Operating Systems

Prof. Yongkun Li
中国科大-计算机学院 教授
http://staff.ustc.edu.cn/~ykli

Chapter 9, part2
File System Layout

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?

File content & attributes

Directory

Allocated Space

Free Space

Things need to be stored.

Outline

- We briefly introduce the evolution of the file system layout:
  - From a dummy way to advanced ways.
  - The pros and cons are covered.

- We begin to look at some details of the FAT file system and EXT file system

How to store data?

- Consider the following case:
  - You are going to design the layout of a FS.
  - You are given the freedom to choose the locations to store files, including directory files.
  - How will you organize the data?

How to store data?

- Some (basic) rules are required:
  - Every data written to the device must be able to be retrieved.
    - Would you use the FS that will lose data randomly?
  - Every FS operation should be done as efficient as possible.
    - Would you use the FS if it takes a minute to retrieve several bytes of data?
  - When a file is removed, the FS should free the corresponding space.
    - Would you use the FS if it cannot free any occupied space?

0

100GB

File System Layout

Trial 1.0

The Contiguous Allocation

Trial 1.0 – the basics

- Just like a book!

```
Table of content

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

Book VS Trial #1	
Book	**Trial #1**
Chapter	Filename
Starting Page	Starting Address
NIL	Ending Address

Trial 1.0 – the basics

- Just like a book!

Book VS Trial #1	
Book	**Trial #1**
Chapter	Filename
Starting Page	Starting Address
NIL	Ending Address

Suppose we have 3 files to store

