3150 - Operating Systems

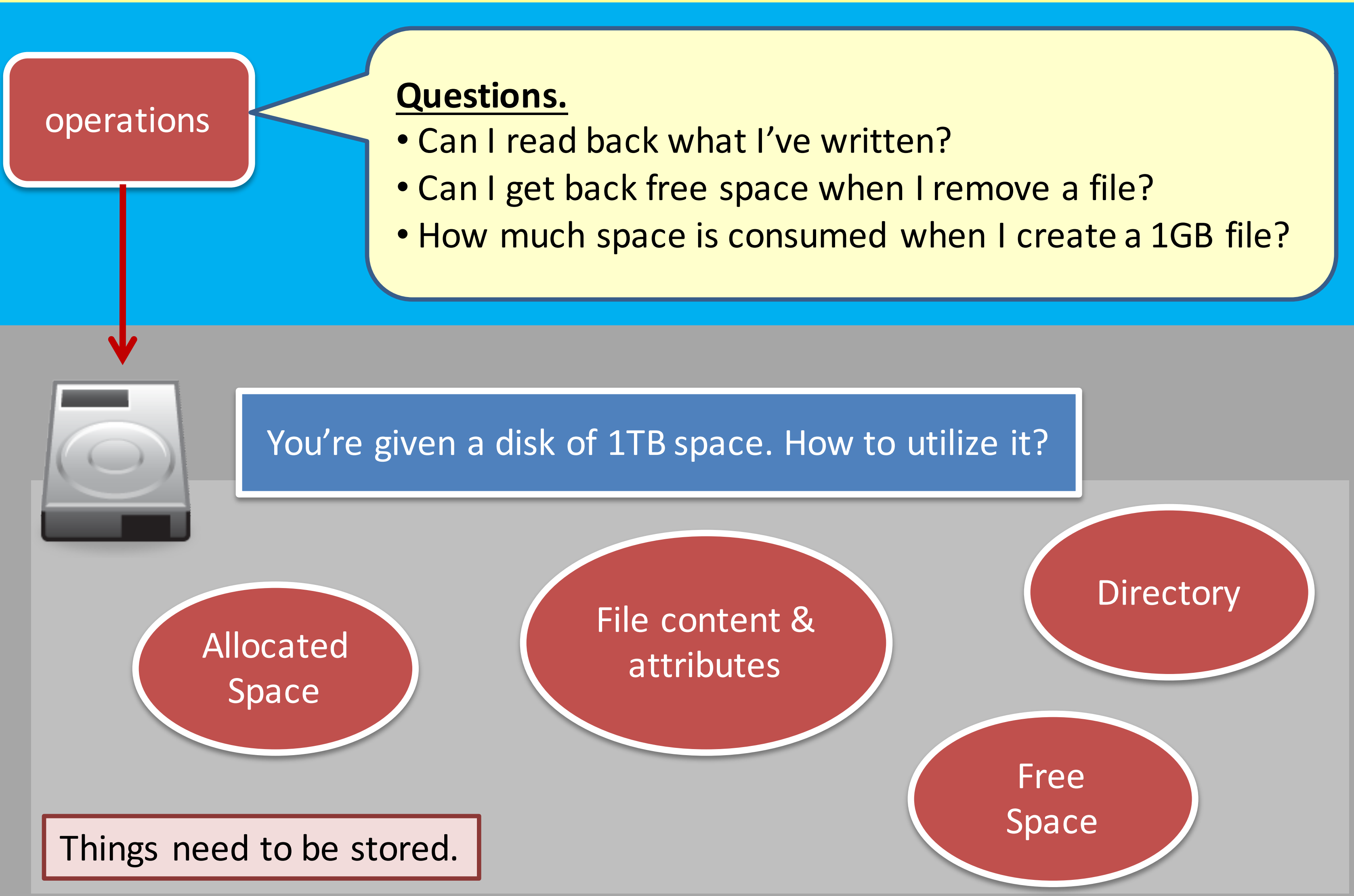
Dr. WONG Tsz Yeung

Due to the lack of time, p.70-88 will only be covered in ESTR Class.

Chapter 3, part 2 - File System Layout

- Here comes the static part of a file system…

Outline



operations

Questions.

- Can I read back what I've written?
- Can I get back free space when I remove a file?
- How much space is consumed when I create a 1GB file?

You're given a disk of 1TB space. How to utilize it?

Allocated Space

File content & attributes

Directory

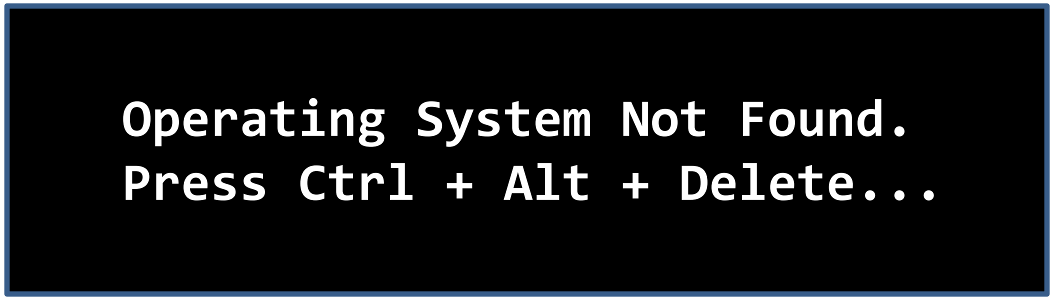
Free Space

Things need to be stored.

Basics…

EXTRA

- Do you remember…what will you do when you buy a new hard disk?
  - – Mount it on to the chassis;
  - – Connect the cables;
  - – Turn on the power;
  - – Oops….

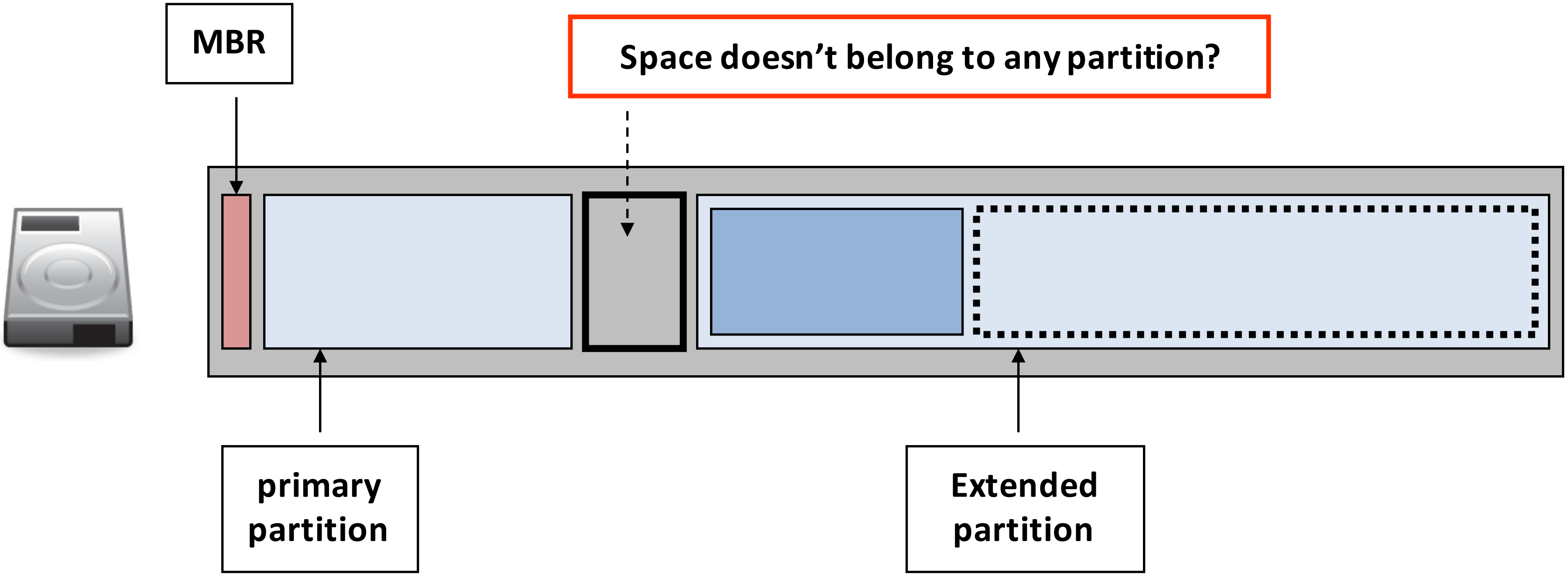


```
Operating System Not Found.
Press Ctrl + Alt + Delete...
```

Basics…disk partitions

EXTRA

- Disk partition?
  - A file system must be stored in a partition.
  - An operating system must be hosted in a partition.



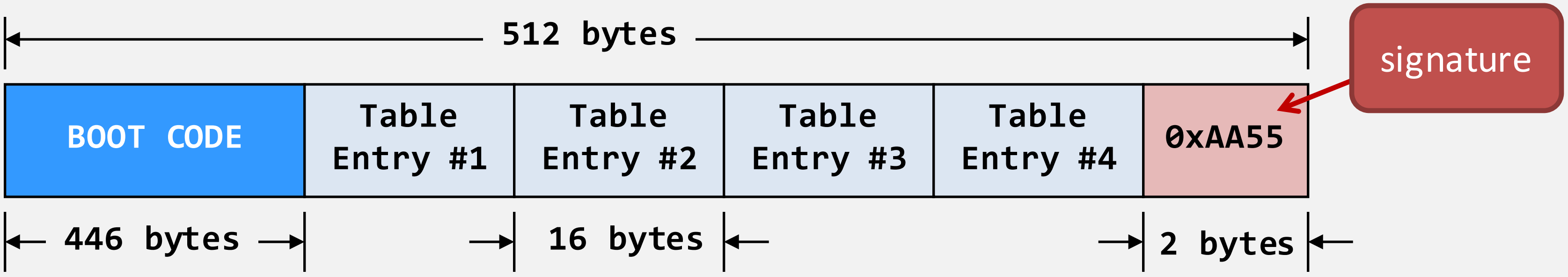
Basics…disk partitions

EXTRA

- Why do we need to have partitions?
  - **Performance**
    - A smaller file system is more efficient!
      - We will look into this problem during the study of FAT32.
  - **Multi-booting**
    - You can have a Windows XP + Linux + Mac installed on a single hard disk (not using VMware).
  - **Data management**
    - You can have one logical drive to store movies, one logical drive to store the OS-related files, etc.

Basics...master boot record...

EXTRA



512 bytes

BOOT CODE | Table Entry #1 | Table Entry #2 | Table Entry #3 | Table Entry #4 | 0xAA55

signature

446 bytes

16 bytes

2 bytes

The range of a partition of is described by the: (offset, length) duple.

Partition Table Entry	
Bytes	**Description**
0-0	Bootable flag; 0x80 means bootable.
1-3	Starting CHS address [not covered]
4-4	Partition type http://www.datarecovery.com/hexcodes.asp
5-7	Ending CHS address [not covered]
8-11	Starting LBA address (measured in # of sectors)
12-15	Sizes in sectors

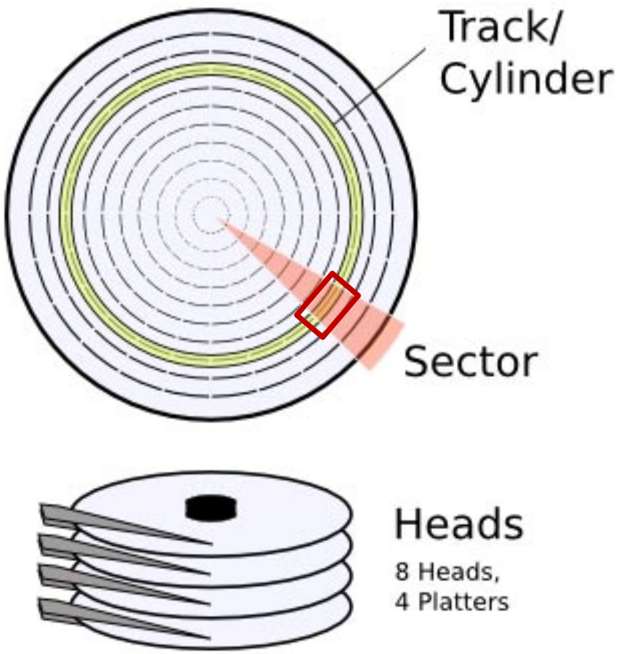
```
[examples@3150] cat read_part.c
```

Basics…what is a sector?

EXTRA

- Have you ever heard of **CHS**: **cylinder-head-sector**?
  - The # of heads defines the **# of platters**, i.e., ***how many pizzas*** are there?
  - The # of cylinders is the circular zones that the head can read/write while
    - (1) the platter is spinning and
    - (2) the head remains stationary.
  - A sector means ***a slice from a cylinder.***

source: wikipedia



Some good movies
http://www.youtube.com/watch?v=9eMWG3fwiEU
http://www.youtube.com/watch?v=L0nbo1VOF4M

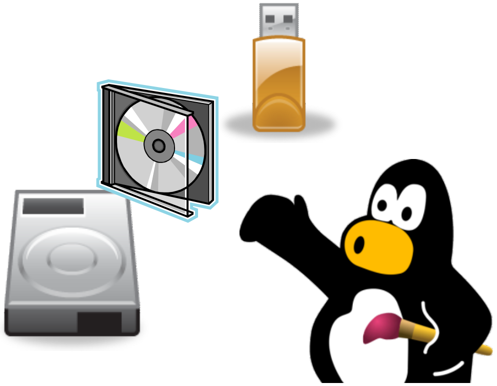
Basics…formatting?

- Do you know what is the meaning of "**formatting a disk**"?
  - **Create and initialize a file system**.
  - In Windows, we have "**`format.exe`**".
  - In Linux, we have "**`mkfs.ext`**2", "**`mkfs.ext3`**", etc.

- In this part, we are going to teach you how a file system utilizes the limited space in a partition.

Different Layouts

- Contiguous allocation;

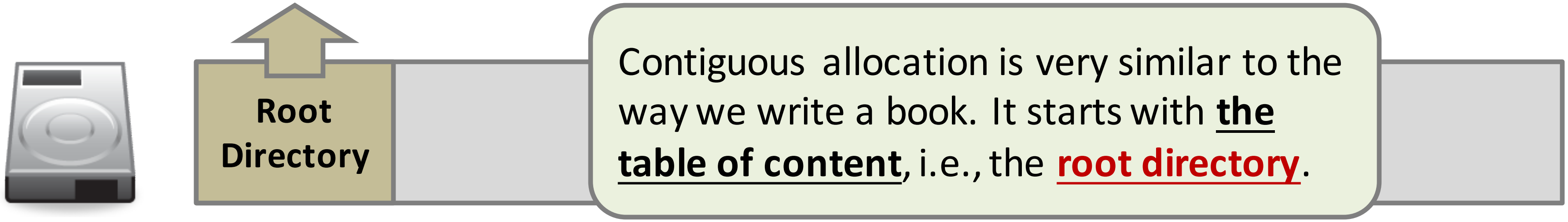


Contiguous allocation – basics

```
Table of content

Chapter 1   ............ p.1
Chapter 2   ............ p.2
Chapter 3   ............ p.10
```

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

File attributes can be found in the root directory!



Contiguous allocation is very similar to the way we write a book. It starts with **the table of content**, i.e., the **root directory**.

Requirements	
Allocated Space Mgt	
Free Space Mgt	
File Content Allocation	
File Attributes	✓
Directory	✓
File growth and shrink	
File creation	
File deletion	

Contiguous allocation – basics

You can locate files easily.

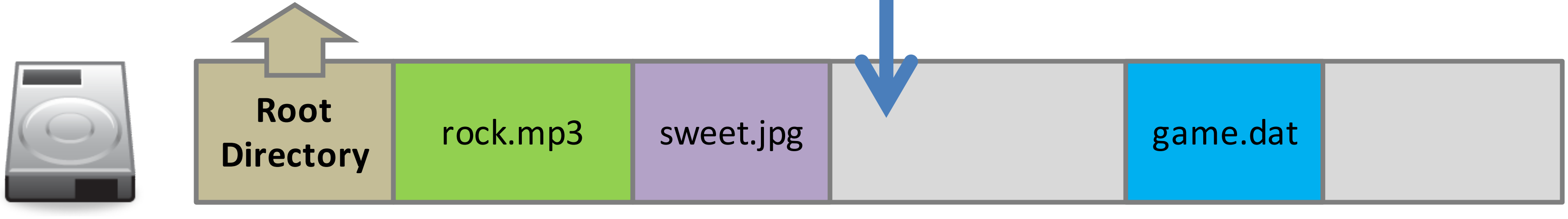
But, can you locate the **allocated space** and the **free space** in a short period of time? I mean in a 1TB HDD!

Requirements	
Allocated Space Mgt	👎
Free Space Mgt	👎
File Content Allocation	✓
File Attributes	✓
Directory	✓
File growth and shrink	
File creation	
File deletion	

Free space is here.

But, it needs an O(n) search, where n is the number of files.

