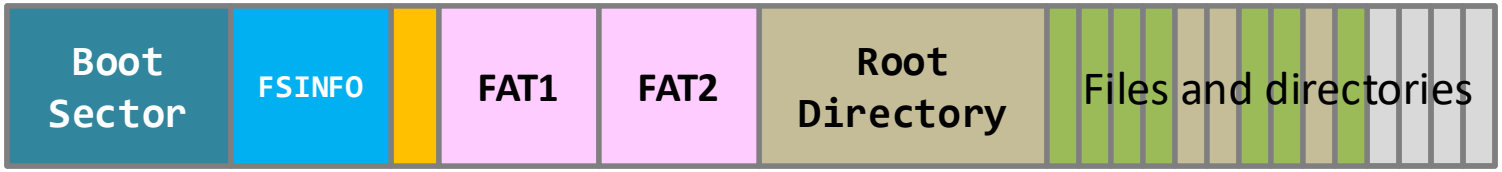
- Anyway, there are X blocks, and each block is Y kb in sizes.
- File system size = XY kbytes
  - i.e., can store up to XY kbytes of data
- Depends on a specific file system implementation
  - It may allow one file that big
  - Or put a limit of the maximum file size
  - Or depends on the dirent structure (we'll see later)


Boot code
1
2
3
4
Primary partition table
MBR
Bootable
Partitions within an extended partition
Boot partition
Partition 1
Partition 2
Partition 3 (Extended)
Partition 4
MBR per storage device
Boot sector per partition
Extended partition boot record

A FAT partition

	Propose	Size
Boot sector	FS-specific parameters	1 sector, 512 bytes
FSINFO	Free-space management	1 sector, 512 bytes
More reserved sectors	Optional	Variable, can be changed during formatting
FAT (2 pieces)	1 copy as backup	Variable, depends on disk size and cluster size.
Root directory	Start of the directory tree.	At least one cluster, depend on the number of directory entries.

Reserved sectors



A FAT partition

FAT series – directory traversal

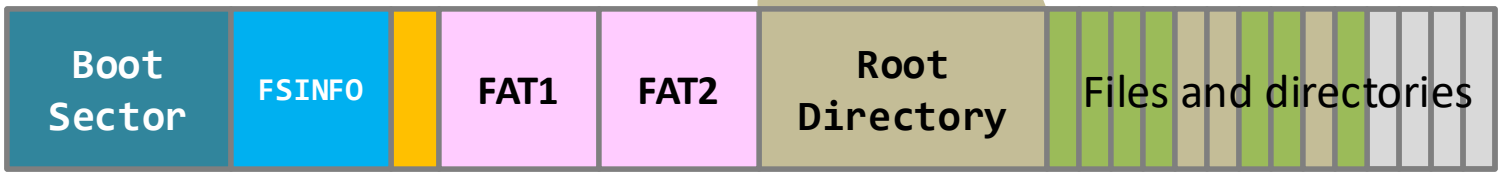
Step (1) Read the directory file of the root directory starting from Cluster #2.

"**C:\windows**" starts from Cluster #123.

```
c:\> dir c:\windows
……
06/13/2007  1,033,216    gamedata.exe
08/04/2004     69,120    notepad.exe
……
c:\> _
```

A directory entry

Cluster #2		
Filename	Attributes	Cluster #
.	……	……
..	……	……
……	……	……
windows	……	123



FAT series – directory traversal

Step (2) Read the directory **file** of the "**C:\windows**" starting from **Cluster #123**.

```
c:\> dir c:\windows
……
06/13/2007   1,033,216     gamedata.exe
08/04/2004      69,120     notepad.exe
……
c:\> _
```

Cluster #123		
Filename	**Attributes**	**Cluster #**
.	……	……
..	……	……
……	……	……
notepad.exe	……	456

FAT series – directory entry

- A 32-byte directory entry in a directory file
- A directory entry is describing a file (or a sub-directory) under a particular directory

Bytes	Description
0-0	1st character of the filename (0x00 or 0xe5 means unallocated)
1-10	remaining characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster number (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster number.
28-31	File size.

Filename	Attributes	Cluster #
gamedata.exe	.....	32

0	g	a	m	e	d	a	t	a	7
8	e	x	e	…	…	…	…	…	15
16	…	…	…	…	00	00	…	…	23
24	…	…	20	00	00	C4	0F	00	31

Note. This is the 8+3 naming convention.

8 characters for name +
3 characters for file extension

FAT series – directory entry

- The 1st block address of that file

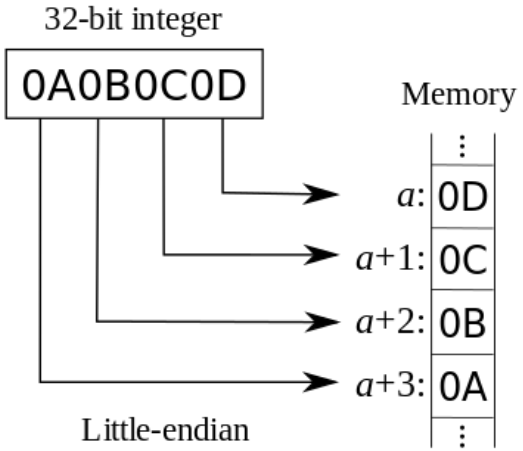
Bytes	Description
0-0	1st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster number (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster number.
28-31	File size.

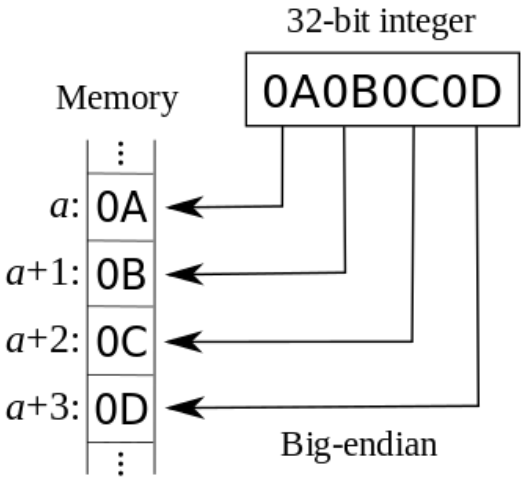
Filename	Attributes	Cluster #
gamedata.exe	.....	32

0	g	a	m	e	d	a	t	a	7
8	e	x	e	…	…	…	…	…	15
16	…	…	…	…	00	00	…	…	23
24	…	…	20	00	00	C4	0F	00	31

Big endian VS little endian

- Endian-ness is about **byte ordering**.
- Little endian:
  - Insignificant (little) byte goes first
    - Stores in lower address
  - FAT uses this
- Big endian:
  - Significant (big) byte goes first





From: wiki

FAT series – directory entry

Bytes	Description
0-0	1st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster number (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster number
28-31	File size.

Filename	Attributes	Cluster #
gamedata.exe	.....	32

0	e	x	p	l	o	r	e	r	7
8	e	x	e	…	…	…	…	…	15
16	…	…	…	…	00	00	…	…	23
24	…	…	20	00	00	C4	0F	00	31

So, what is the largest size of a FAT32 file?

4G - 1 bytes

Bounded by the file size attribute!

Why "– 1"?

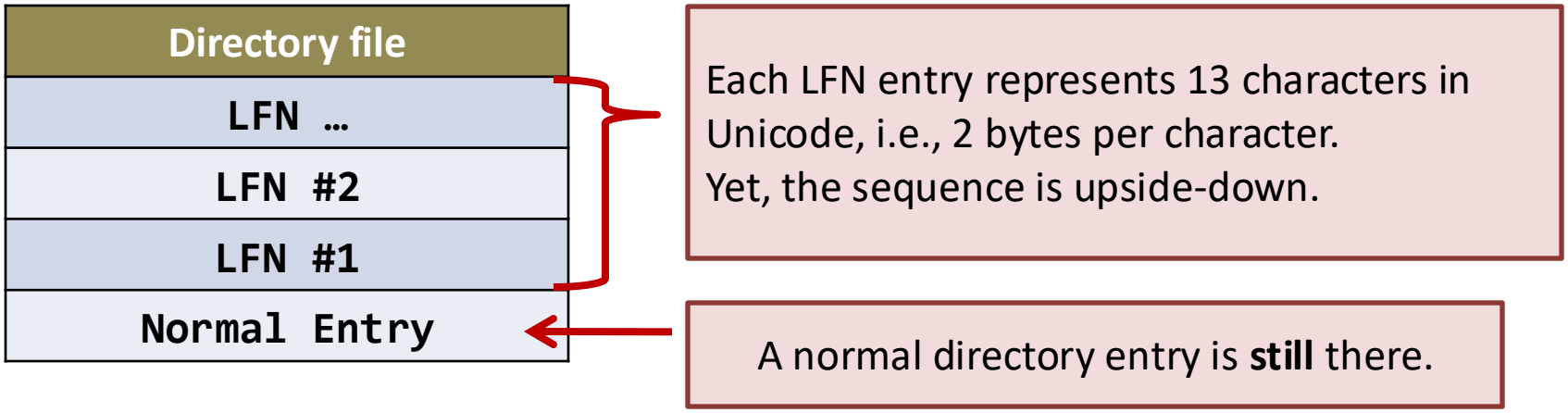
- Imagine 3 bits: 000, 001, …, 110, 111
- Largest number is 111 = 2^3-1
- i.e., we also need to represent "0 bytes"

FAT series – LFN directory entry

Optional

- LFN: Long File Name.
  - In old days, Uncle Bill set the rule that every file should follow the 8+3 naming convention.
  - To support LFN

- **Abuse directory entries to store the remaining characters**!
- Allow to use up to 20 entries for one LFN



FAT series – LFN directory entry

Optional

- Normal directory entry vs Long File Name dirent

Bytes	Description
0-0	1st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster number (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster number.
28-31	File size.