```
rock.mp3
sweet.jpg
same.exe
```

We do not consider the directory structure at this moment

Like a book, we need to some space to store the **table of content,** which records the filename and the (starting and ending) addresses of the file content.

Trial 1.0 – the basics

- Just like a book!

Book VS Trial #1	
Book	**Trial #1**
Chapter	Filename
Starting Page	Starting Address
NIL	Ending Address

The table of content!

Filename	Starting Address	Ending Address
rock.mp3	0	2000
sweet.jpg	2001	3456
game.exe	5000	5678

File attributes

Trial 1.0 – the basics

- Just like a book!

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

The table of content!

Filename	Starting Address	Ending Address
rock.mp3	0	2000
sweet.jpg	2001	3456
game.exe	5000	5678

File attributes

Trial 1.0 – the basics

You can locate files easily (with a directory structure).

But, can you locate the **allocated space** and the **free space** in a short period of time?

Filename	Starting Address	Ending Address
rock.mp3	0	2000
sweet.jpg	2001	3456
game.exe	5000	5678

Free space is here.

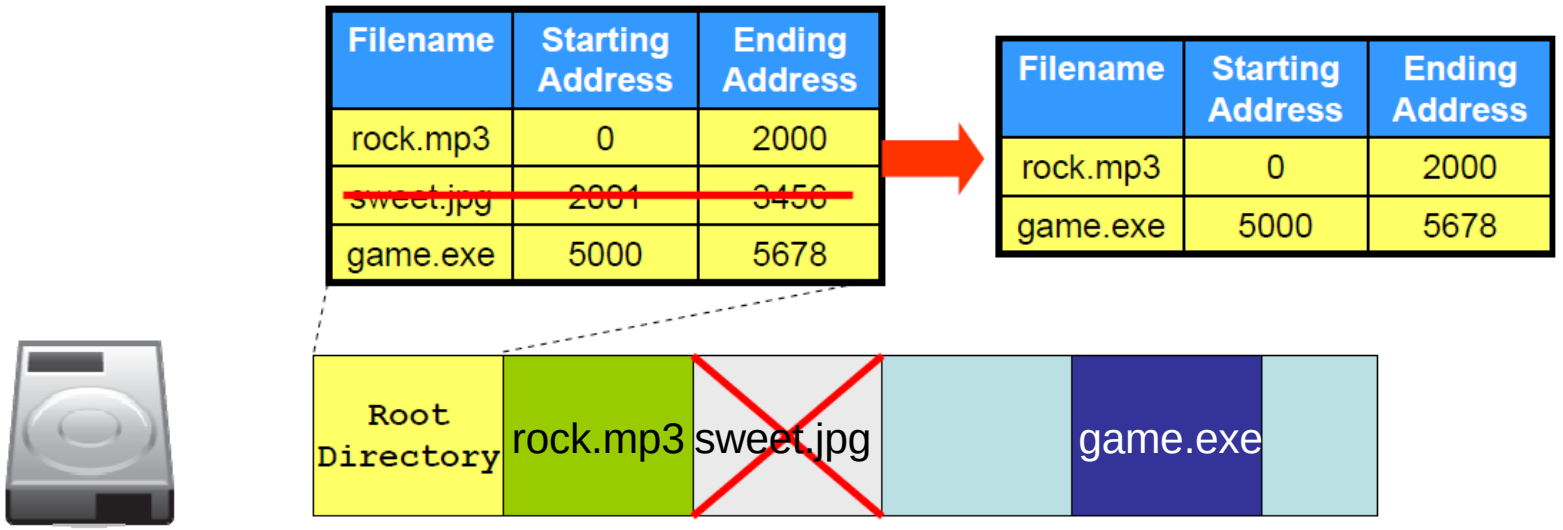
But, it needs an O(n) search, where n is the total number of files.

What if the disk is large and the files are small?

Trial 1.0 – the basics

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

Yet, how about file creation?



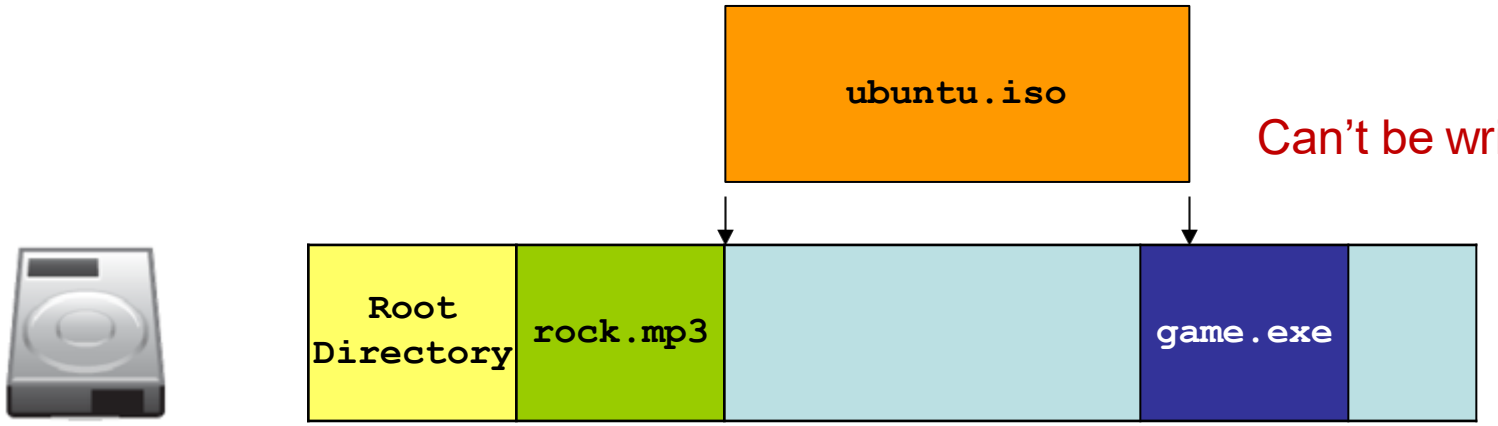
Trial 1.0 – the bad #1

- Suppose we need to write a new, but large file?

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

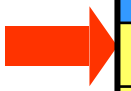
Any solution?



Trial 1.0 – the bad #1

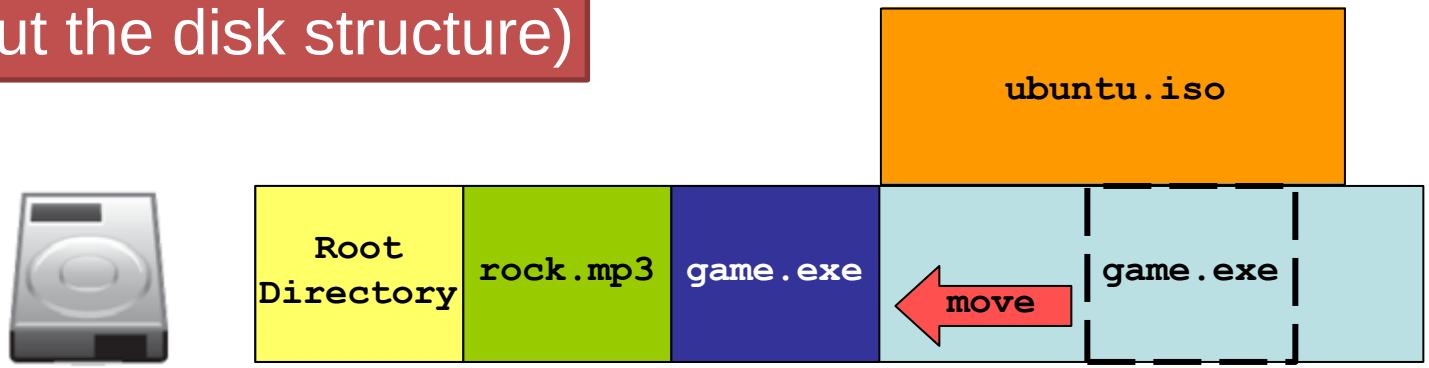
- The **defragmentation process** may help.

Filename	Starting Address	Ending Address
rock.mp3	0	2000
game.exe	5000	5678



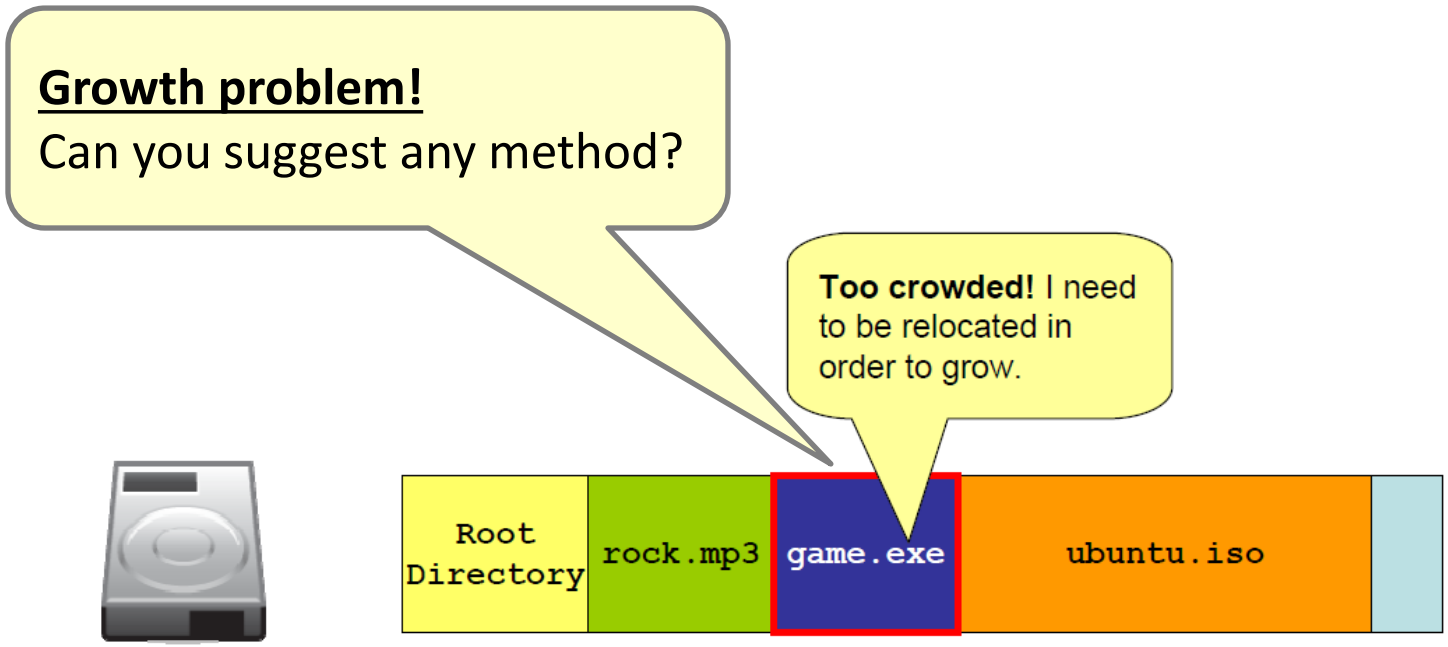
Filename	Starting Address	Ending Address
rock.mp3	0	2000
game.exe	**2001**	**2679**
ubuntu.iso	2680	6000

Very expensive (think about the disk structure)



Trial 1.0 – the bad #2

- Comment:
  - Also, the growth problem…there is no space for files to grow.



Trial 1.0 – the reality

- This kind of file systems has a name called the **contiguous allocation**.

- This kind of file system is not totally useless…
  - The suitable storage device is something that is…
  - **read-only** (just like a book)

Trial 1.0 – the reality

- Can you think of any real life example?