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



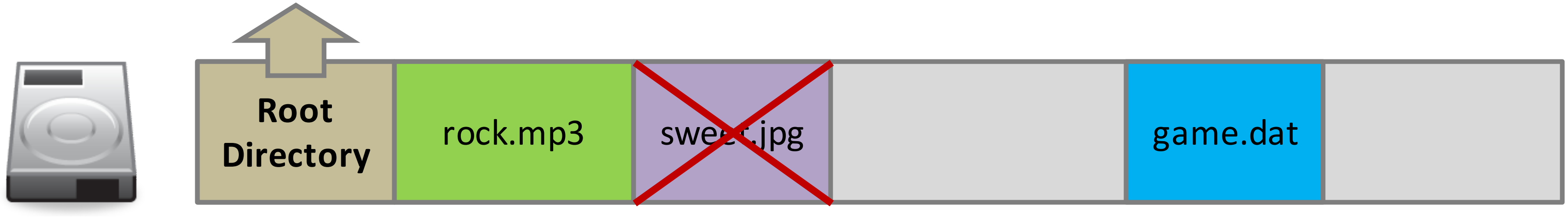
Contiguous allocation – basics

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



Requirements	
Allocated Space Mgt	👎
Free Space Mgt	👎
File Content Allocation	✓
File Attributes	✓
Directory	✓
File growth and shrink	
File creation	
File deletion	✓

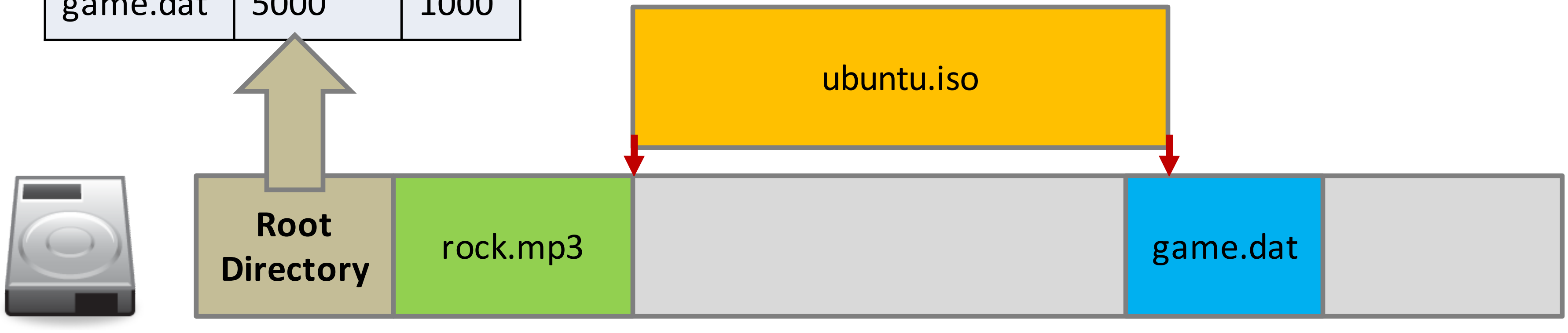
Contiguous allocation – basics

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

Requirements	
Allocated Space Mgt	👎
Free Space Mgt	👎
File Content Allocation	✓
File Attributes	✓
Directory	✓
File growth and shrink	
File creation	×
File deletion	✓

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

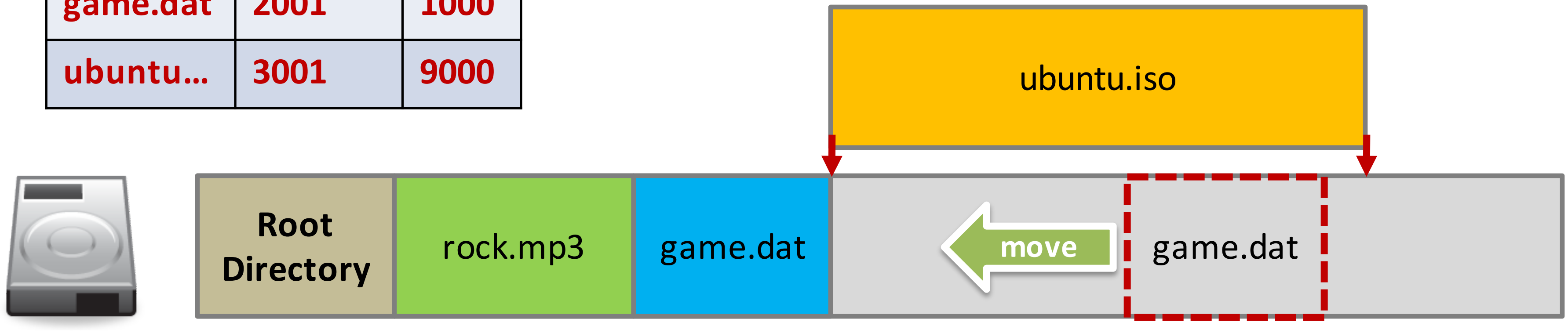


Contiguous allocation – basics

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

Requirements	
Allocated Space Mgt	👎
Free Space Mgt	👎
File Content Allocation	✓
File Attributes	✓
Directory	✓
File growth and shrink	
File creation	👎
File deletion	✓

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

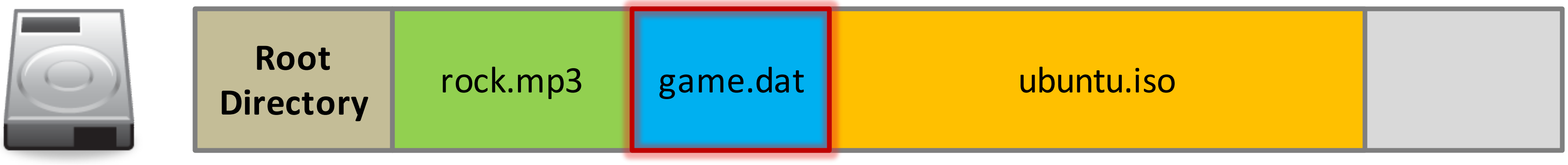


Contiguous allocation – basics

Requirements	
Allocated Space Mgt	👎
Free Space Mgt	👎
File Content Allocation	✓
File Attributes	✓
Directory	✓
File growth and shrink	✗
File creation	👎
File deletion	✓

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

Growth problem!
Can you suggest any method?



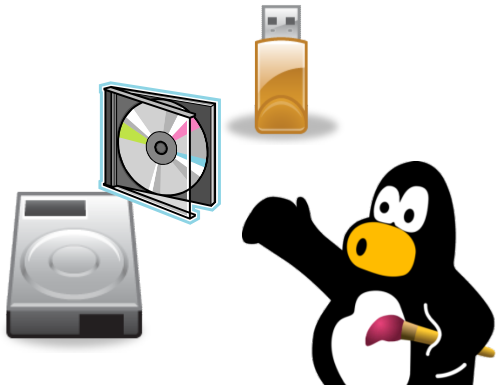
Contiguous allocation – application?

- Does this file system **only exist during the lecture** (or for the lecturer to contrast this FS to others)?
  - No…
  - Hint #1: better not grow any files.
  - Hint #2: OK to delete files.
  - Hint #3: better not add any files; or just add to the tail.

- Can you think of anything?
  - **ISO9660** and **MS Juliet** file systems.

Different Layouts

- Contiguous allocation;
- Linked list allocation;



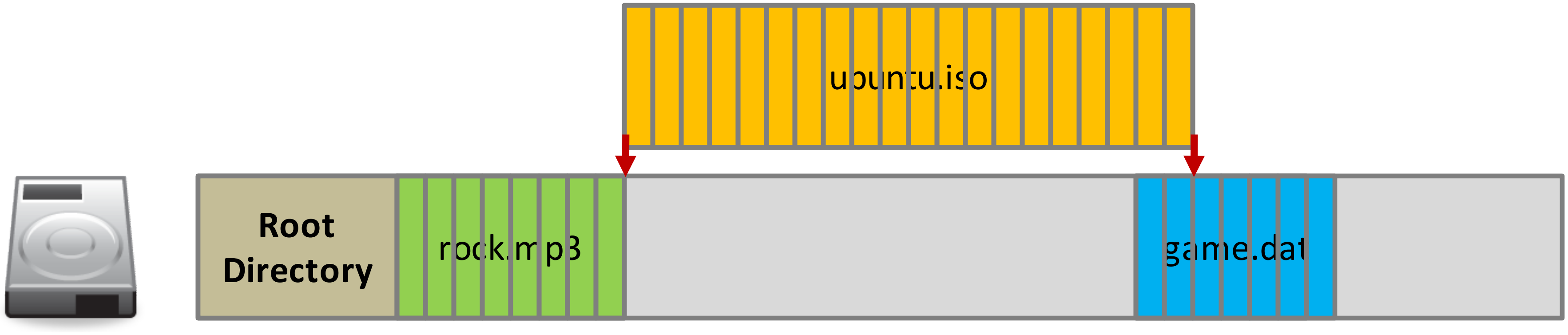
Approaching to a new design…

- Lessons learned from contiguous allocation:
  - External fragmentation.
    - Can we reduce its damage?
  - File growth problem.
    - Can we let every file to grow without paying an expensive overhead?

- One goal: **to avoid allocating space in a contiguous manner**!

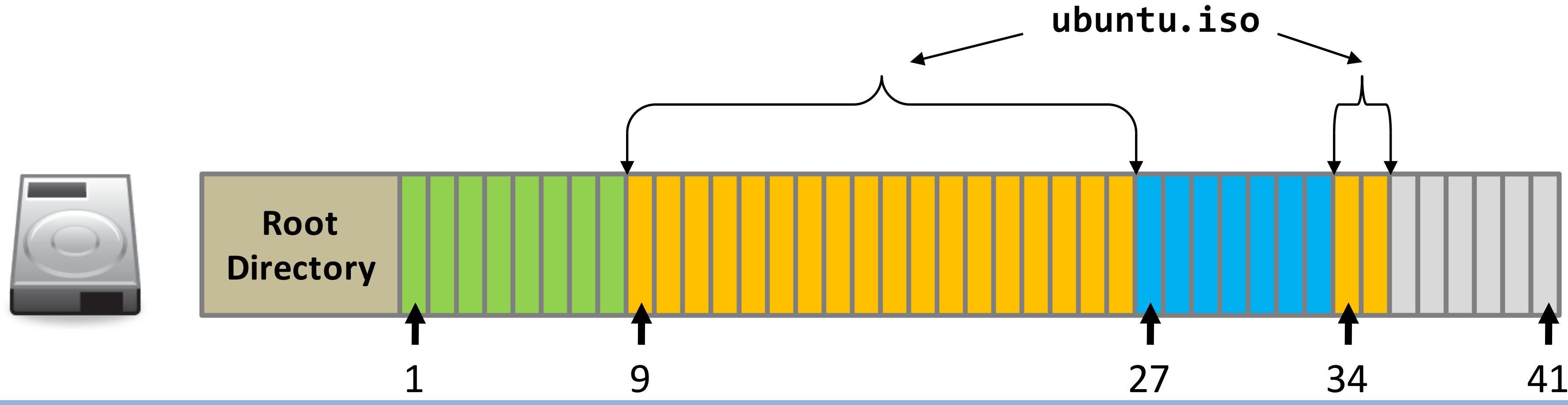
Linked list allocation – trial implementation

- Let’s borrow the idea from the <u>linked list</u>…
  - – Step (1) Chop the storage device into **equal-sized blocks**.



Linked list allocation – trial implementation

- 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.

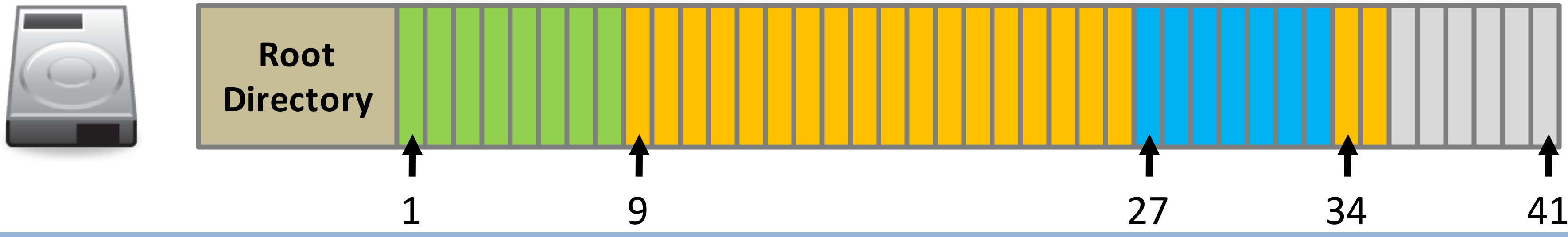


Linked list allocation – trial implementation

- 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.
  - Step (3) Root directory…well…it is strange…

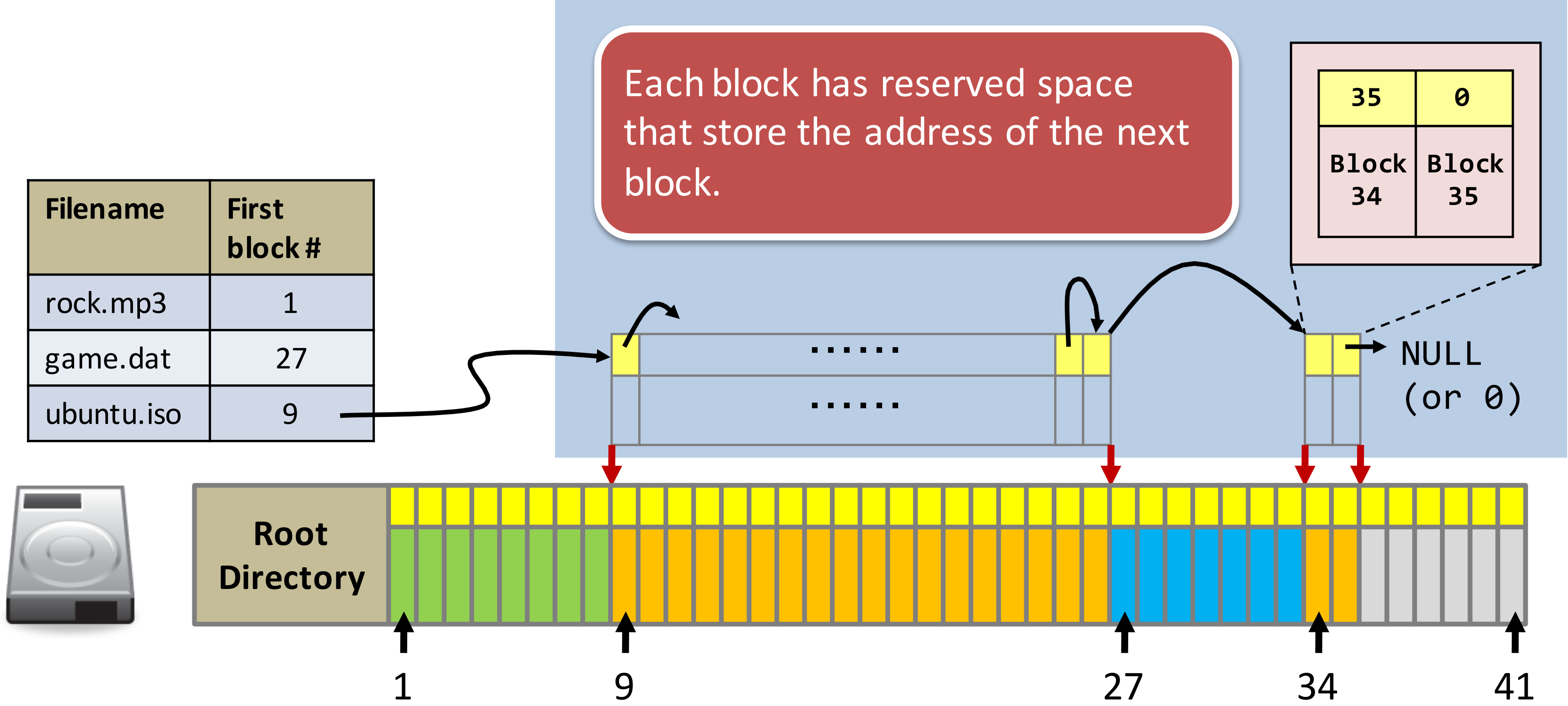
Filename	Block seq. #1	Block seq. #2	Block seq. #3
rock.mp3	1-8	NULL	NULL
game.dat	27-33	NULL	NULL
ubuntu.iso	9-26	34-35	NULL

So, any limit on the length of the block sequence?



Linked list allocation – trial implementation

- Let’s borrow **<u>4 bytes from each block</u>**!
  - To write the block # of the next block into the first 4 bytes of each block.



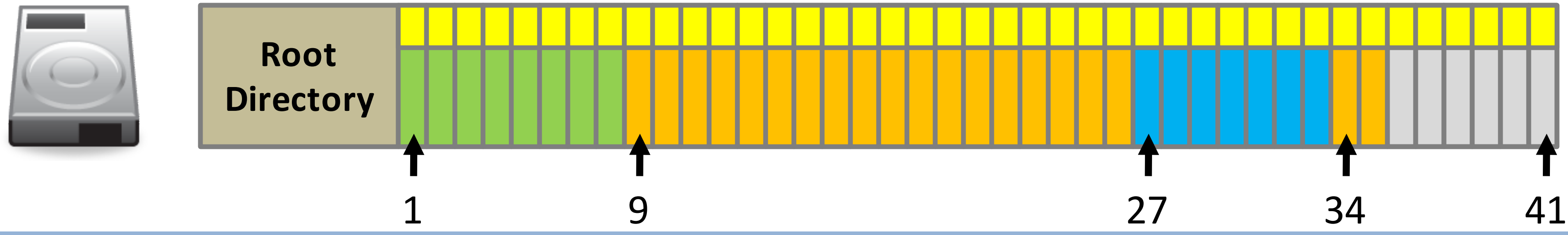
Filename	First block #
rock.mp3	1
game.dat	27
ubuntu.iso	9

Each block has reserved space that store the address of the next block.

Linked list allocation – trial implementation

- Note that the **file size is very important**, and, therefore, must be stored in the root directory.
  - **Class discussion.** What will happen if the size is removed?

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



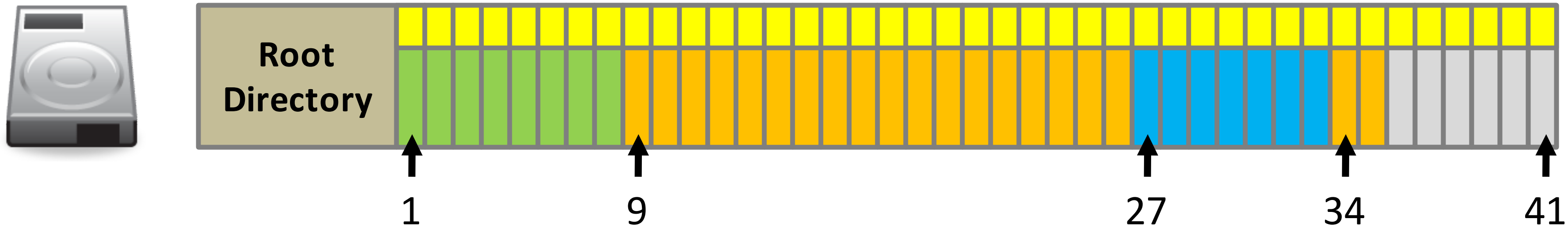
Linked list allocation – trial implementation

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

Requirements	
Allocated Space Mgt	👎
Free Space Mgt	👎
File Content Allocation	✓
File Attributes	✓
Directory	✓
File growth and shrink	✓
File creation	✓
File deletion	✓

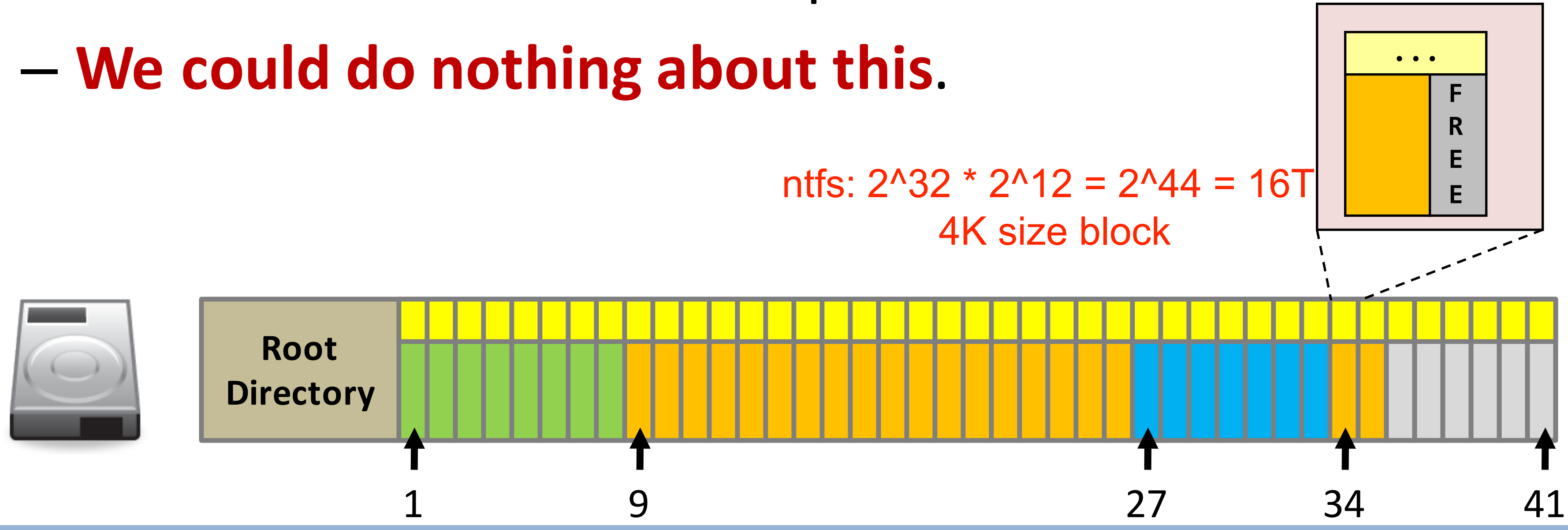
Filename	First block #	Size
rock.mp3	1	1900
game.dat	27	1000
ubuntu.iso	9	9000

Yet, there are two hidden problems.