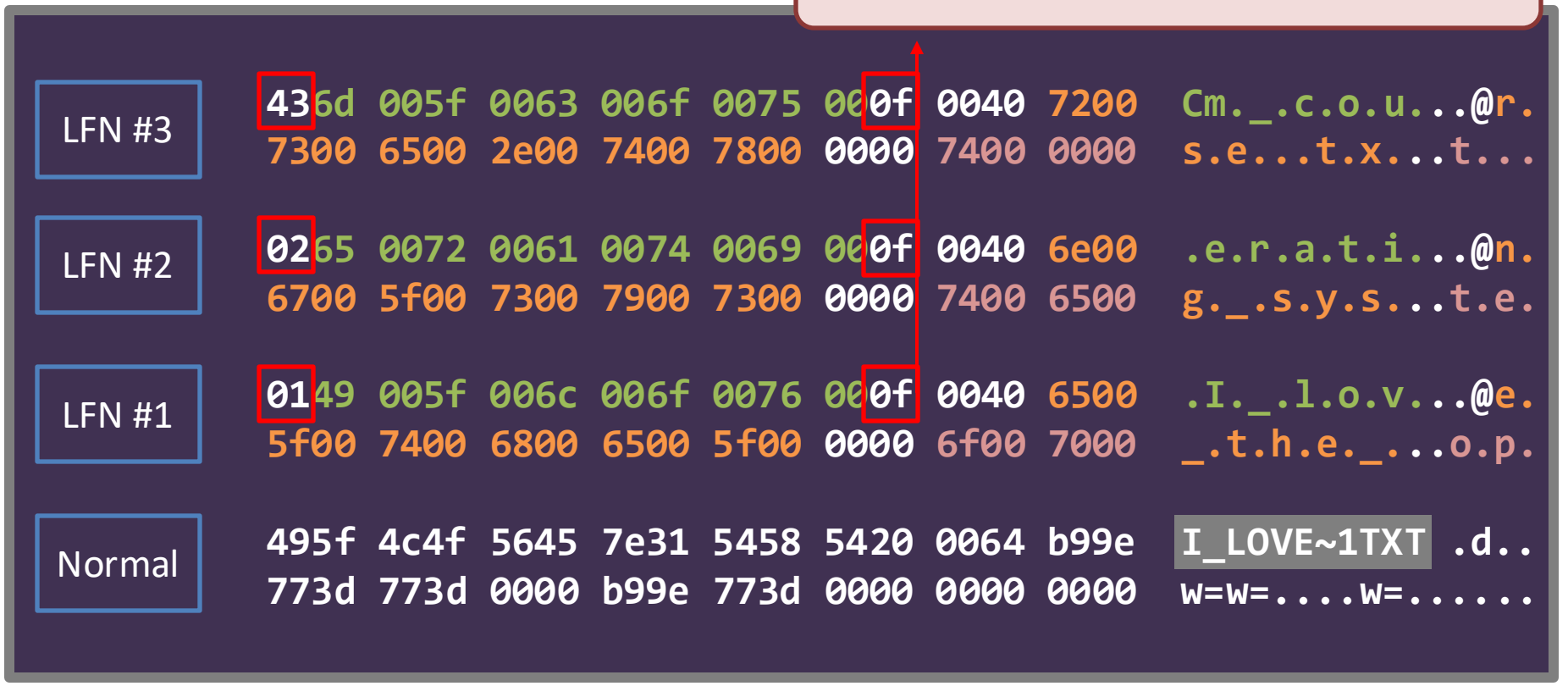
Bytes	Description
0-0	Sequence Number
1-10	File name characters (5 characters in Unicode)
11-11	File attributes - always 0x0F (to indicate it is a LFN)
12-12	Reserved.
13-13	Checksum
14-25	File name characters (6 characters in Unicode)
26-27	Reserved
28-31	File name characters (2 characters in Unicode)

FAT series – LFN directory entry

Optional

- Filename:
  “**I_love_the_operating_system_course.txt**”.

Byte 11 is always 0x0F to indicate that is a LFN.



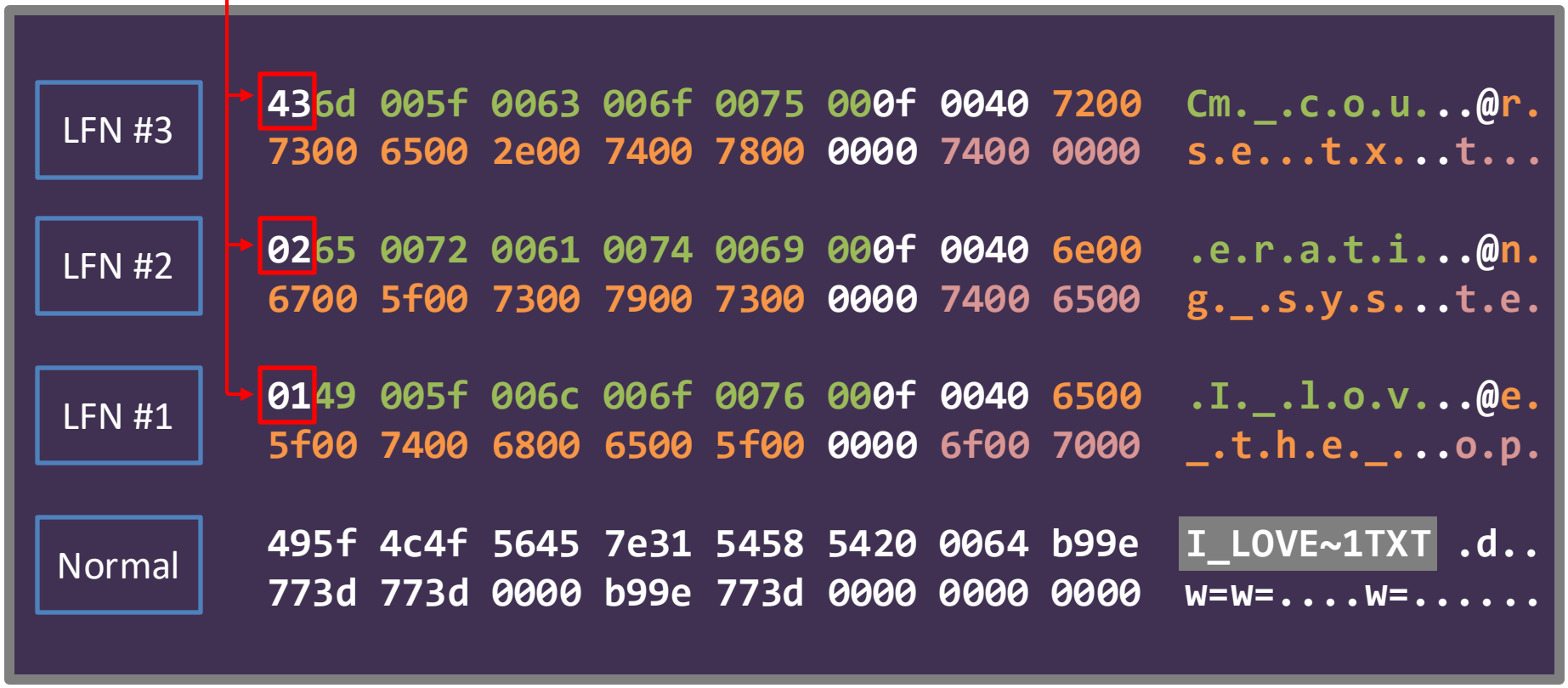
FAT series – 1 directory entry can hold 5+6+2 = 13 characters for file name

Optional

LFN #3: “m_cou” “rse.tx” “t”
LFN #2: “erati” “ng_sys” “te”
LFN #1: “I_lov” “e_the_” “op”
Normal Entry

This is the sequence number, and they are arranged in descending order.

The terminating directory entry has the sequence number **OR-ed with 0x40**.



FAT series – directory entry: a short summary

- A directory entry is an extremely important part of a FAT-like file system.
  - It stores the start cluster number.
  - It stores the **file size**; without the file size, how can you know when you should stop reading witin a cluster?
  - It stores **all file attributes**.

FAT series – reading a file

Task: read "C:\windows\gamedata.exe" sequentially.

FAT1

0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

Filename	Attributes	Cluster #
gamedata.exe	.....	32

Damaged	= 0x0ffffff7
EOF	>= 0x0ffffff8
Unallocated	= 0x0

Step 1. Read the content from Cluster #32.
Note. The **file size** may also help determining if the last cluster is reached.

Step 2. Look for the next cluster and it is Cluster #33.

FAT series – reading a file

Task: read "C:\windows\gamedata.exe" sequentially.

0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

Filename	Attributes	Cluster #
gamedata.exe	.....	32

Step 3. Since the FAT has marked "EOF", we have reached the last cluster of that file.

Note. The file size help determining **how many bytes to read** from the last cluster.

FAT series – writing a file

Task: append data to "C:\windows\gamedata.exe".

0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

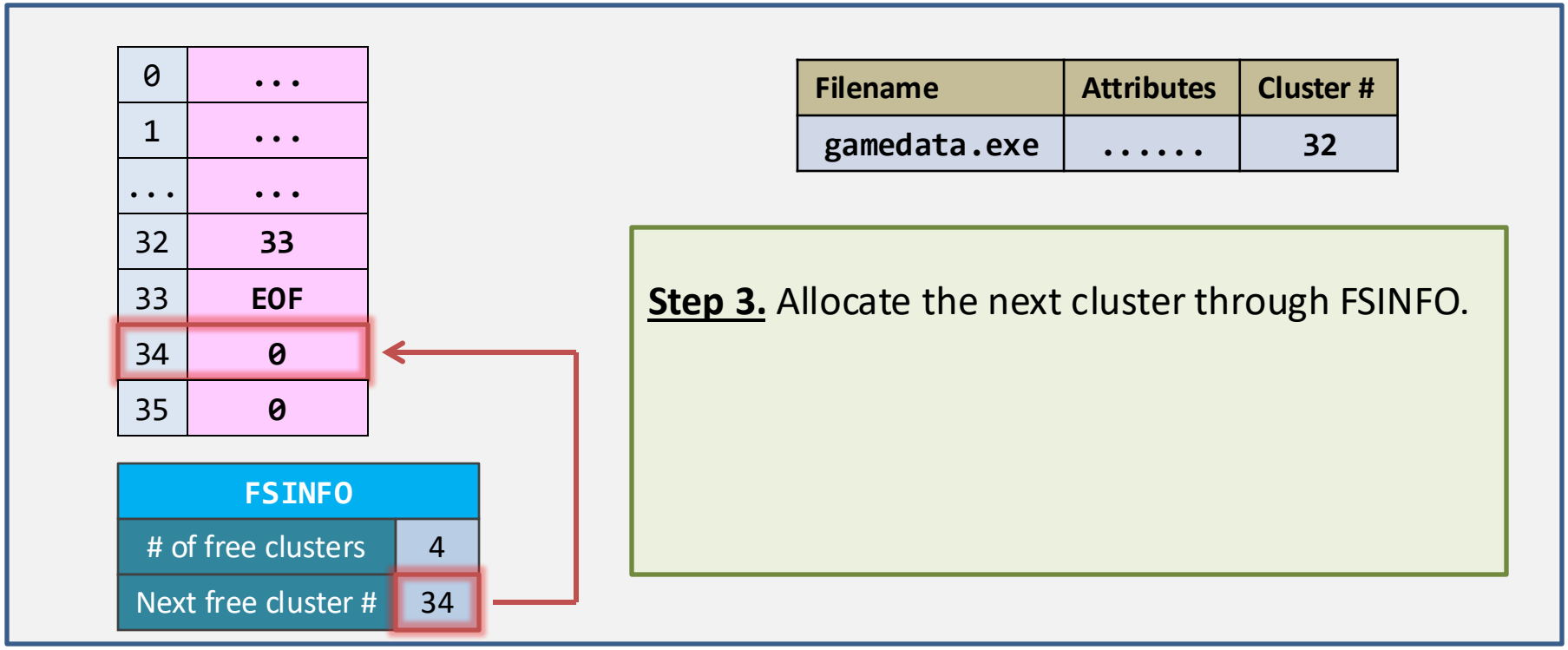
Filename	Attributes	Cluster #
gamedata.exe	.....	32

Step 1. Locate the last cluster.