  - 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.
  - ISO9660.

File System Layout

Trial 2.0

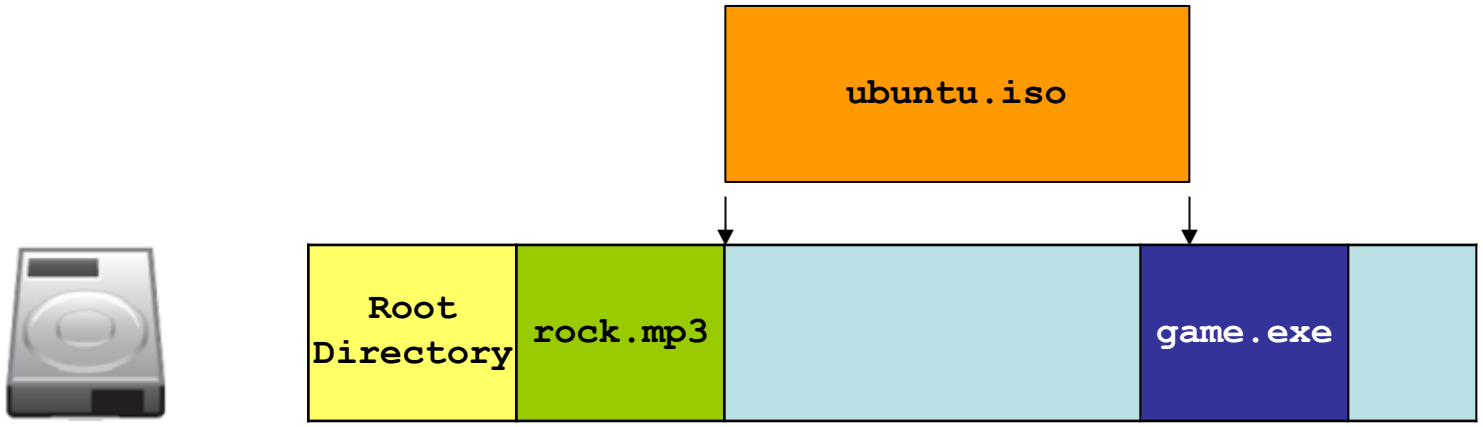
The Linked List Allocation

From Trial 1.0 to Trial 2.0…

- Lessons learned from Trial 1.0:
  - **File Size Growth**:
    - Can we let every file to grow without paying an experience overhead?
  - **External fragmentation**:
    - Can we reduce its damage?
- One goal
  - **To avoid allocating space in a contiguous manner!**

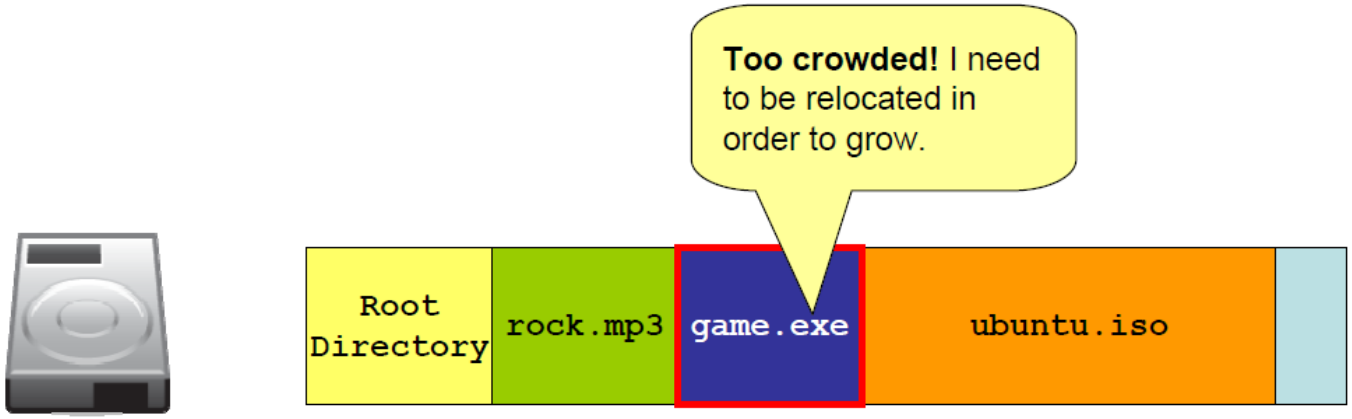
Trial 2.0 – the basics

- How?
  - The first undesirable case in trial 1.0 is to **write a large file** (as it may fail or need defragmentation)
  - So, can we write small files/units only?
    - For large files, let us break them into small pieces…



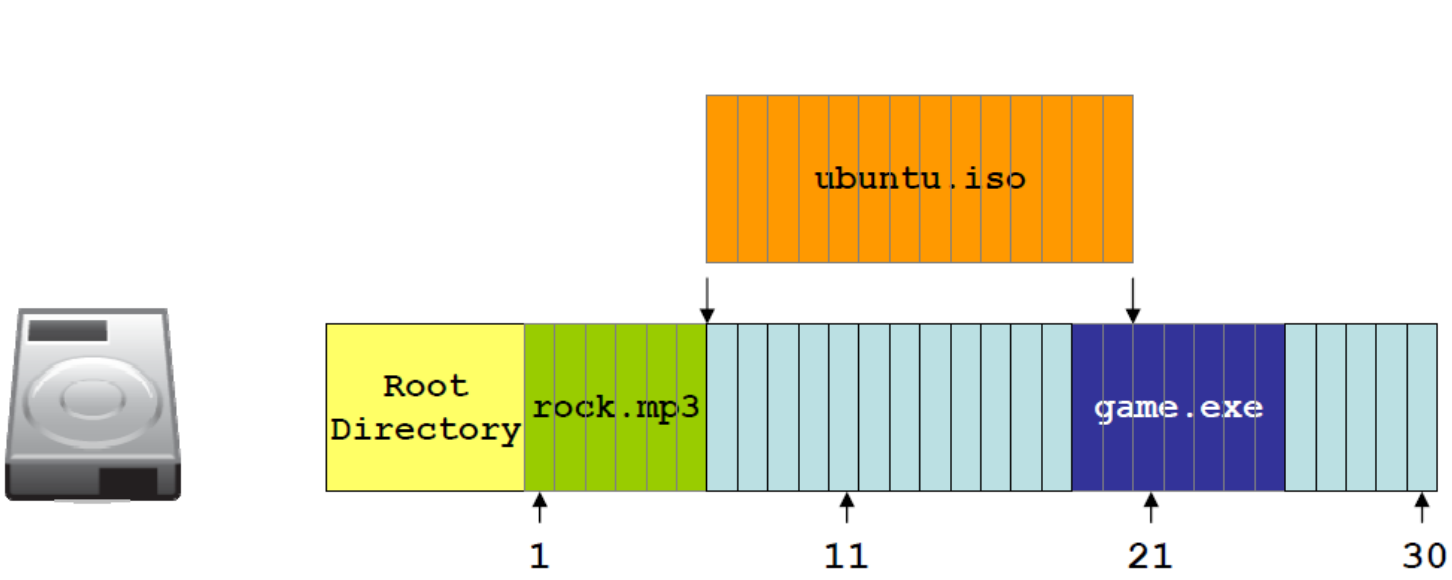
Trial 2.0 – the basics

- How?
  - The second undesirable case in trial 1.0 is when **file grows** (as it needs reallocation)
  - So, how can we support dynamic growth?
    - Let's borrow the idea from the linked list…



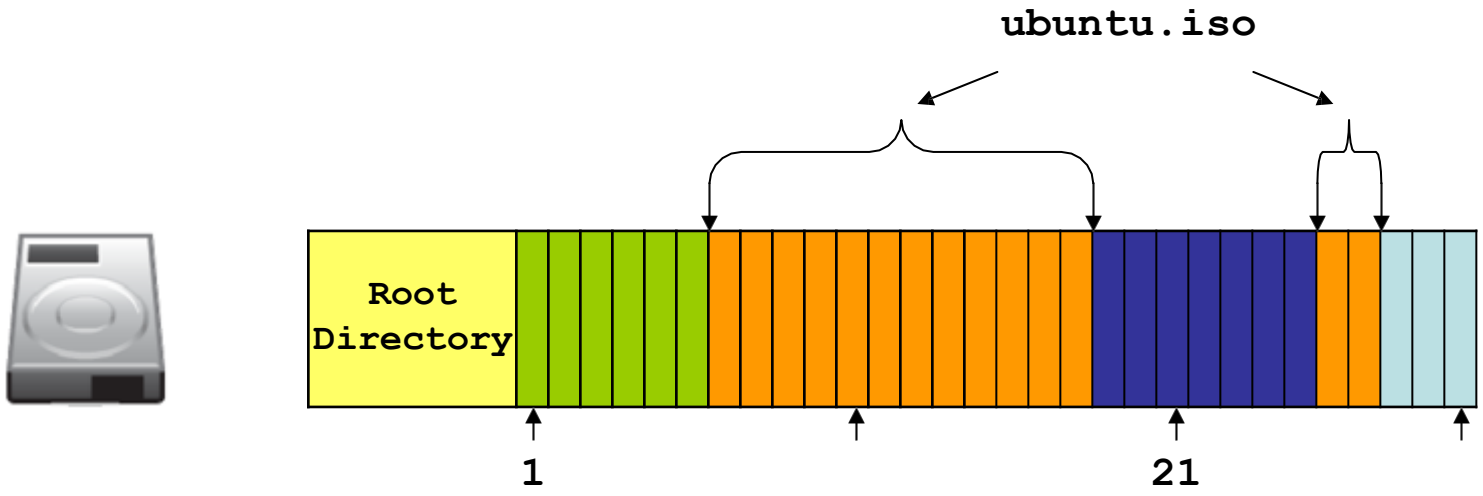
Trial 2.0 – the basics

- Linked list allocation…
  - Step (1): Chop the storage device into **equal-sized blocks**.



Trial 2.0 – the basics

- Linked list allocation…
  - Step (2): Fill the new file into the empty space in a **block-by-block** manner.



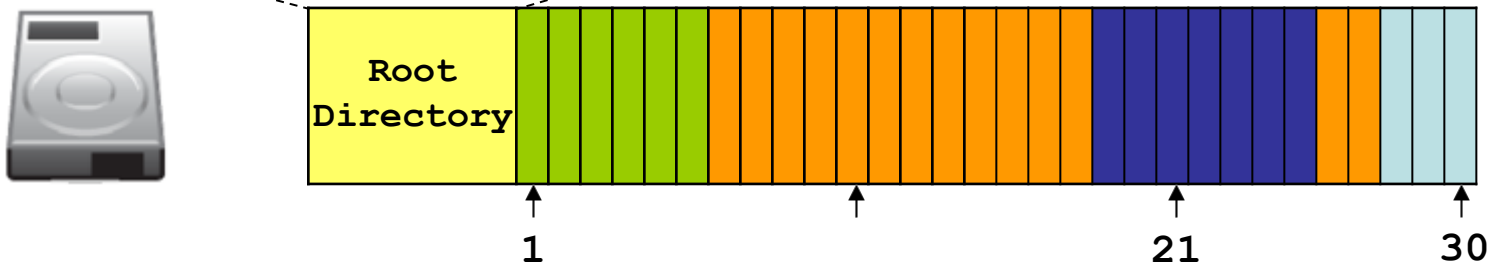
Trial 2.0 – the basics

- Linked list allocation…
  - Step (3): The root directory…
    - becomes strange/complicated.

Since a directory file is an array, it is difficult to **pretend** to be a linked list….

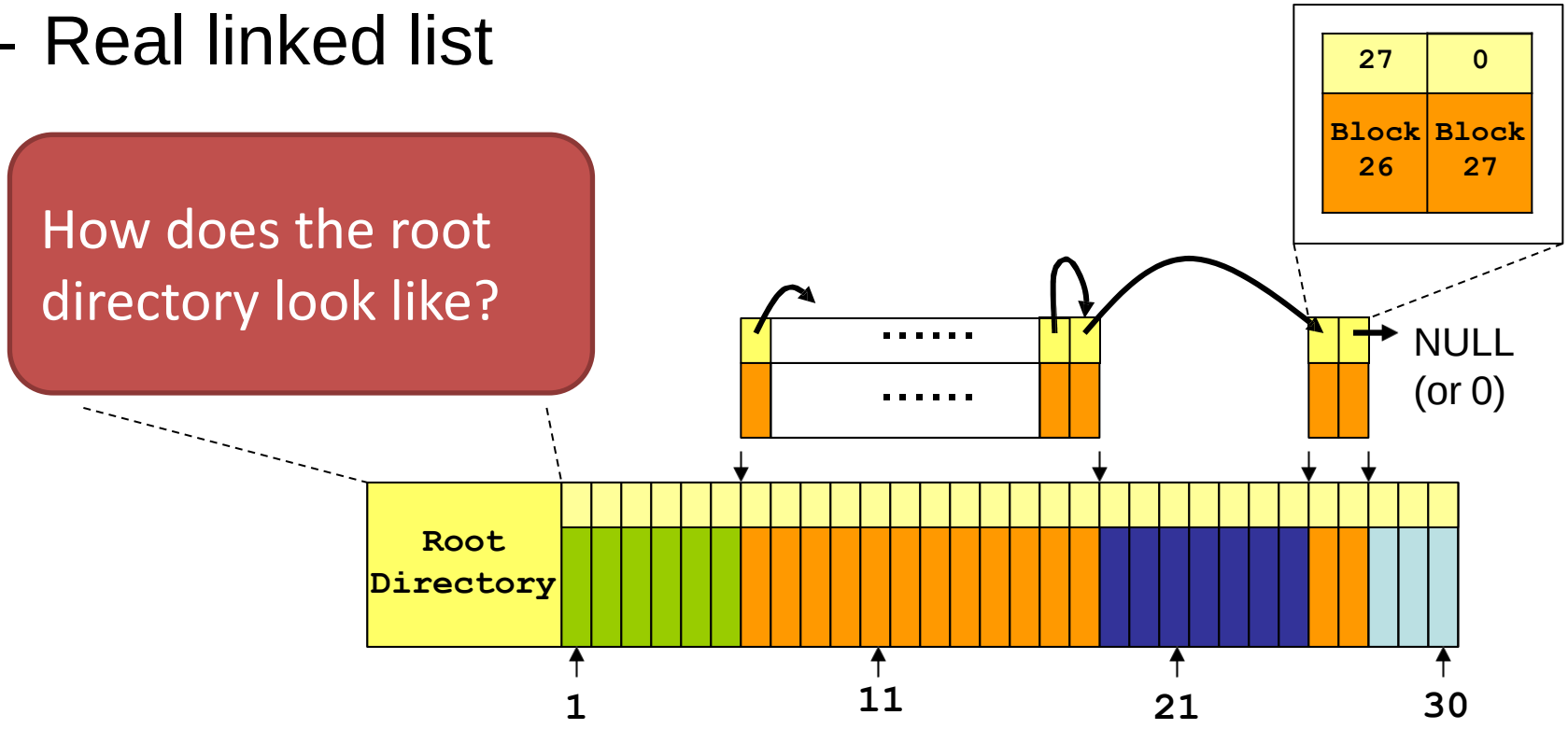
Filename	Sequence of Block #	Sequence of Block #
rock.mp3	1-6	NULL
game.exe	19-25	NULL
ubuntu.iso	**7-18**	**26-27**

Can we have a better solution to optimize the directory?



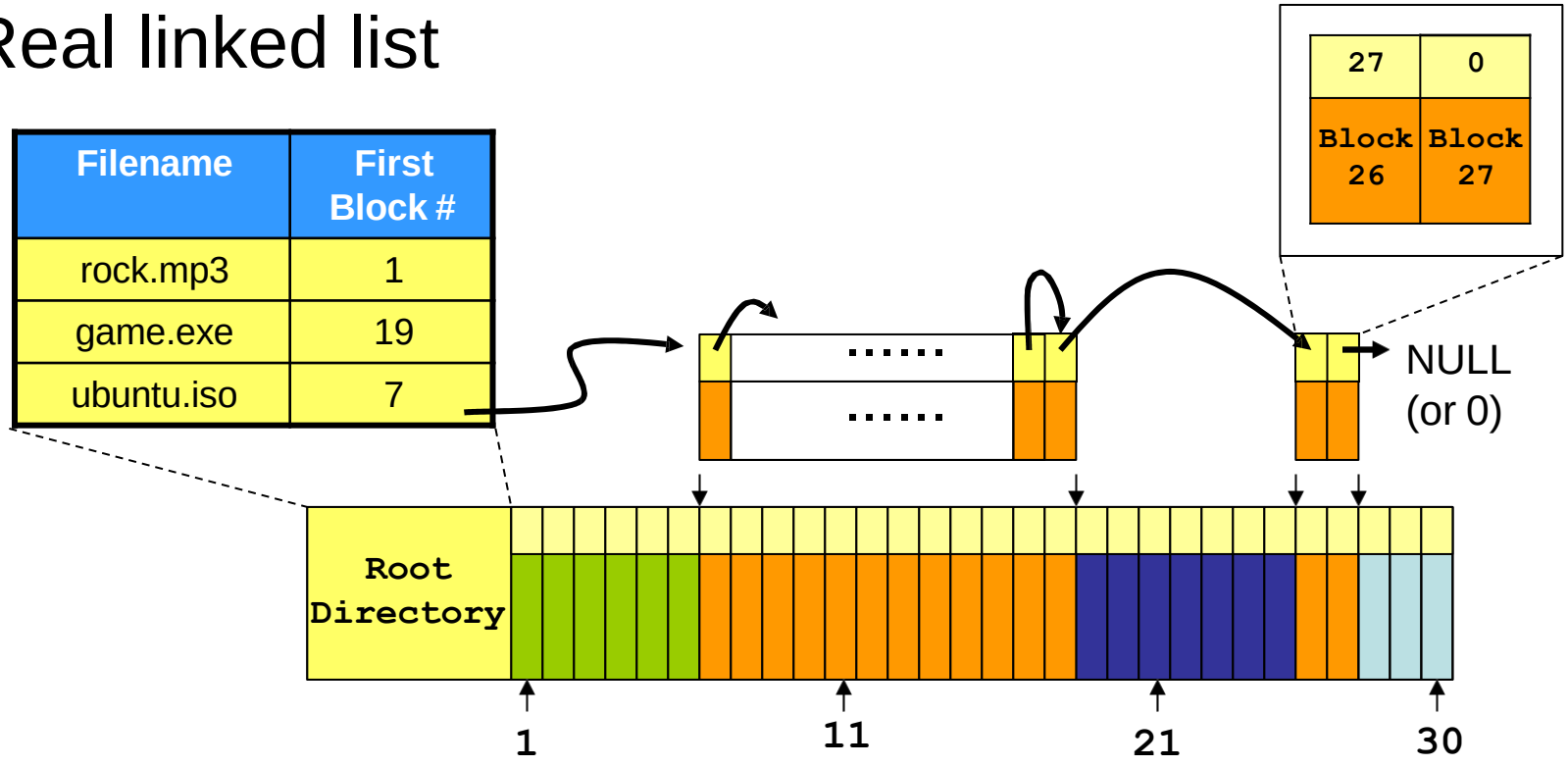
Trial 2.1 – the linked list

- Let's borrow 4 bytes from each block.
  - To write **the block # of the next block** into the first 4 bytes of each block.
  - Real linked list



Trial 2.1 – the linked list

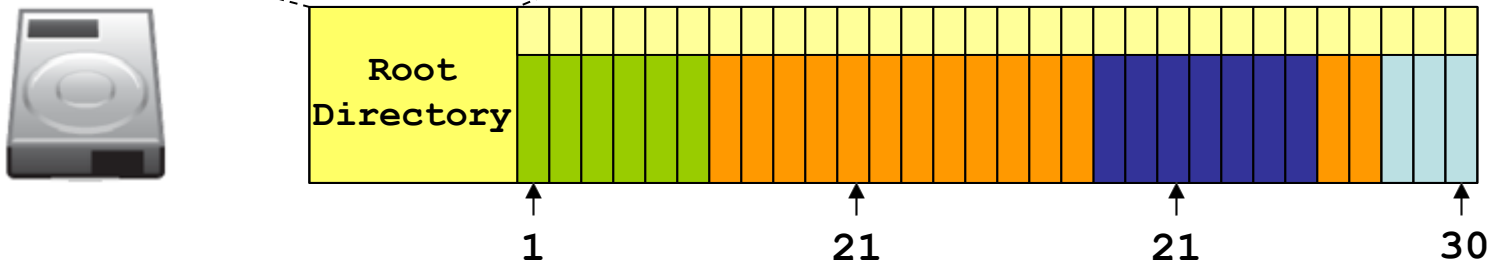
- Let's borrow 4 bytes from each block.
  - To write **the block # of the next block** into the first 4 bytes of each block.
  - Real linked list



Trial 2.1 – the file size

- Note that we need the **file size stored in the root directory** because…
  - The last block of a file **may not be fully filled**.