Linked list allocation – trial implementation

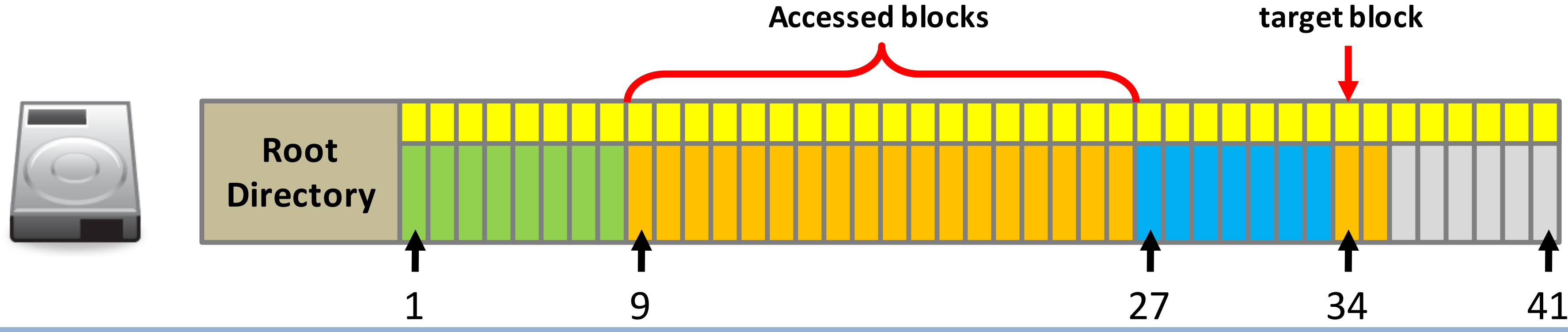
- Hidden problem #1: **Internal Fragmentation**.
  - A file is not always a multiple of the block size.
    - The last block of a file may not be **fully filled**.
    - E.g., a file of size 1 byte still occupies one block.
  - The remaining space will be wasted since no other files can be allowed to fill such space.
  - **We could do nothing about this**.



ntfs: 2^32 * 2^12 = 2^44 = 16T
4K size block

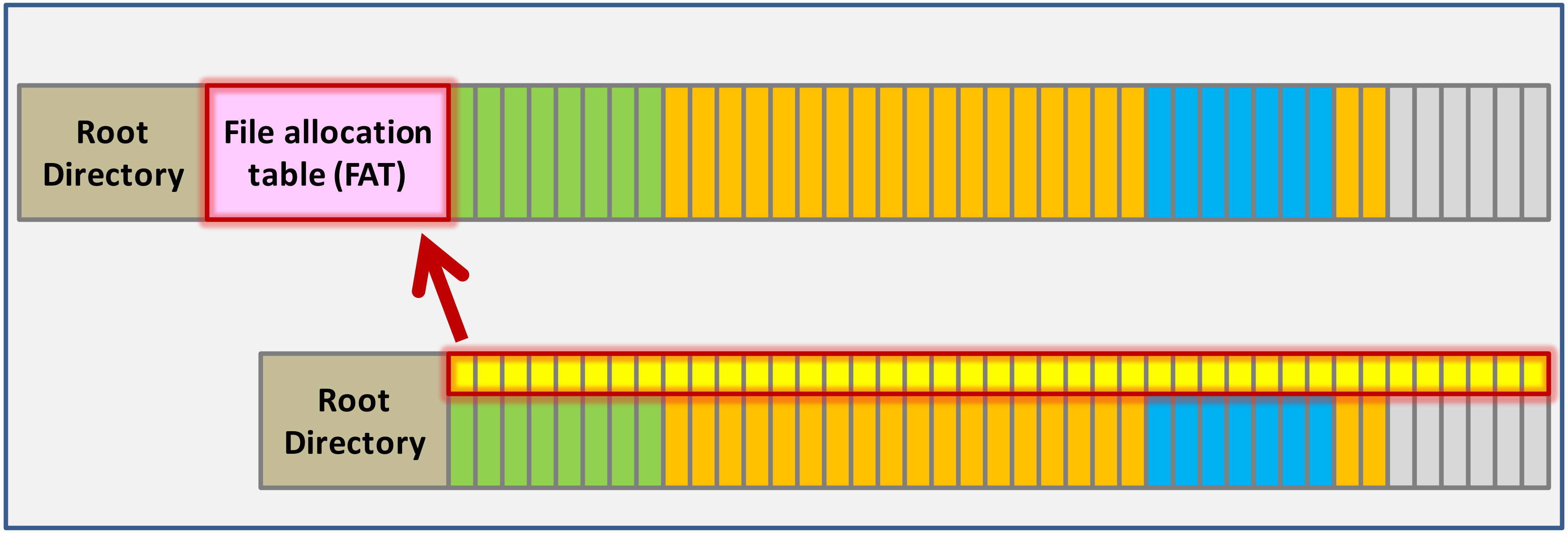
Linked list allocation – trial implementation

- Hidden problem #2: **random access performance**.
  - The random access mode means accessing a file at random locations, instead of a sequential manner.
  - The OS needs to access a series of blocks before it can access a target block.
    - Complexity: **O(n) number of I/O accesses**, where n is the number of blocks of the file.
  - **Can we do anything about this?**



Linked list allocation – trial implementation

- We are very wrong at the very beginning:
  - **The linked list information should be centralized!**
  - **This is the FAT-based file system!**



Linked list allocation – FAT implementation

Task: read "ubuntu.iso" sequentially.

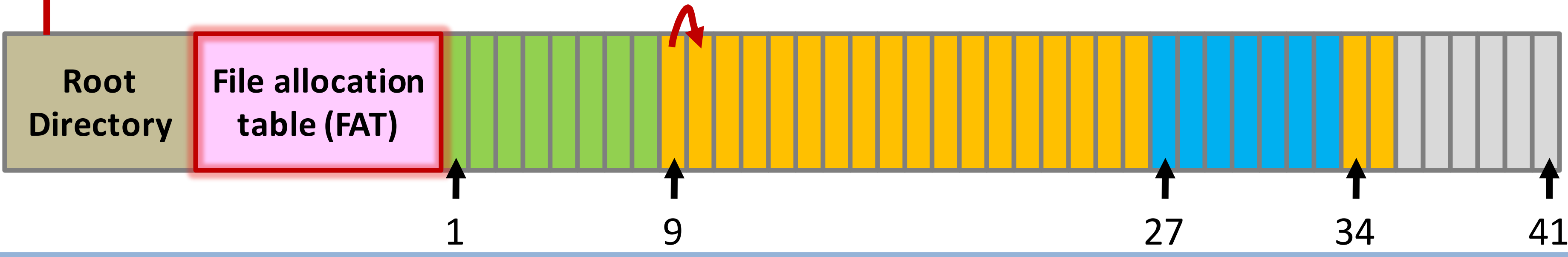
<u>Step 1.</u> Read the root directory and retrieve the **first block number**.

<u>Step 2.</u> Read the FAT to determine the location of next block.

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

1 row 1D array

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



Linked list allocation – FAT implementation

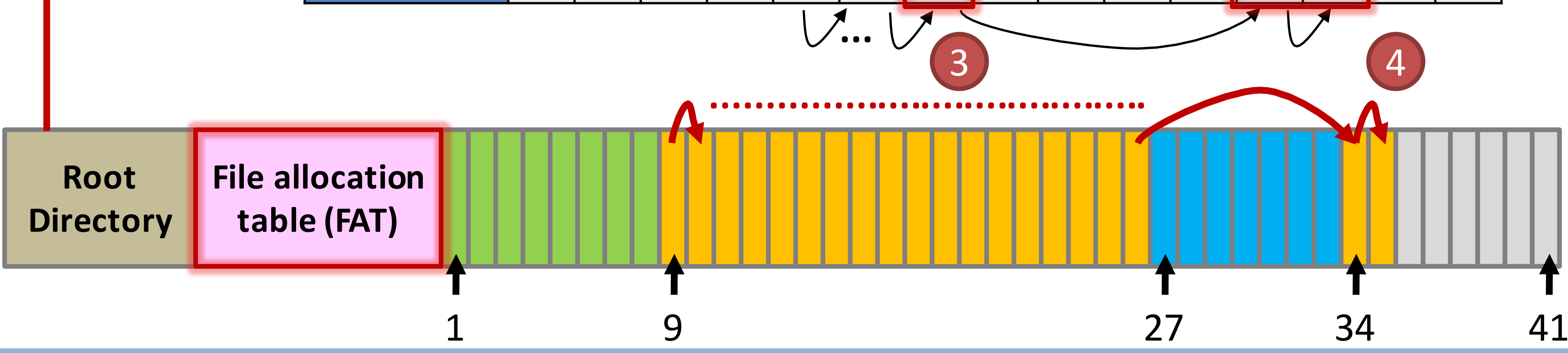
Task: read “ubuntu.iso” sequentially.

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

<u>Step 4.</u> The process stops until the FAT says the next block # is 0.

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	0	10	…	34	28	…	33	0	35	0	…	0

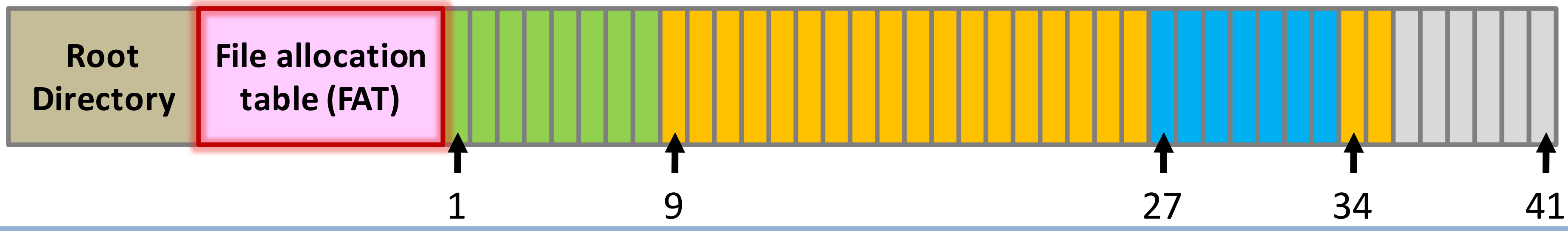


Linked list allocation – FAT implementation

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	0	10	…	34	28	…	33	0	35	0	…	0



Linked list allocation – FAT implementation

- Does this stop or mitigate the **random access problem**?
  - Only if the FAT is presented as **an array**.
  - Then, reaching an arbitrary location is as simple as doing a **pointer addition operation**.

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

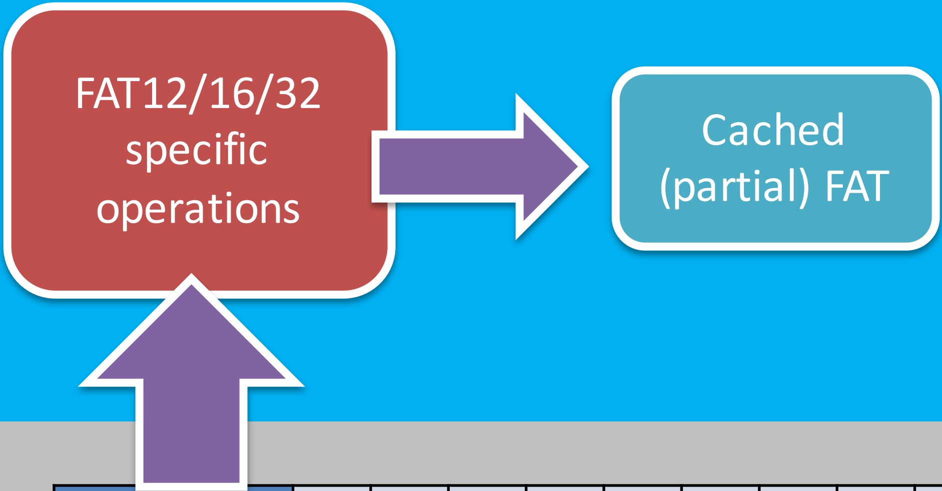
I know the starting position, ‘p’

I know the width. (4 bytes)

i-th FAT entry = p + 4i.

File Allocation Table (FAT)

Linked list allocation – FAT implementation



The random access problem can be eased by keeping a **cached version of FAT** inside the kernel.

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

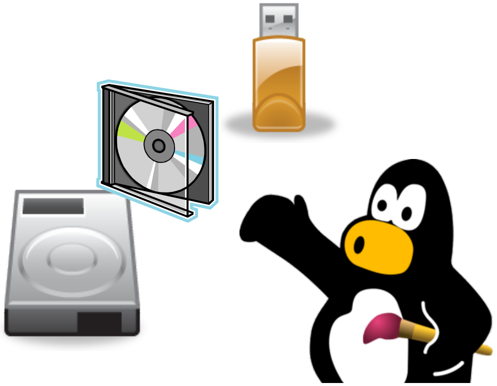
If this table is partially kept on the cache, then **extra I/O requests** will be generated in locating the next block #.

File Allocation Table (FAT)

Different Layouts

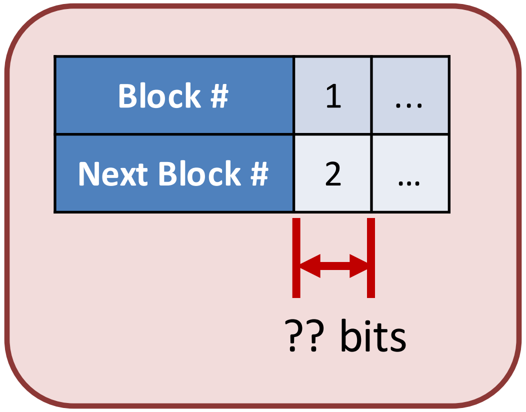
- Contiguous allocation;

- Linked list allocation;
  - FAT12/16/32;



FAT series – at a glance

- Uncle Bill named a block – a **cluster**.
- Different versions of FAT have different cluster address sizes.
  - The larger the cluster address size is, the larger **the size of the file allocation table** is resulted.



Block #	1	…
Next Block #	2	…

	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} (256M)

Ask Bill why…

FAT series – interesting facts

- Available cluster size:

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

Reference:
`help format.exe`

`Cluster size: 32KB`

`Cluster address: 28 bits`

File system size.

$$\begin{aligned}(32 \times 2^{10}) \times 2^{28} &= 2^5 \times 2^{10} \times 2^{28} \\ &= 2^{43} \quad (8\ TB)\end{aligned}$$

FAT series – interesting facts

- "*I've heard & tried that Windows XP only allows a partition of a maximum size 32GB!*" you said!

- True, but:

Fact #1	Using 3rd party tools, e.g., PartitionMagic, can create and format a partition > 32GB!
Fact #2	Windows XP allows you to mount a pre-formatted partition of size > 32GB!!
Fact #3	Windows 98 & ME setup disks know how to create and format a partition > 32 GB!!!

- Rumors:

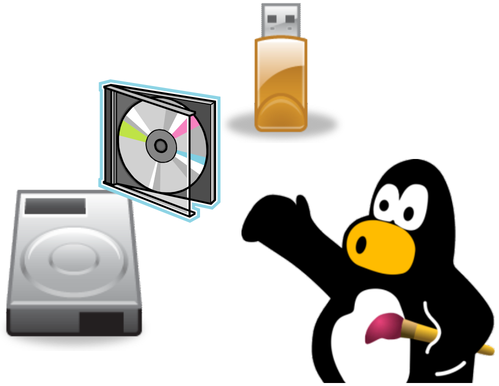
Micro$oft did it **on purpose**!
She wanted to *persuade* you to switch to NTFS and let FAT32 to fade out quietly (**and she failed**).