Step 2. Start writing to the non-full cluster.

FAT series – writing a file

Task: append data to “C:\windows\gamedata.exe”.



FAT series – writing a file

Task: append data to “C:\windows\gamedata.exe”.

0	...
1	...
...	...
32	33
33	34
34	EOF
35	0

Filename	Attributes	Cluster #
gamedata.exe	.....	32

FSINFO	
# of free clusters	3
Next free cluster #	35

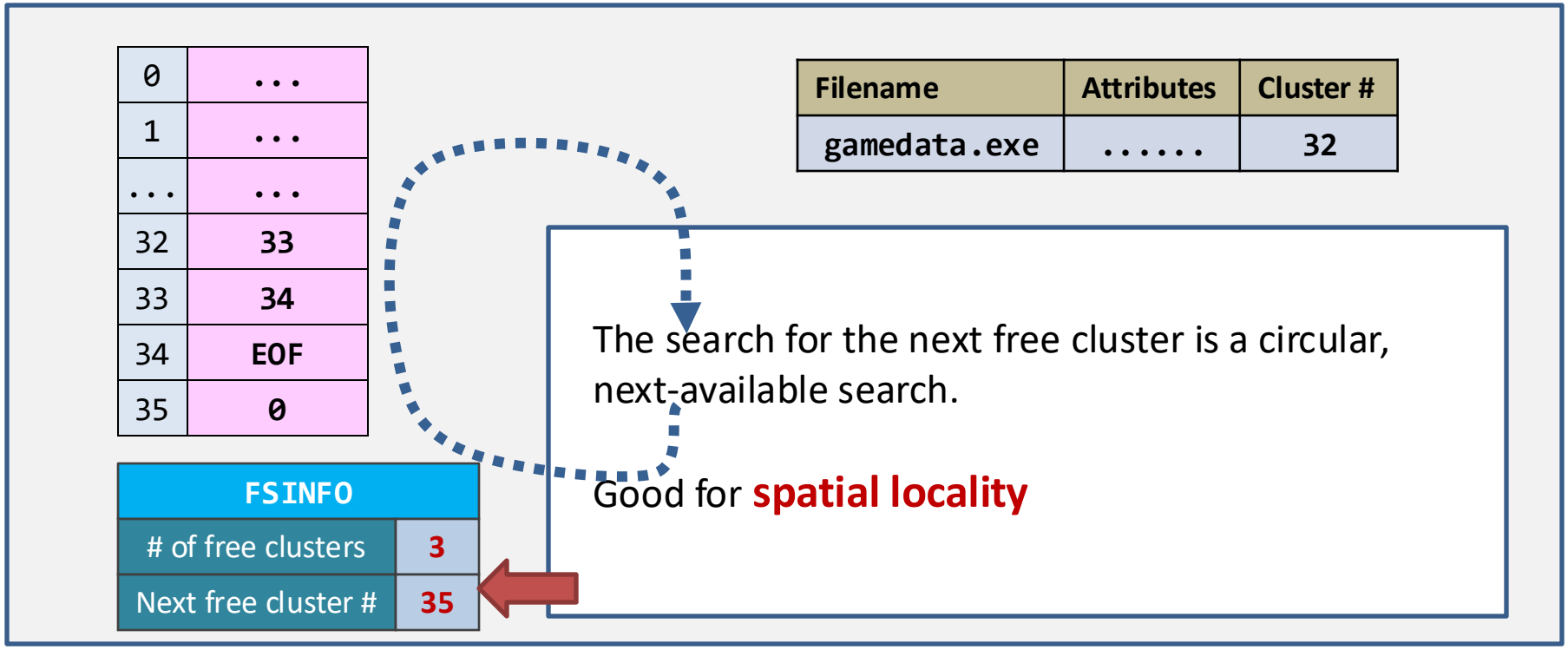
Step 3. Allocate the next cluster through FSINFO.

Step 4. Update the FATs and FSINFO.

Step 5. When write finishes, update the file size.

FAT series – writing a file

Task: append data to “C:\windows\gamedata.exe”.



FAT series – delete a file

Task: delete "C:\windows\gamedata.exe".

0	...
1	...
...	...
32	33
33	34
34	EOF
35	0

0	...
1	...
...	...
32	0
33	0
34	0
35	0

Filename	Attributes	Cluster #
gamedata.exe	.....	32

Step 1. De-allocate all the blocks involved. Update FSINFO and FATs.

FSINFO	
# of free clusters	3
Next free cluster #	35

FSINFO	
# of free clusters	6
Next free cluster #	32

FAT series – delete a file

Task: delete "C:\windows\gamedata.exe".

Directory "windows"		
Filename	**Attributes**	**Cluster #**
.	.....	?
..	.....	?
_amedata.dat	.....	32
notepad.exe	.....	456

Step 2. Change the first byte of the directory entry to _ (0xE5)

That's the end of deletion!

FAT series – really delete a file?

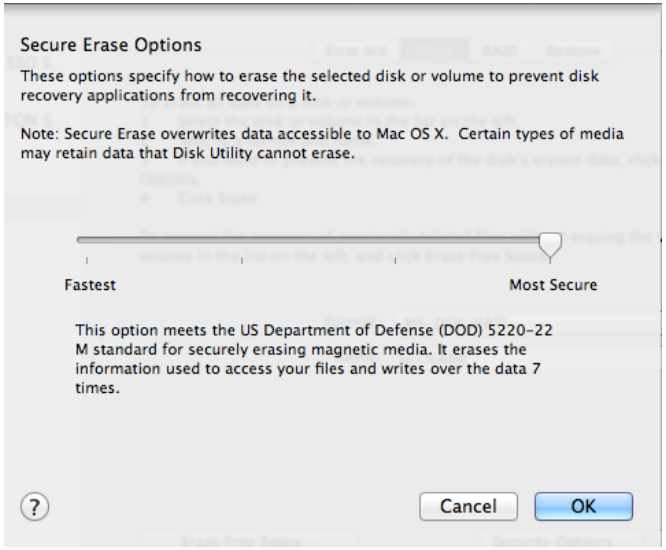
- Can you see that: **the file is not really removed from the FS layout?**
  - Perform a search in all the free space. Then, you will find all deleted file contents.

- "*Deleted data*" persists until the de-allocated clusters **are reused**.
  - This is an issue between performance (during deletion) and security.

- Any way(s) to delete a file **securely**?

FAT series – really delete a file?

Mac OS X Secure Disk Erase

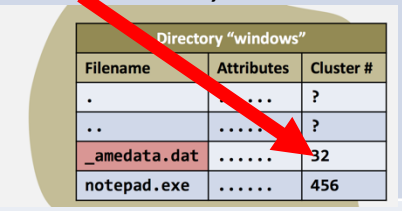


Hard disk Degausser?
http://www.youtube.com/watch?v=5zKjGQAPhUs

Brute Force?
http://www.ohgizmo.com/2009/06/01/manual-hard-drive-destroyer-looks-like-fun/

FAT series – how to recover a deleted file?

- If you're really care about the deleted file, then…
  - **PULL THE POWER PLUG AT ONCE!**
  - Pulling the power plug stops the target clusters from being over-written.

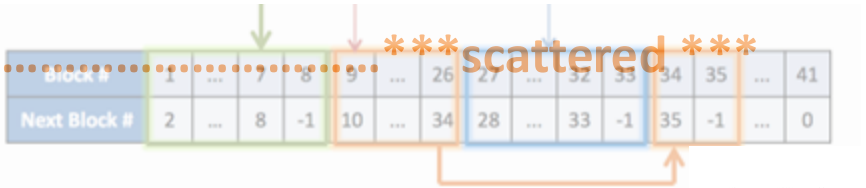
File size is within one block (cluster)	Because **the first cluster address** in the dirent is still **readable**, the recovery is having a very high successful rate. 
File size spans more than 1 block	Because of the next-available search, clusters of a file are likely to be contiguous allocated. This provides a hint in looking for deleted blocks. Can you devise an undelete algorithm for FAT32?

FAT series – conclusion

- Space efficient:
  - 4 bytes overhead (FAT entry) per data cluster.
- Delete:
  - Lazy delete efficient
  - Insecure
    - designed for single-user 30+ years ago
- Deployment:
  - It is still supported everywhere for backward compatibility: CF cards, SD cards, USB drives
- Search:
  - Block addresses of a file may scatter discontinuously
  - To locate the 888-th block of a file?
    - Start from the first FAT entry and follow 888 pointers

Ext File System

- All pointers of a file are located together in an *iNode*
  - Vs. **FAT: pointers of a file are ……** ***scattered***
- Each directory/file has one Inode



Directory inode (128B)

Type	Mode
User ID	Group ID
File size	# blocks
# links	Flags
Timestamps (×3)	
Direct blocks (×12)	
Single indirect	
Double indirect	
Triple indirect	