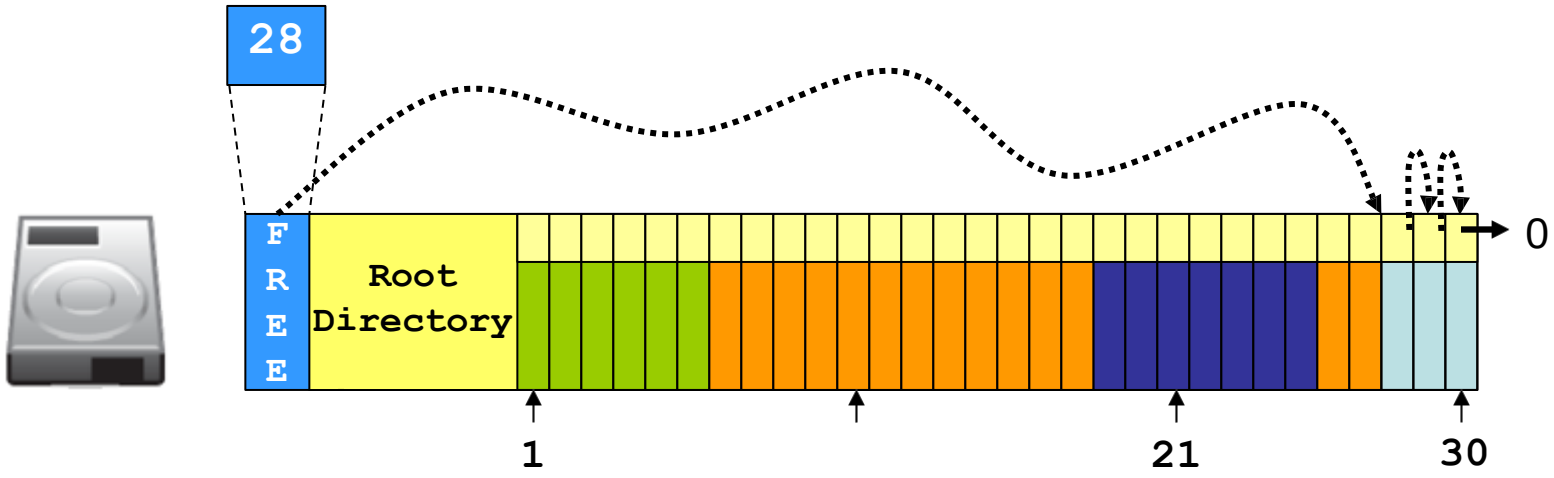
Filename	First Block #	File Size
rock.mp3	1	600M
game.exe	19	2000M
ubuntu.iso	7	700M



Trial 2.1 – the free space

- One more thing: free space management.
  - Extra data is needed to maintain a free list.

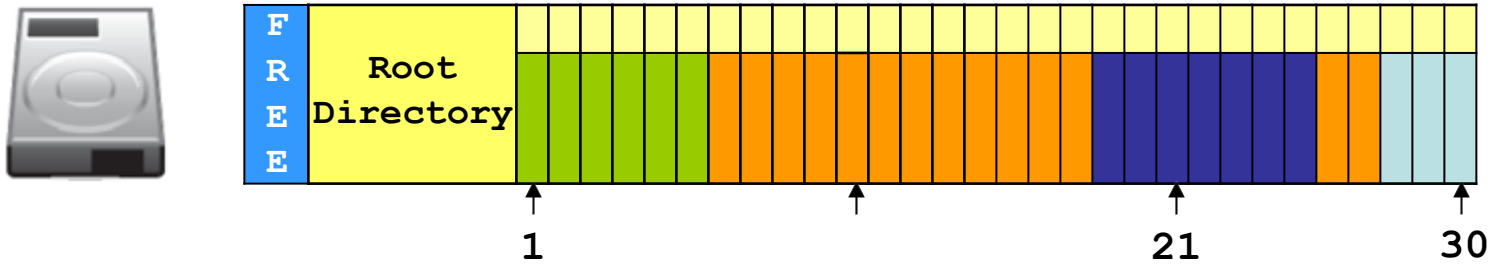
We can also maintain the free blocks as a linked list, too.



Trial 2.1 – the good

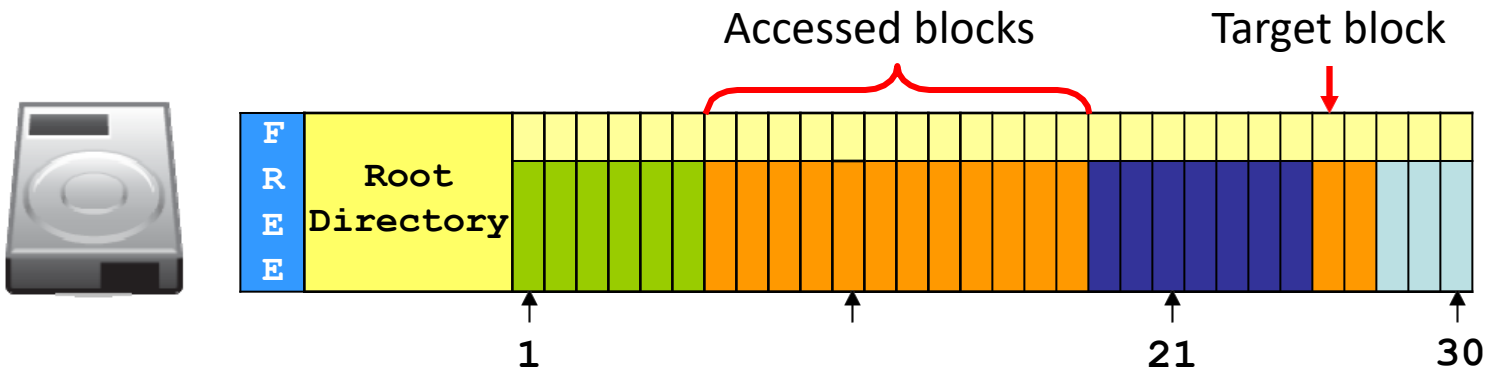
- Pros:

External fragmentation problem is solved.	Files can grow and shrink freely.	Free block management is easy to implement.



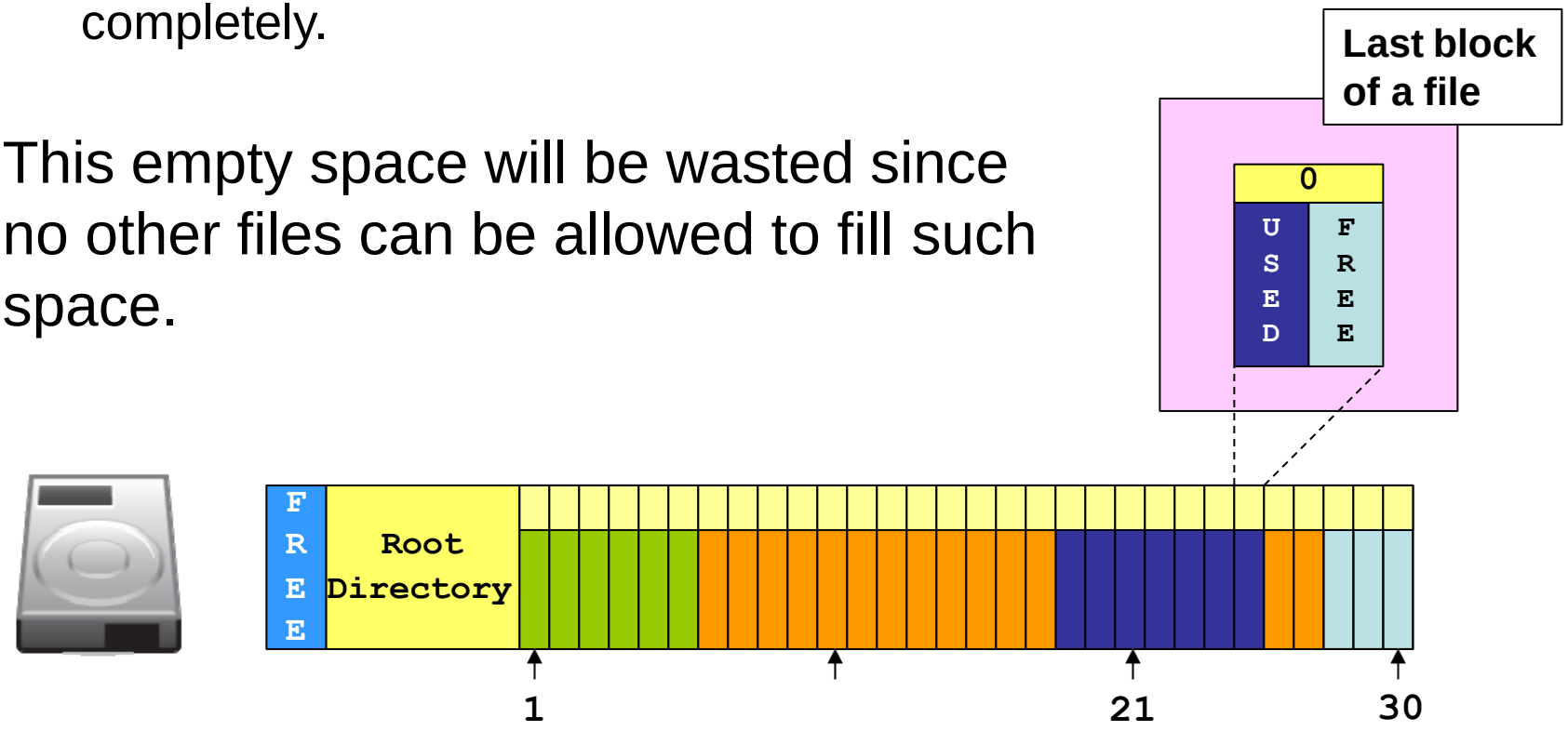
Trial 2.1 – the bad #1

- Cons:
  - **Random access performance problem**.
    - The random access mode is to access a file at random locations.
  - The OS needs to access a series of blocks before it can access an arbitrary block.
    - Worst case: **O(n) number of I/O accesses**, where n is the number of blocks of the file.



Trial 2.1 – the bad #2

- Cons (recall why we record file size?):
  - **Internal Fragmentation**.
    - A file is not always a multiple of the block size
    - The last block of a file may not be fill completely.
  - This empty space will be wasted since no other files can be allowed to fill such space.



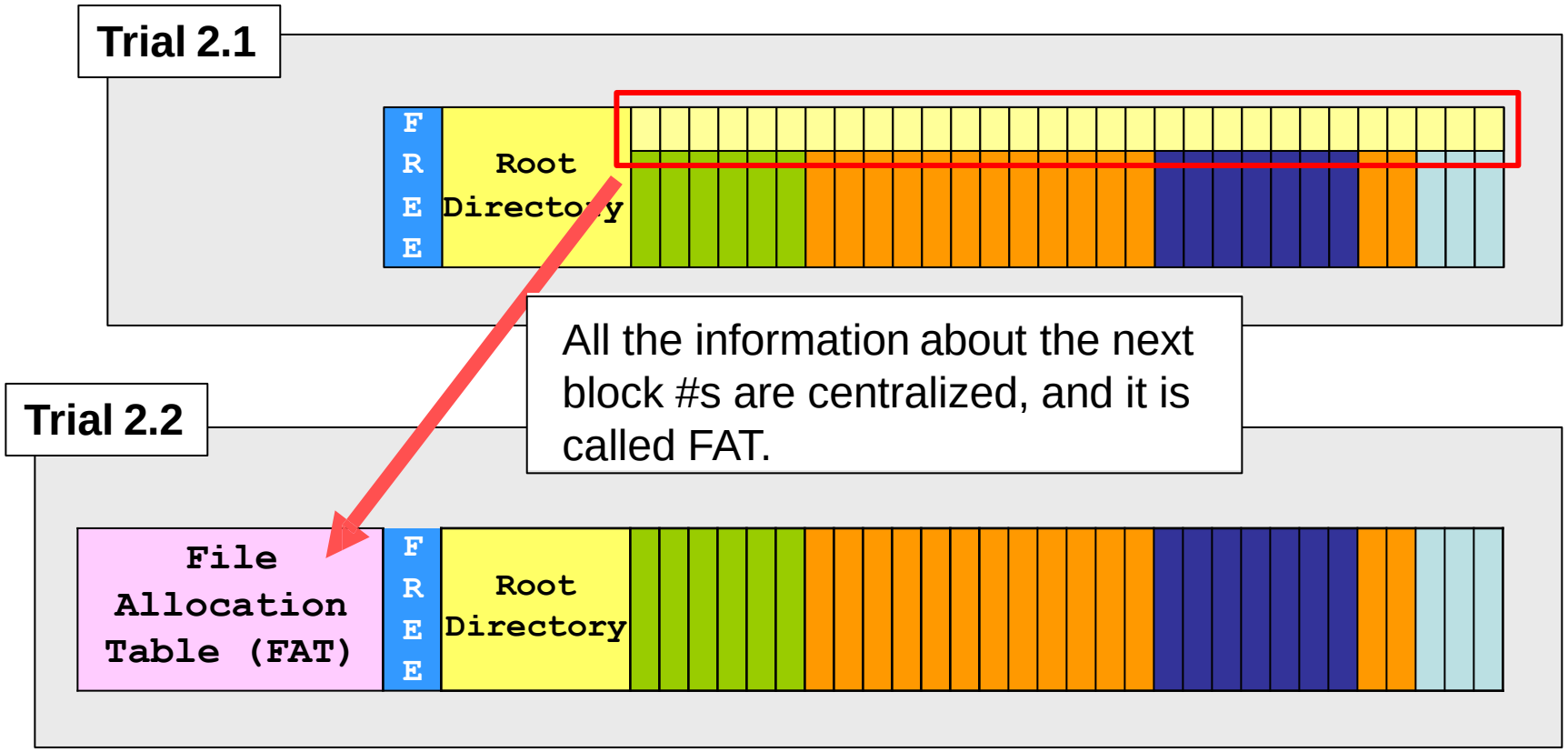
From Trial 2.1 to Trial 2.2

- Can we further improve?
  - We know that the internal fragmentation problem is here to stay.
  - How about the random access problem?
    - We are very wrong at the very beginning…decentralized next block location

The information about the next block should be centralized

Trial 2.2 – the FAT

- The only difference between 2.1 and 2.2…

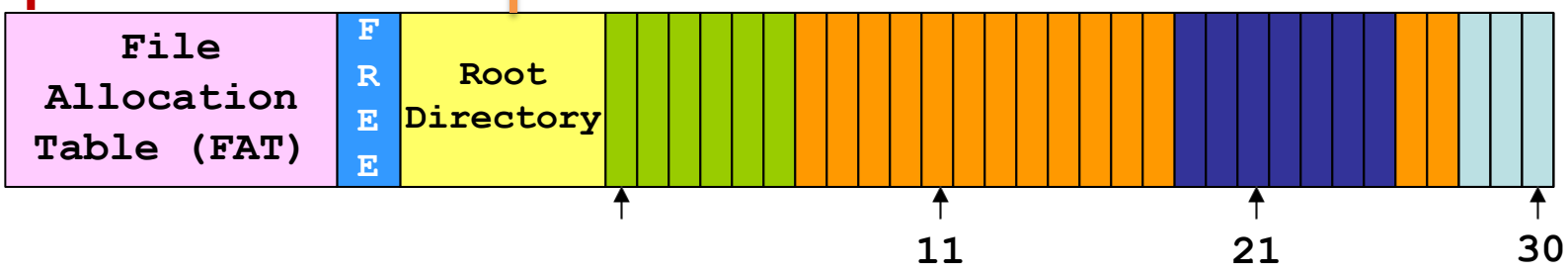


Trial 2.2 – the FAT implementation

Task: read “ubuntu.iso” sequentially.

Filename	First Block #
rock.mp3	1
game.exe	19
ubuntu.iso	7

Block #	1	…	6	7	…	18	19	…	25	26	27	28	29	30
Next Block #	2	…	0	8	…	26	20	…	0	27	0	29	30	0



Trial 2.2 – the FAT

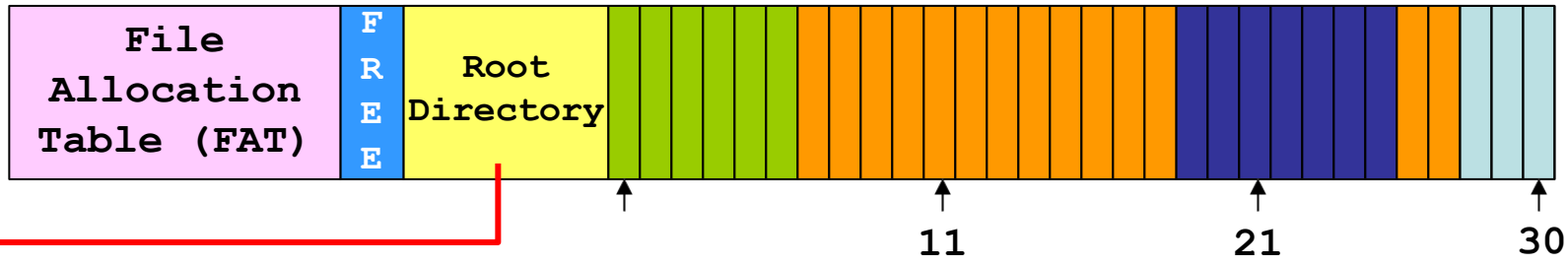
Task: read "ubuntu.iso" sequentially.

Step (1). Look for the first block # of the file.

Step (1)

Filename	First Block #
rock.mp3	1
game.exe	19
ubuntu.iso	7

Block #	1	…	6	7	…	18	19	…	25	26	27	28	29	30
Next Block #	2	…	0	8	…	26	20	…	0	27	0	29	30	0



Trial 2.2 – the FAT

Task: read "ubuntu.iso" sequentially.

Step (2). Read the file allocation table to determine the location of the next block.