Reference: http://support.microsoft.com/kb/314463

Different Layouts

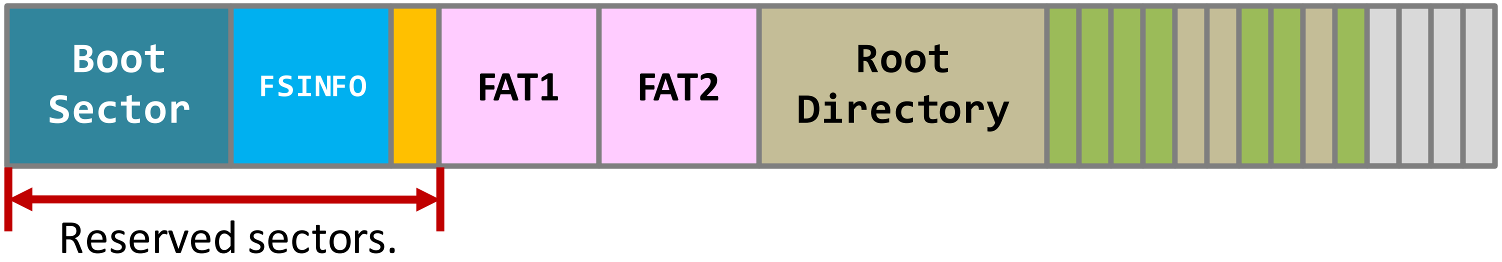
- Contiguous allocation;

- Linked list allocation;
  - FAT12/16/32
    - FS Layout

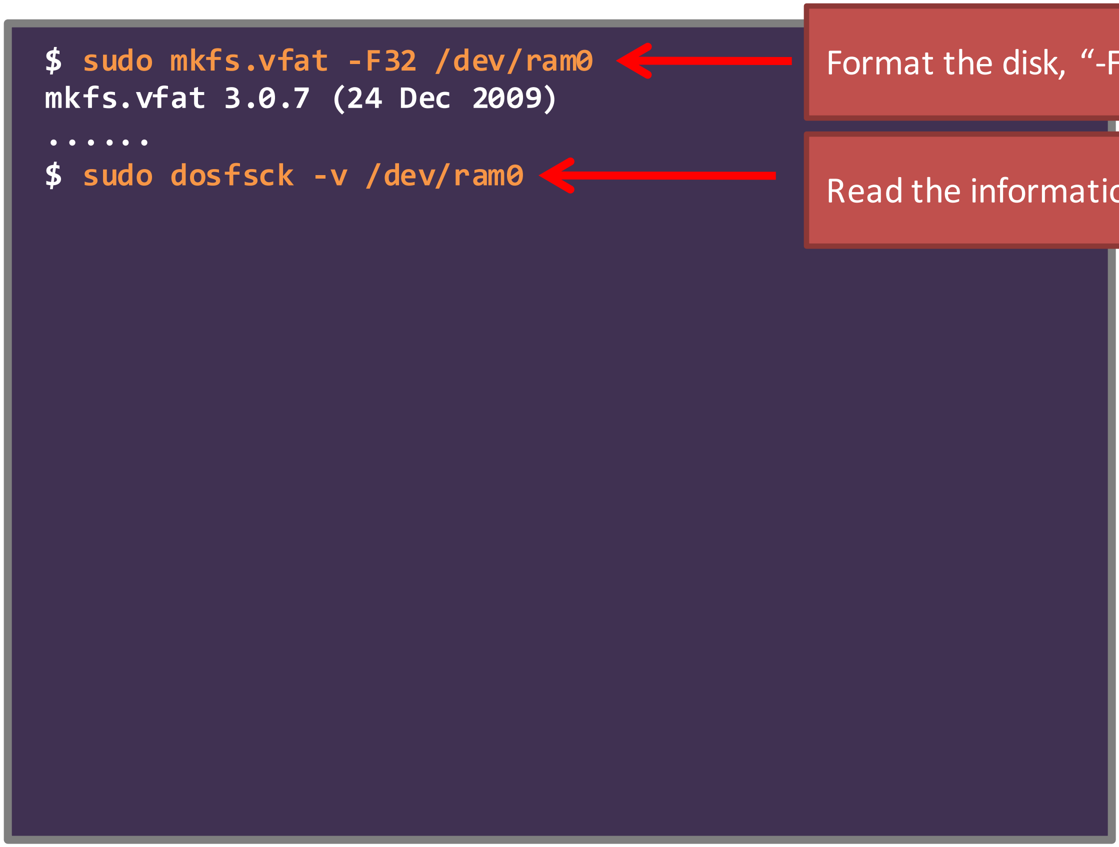


FAT series – layout overview

	Propose	Size
Boot sector	Store FS-specific parameters	1 sector, 512 bytes
FSINFO	Free-space management (seldom used)	1 sector, 512 bytes
Reserved sectors	Don't ask me, ask Micro$oft!	Variable, can be changed during format.
FAT (2 pieces)	A robust design; Number of FATs can be change during format.	Variable, depends on disk size and cluster size.
Root directory	Start of the directory tree.	At least one cluster, depend on the number of director entries.



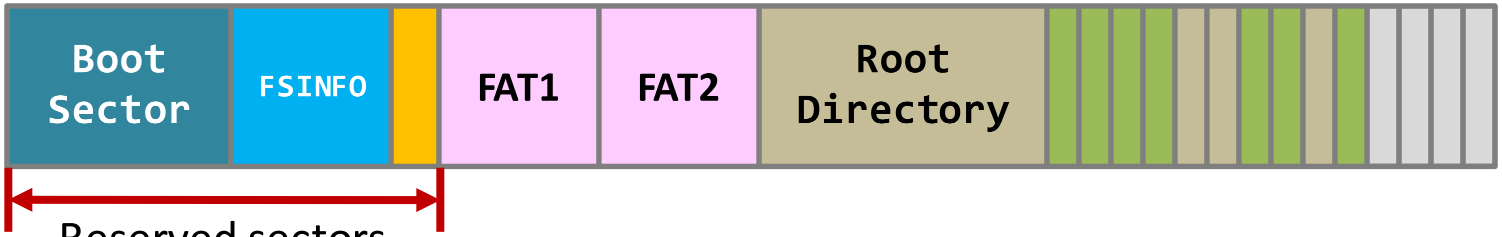
FAT series – layout overview



```
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.vfat 3.0.7 (24 Dec 2009)
......
$ sudo dosfsck -v /dev/ram0
```

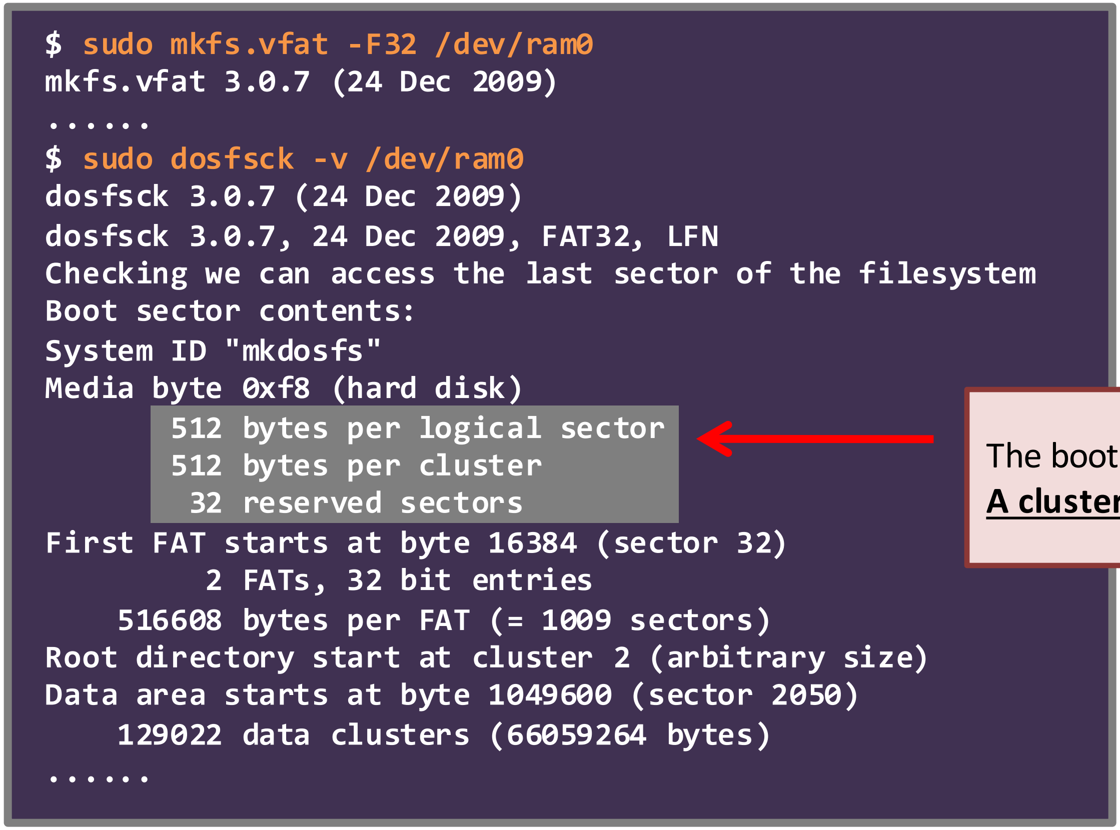
Format the disk, “-F32” means FAT32.

Read the information stored in the boot sector.



Reserved sectors.

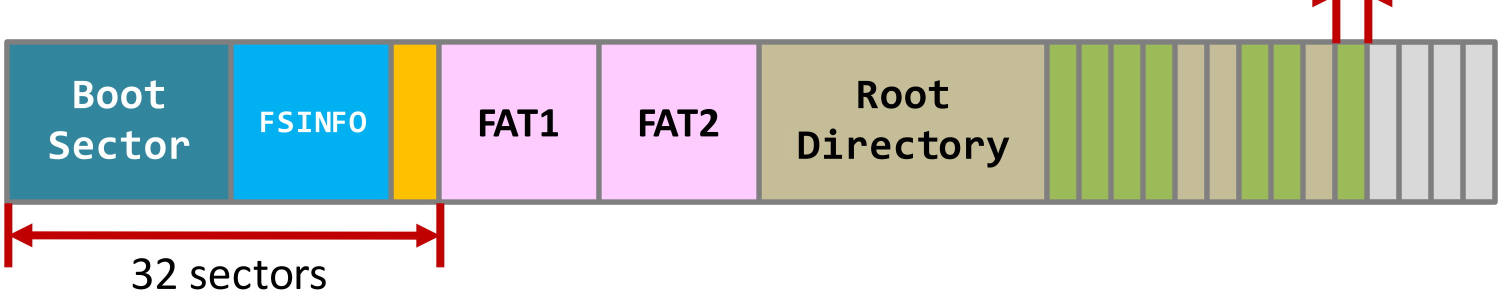
FAT series – layout overview



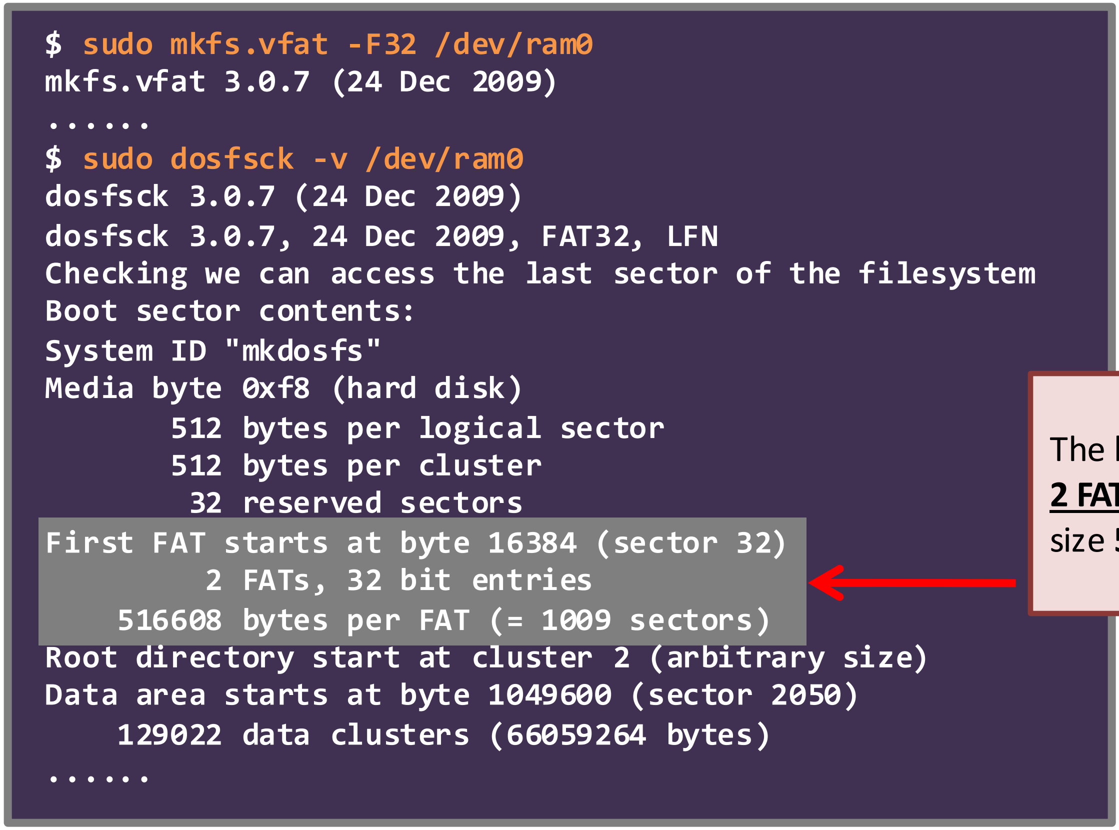
```
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.vfat 3.0.7 (24 Dec 2009)
......
$ sudo dosfsck -v /dev/ram0
dosfsck 3.0.7 (24 Dec 2009)
dosfsck 3.0.7, 24 Dec 2009, FAT32, LFN
Checking we can access the last sector of the filesystem
Boot sector contents:
System ID "mkdosfs"
Media byte 0xf8 (hard disk)
       512 bytes per logical sector
       512 bytes per cluster
        32 reserved sectors
First FAT starts at byte 16384 (sector 32)
         2 FATs, 32 bit entries
    516608 bytes per FAT (= 1009 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1049600 (sector 2050)
    129022 data clusters (66059264 bytes)
......
```

The boot sector says:
A cluster is made of 1 sector.

One cluster size: 512 bytes in this case

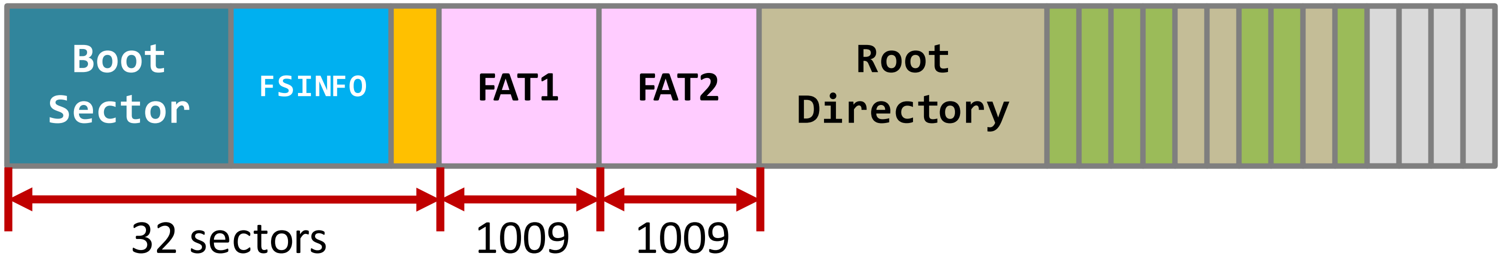


FAT series – layout overview

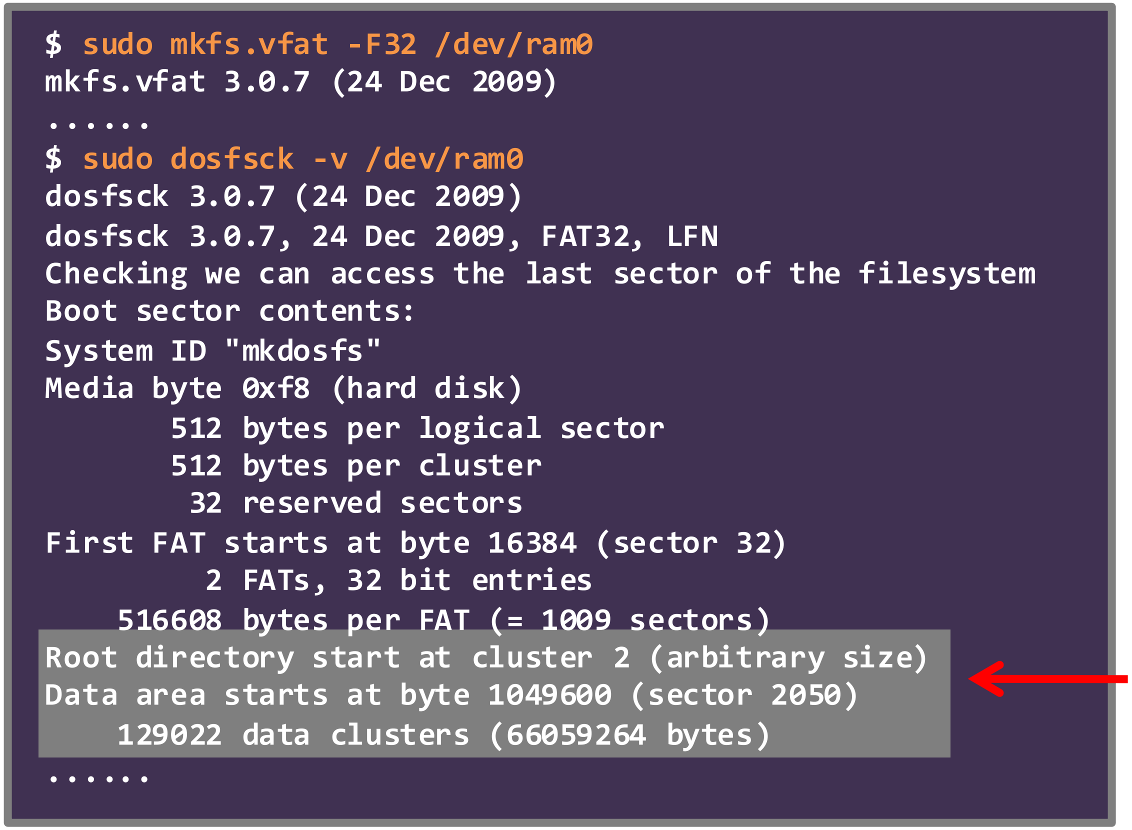


```
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.vfat 3.0.7 (24 Dec 2009)
......
$ sudo dosfsck -v /dev/ram0
dosfsck 3.0.7 (24 Dec 2009)
dosfsck 3.0.7, 24 Dec 2009, FAT32, LFN
Checking we can access the last sector of the filesystem
Boot sector contents:
System ID "mkdosfs"
Media byte 0xf8 (hard disk)
       512 bytes per logical sector
       512 bytes per cluster
        32 reserved sectors
First FAT starts at byte 16384 (sector 32)
         2 FATs, 32 bit entries
    516608 bytes per FAT (= 1009 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1049600 (sector 2050)
    129022 data clusters (66059264 bytes)
......
```

The boot sector says:
2 FATs and each of them is of size **516,608 bytes**.



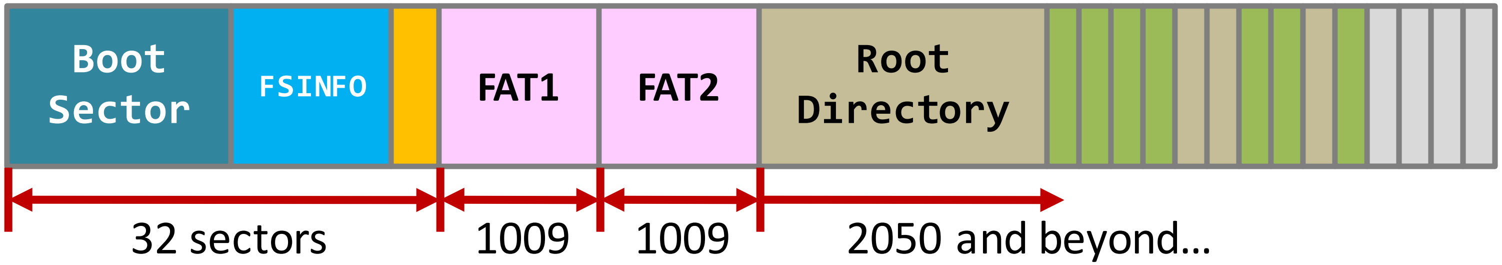
FAT series – layout overview



```
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.vfat 3.0.7 (24 Dec 2009)
......
$ sudo dosfsck -v /dev/ram0
dosfsck 3.0.7 (24 Dec 2009)
dosfsck 3.0.7, 24 Dec 2009, FAT32, LFN
Checking we can access the last sector of the filesystem
Boot sector contents:
System ID "mkdosfs"
Media byte 0xf8 (hard disk)
       512 bytes per logical sector
       512 bytes per cluster
        32 reserved sectors
First FAT starts at byte 16384 (sector 32)
         2 FATs, 32 bit entries
    516608 bytes per FAT (= 1009 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1049600 (sector 2050)
    129022 data clusters (66059264 bytes)
......
```

The first data cluster is **<u>Cluster #2</u>** and it is usually, not always, the root directory.