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all file system types */
    char           d_name[256]; /* filename */
};
```

Directory block (The directory "file")

.	*inode #*
..	*inode #*
passwd	*inode #*
fstab	*inode #*
…	…

Indirect block

Direct blocks (×512)

Block #s of more directory blocks

Block # of block with 512 double indirect entries

Block # of block with 512 single indirect

Inode Array

File inode (128B)

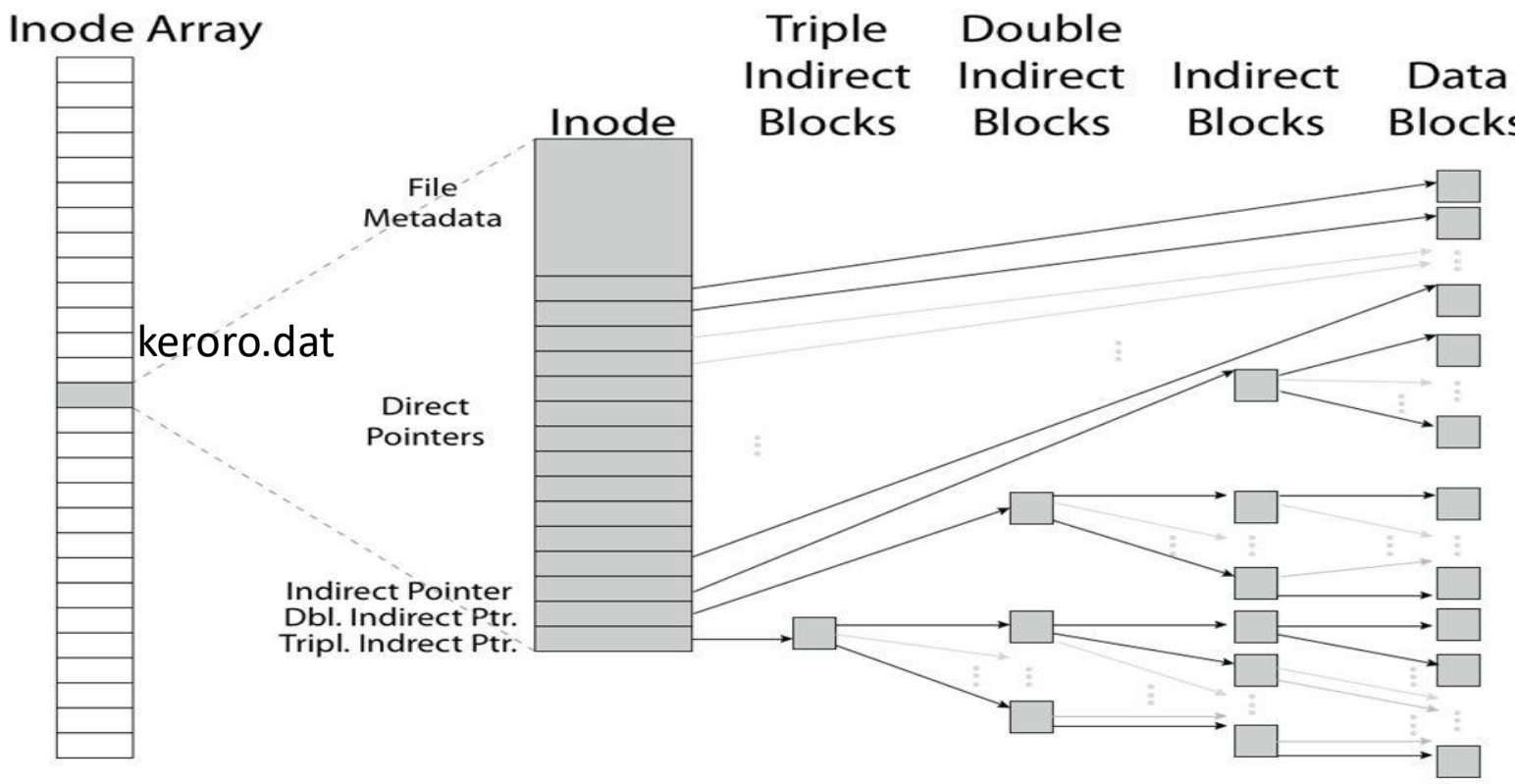
Type	Mode
User ID	Group ID
File size	# blocks
# links	Flags
Timestamps (×3)	
Direct blocks (×12)	
Single indirect	
Double indirect	
Triple indirect	

File data block

Data

iNode

- Inode Table (a.k.a. Inode Array) is an array of Inodes
- Pointers are unbalanced tree-based



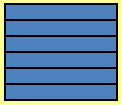
Index-node – file size

Reminder: Max file size != FS size

Number of direct blocks	12
Number of indirect blocks	1
Number of double indirect blocks	1
Number of triple indirect blocks	1
Block size	2^x bytes
Address length	4 bytes

$$12 \times 2^x +$$
$$1 \times 2^x/4 \times 2^x +$$
$$1 \times (2^x/4)^2 \times 2^x +$$
$$1 \times (2^x/4)^3 \times 2^x$$

File size = number of data blocks * Block size



contains "**2^x / 4**" addresses

Block size 2^x	Max size
1024 bytes = 2^{10}	approx. 16 GB
4096 bytes = 2^{12}	approx. 4 TB

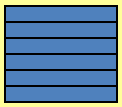
Index-node – file size

Number of direct blocks	12
Number of indirect blocks	1
Number of double indirect blocks	1
Number of triple indirect blocks	1
Block size	2^x bytes
Address length	4 bytes

File size = number of data blocks x 2^x