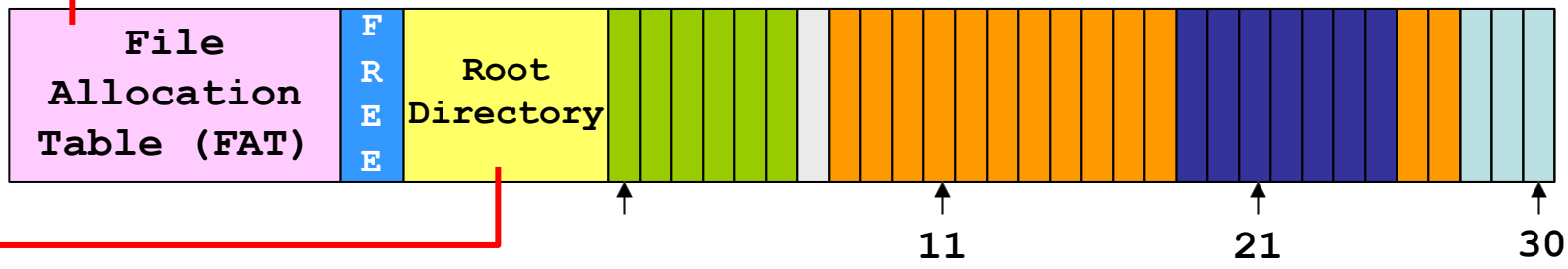
Step (1)

Filename	First Block #
rock.mp3	1
game.exe	19
ubuntu.iso	7

The next block of 7 is 8.

Block #	1	…	6	7	…	18	19	…	25	26	27	28	29	30
Next Block #	2	…	0	8	…	26	20	…	0	27	0	29	30	0

Step (2)



Trial 2.2 – the FAT

Task: read "ubuntu.iso" sequentially.

Step (2). Read the file allocation table to determine the location of the next block.

Note that the next block is not necessarily the adjacent one.

Step (1)

Filename	First Block #
rock.mp3	1
game.exe	19
ubuntu.iso	7

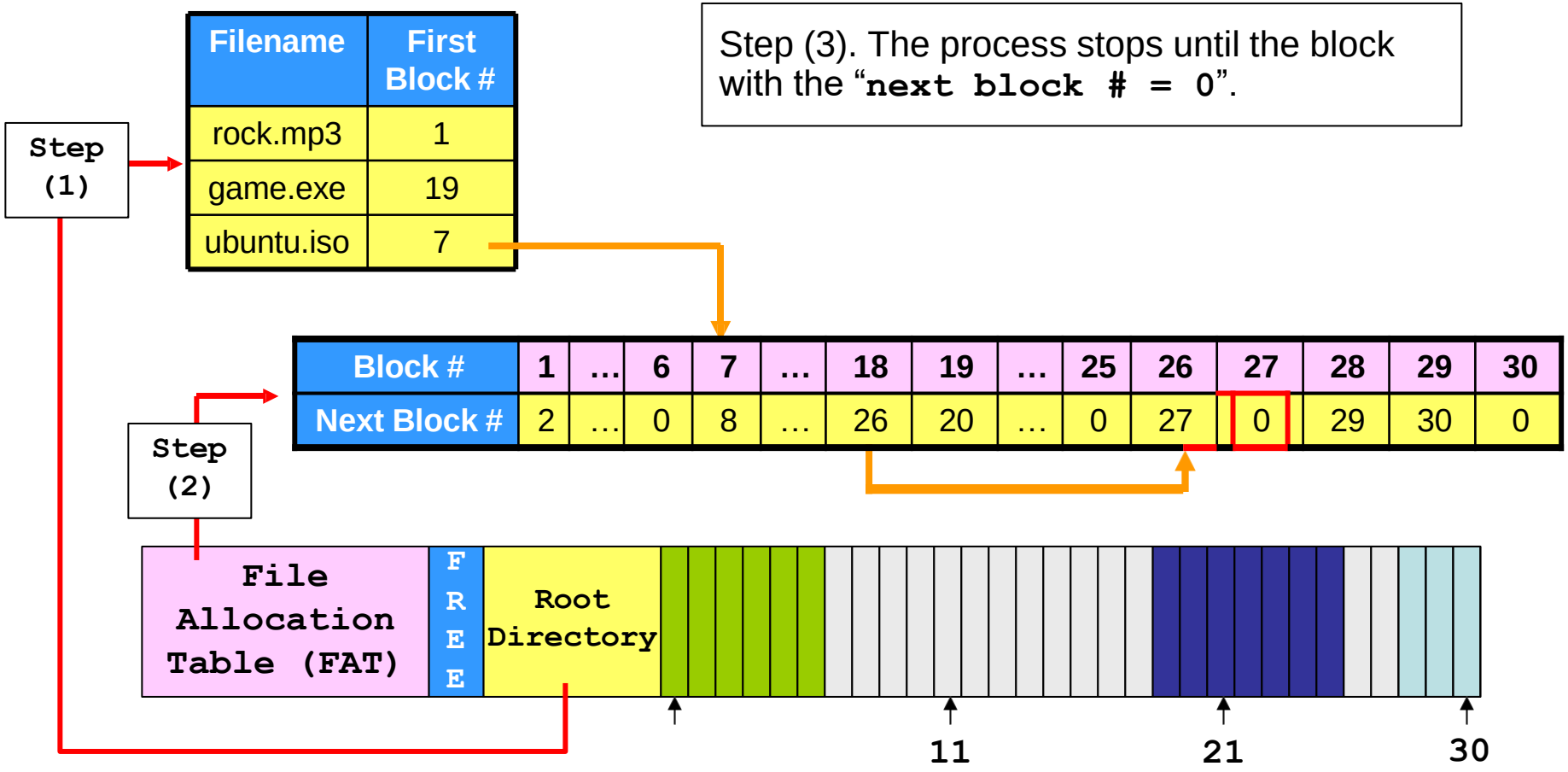
Step (2)

Block #	1	…	6	7	…	18	19	…	25	26	27	28	29	30
Next Block #	2	…	0	8	…	26	20	…	0	27	0	29	30	0

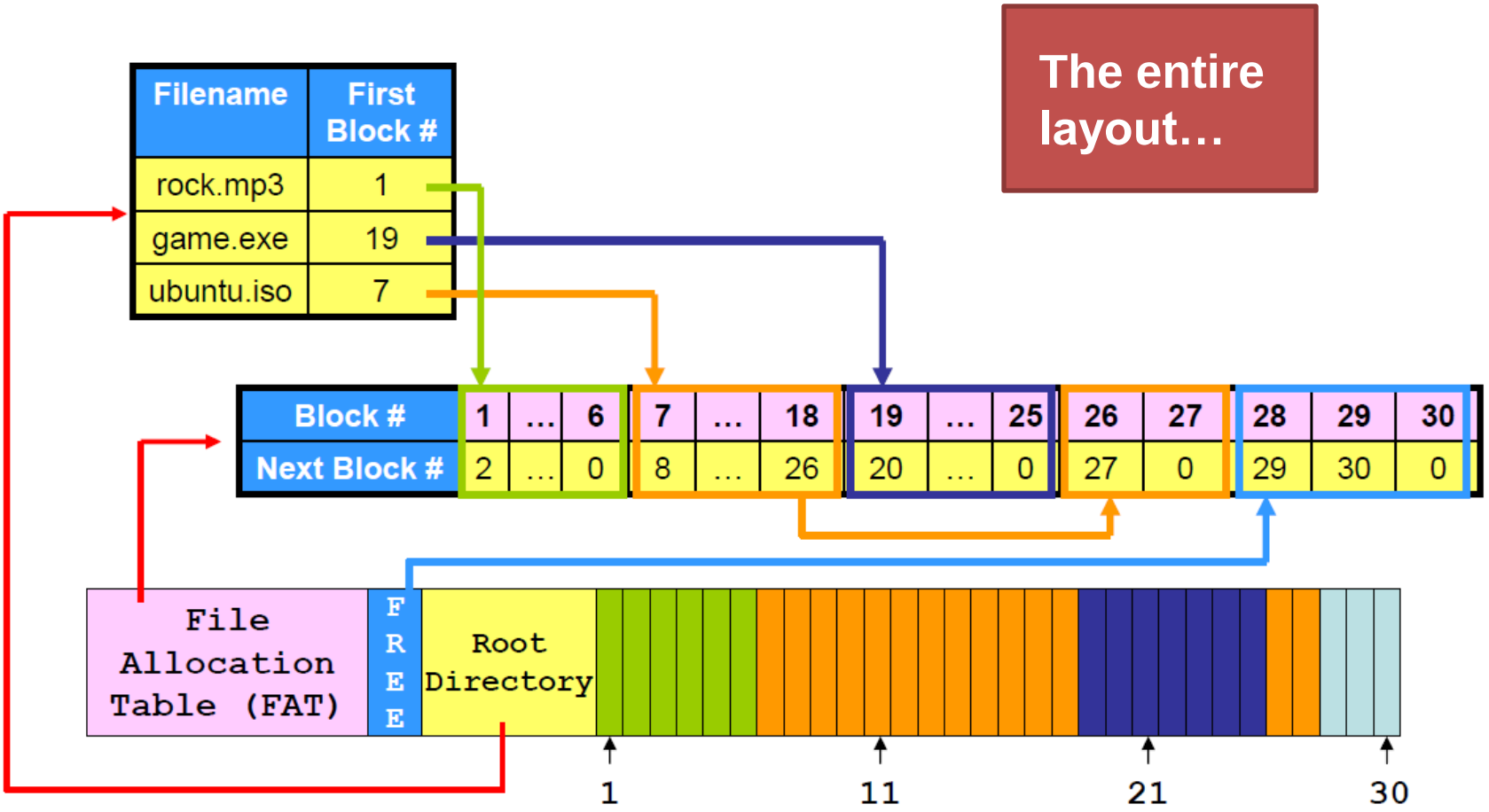
Trial 2.2 – the FAT

Task: read "ubuntu.iso" sequentially.

Step (3). The process stops until the block with the "`next block # = 0`".



Trial 2.2 – the FAT



Trial 2.2 – the lookup

- A point to look into:
  - Centralizing the data does not mean that the *random access problem* will be gone automatically, unless…
  - the file allocation table is presented as **an array**.

File Allocation Table

Block #	1	…	6	7	…	18	19	…	25	26	27	28	29	30
Next Block #	2	…	0	8	…	26	20	…	0	27	0	29	30	0

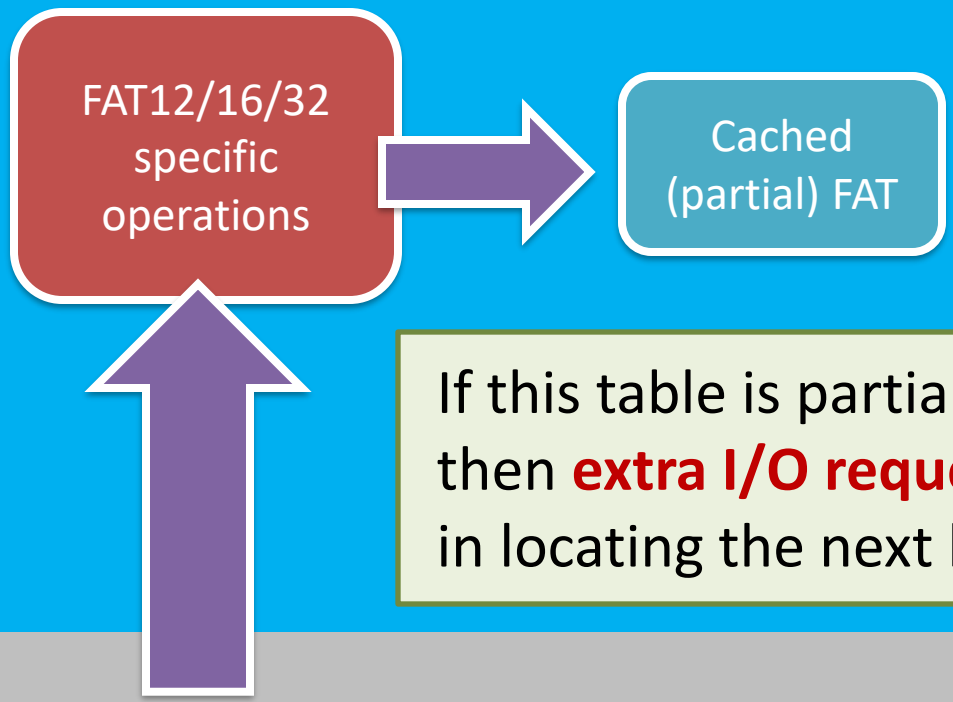
I know the starting position.

I know the width.

So, going to an arbitrary location is as simple as **doing a pointer addition operation**.

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

Trial 2.2 – the lookup



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

Block #	1	…	6	7	…	18	19	…	25	26	27	28	29	30
Next Block #	2	…	0	8	…	26	20	…	0	27	0	29	30	0

File Allocation Table (FAT)

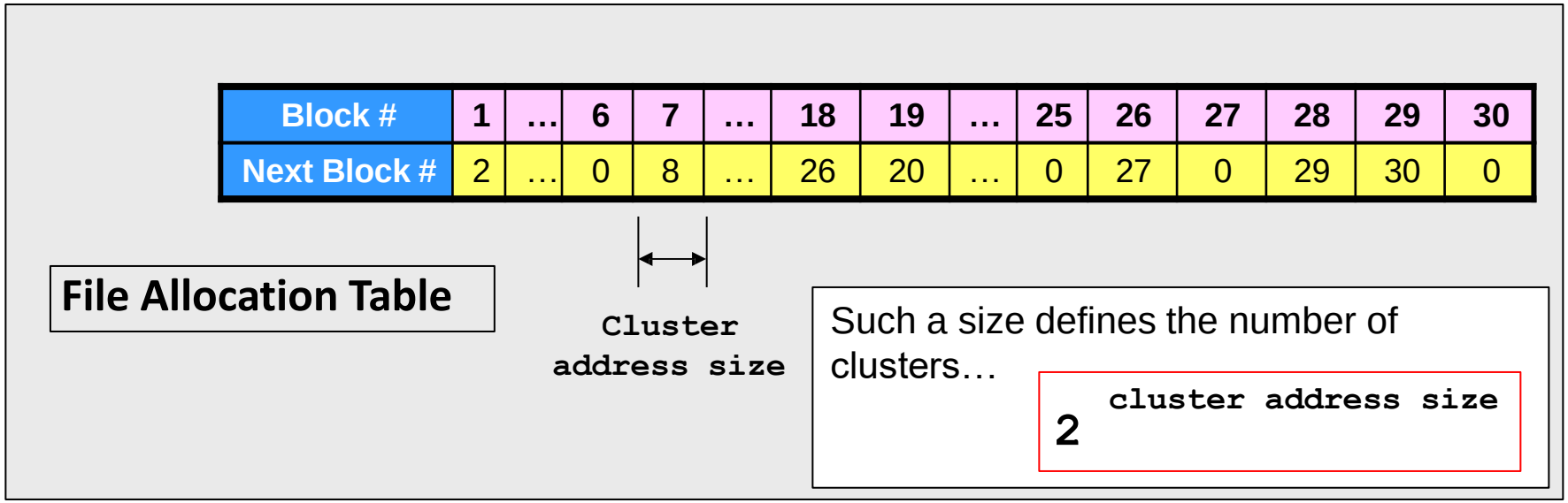
Trial 2.2 and the reality

- Every file system supported by MSDOS and the Windows family is implementing the linked list allocation.

- The file systems are:
  - The FAT family: FAT12, FAT16, and FAT32;
  - The New Technology File System: NTFS.

FATs Brief Introduction

- What is the meaning of the numbers (12/16/32)?
  - A block is named a **cluster**.
  - The main difference among all the versions of FAT FS-es is the **cluster address size**.



FATs Brief Introduction

- **Cluster address sizes**

File System	FAT12	FAT16	FAT32
Cluster address length	12 bits	16 bits	32 bits (28?)
Number of clusters	4K	64K	256M

– The larger the cluster address size is, the larger **the size of the file allocation table**.