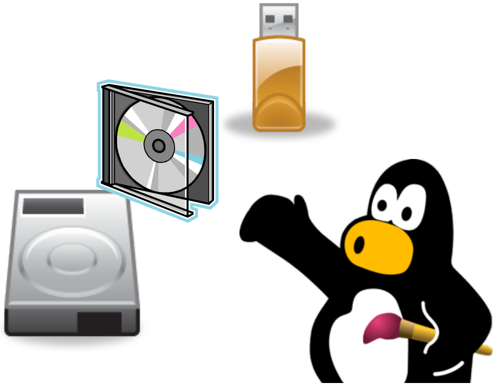
<u>Cluster #0 & #1</u> are reserved.



Different Layouts

- Contiguous allocation;

- Linked list allocation;
  - FAT12/16/32
    - FS Layout;
    - Directory;



FAT series – directory traversal

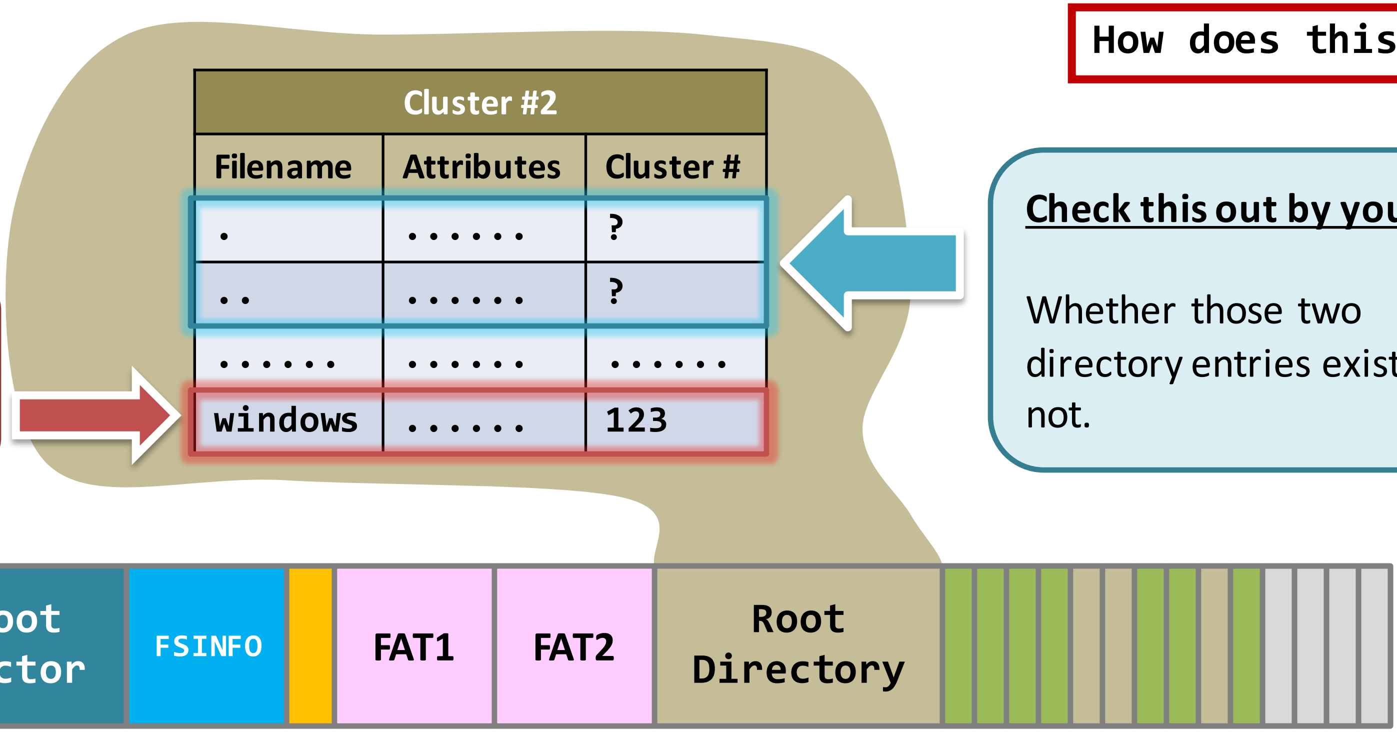
Step (1) Read the directory file of the root directory starting from **Cluster #2**.

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

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

How does this work?

A directory entry



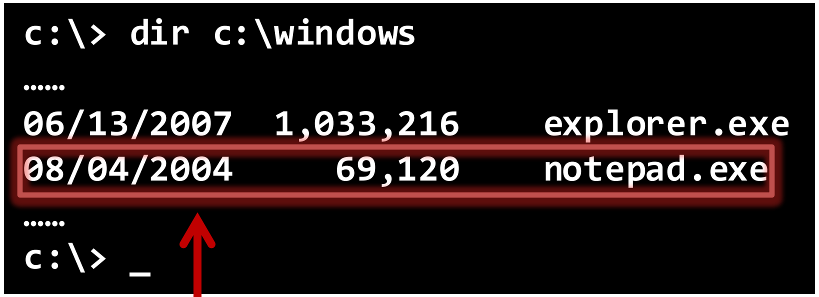
Cluster #2		
Filename	Attributes	Cluster #
.	......	?
..	......	?
......	......	......
windows	......	123

Check this out by yourself.

Whether those two directory entries exist or not.

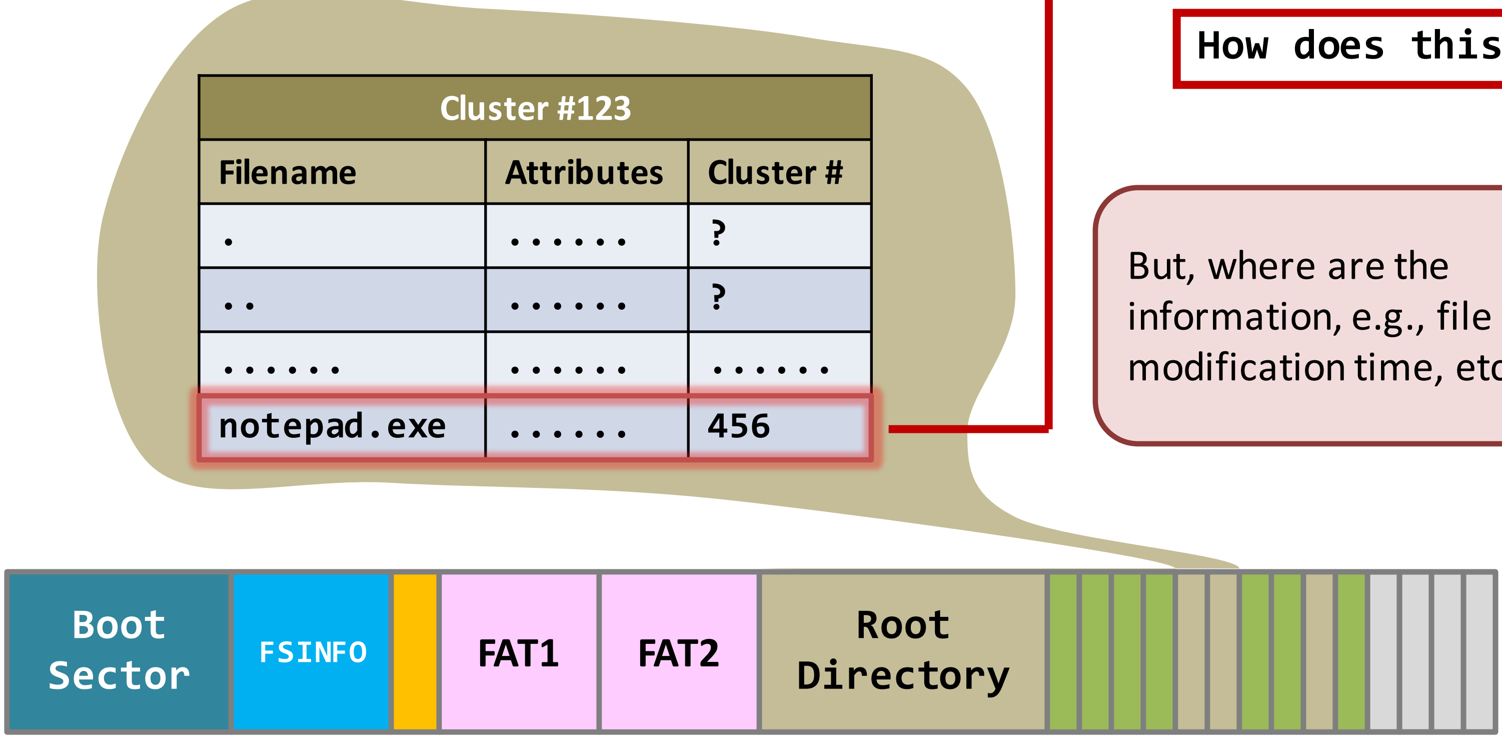
FAT series – directory traversal

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



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

How does this work?



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

But, where are the information, e.g., file size, modification time, etc?

FAT series – directory entry

- Directory entry is just a structure.

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 address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

what?

Filename	Attributes	Cluster #
explorer.exe	......	32

How?

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

Note. This is the 8+3 naming convention.

8 characters for name +
3 characters for file extension

FAT series – directory entry

- Directory entry is just a structure.

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 address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

what?

Filename	Attributes	Cluster #
explorer.exe	......	32

How?

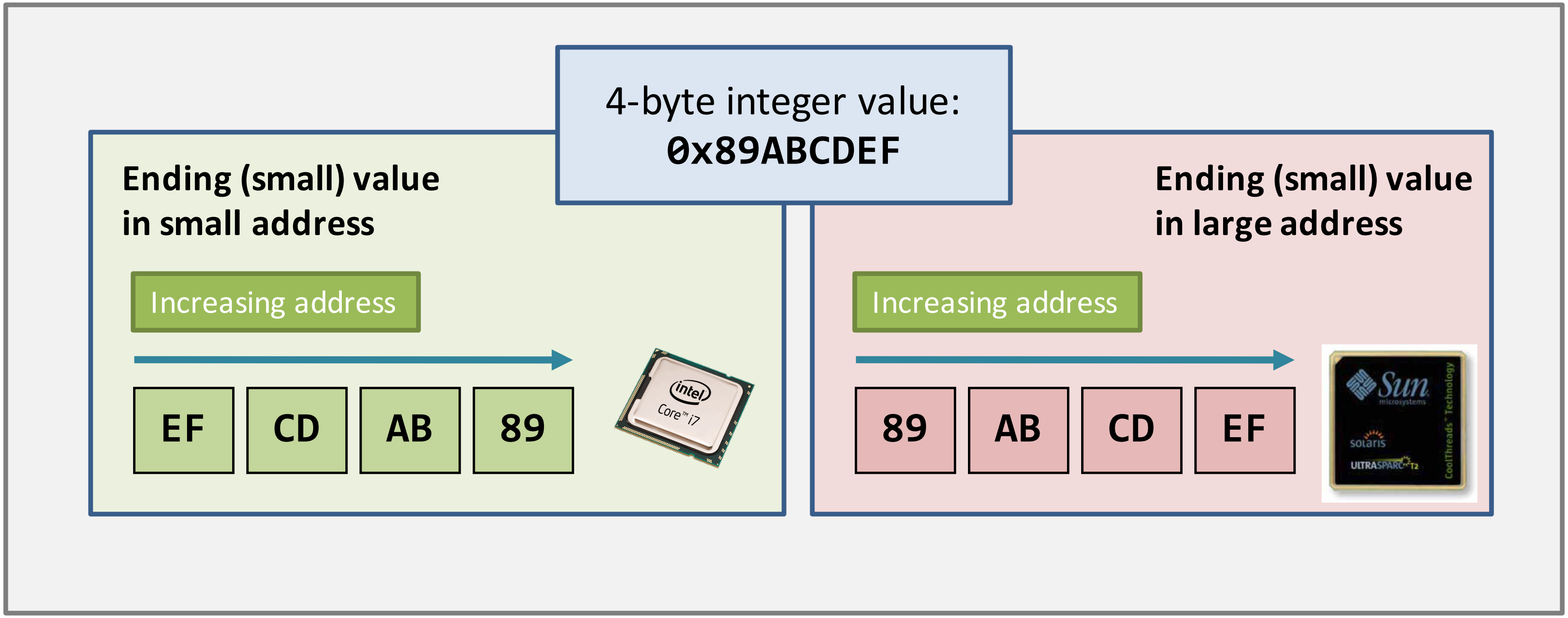
0	e	x	p	l	o	r	e	r	7
8	e	x	e	…	…	…	…	…	15
16	…	…	…	…	00	00	…	…	23
24	…	…	20	00	00	C4	0F	00	31

Higher 2 bytes		Lower 2 bytes			Cluster address
00	00	20	00	=	8192

Sidetrack – big endian VS little endian

EXTRA

- Endian-ness is about **byte ordering**.
  - It means the way that a machine (we mean the entire computer architecture) orders the bytes.



4-byte integer value: `0x89ABCDEF`

Ending (small) value in small address

Increasing address

EF	CD	AB	89

Ending (small) value in large address

Increasing address

89	AB	CD	EF

```
[examples@3150] cat endian.c
```

FAT series – directory entry

- Directory entry is just a structure.

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 address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

what?

Filename	Attributes	Cluster #
explorer.exe	......	32

How?

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

Big endian	00	00	20	00	=	8192
Little endian	00	00	00	20	=	32

FAT series – directory entry

- Directory entry is just a structure.

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 address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

what?

Filename	Attributes	Cluster #
explorer.exe	......	32

How?

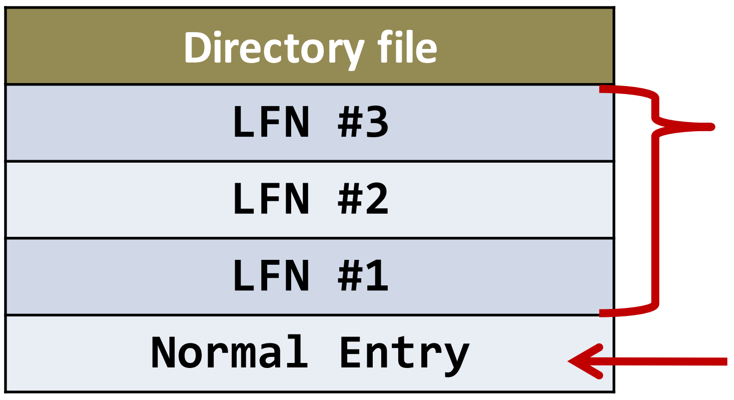
0	e	x	p	l	o	r	e	r	7
8	e	x	e	…	…	…	…	…	15
16	…	…	…	…	00	00	…	…	23
24	…	…	20	00	00	C4	0F	00	31

So, what is the largest size of a file?

4G – 1 bytes

FAT series – LFN directory entry

- LFN: Long File Name.
  - In old days, Uncle Bill set the rule that every file should follow the 8+3 naming convention.
  - Yet, he removed such a constraint in FAT32 by a **super ugly design**!



Directory file
LFN #3
LFN #2
LFN #1
Normal Entry

Each LFN entry represents 12 characters in Unicode, i.e., 2 bytes per character.
Yet, the sequence is upside-down!

The normal directory entry is still there.

FAT series – LFN directory entry

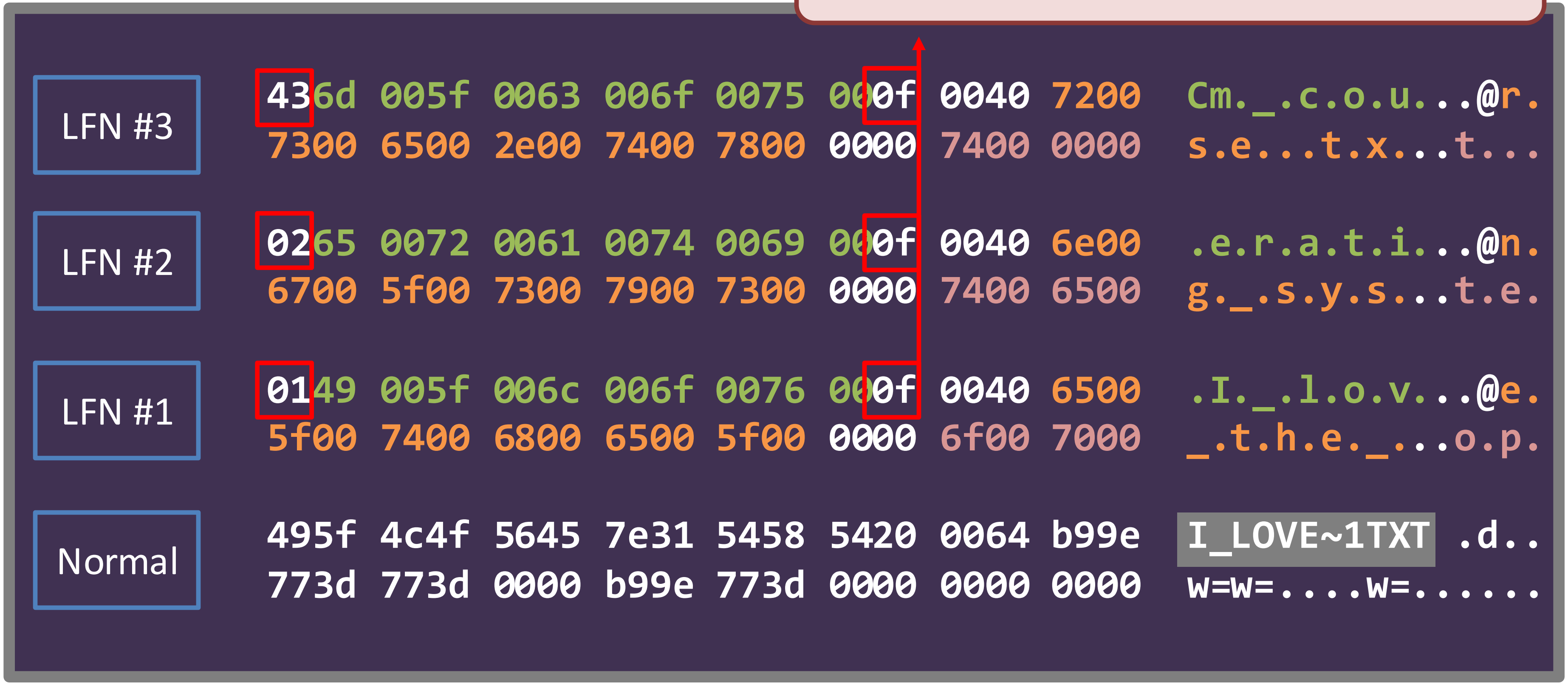
- LFN: Long File Name.