$$12 \times 2^x + 2^{2x-2} + 2^{3x-4} + 2^{4x-6}$$

The dominating factor.



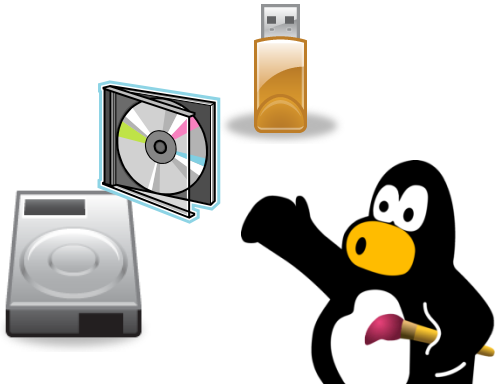
contains "2^x / 4" addresses

Block size 2^x	Max size
1024 bytes = 2^{10}	approx. 16 GB
4096 bytes = 2^{12}	approx. 4 TB

Reminder: Max file size != FS size

Ext 2/3/4

- Disk layout
- Directory
- Hard and Soft Links
- Consistency



File System Ext

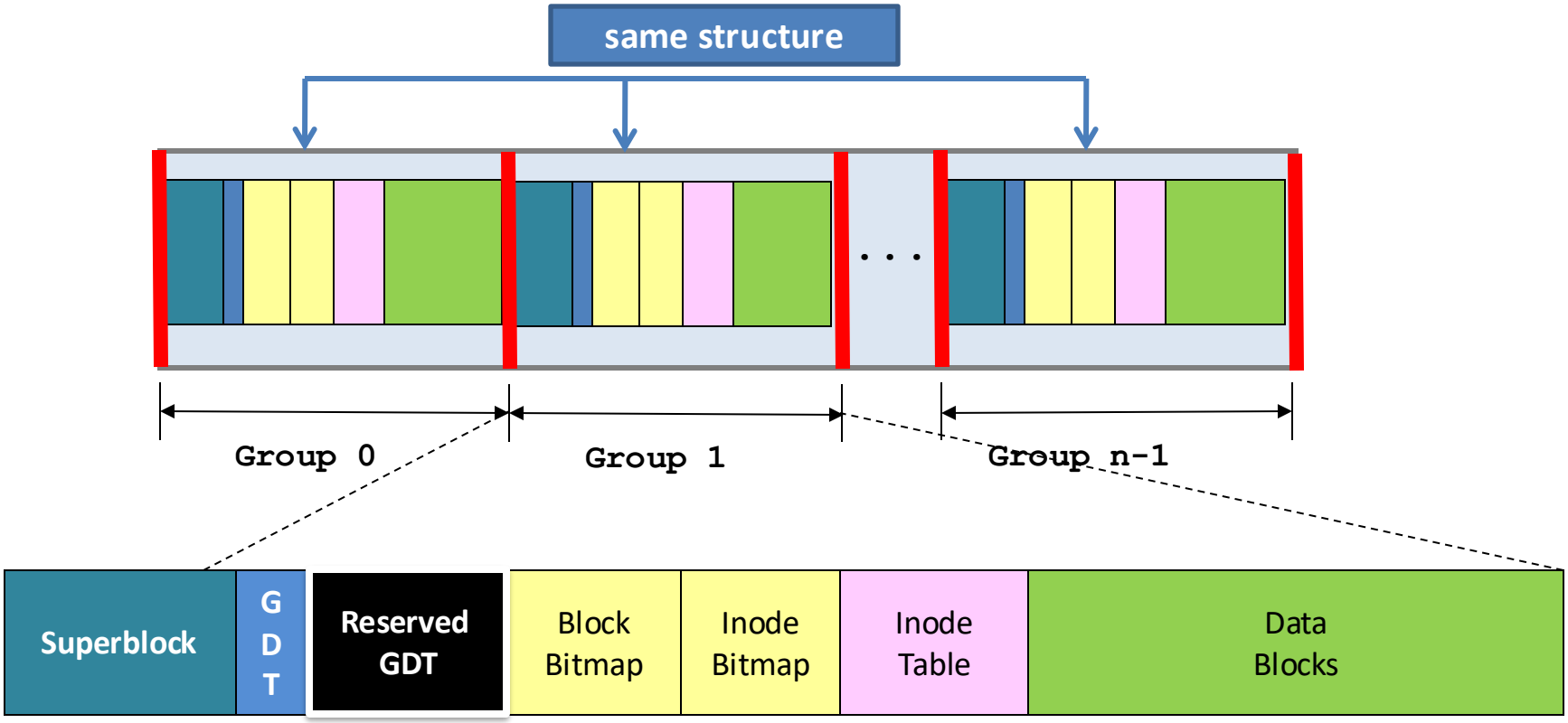
- The latest default FS for Linux distribution is the **Fourth Extended File System**, **Ext4** for short.

- For Ext2 & Ext3:
  - Block size: 1,024, 2,048, or 4,096 bytes.
  - Block address size: 4 bytes => # of block addresses = 2^{32}

$2^x \times 2^{32} = 2^{32+x}$			
Block size	$2^x = 1024$	$2^x = 2048$	$2^x = 4096$
Max file size	4 TB	8 TB	16 TB

Ext2/3 – Block groups

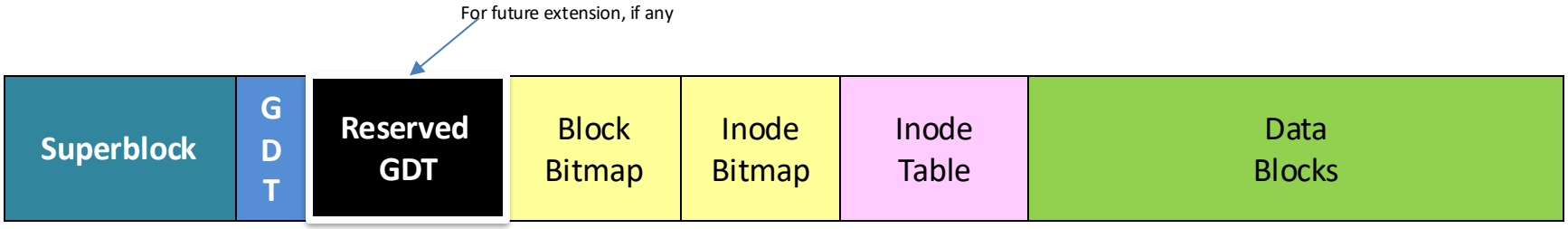
- The file system is divided into **block groups** and every block group has the **same structure**



Ext2/3 – FS layout

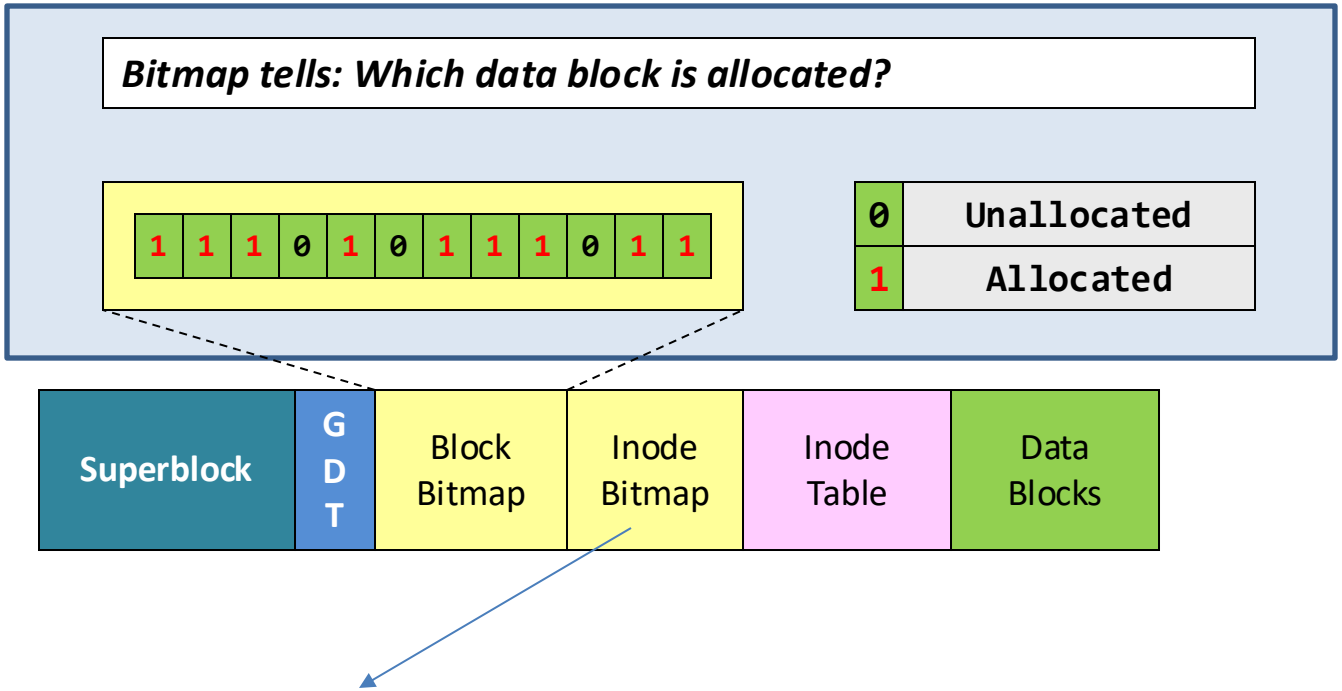
- Layout of one block group is as follows:

Superblock	Stores FS specific data. E.g., the total number of blocks, etc.
GDT – Group Descriptor Table	It stores: - The locations of the **block bitmap**, the **inode bitmap**, and the **inode table**. - Free block count, free inode count, etc…
Block Bitmap	A bit string that represents if a block is allocated or not.
Inode Bitmap	A bit string that represents if an inode (index-node) is allocated or not.
Inode Table	An array of inodes ordered by the inode #.
Data Blocks	An array of blocks that stored files.



https://www.ibm.com/developerworks/cn/linux/l-cn-filesrc5/