– The larger the cluster size is, the larger **the size of the disk partition** is.

We will look into more details of FAT32 in later lectures

Summary of Trial 2.2

- Is FAT a perfect solution…
  - Tradeoff: **trade space for performance**
    - The entire FAT has to be stored in memory so that…
    - the performance of looking up of an arbitrary block is satisfactory.

- Can we have a solution that stands in middle?
  - Not store *the entire set* of block locations in mem…
  - I don't need an *extremely high performance* in block lookups.

File System Layout

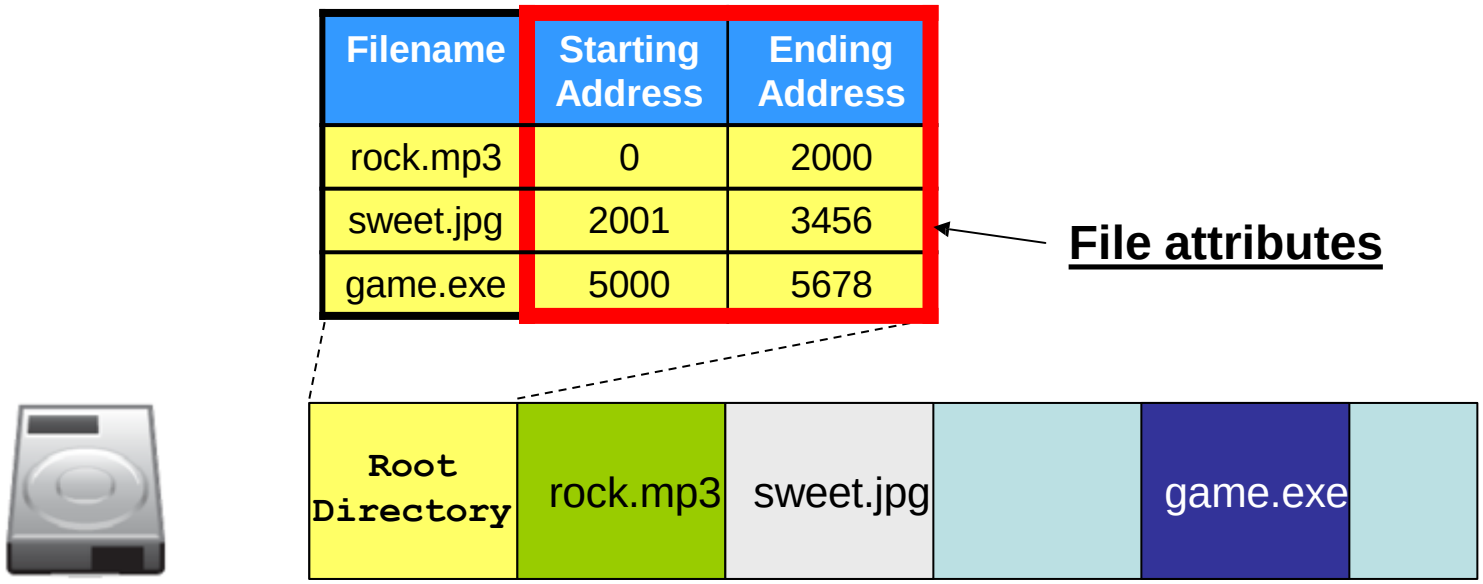
Trial 3.0

The Index-Node Allocation

Back to Trial 1.0-2.2

- File system layout: how to store file and directory
  - 1.0: Contiguous allocation (just like a book)

Two key problems: External fragmentation + file growth

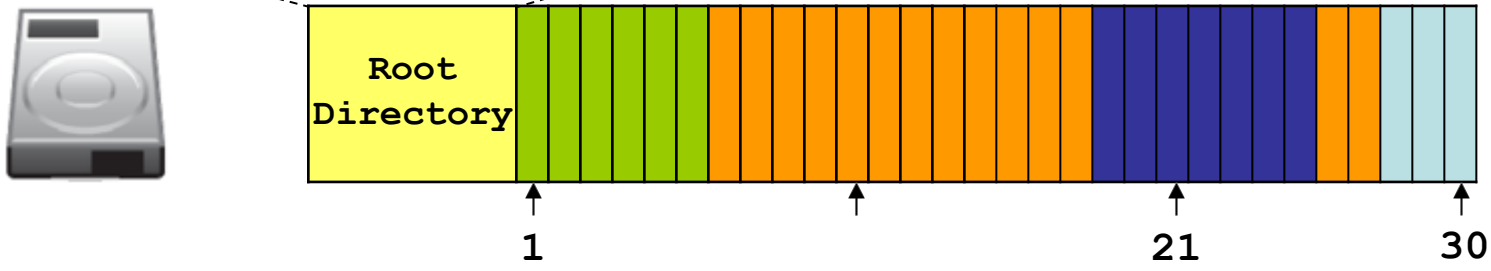


Back to Trial 1.0-2.2

- File system layout: how to store file and directory
  - 2.0: Linked-list allocation: blocking

Key problem: complicated root directory

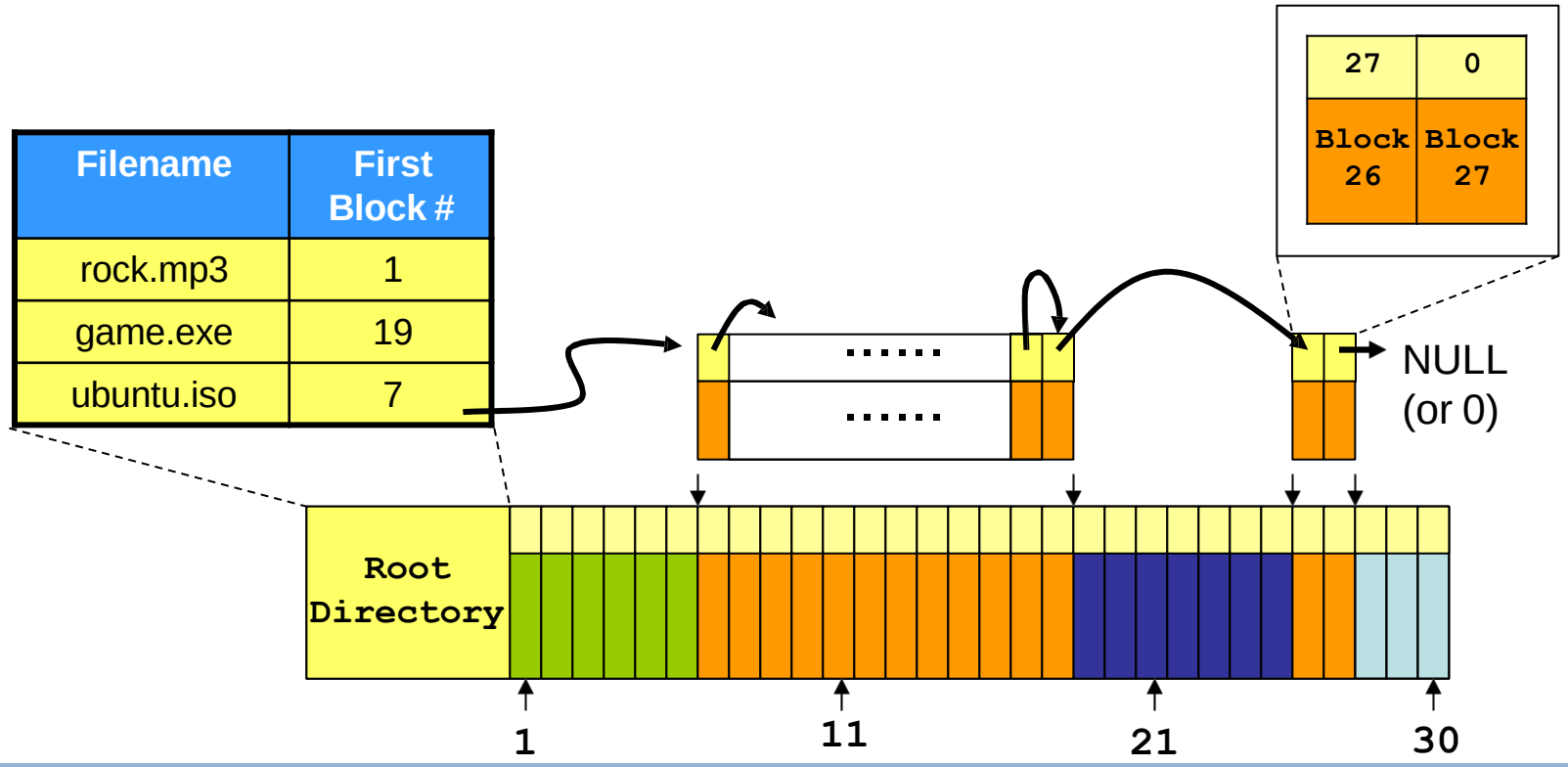
Filename	Sequence of Block #	Sequence of Block #
rock.mp3	1-6	NULL
game.exe	19-25	NULL
ubuntu.iso	**7-18**	**26-27**



Back to Trial 1.0-2.2

- File system layout: how to store file and directory
  - 2.1: Linked-list allocation: blocking + linked list

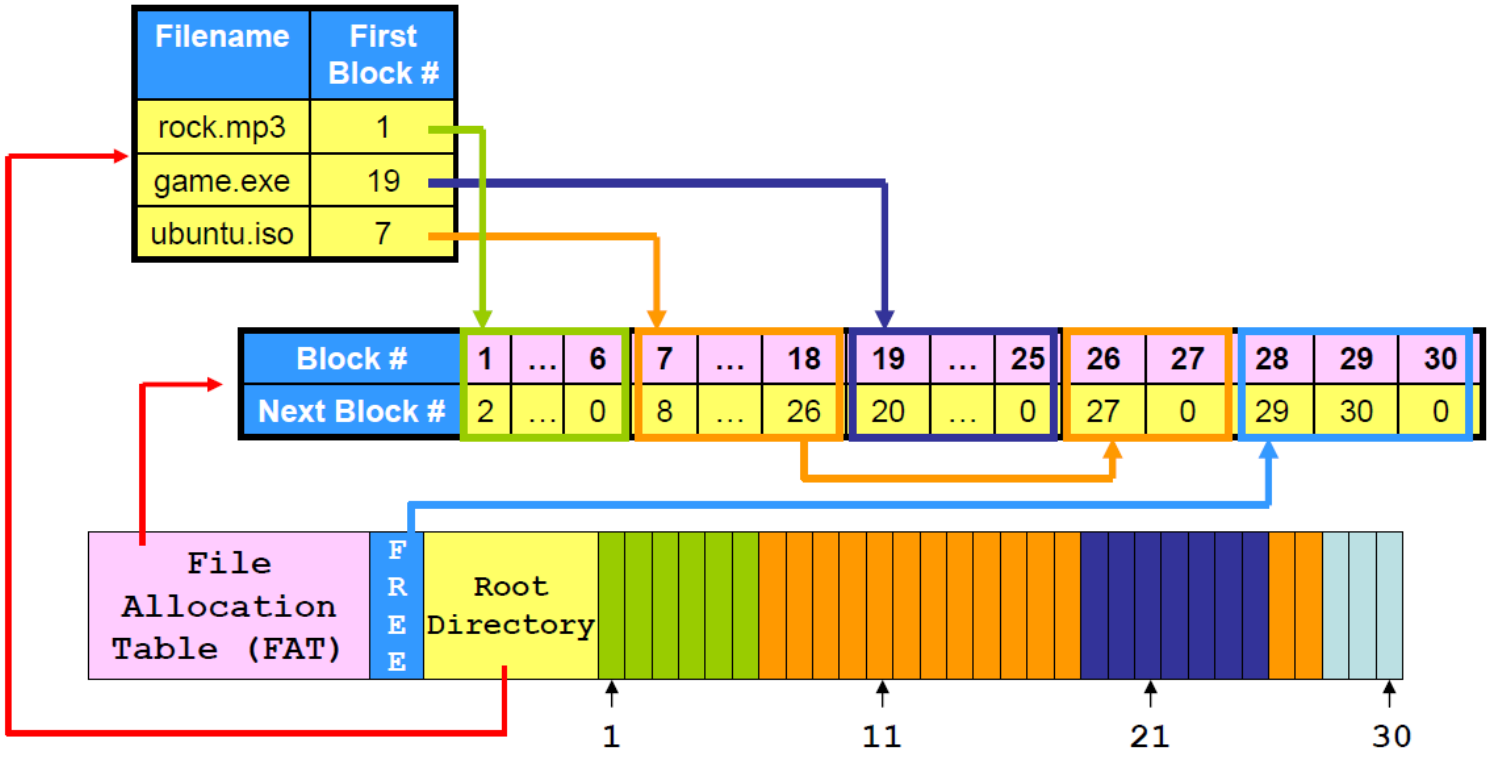
Key problem: random access problem



Back to Trial 1.0-2.2

- File system layout: how to store file and directory
  - 2.2: Linked-list allocation: centralized next-block # (FAT)

Requirement: FAT Caching

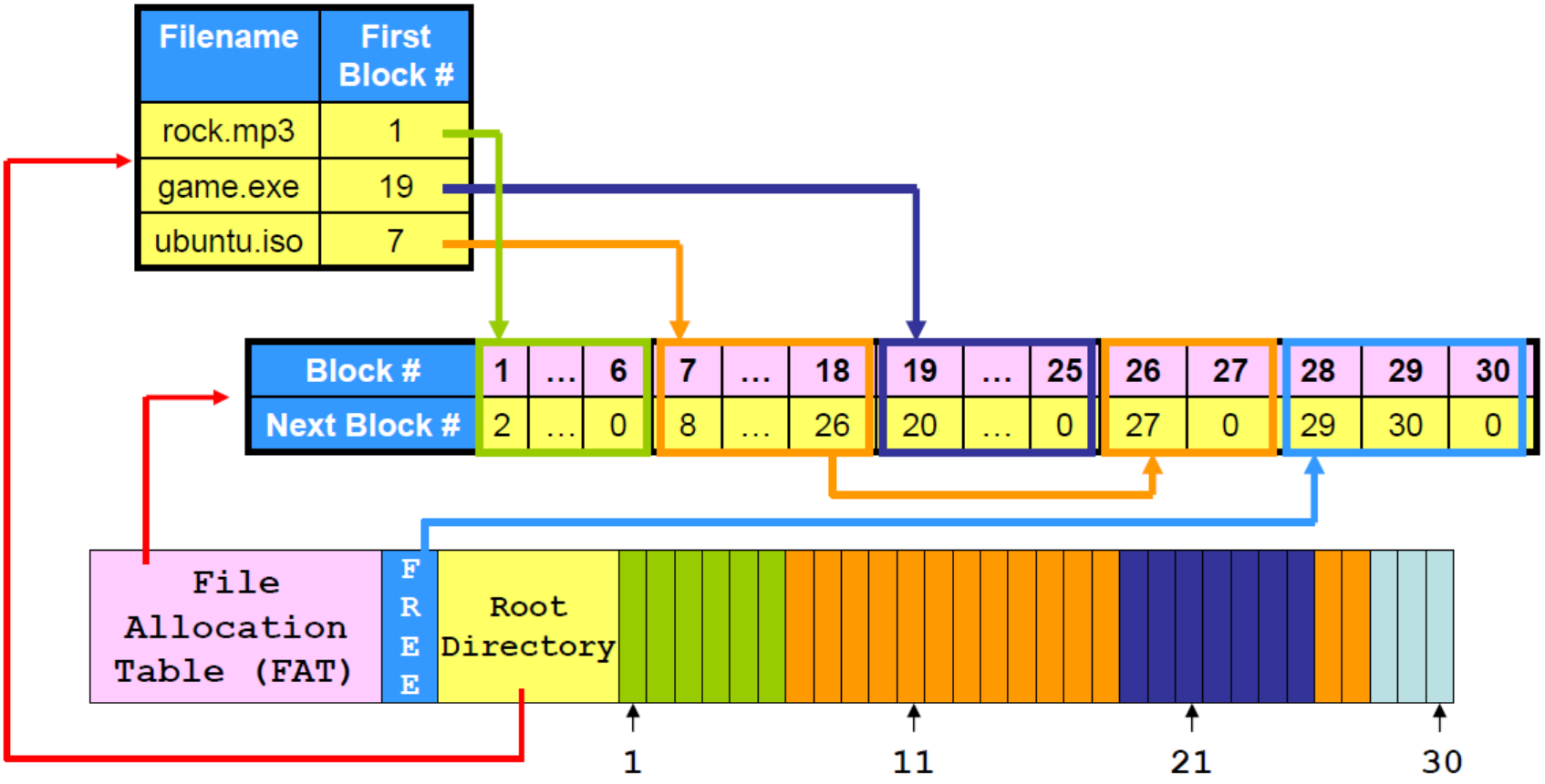


Trial 2.2 - FAT

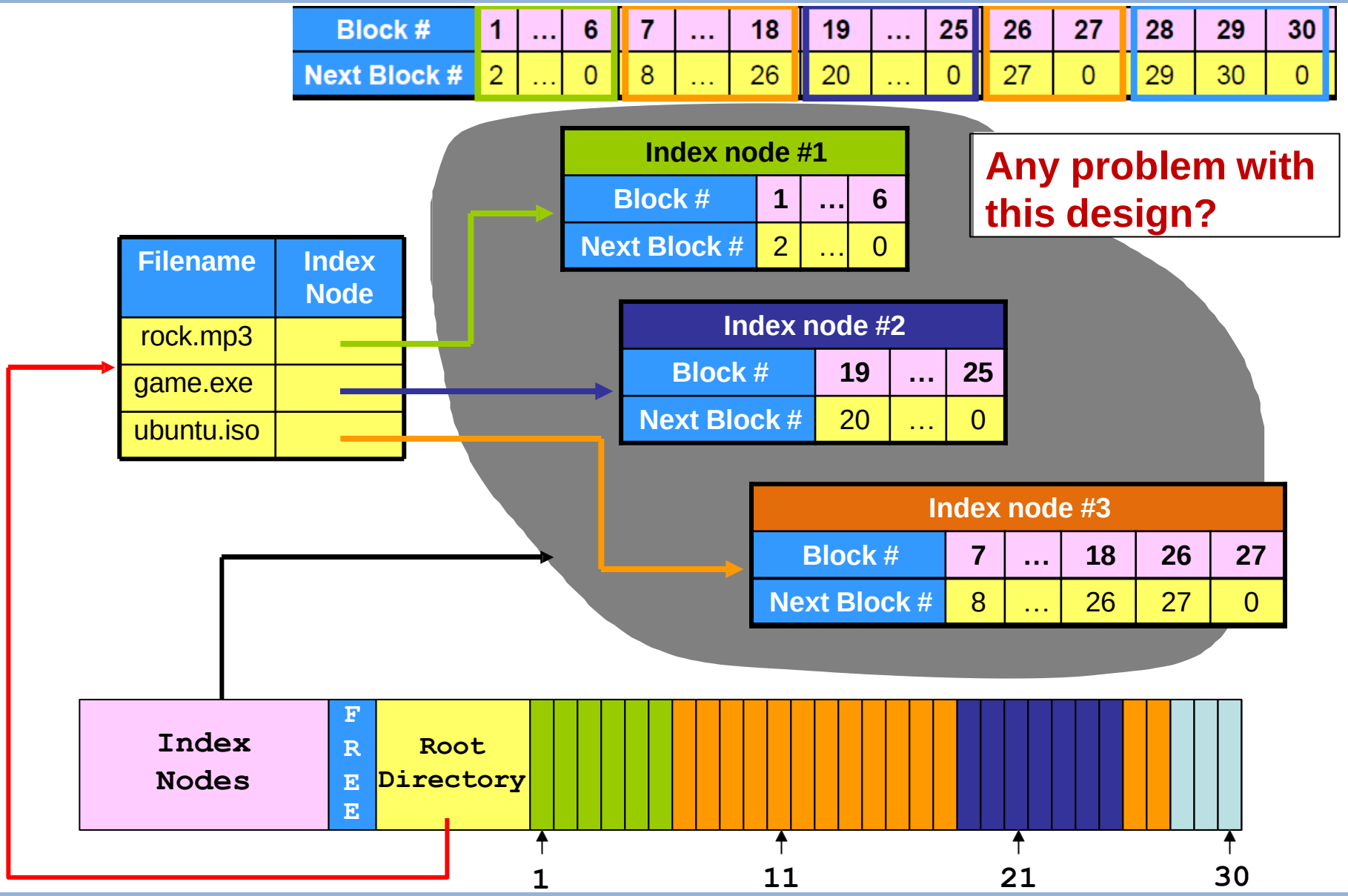
- FAT provides a good performance in all aspects
  - File creation, file growth/shrink, file deletion …
  - Random access performance…but requires to
    - **cache the FAT**
- Balance the tradeoff between Performance and memory space
  - Partial caching
  - How?

Trial 2.2 - FAT

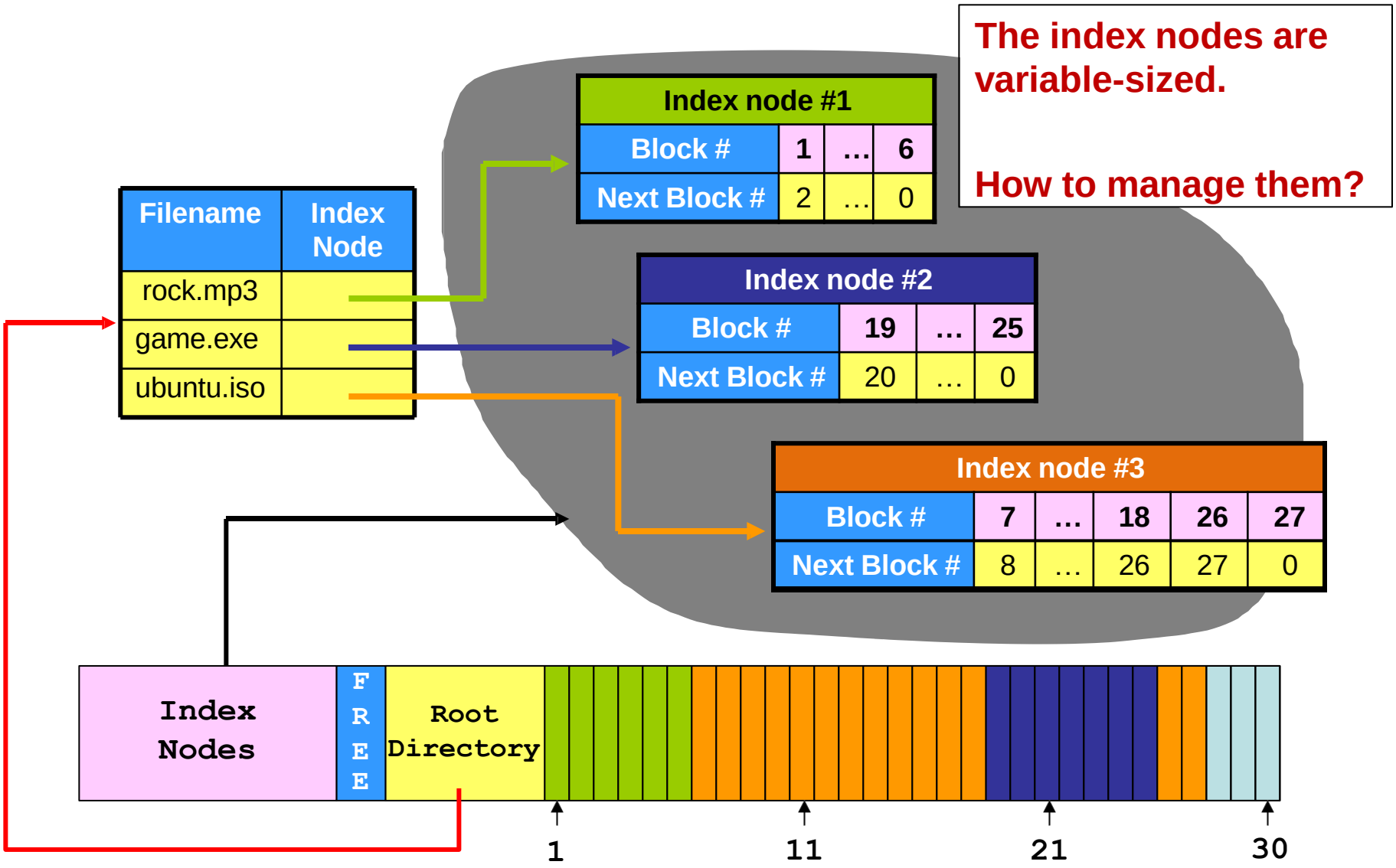
We are going to **break the FAT into pieces**…Trial 3.0



Trial 3.0 – the beginning



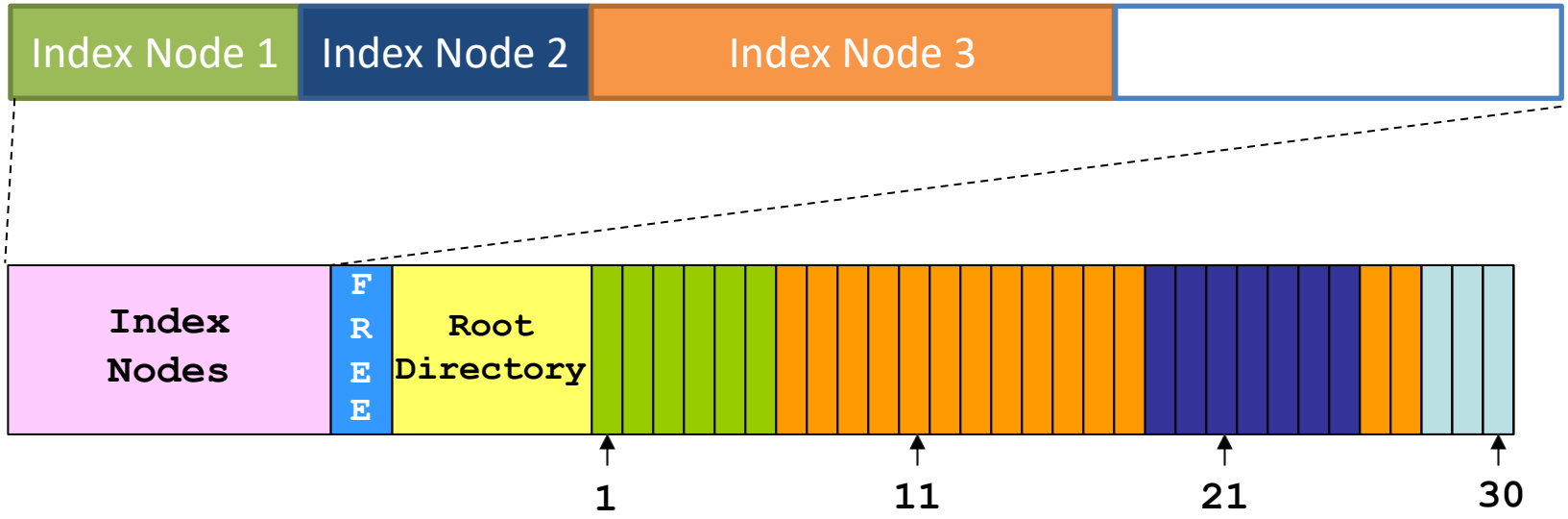
Trial 3.0 – the beginning



Trial 3.0 – the beginning

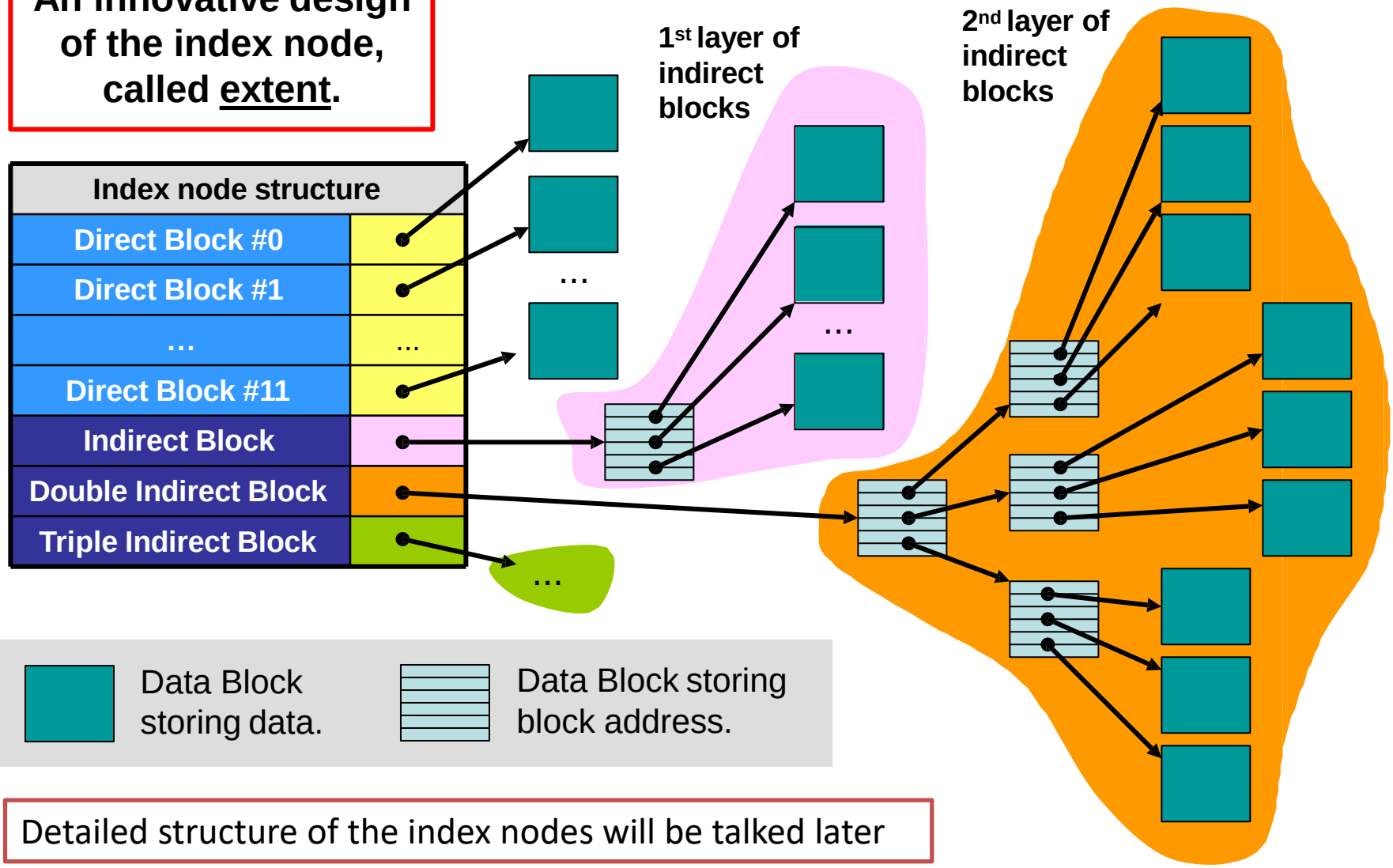
- Problems with variable-sized index nodes
  - How to locate an index node?