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 address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
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
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

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



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

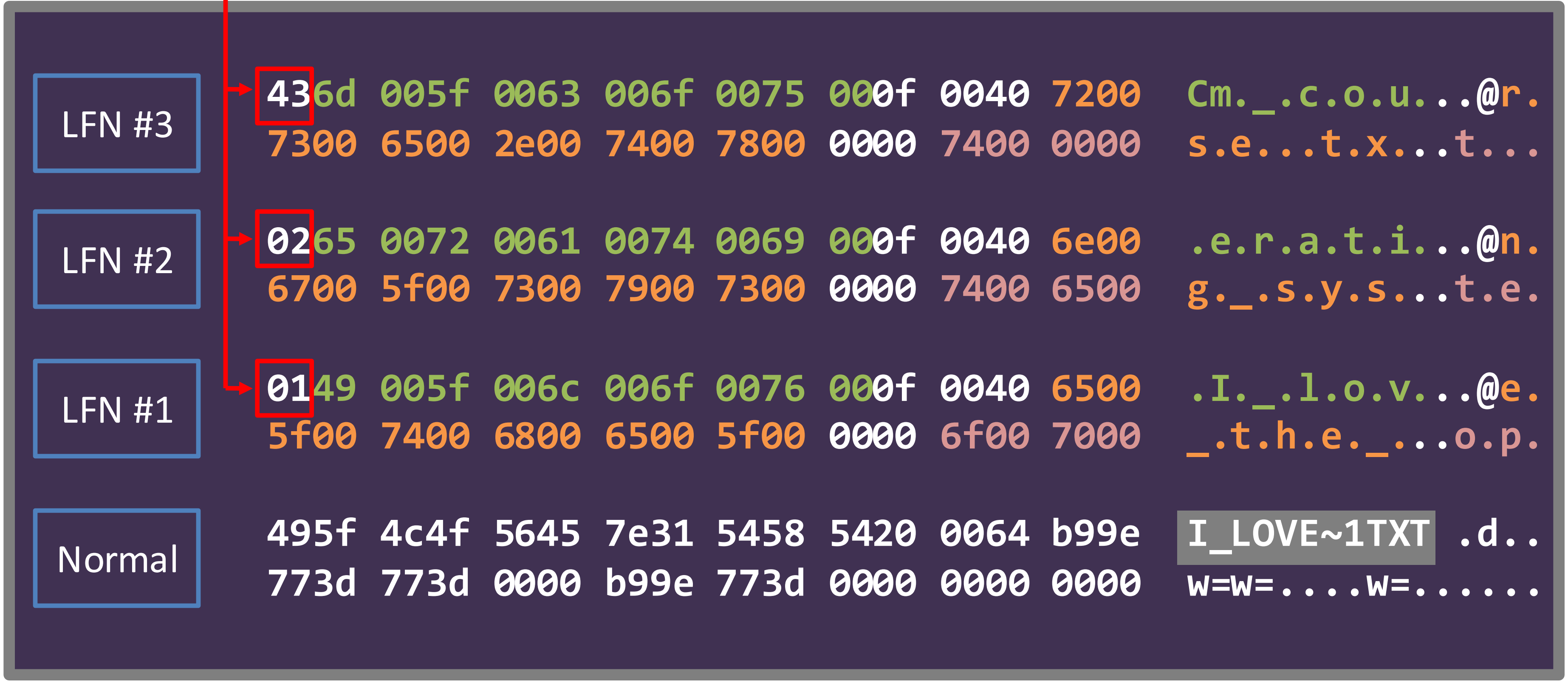
```
LFN #3   436d 005f 0063 006f 0075 000f 0040 7200  Cm._.c.o.u...@r.
         7300 6500 2e00 7400 7800 0000 7400 0000  s.e...t.x...t...

LFN #2   0265 0072 0061 0074 0069 000f 0040 6e00  .e.r.a.t.i...@n.
         6700 5f00 7300 7900 7300 0000 7400 6500  g._.s.y.s...t.e.

LFN #1   0149 005f 006c 006f 0076 000f 0040 6500  .I._.l.o.v...@e.
         5f00 7400 6800 6500 5f00 0000 6f00 7000  _.t.h.e._...o.p.

Normal   495f 4c4f 5645 7e31 5458 5420 0064 b99e  I_LOVE~1TXT .d..
         773d 773d 0000 b99e 773d 0000 0000 0000  w=w=....w=......
```

FAT series – LFN directory 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**.

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



```
436d 005f 0063 006f 0075 000f 0040 7200  Cm._.c.o.u...@r.
7300 6500 2e00 7400 7800 0000 7400 0000  s.e...t.x...t...

0265 0072 0061 0074 0069 000f 0040 6e00  .e.r.a.t.i...@n.
6700 5f00 7300 7900 7300 0000 7400 6500  g._.s.y.s...t.e.

0149 005f 006c 006f 0076 000f 0040 6500  .I._.l.o.v...@e.
5f00 7400 6800 6500 5f00 0000 6f00 7000  _.t.h.e._...o.p.

495f 4c4f 5645 7e31 5458 5420 0064 b99e  I_LOVE~1TXT .d..
773d 773d 0000 b99e 773d 0000 0000 0000  w=w=....w=......
```

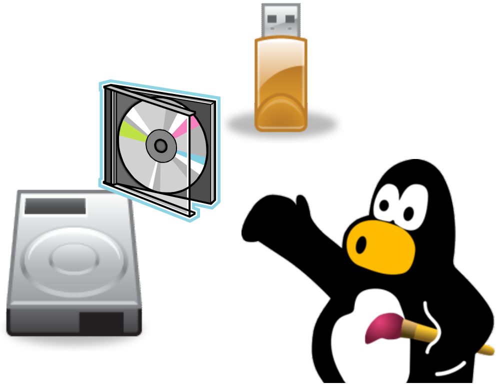
FAT series – directory entry: a short summary

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

Different Layouts

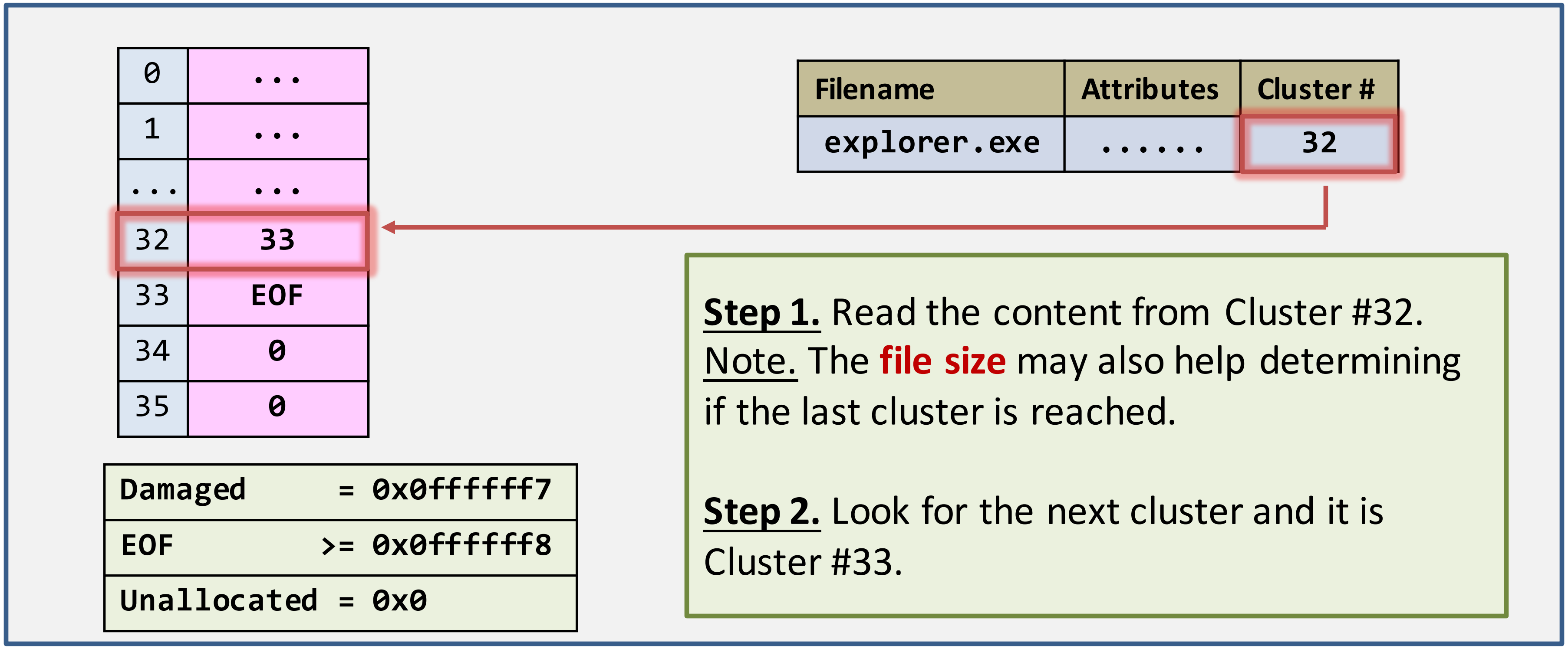
- Contiguous allocation;

- Linked list allocation;
  - FAT12/16/32
    - FS Layout
    - Directory
    - Read & write files



FAT series – reading a file

Task: read “C:\windows\explorer.exe” sequentially.



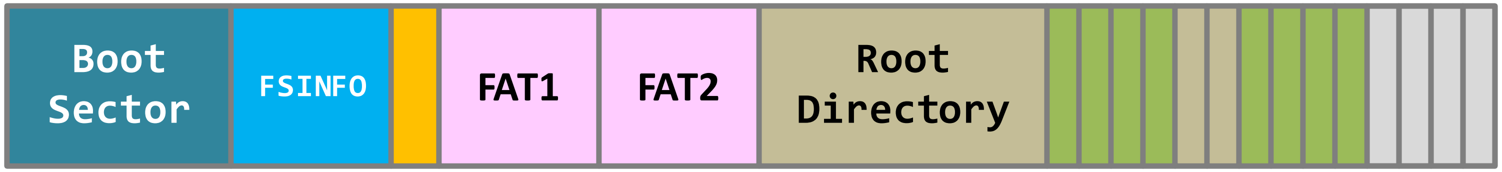
0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

Filename	Attributes	Cluster #
explorer.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

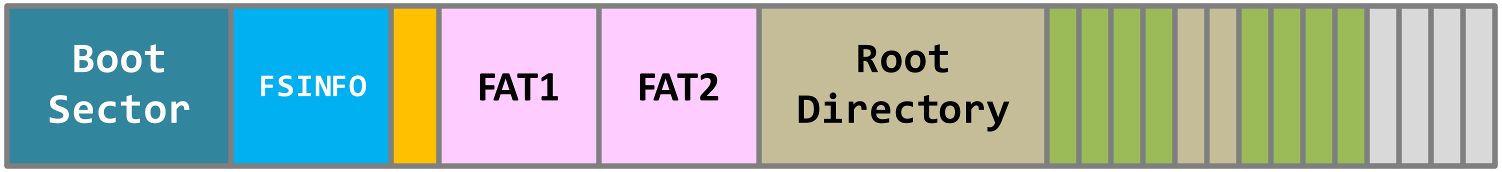
Task: read "C:\windows\explorer.exe" sequentially.

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

Filename	Attributes	Cluster #
explorer.exe	......	32

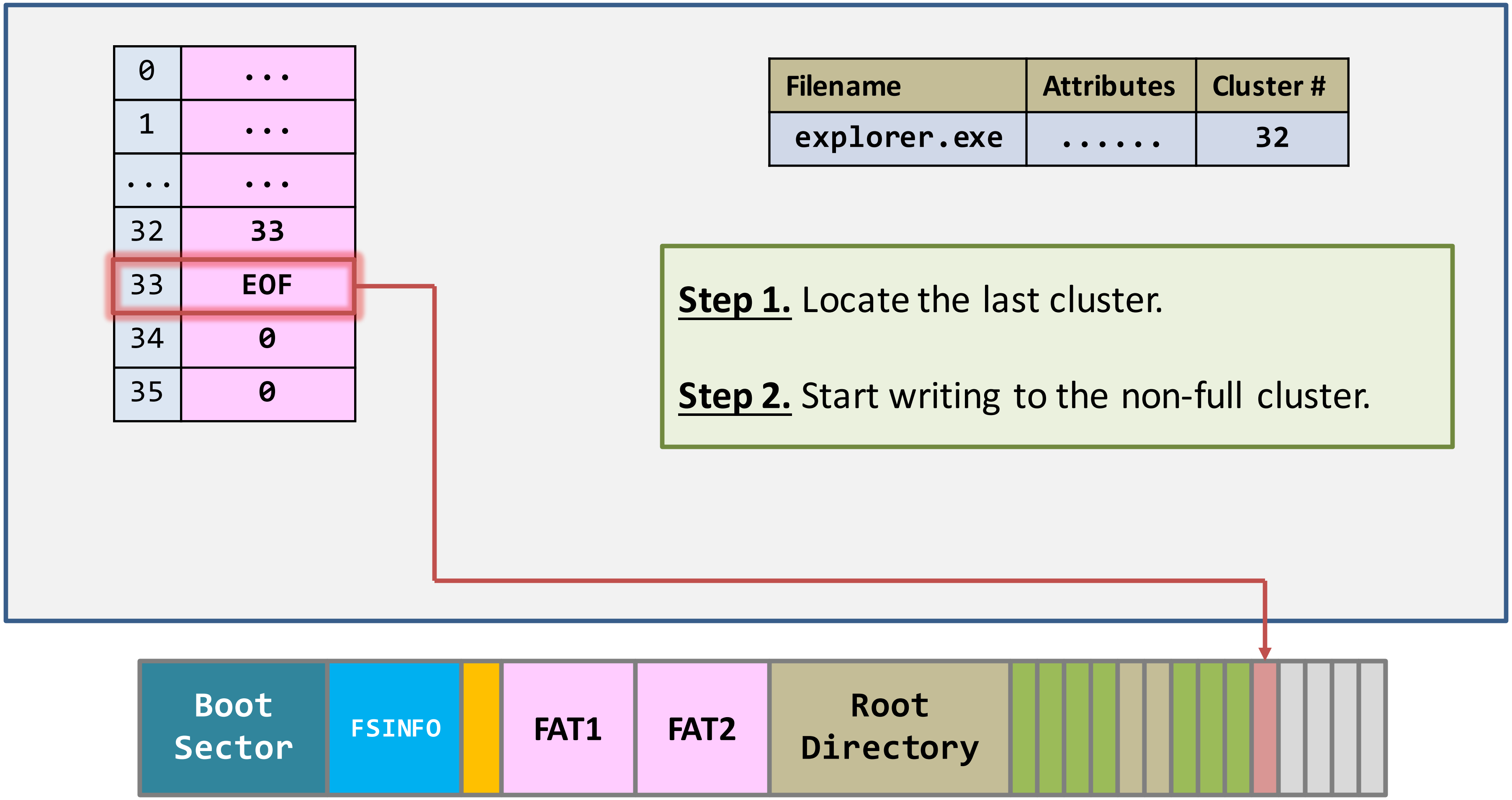
Step 3. Since the FAT has marked "EOF", we have reached the last cluster.

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\explorer.exe".



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

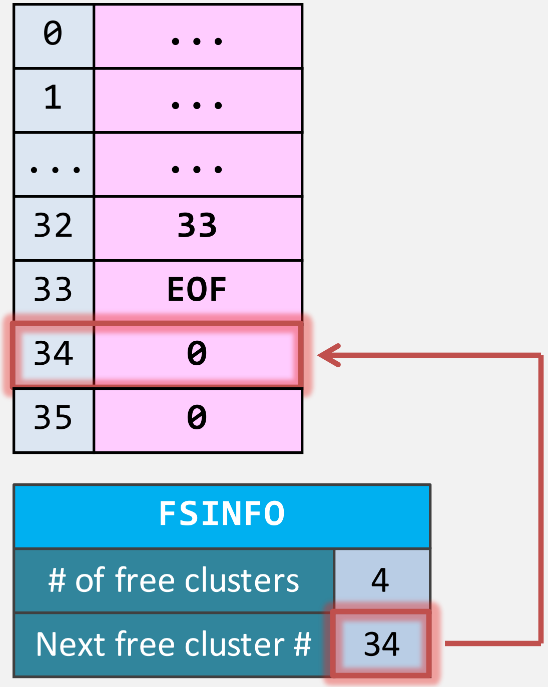
Filename	Attributes	Cluster #
explorer.exe	......	32

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\explorer.exe".

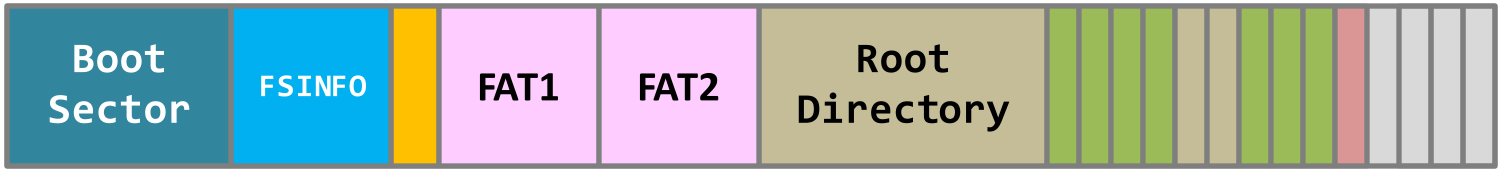


0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

FSINFO	
# of free clusters	4
Next free cluster #	34

Filename	Attributes	Cluster #
explorer.exe	......	32

Step 3. Allocate the next cluster through FSINFO.



FAT series – writing a file

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

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

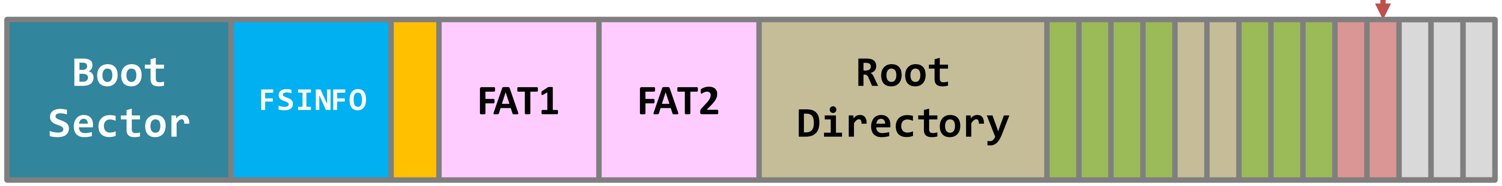
FSINFO	
# of free clusters	3
Next free cluster #	35

Filename	Attributes	Cluster #
explorer.exe	......	32

Step 3. Allocate the next cluster through FSINFO.

Step 4. Update the FATs and FSINFO.