Ext2/3 – Block Bitmap & Inode Bitmap

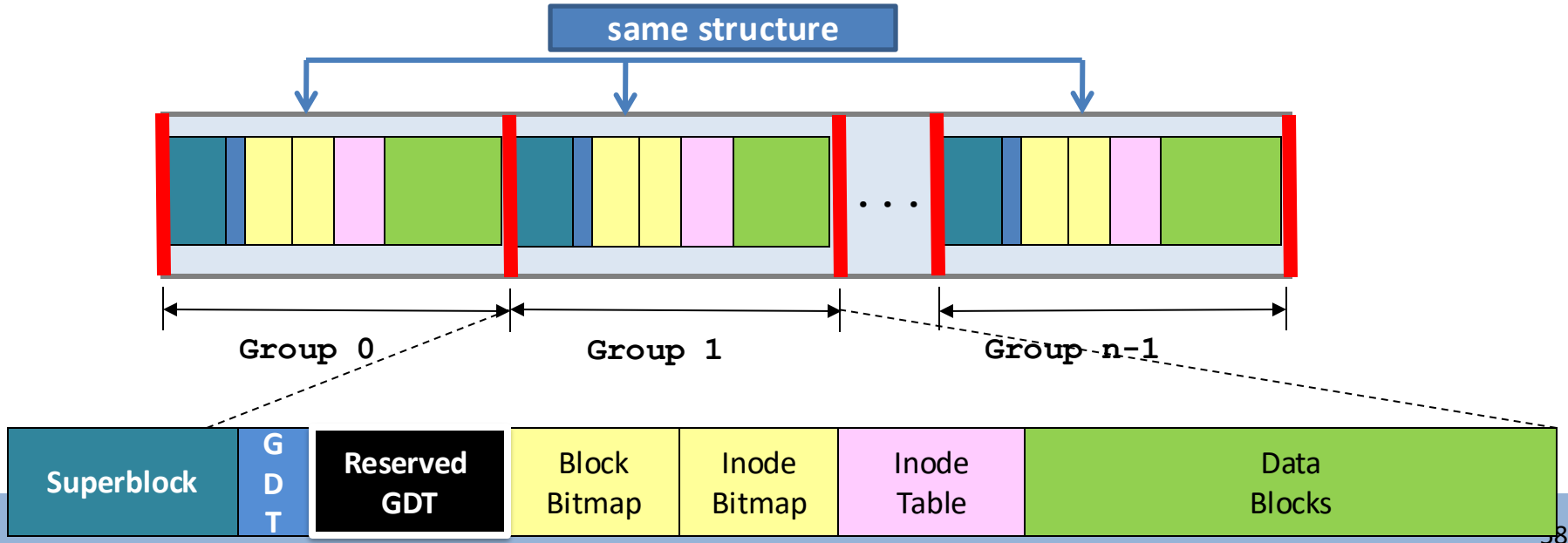


- Inode Bitmap
  - A bit string that represents if an inode (index-node) is allocated or not

  ➔ implies that the **number of files in the file system is fixed**!

Ext2/3 – Block groups

- Why having groups?
- For **(1) performance** and **(2) reliability**
  - (1) Performance: spatial locality.
    - Group inodes and data blocks of related files together
  - (2) Reliability: superblock and GDT are **replicated** in some block groups

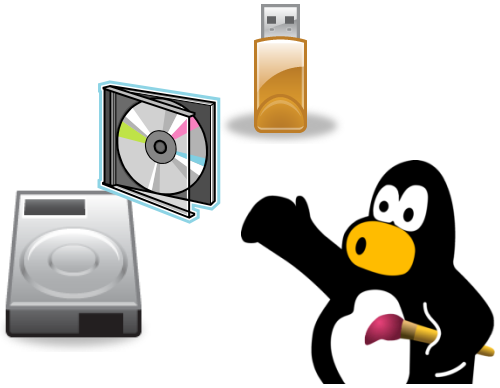


Ext 2/3

- Disk layout;

- Directory;

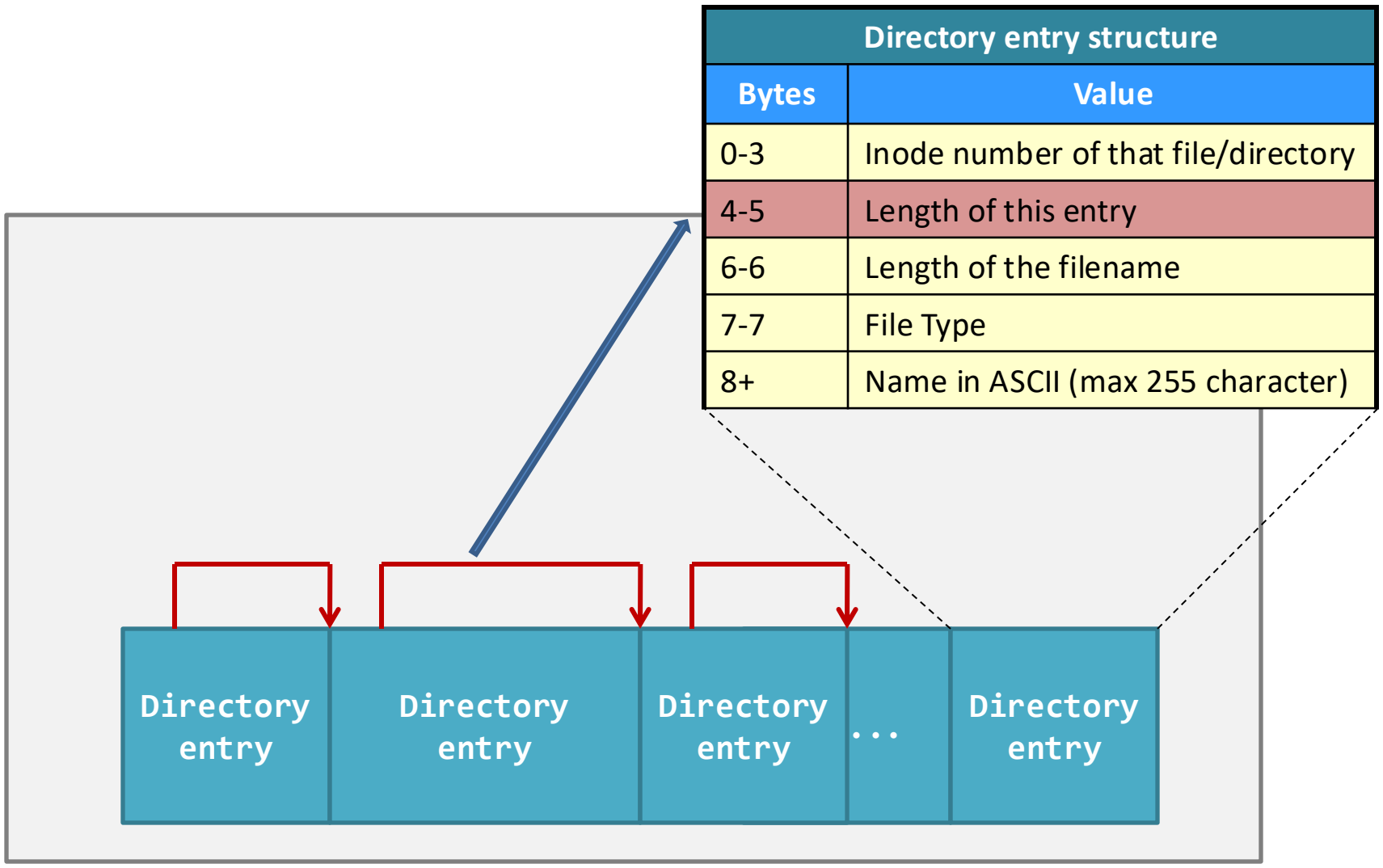
- Hard and Soft Links.



Ext2/3 – inode structure (for 1 file)

Inode Structure (128 bytes long)	
Bytes	**Value**
0-1	File type and permission
2-3	User ID
4-7	Lower 32 bits of file sizes in bytes
8-23	Time information
24-25	Group ID
26-27	Link count (will discuss later)
…	…
40-87	12 direct data block pointers
88-91	Single indirect block pointer
92-95	Double indirect block pointer
96-99	Triple Indirect block pointer
…	…
108-111	Upper 32 bits of file sizes in bytes

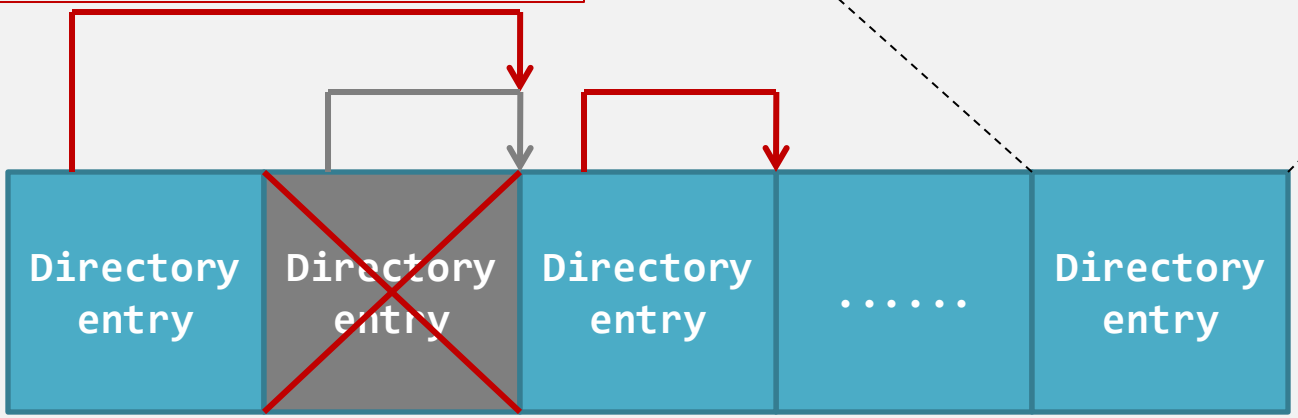
Ext2/3 –directory entry in a directory block



Ext2/3 – File Deletion

Directory entry structure	
Bytes	**Value**
0-3	Inode number of that file/directory
4-5	Length of this entry
6-6	Length of the filename
7-7	File Type
8+	Name in ASCII (max 255 character)

Ext 2: File deletion is just an update of the “entry length” of the previous entry.
Ext 3/4: the inode’s data block pointers are zeroed out