  - How to support file growth…size of index nodes depends on file size

Fix-sized index nodes are preferable, how to achieve?



Trial 3.0 – the heart

An innovative design of the index node, called extent.



Detailed structure of the index nodes will be talked later

Trial 3.0 – the two kinds of blocks

Indirect block

Stores an array of **block addresses**.

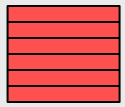
An address may point to either a data block or another indirect block.

However, in a block, **all** the addresses are either pointing to indirect blocks or data blocks.

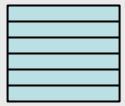
Data block

Stores file data.

Keys



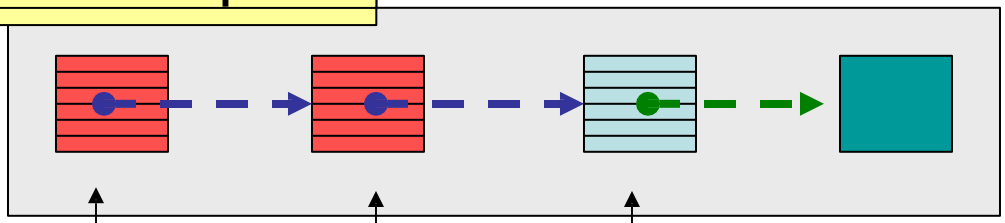
Indirect blocks that point to indirect blocks



Indirect blocks that point to data blocks

Data blocks

The consequence



3rd layer indirect

2nd layer indirect

1st layer Indirect

Where are the (indirect) blocks stored?

Trial 3.0 – the file size

How large files can be supported?

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 * block size

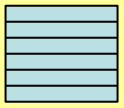
$12 \times 2^x +$

$2^{x-2} * 2^x = 2^{2x-2}$ +

$2^{x-2} * 2^{x-2} * 2^x = 2^{3x-4}$ +

2^{4x-6}

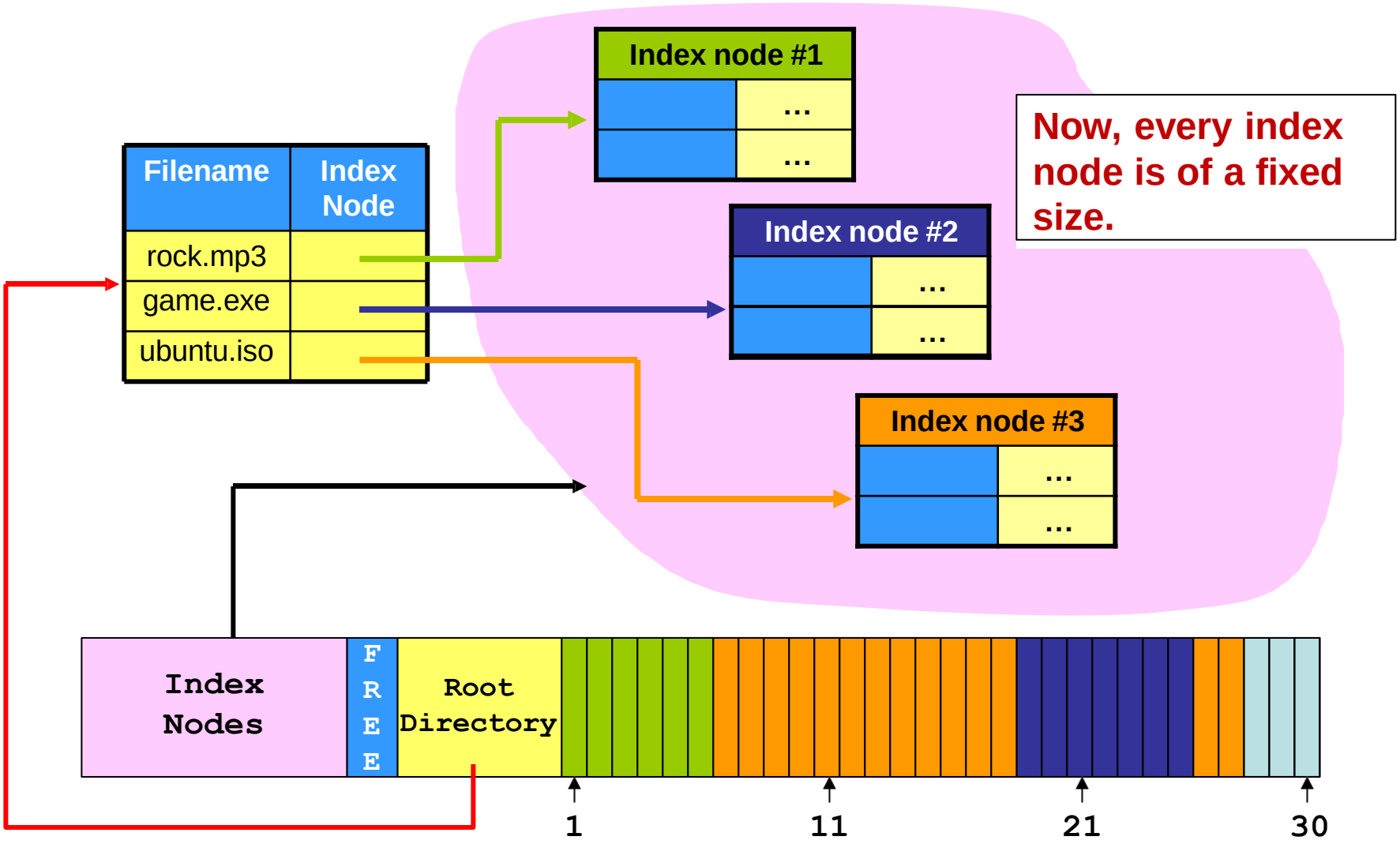
The dominating factor.



"$2^x / 4 = 2^{x-2}$" addresses

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

Trial 3.0 – the final design



Trial 3.0 – the final design

Layout & read process

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

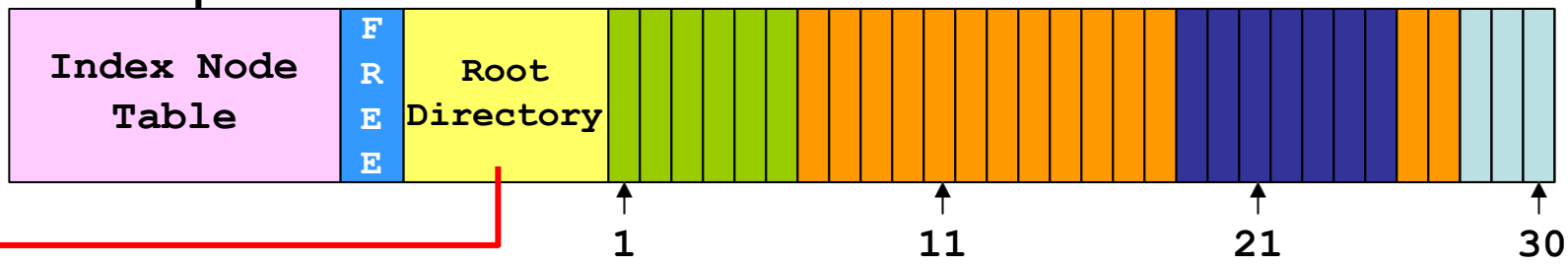
Filename	Index Node #
rock.mp3	1
game.exe	2
ubuntu.iso	3

Inside the index node table …

Index node #1		Index node #2		…		Index node #n-1	
	…		…				…
	…		…				…

Searching the index nodes **using the index node #**.

It is arranged as **an array**. So, looking up an index node will be fast.



Trial 3.0

- How about the tradeoff between performance and memory usage?
  - Partial caching is easy

- Any overhead of Trial 3.0?
  - The index-node allocation uses more storage:
    - **to trade for a larger file size (with fixed-size index nodes)**.
  - The indirect blocks are the **extra things**.

Trial 3.0 – Storage Overhead

- The indirect blocks are the **extra things**.

File Size	$12 \times 2^x + 2^{2x-2}$	~4M (x=12)
# of Indirect Blocks	$(2^{x-2})^0$	1 block

File Size	$12 \times 2^x + 2^{2x-2} + 2^{3x-4}$	~4G (x=12)