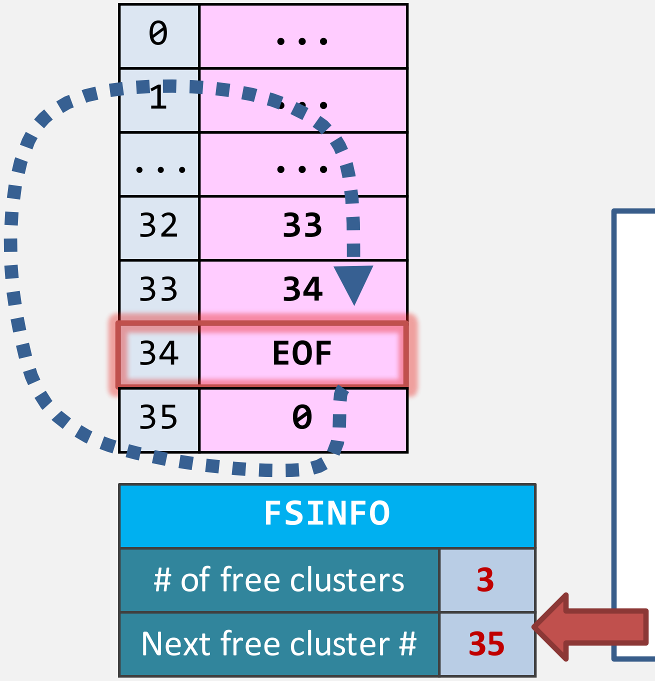
Step 5. When write finishes, update the file size.



FAT series – writing a file

EXTRA

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

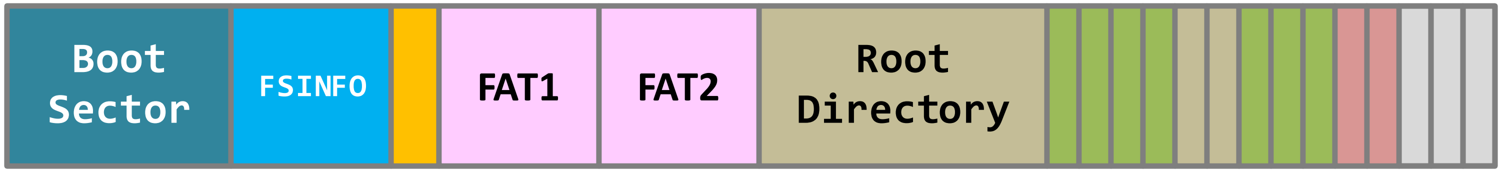


Filename	Attributes	Cluster #
explorer.exe	......	32

Reasons for obtaining the Cluster #35 is not trivial.

The search for the next free cluster is a circular, next-available search.

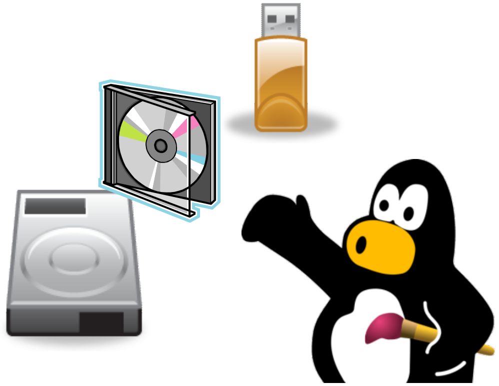
Do you know what is **spatial locality**?



Different Layouts

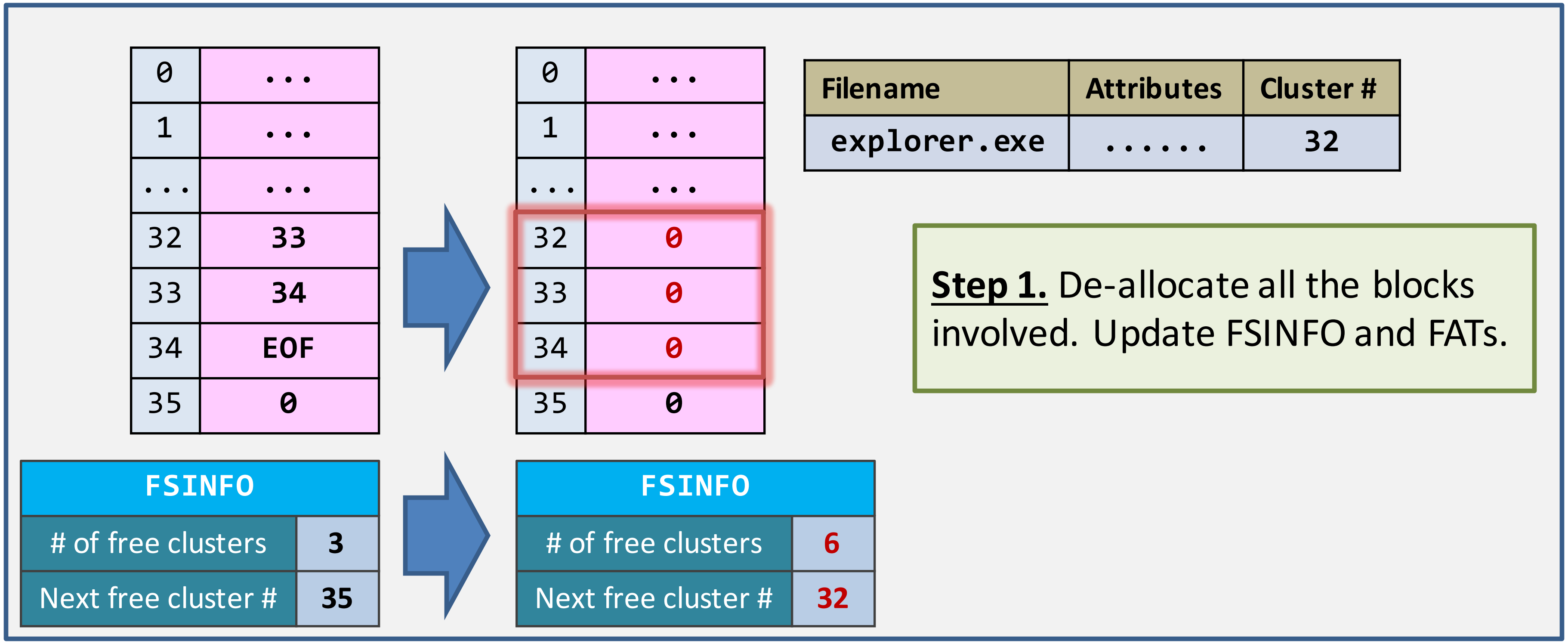
- Contiguous allocation;

- Linked list allocation;
  - FAT12/16/32
    - FS Layout
    - Directory
    - Read & write files
    - Delete & recover deleted files



FAT series – delete a file

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



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

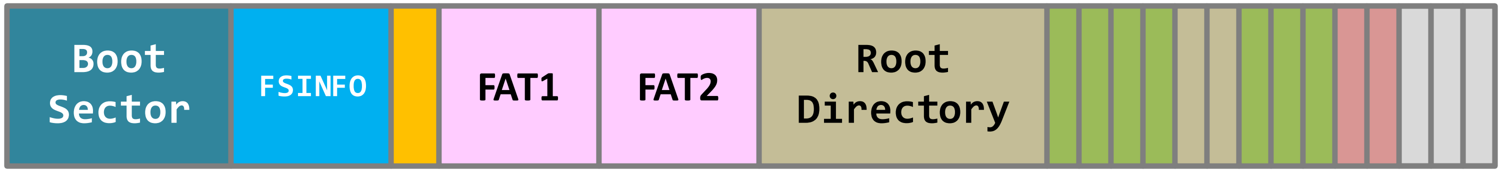
0	...
1	...
...	...
32	**0**
33	**0**
34	**0**
35	**0**

Filename	Attributes	Cluster #
explorer.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

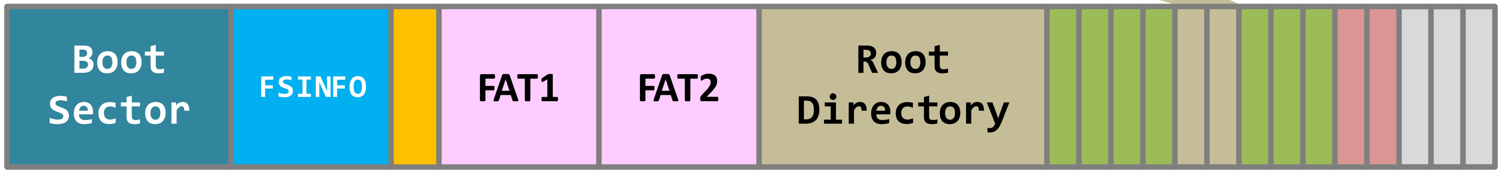
Task: delete “C:\windows\explorer.exe”.

Cluster #123		
Filename	**Attributes**	**Cluster #**
.	......	?
..	......	?
_xplorer.exe	......	32
notepad.exe	......	456

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

LFN entries also receive the same treatment.

That’s the end of deletion!



FAT series – really delete a file?

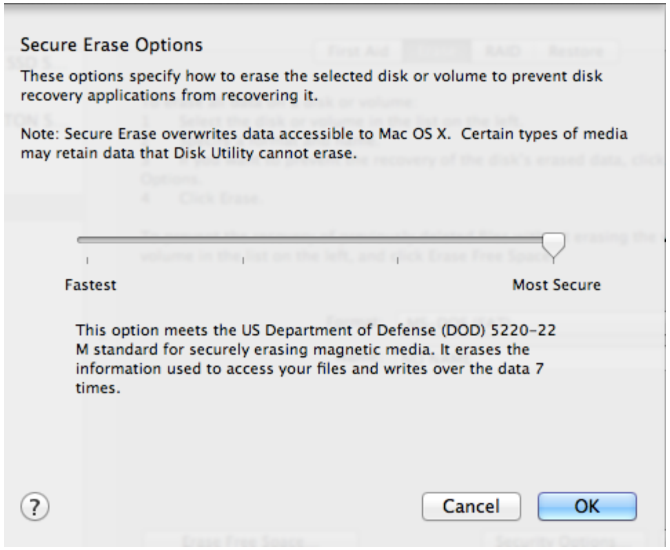
- Can you see that: **the file is not really removed from the FS layout?**
  - Perform a search in all the free space. Then, you will find all deleted file contents.

- “*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



Secure Erase Options

These options specify how to erase the selected disk or volume to prevent disk recovery applications from recovering it.

Note: Secure Erase overwrites data accessible to Mac OS X. Certain types of media may retain data that Disk Utility cannot erase.

This option meets the US Department of Defense (DOD) 5220-22 M standard for securely erasing magnetic media. It erases the information used to access your files and writes over the data 7 times.

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/

What will the research community tell you?

http://cdn.computerscience1.net/2006/fall/lectures/8/articles8.pdf

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 <= 1 cluster	Because **the first cluster address** is still readable, the recovery is having a very high successful rate. Note that filenames with **the same postfix** may also be found.
File size > 1 cluster	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. If not, you'd better have the **checksum** and **the exact file size** of the deleted file beforehand, so that you can use a *brute-force method* to recover the file.

FAT series – conclusion

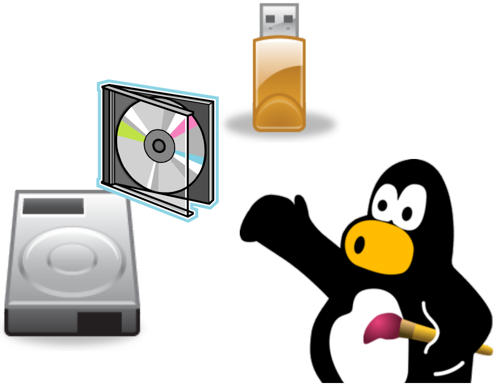
- It is a “nice” file system:
  - Space efficient: 4 bytes overhead (FAT entry) per data cluster.
- Deletion problem:
  - This is a lazy yet fast implementation.
  - Need extra protection for deleted data.
- Deployment:
  - It is everywhere: CF cards, SD cards, USB drives, iPod.

Requirements	
Allocated Space Mgt	✓
Free Space Mgt	✓
File Content Allocation	✓
File Attributes	✓
Directory	✓
File growth and shrink	✓
File creation	✓
File deletion	?

EXTRA

Different Layouts

- Contiguous allocation;
- Linked list allocation;
  - FAT12/16/32;
- Index-node allocation;

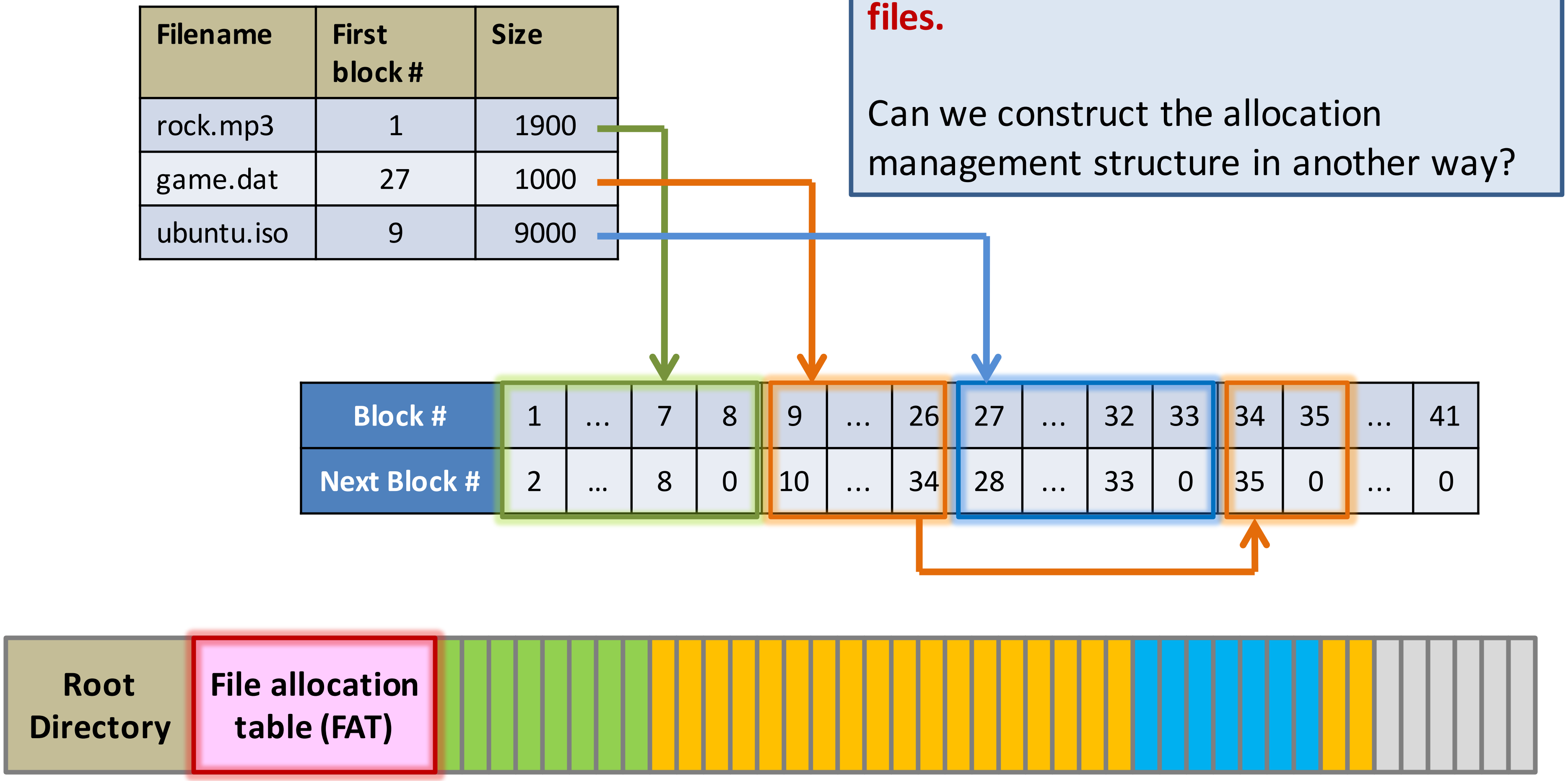


Back to the basics…

EXTRA

The allocation management structure is the FAT, which **doesn’t emphasize on files.**

Can we construct the allocation management structure in another way?