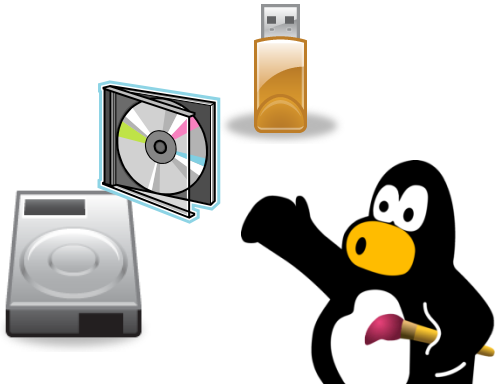


Ext 2/3

- Disk layout;

- Directory;

- **Hard and Soft Links.**



Ext2/3 – link file: what is a hard link

- A hard link is a **directory entry** pointing to the inode of an existing file.

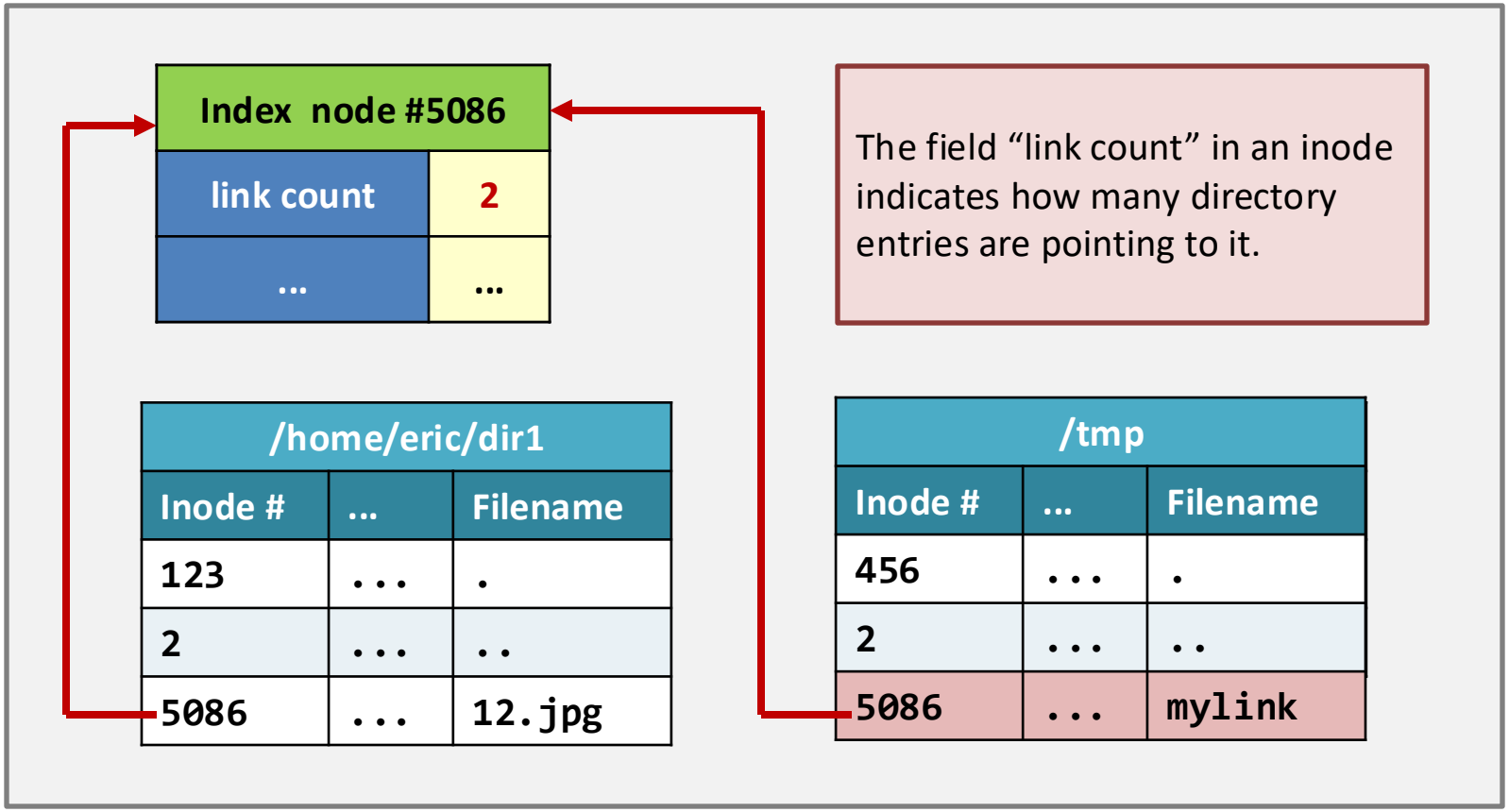
```
# ln /home/eric/dir1/12.jpg /tmp/mylink
```

/home/eric/dir1		
Inode #	...	Filename
123	...	.
2	...	..
5086	...	12.jpg

/tmp		
Inode #	...	Filename
456	...	.
2	...	..
5086	...	mylink

Ext2/3 – link file: what is a hard link

- That **file can accessed through two different pathnames**.

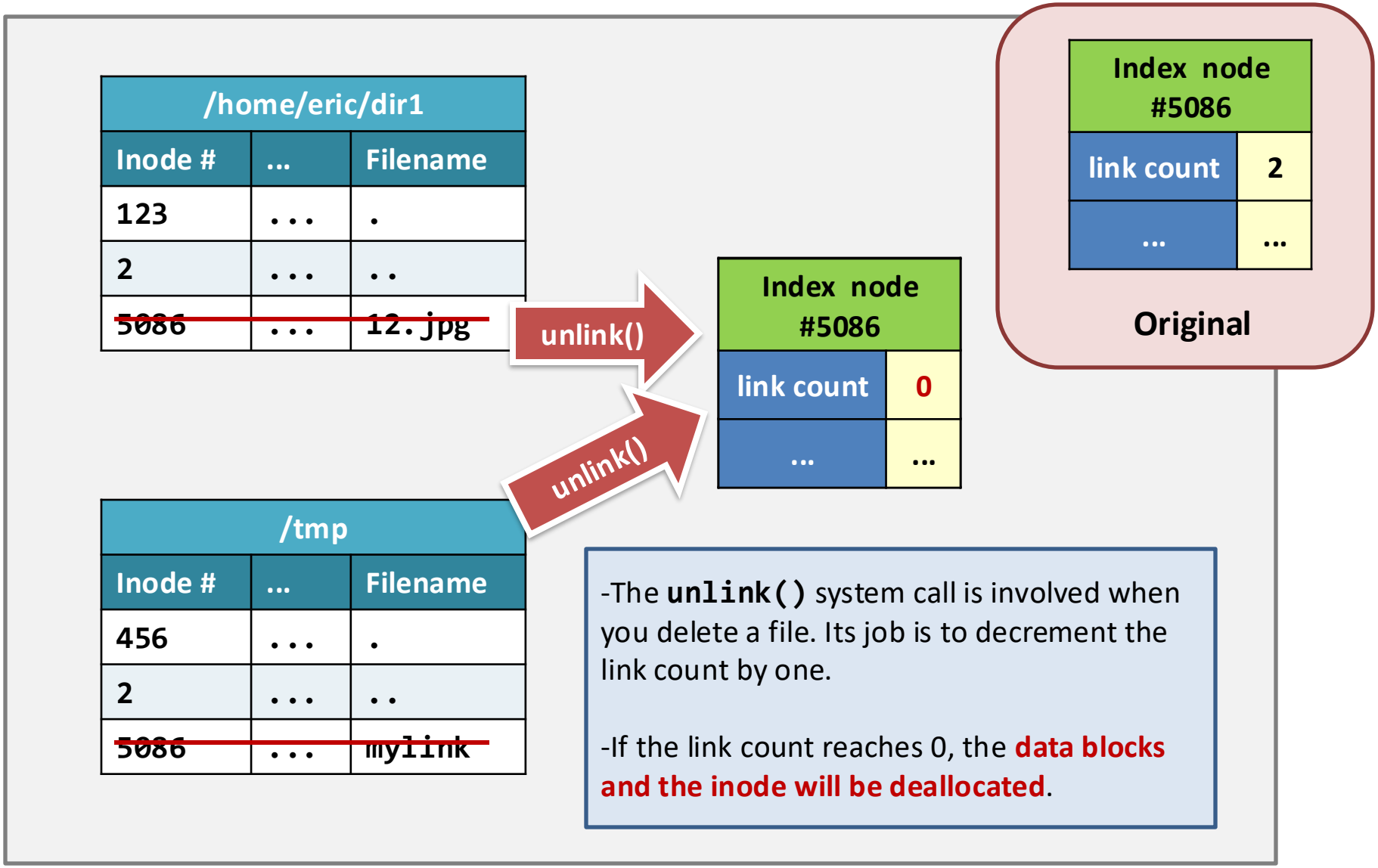


Ext2/3 – link file: examples on hard link

- Let's look at the link count of the root directory.
  - **20 sub-directories**: have a link "..";
  - **Root directory**: "." and ".." pointing to itself;
  - 20 + 2 = 22.

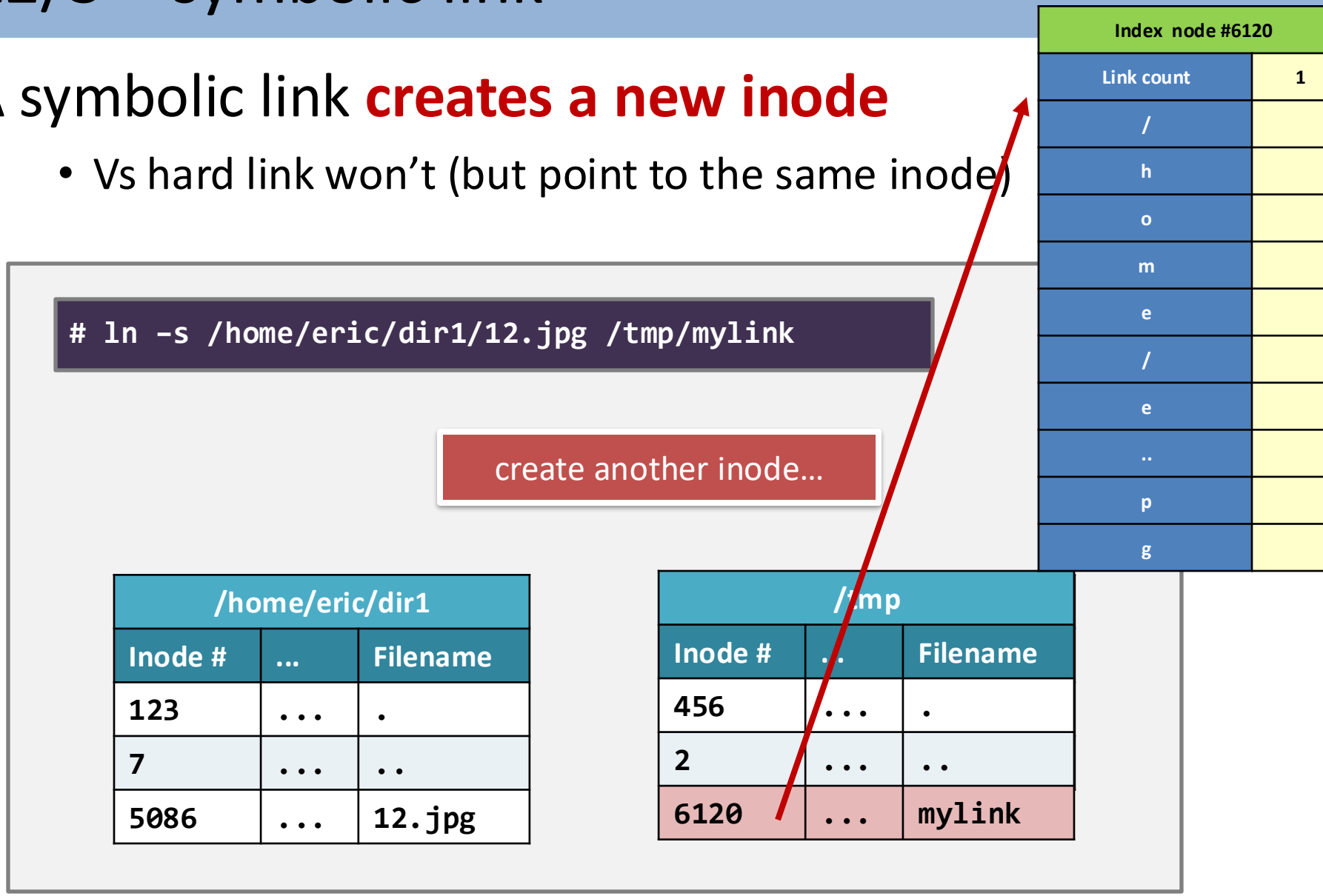
```
# ls -F /
bin/    home/            media/  rules.log  tmp/
boot/   initrd.img@      mnt/    sbin/      usr/
cdrom/  initrd.img.old@  opt/    selinux/   var/
dev/    lib/             proc/   srv/       vmlinuz@
etc/    lost+found/      root/   sys/       vmlinuz.old@
# stat /
  File: `/'
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 806h/2054d      Inode: 2           Links: 22
......
$ _
```

Ext2/3 – removing file and link count



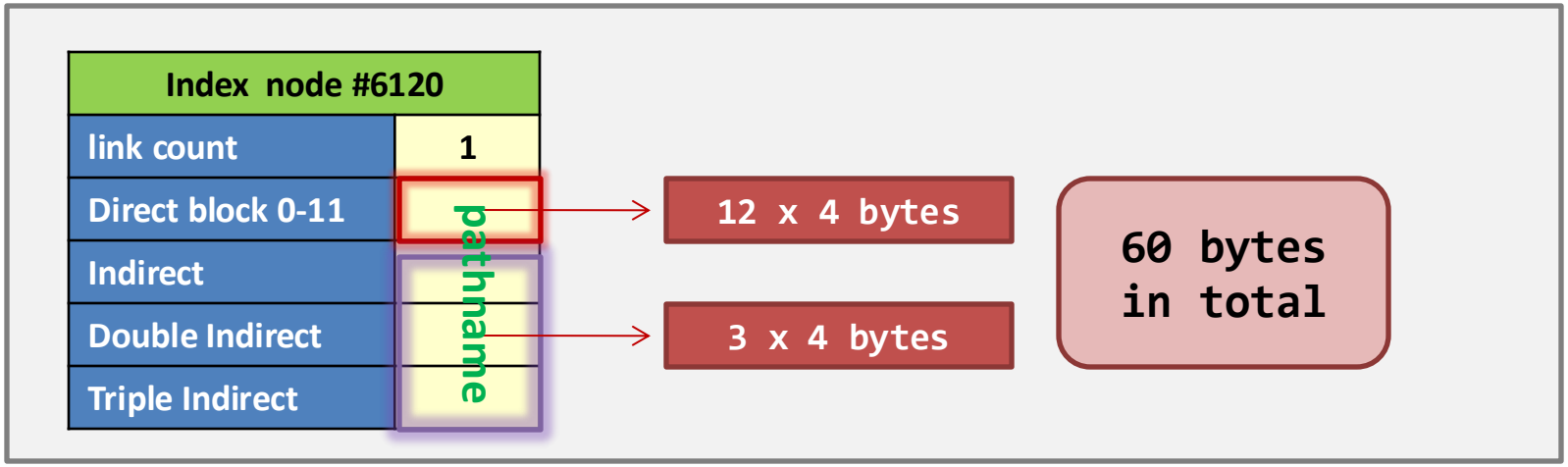
Ext2/3 – symbolic link

- A symbolic link **creates a new inode**
  - Vs hard link won't (but point to the same inode)



Ext2/3 – symbolic link

- Symbolic link is pointing to a newly created Inode whose target's **pathname** are stored by abusing the space originally designed for **12 direct block and the 3 indirect block pointers** if the pathname is shorter than 60 characters.
    - Use back a normal inode + **one direct data block** to hold the long pathname otherwise

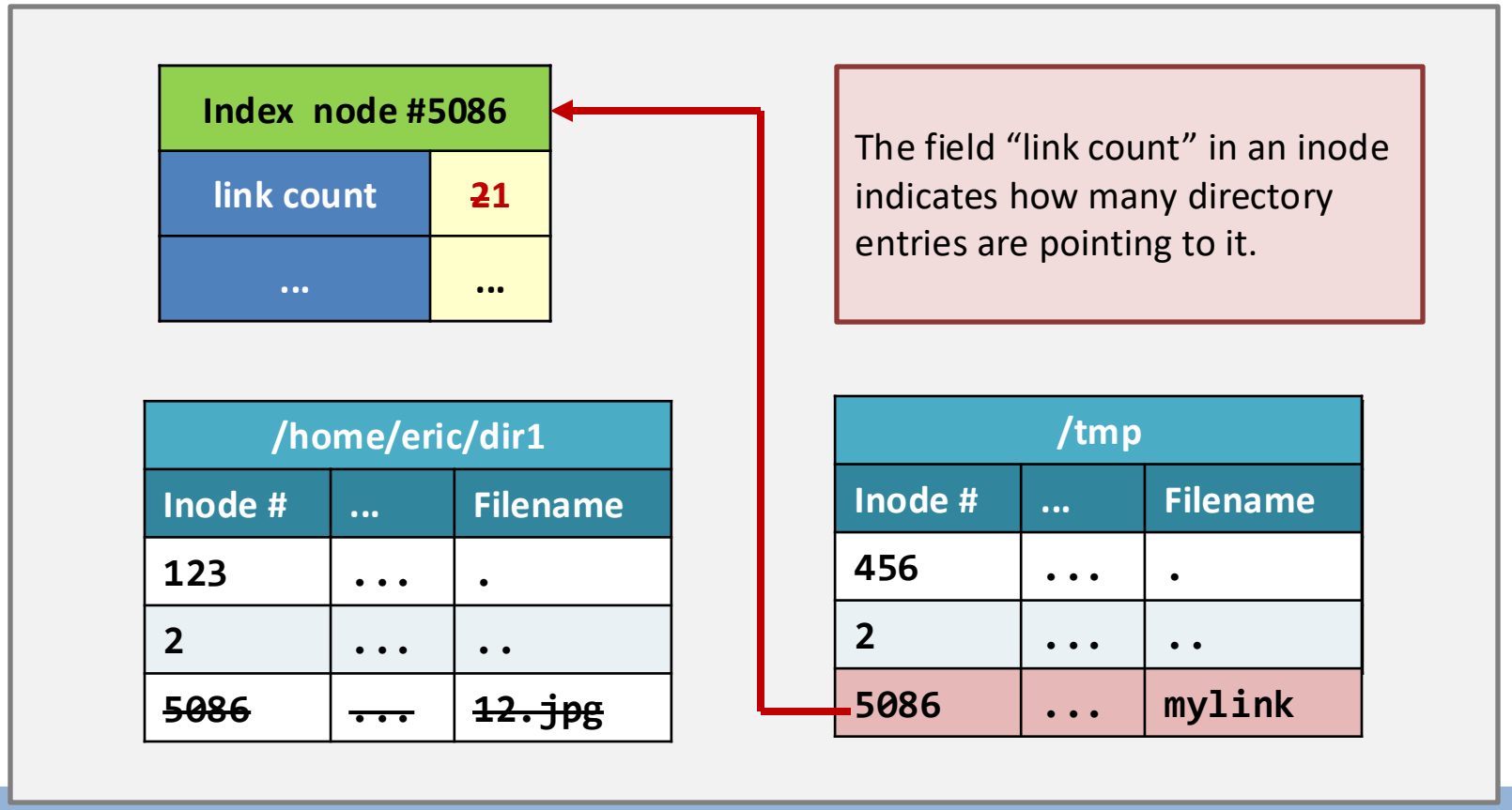


Symbolic link vs Hard link

- Hard link:

  rm /home/eric/dir1/12.jpg [removing the original]

  – Still accessible through /tmp/mylink

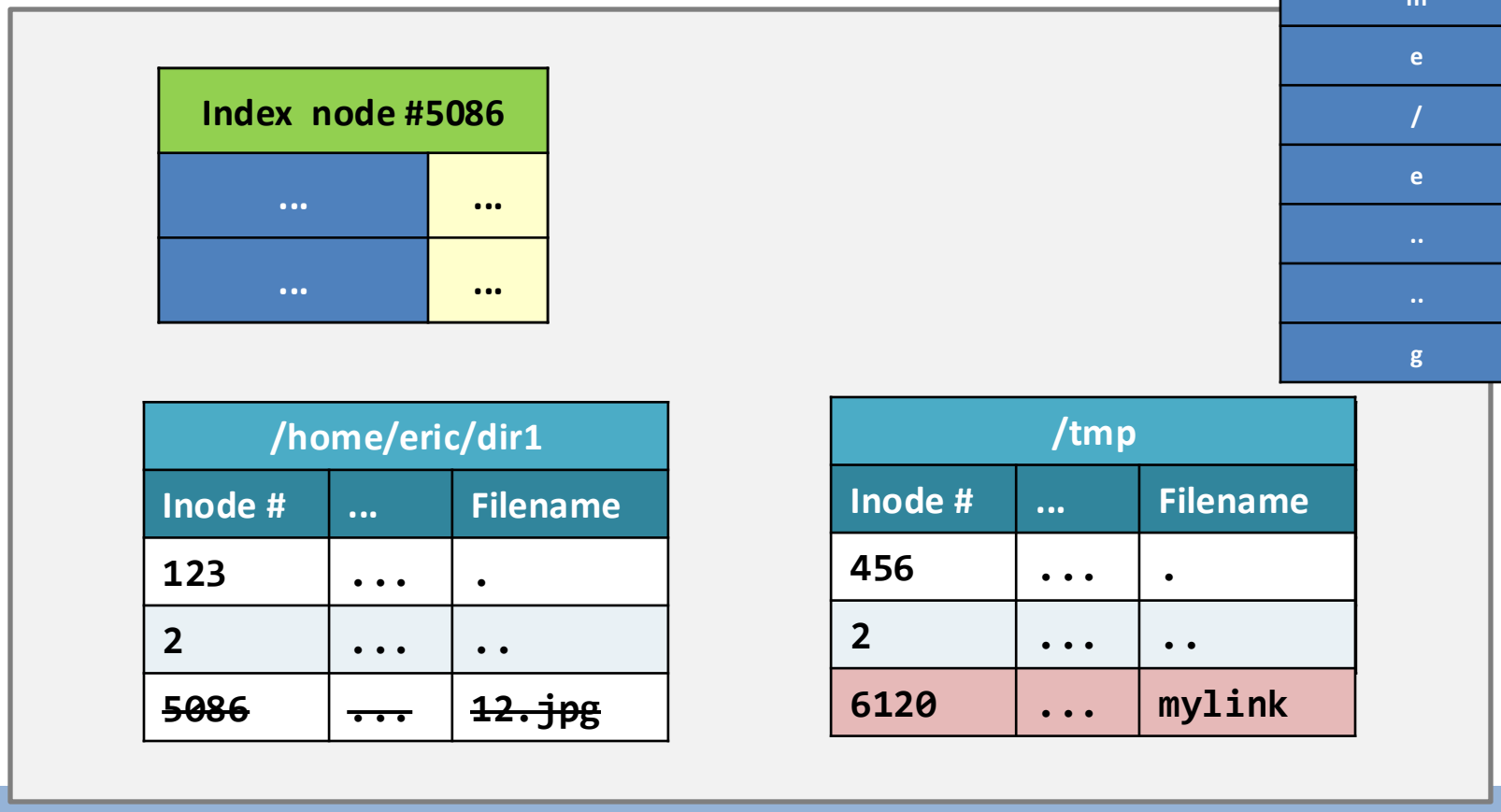


Symbolic link vs Hard link

- Symbolic link:

  rm /home/eric/dir1/12.jpg
  [removing the original]

Does this pathname still exist?



Index node #6120	
Link count	1
/	
h	
o	
m	
e	
/	
e	
..	
..	
g	

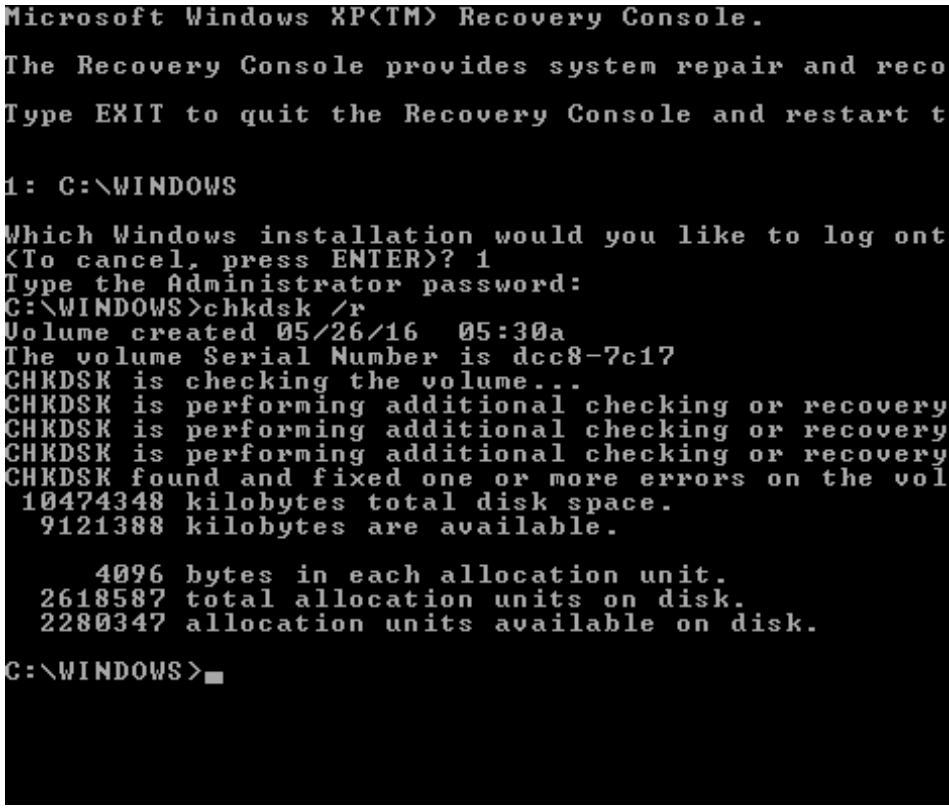
Index node #5086	
…	…
…	…

/home/eric/dir1		
Inode #	…	Filename
123	. . .	.
2	. . .	. .
~~5086~~	~~. . .~~	~~12.jpg~~

/tmp		
Inode #	…	Filename
456	. . .	.
2	. . .	. .
6120	. . .	mylink

File system consistency

- It is about how to **detect** and how to **recover** **inconsistency in a file system**.
  - But, why does inconsistency exist?
    - E.g., Deleting a file involves three steps:
      1. Removing its directory entry.
      2. Releasing the Inode entry.
      3. Returning all used disk blocks to the pool of free disk blocks (updating GDT).



Power failure, Pressing reset button accidentally, etc.

If power-down between Steps 1 & 2 → Orphan Inode
If power-down between Steps 2 & 3 → Leak Storage

Journal

- The **file system journal** is the current, state-of-the-art practice (e.g., ext3/4)
- = To-do list
- During recovery
  - Would either continue to do your work or rollback your work