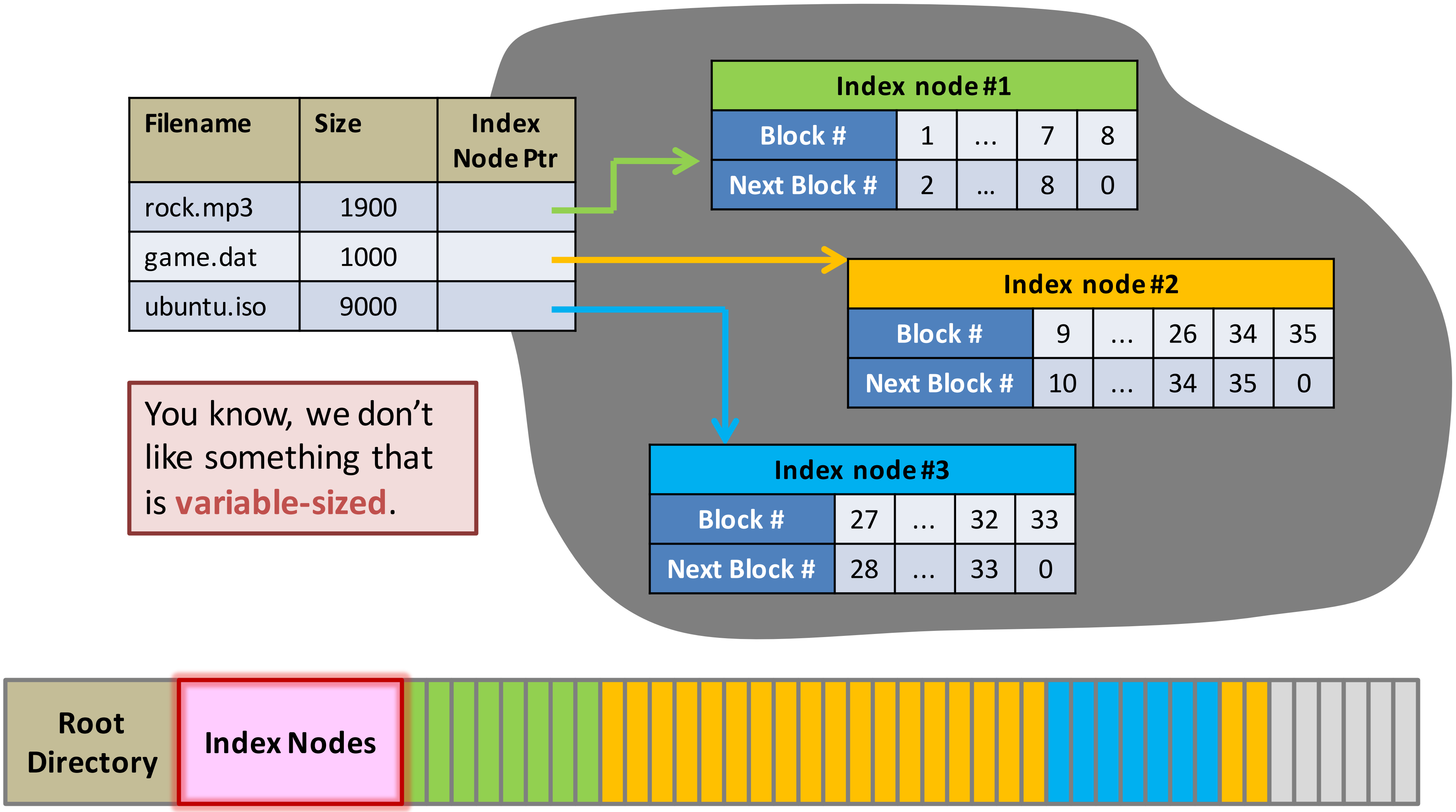


Filename	First block #	Size
rock.mp3	1	1900
game.dat	27	1000
ubuntu.iso	9	9000

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

Back to the basics…

EXTRA



Filename	Size	Index Node Ptr
rock.mp3	1900	
game.dat	1000	
ubuntu.iso	9000	

Index node #1

Block #	1	…	7	8
Next Block #	2	…	8	0

Index node #2

Block #	9	…	26	34	35
Next Block #	10	…	34	35	0

Index node #3

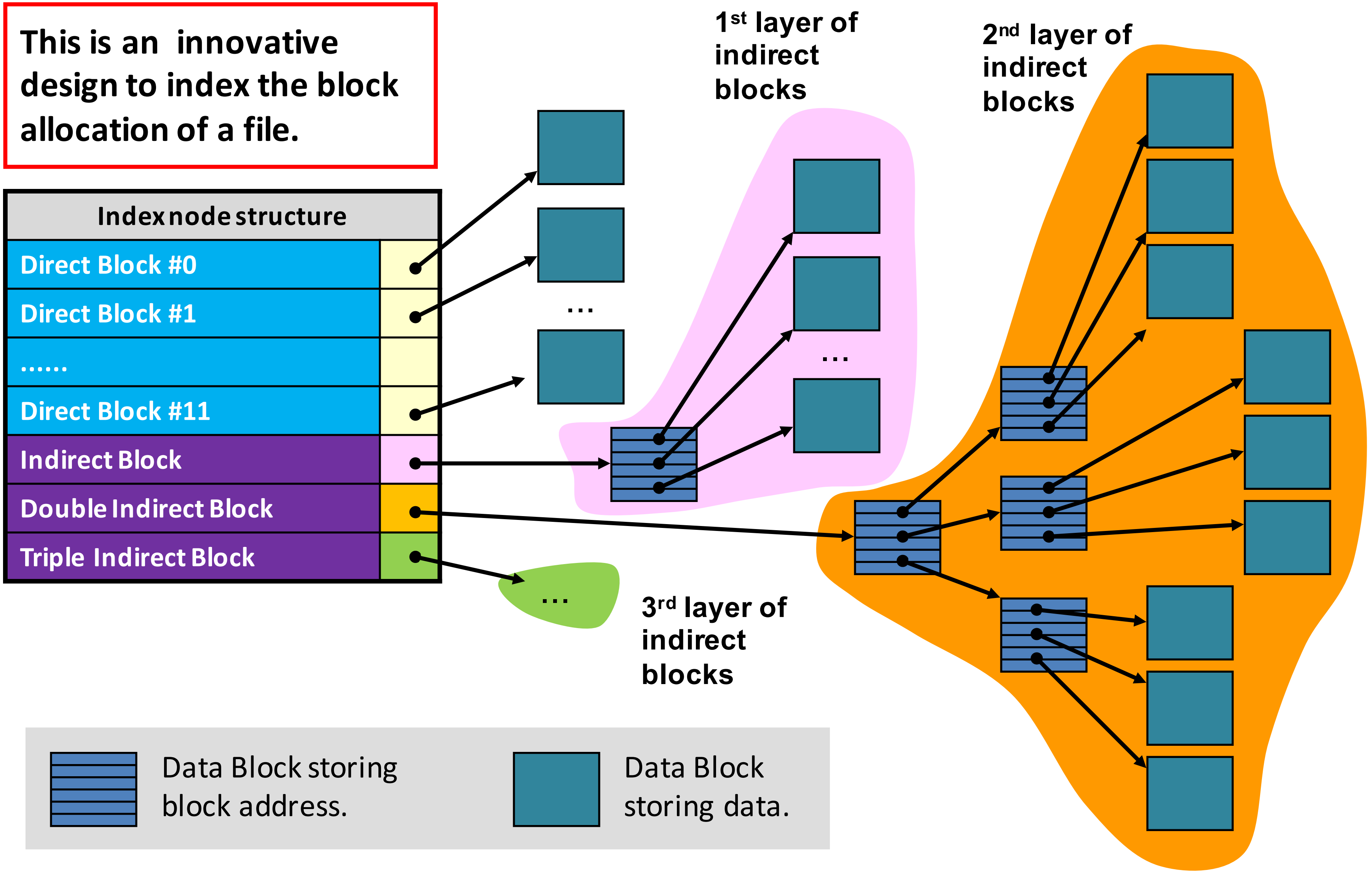
Block #	27	…	32	33
Next Block #	28	…	33	0

You know, we don’t like something that is **variable-sized**.

Index-node allocation – the core

EXTRA

This is an innovative design to index the block allocation of a file.



Index node structure
Direct Block #0
Direct Block #1
……
Direct Block #11
Indirect Block
Double Indirect Block
Triple Indirect Block

…
1st layer of indirect blocks
…
2nd layer of indirect blocks
…
3rd layer of indirect blocks

▤ Data Block storing block address.

■ Data Block storing data.

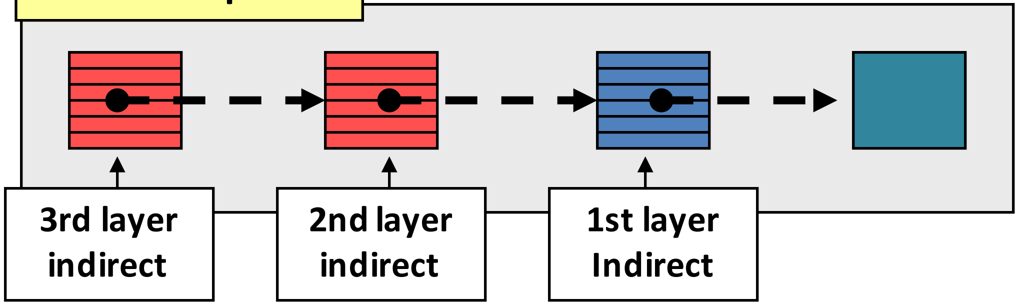
Index-node allocation – the core

EXTRA

Indirect block
Stores an array of block addresses.
An address may point to either a data block or another indirect block.
However, in a block, all the addresses are either pointing to indirect blocks or data blocks.

Data block
Stores file data.

The consequence



Keys

🟥	Indirect blocks that point to indirect blocks
🟦	Indirect blocks that point to data blocks
■	Data blocks

Index-node allocation – the file size

EXTRA

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

▤ contains “**2^x / 4**” addresses

File size = number of data blocks * 2^x

$$
\begin{aligned}
&12 \times 2^x &+\\
&2^x/4 \times 2^x &+\\
&(2^x/4)^2 \times 2^x &+\\
&(2^x/4)^3 \times 2^x &
\end{aligned}
$$

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

Index-node allocation – the file size

EXTRA

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

▤ contains “2^x / 4” addresses

File size = number of data blocks * 2^x

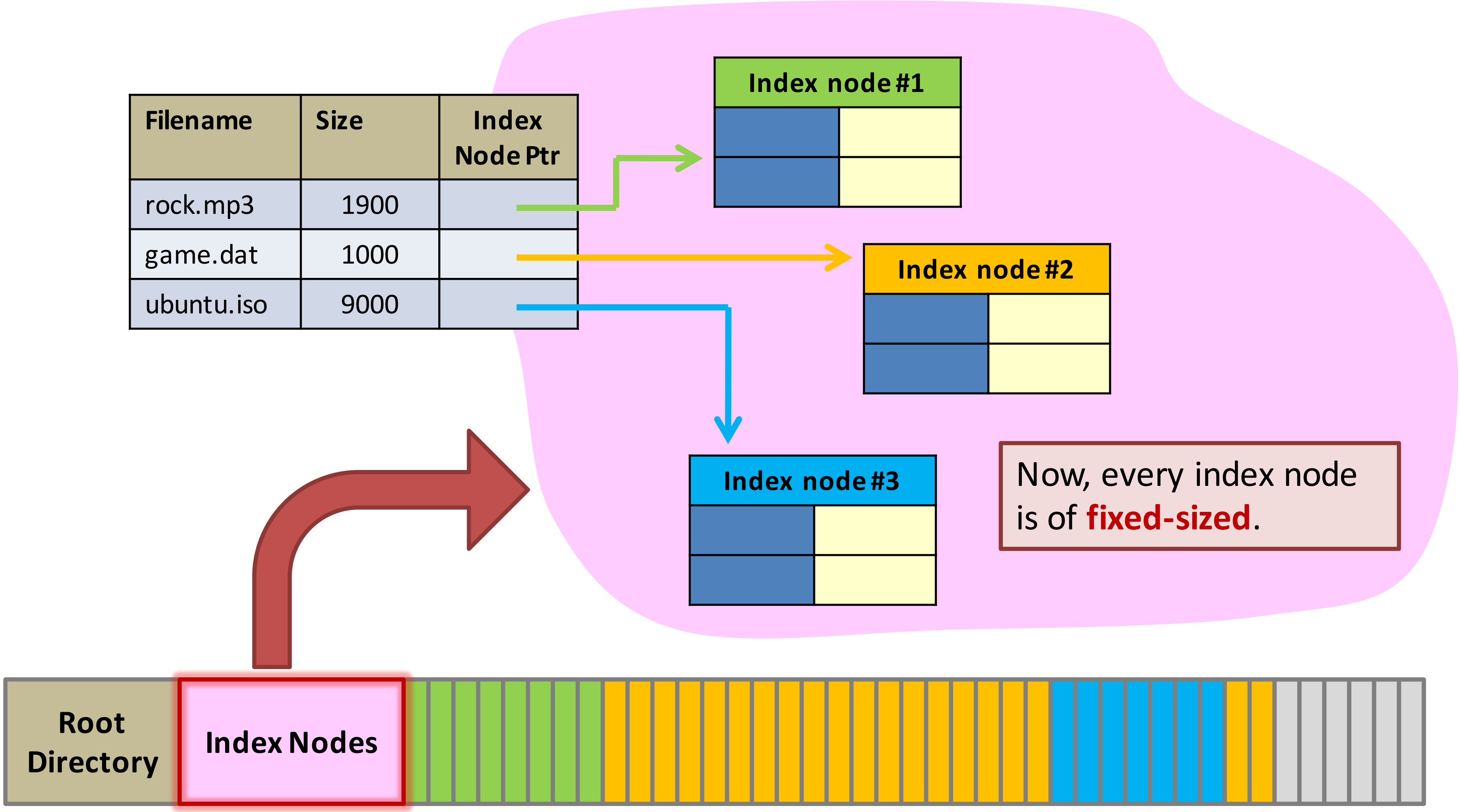
$$12 \times 2^x + 2^{2x-2} + 2^{3x-4} + \boxed{2^{4x-6}}$$

The dominating factor.

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

Index-node allocation – finishing touch

EXTRA



Filename	Size	Index Node Ptr
rock.mp3	1900	
game.dat	1000	
ubuntu.iso	9000	

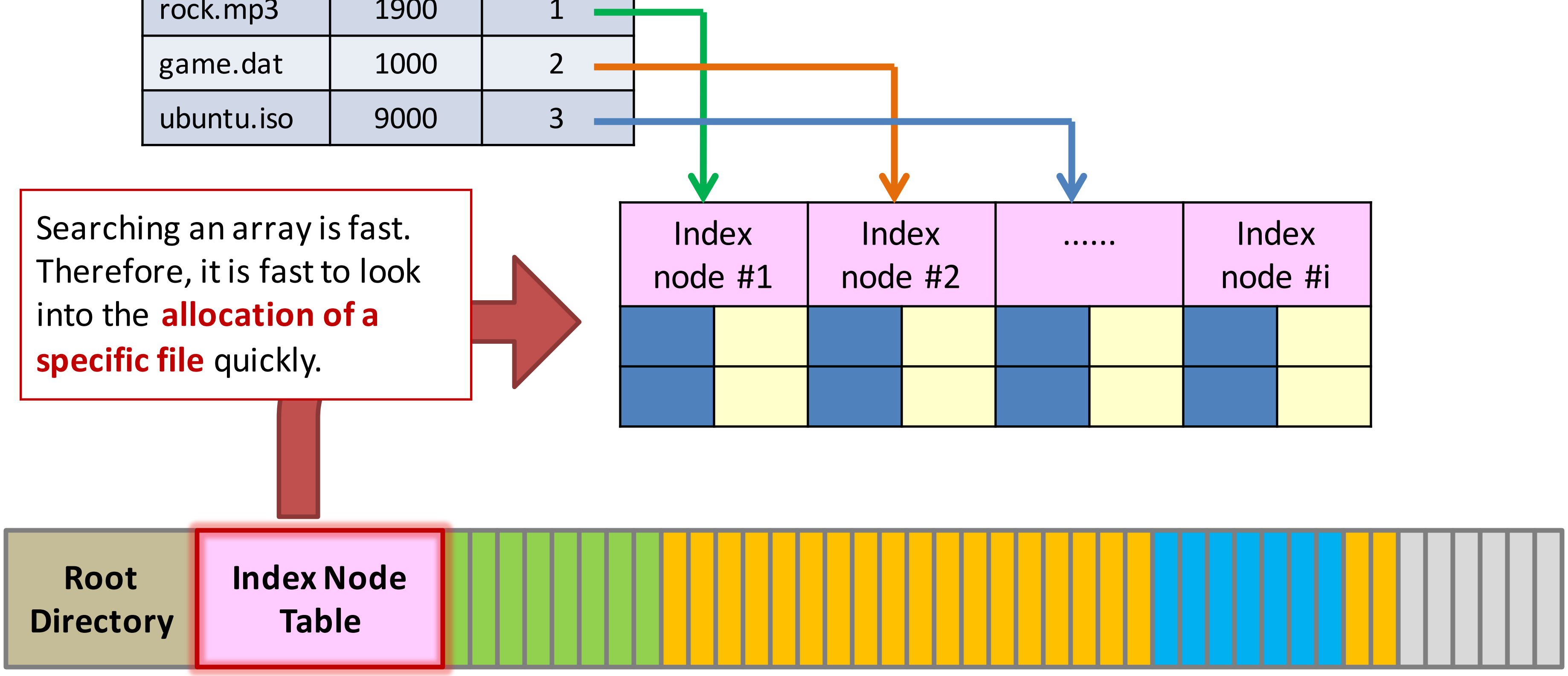
Now, every index node is of **fixed-sized**.

Index-node allocation – finishing touch

EXTRA

Now, this column stores the **index node #**.

Filename	Size	Index Node #
rock.mp3	1900	1
game.dat	1000	2
ubuntu.iso	9000	3



Searching an array is fast. Therefore, it is fast to look into the **allocation of a specific file** quickly.

Index-node allocation – overhead

EXTRA

- The index-node allocation actually uses more storage: **using space to trade for a larger file size**.
  - The indirect blocks are the **overhead**.

With 1st indirection layer	**File Size**	$12 \times 2^x + \boxed{2^{2x-2}}$
	# of Indirect Blocks	$\left(2^{x-2}\right)^0$

With 2nd indirection layer	**File Size**	$12 \times 2^x + \boxed{2^{2x-2}} + \boxed{2^{3x-4}}$
	# of Indirect Blocks	$\boxed{\left(2^{x-2}\right)^0} + \boxed{\left(2^{x-2}\right)^0 + \left(2^{x-2}\right)^1}$

Index-node allocation – overhead

EXTRA

- The index-node allocation actually uses more storage: **using space to trade for a larger file size**.
  - The indirect blocks are the **overhead**.

With 3rd indirection layer	**File Size**	$12 \times 2^x + \boxed{2^{2x-2}} + \boxed{2^{3x-4}} + \boxed{2^{4x-6}}$
	# of Indirect Blocks	$\boxed{\left(2^{x-2}\right)^0} + \boxed{\left(2^{x-2}\right)^0 + \left(2^{x-2}\right)^1} +$ $\boxed{\left(2^{x-2}\right)^0 + \left(2^{x-2}\right)^1 + \left(2^{x-2}\right)^2}$

Index-node allocation – overhead

EXTRA

- The index-node allocation actually uses more storage: **using space to trade for a larger file size**.
  - The indirect blocks are the **overhead**.

Block size	Max. # of indirect blocks	Max. overhead involved
1024 bytes = 2^{10}	approx. 2^{16}	approx. 64 MB
4096 bytes = 2^{12}	approx. 2^{20}	approx. 4 GB

Remember, they are not static and they grow/shrink with the file size.

Conclusion

EXTRA

- Different file system layouts present to you one important concept in data management: **indexing**!
  - How can you retrieve something in a fast manner?
  - How can you allocate space in a fast manner?

- Other issues including:
  - How can you reduce the overhead per data blocks?
  - How can you delete a file efficiently?

Interesting but missed topics…

EXTRA

- File system consistency:
  - It is about how to **detect** and how to **recover inconsistency in a file system**.
  - But, why does inconsistency exist?
    - Power failure;
    - Pressing reset button accidentally; etc.

- The **file system journal** is the current, state-of-the-art practice.

Interesting but missed topics…

EXTRA

You write down all the tasks assigned to you into a log book.

Task list:
- Go to Mongkok and buy boss a DC.
- Pick up boss' friend.
- Drive his friend back to his home.
- Buy boss a coffee when I return.

Your boss orders
you to do a set of tasks!

Interesting but missed topics…

EXTRA

Task list:

- ~~Go to Mongkok and buy boss a DC.~~
- Pick up boss’ friend.
- Drive his friend back to his home.
- Buy boss a coffee when I return.

You cross out a task when it is completed.

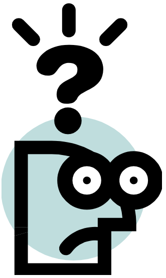
Interesting but missed topics…

EXTRA

Unfortunately, a car accident happens!

You lost all your memory!!

The log book comes in handy!



Your boss sends your colleague to finish your job. But, he doesn’t know about your progress.

Worse, your boss has forgotten what are the tasks given to you!

Interesting but missed topics…

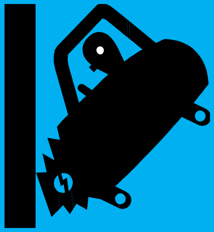
EXTRA

User Program

FS operations invoked by the user program

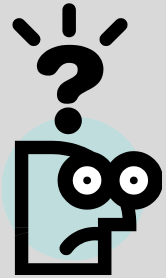
Task list:
- Go to Mongkok and buy boss a DC.
- Pick up boss' friend.
- Drive his friend back to his home.
- Buy boss a coffee when I return.

OS



System crash!
All memory lost!

File system
recovery tool



The journal!

Interesting but missed topics…

EXTRA

- #2 Kernel Buffer Cache.
  - The kernel will keep a set of copies of the read/written data blocks.
  - The space that stores those blocks are called the **buffer cache**.
  - It is used for **reducing the time** in accessing those blocks in the near future.

- For the writing performance…
  - the data is not required to write to the device immediately…
  - So, the user will "***feel***" that **the write operation is quick**!