# of Indirect Blocks	$(2^{x-2})^0 + (2^{x-2})^0 + (2^{x-2})^1$	~1K blocks

Trial 3.0 – Storage Overhead

- The indirect blocks are the **extra things**.

~4T (x=12)

File Size	$12 \times 2^x + 2^{2x-2} + 2^{3x-4} + 2^{4x-6}$
# of Indirect Blocks	$(2^{x-2})^0 + (2^{x-2})^0 + (2^{x-2})^1 + (2^{x-2})^0 + (2^{x-2})^1 + (2^{x-2})^2$

~1M blocks

Trial 3.0 – Storage Overhead

- The indirect blocks are the **extra things**.
  - Max. number of indirect blocks depends on
    - Block size
    - File size

$$\left(2^{x-2}\right)^0 + \left(2^{x-2}\right)^1 + \left(2^{x-2}\right)^2$$

Block size	Max. # of indirect blocks	Max. Extra Size involved
1024 bytes = 2^{10}	approx. 2^{16}	approx. 256 Mbytes
4096 bytes = 2^{12}	approx. 2^{20}	approx. 4 Gbytes

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

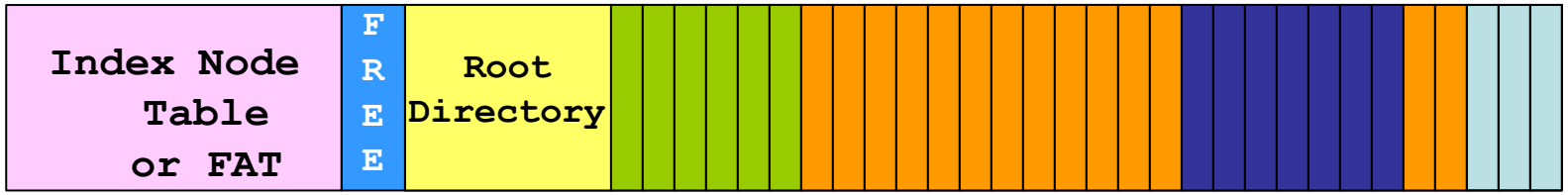
Trial 3.0 – the summary

- FSes in UNIX and Linux use the index-node allocation method.
  - The **`Ext2/3/4`** file systems.
    - The index node is called **inode** in those systems.
    - Ext4 uses extent, not indirect blocks
  - We will discuss the details of Ext file system later.

From Trial 1.0 to Trial 3.0…

- We studied what are the possible ways to store data in the storage device.
  - The things stored are usually:

Root directory	**File attributes**
Hey, where are the sub-directories? Still remember the directory traversal	Except the **file size** and **the locations** of the data blocks, where and what are the **other attributes**?
Free space management	**Data block management**
Actually, we didn't cover that much…	The FAT, the extents, the table of content.

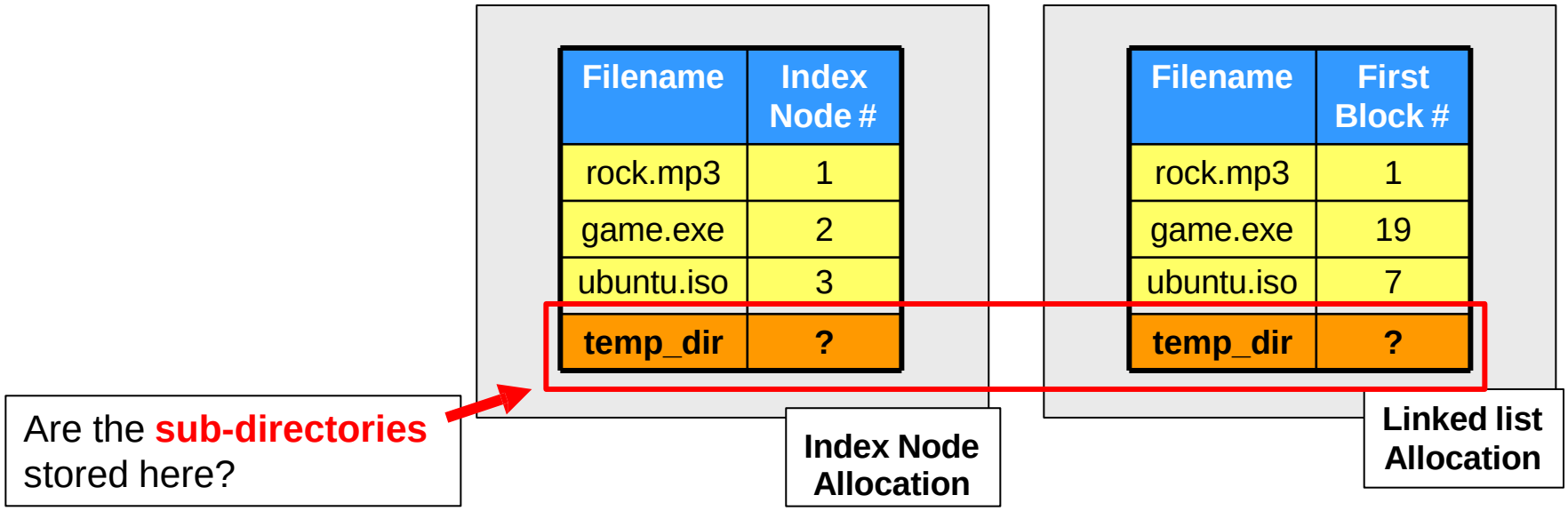


File System Layout

Root Directory and
Sub-directories

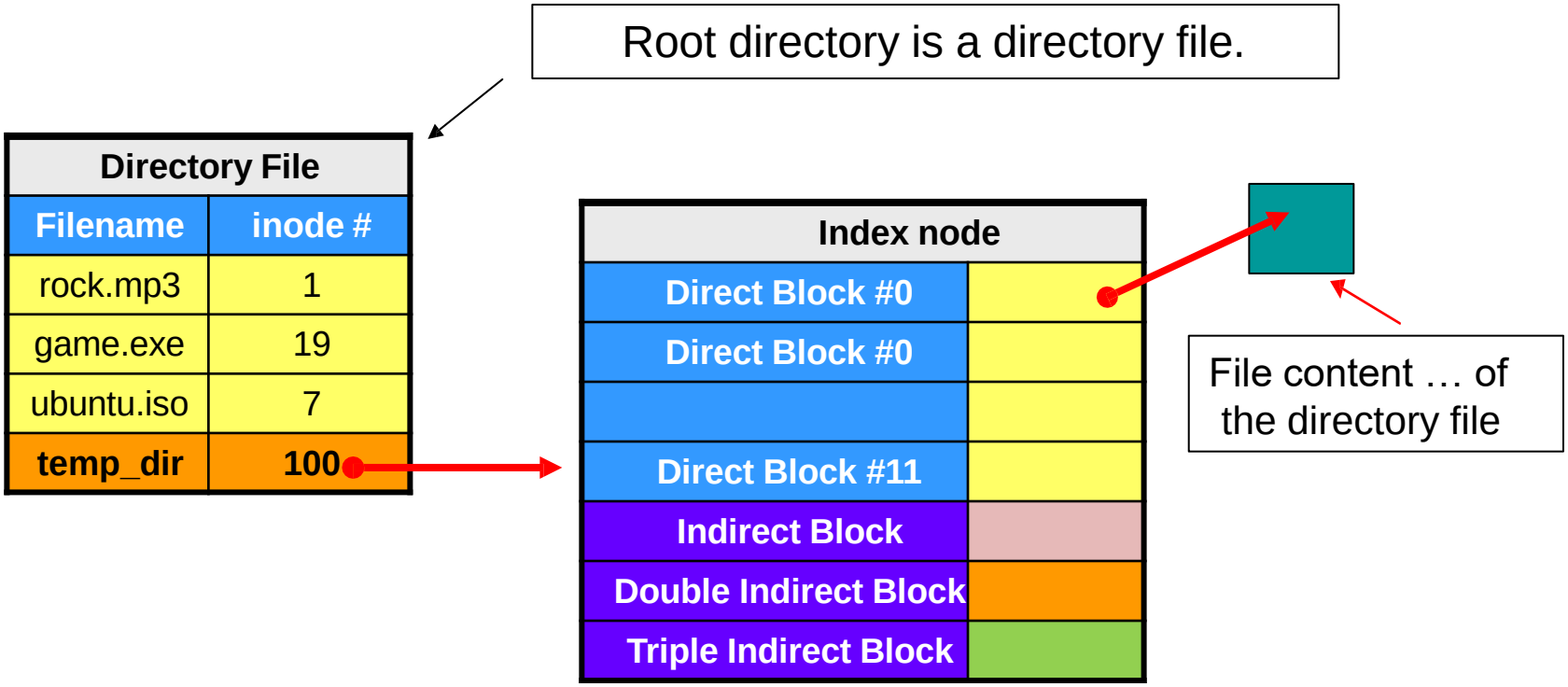
Root directory

- We know that the root directory is vital.
  - However, we have sub-directories…
  - Where are they?



Sub-directories?

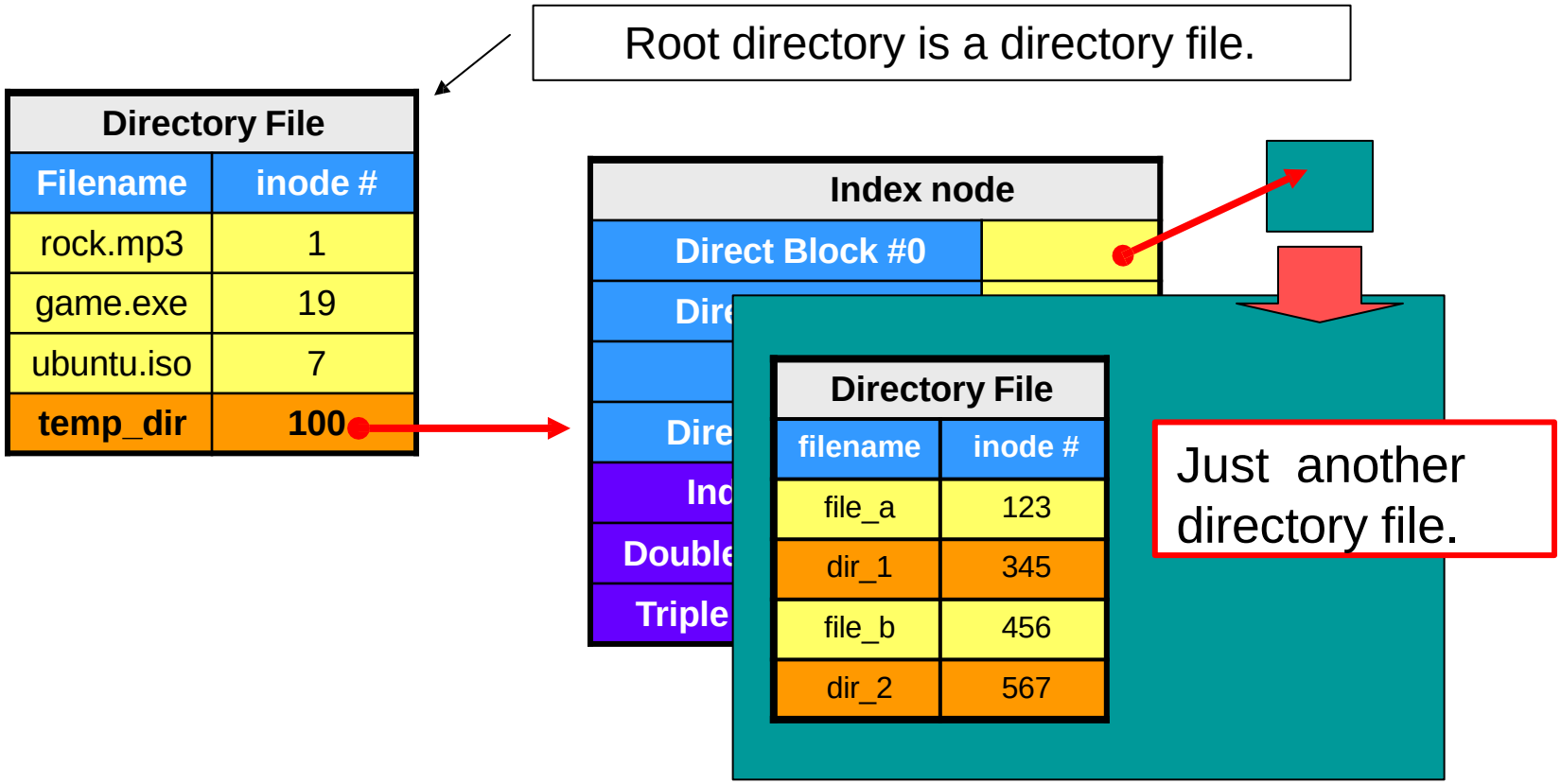
- Let's take the index-node allocation as an example…



Directory is also a file, so it has an inode too

Sub-directories?

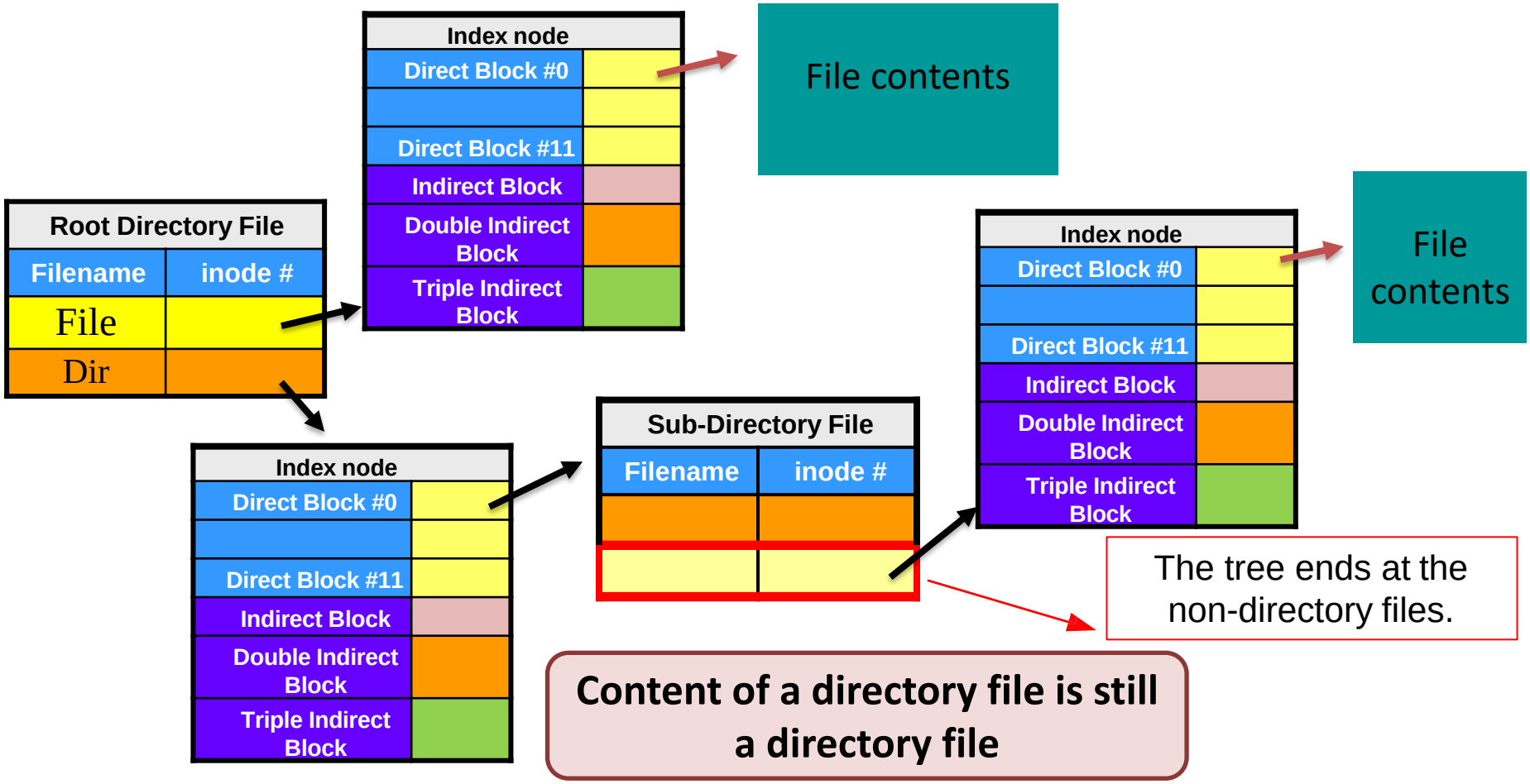
- Let's take the index-node allocation as an example…



See, each directory entry keeps the **address** of the file attributes, not the attributes themselves (how about FAT file systems?)

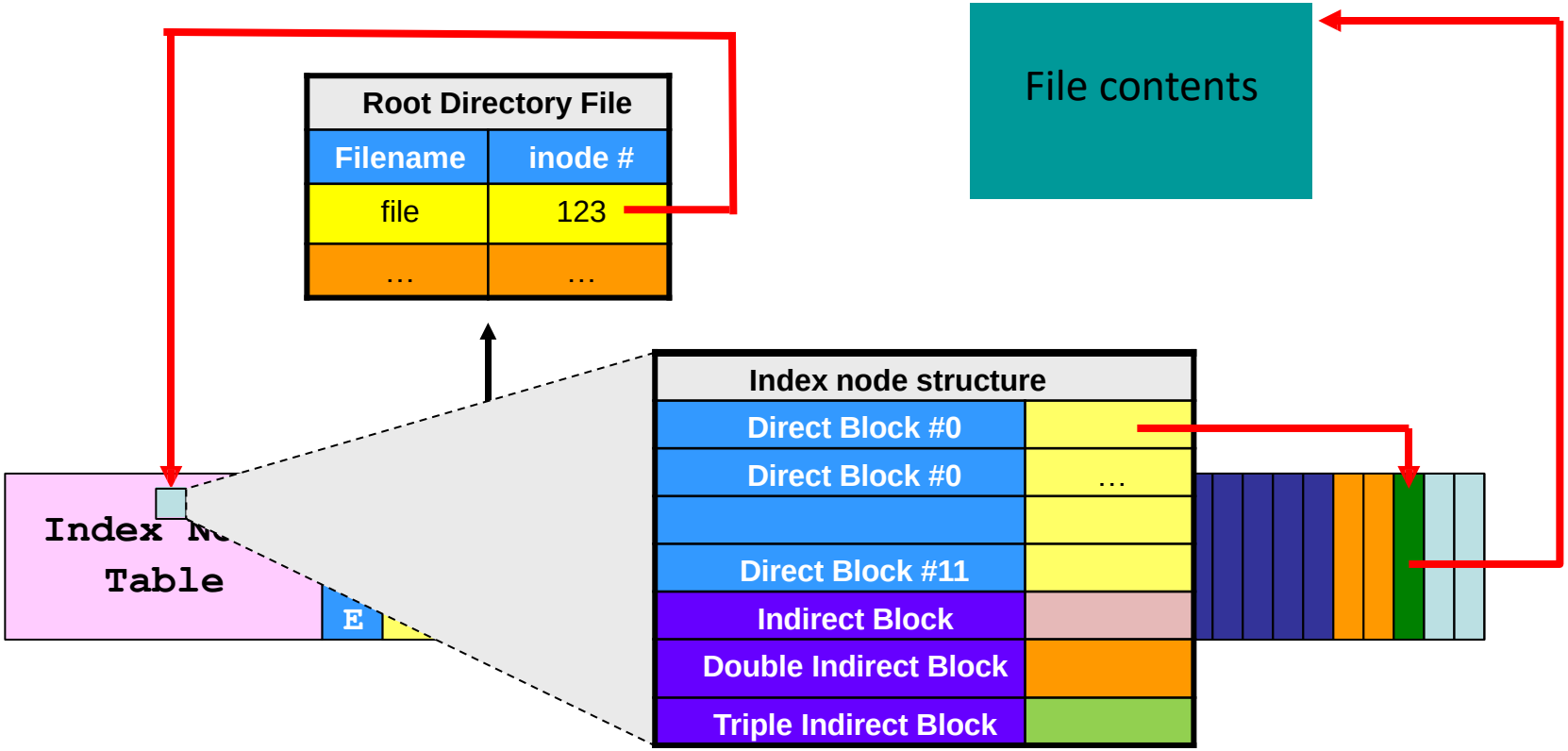
Traversing directory structure…

- Let's take index-node allocation as an example…



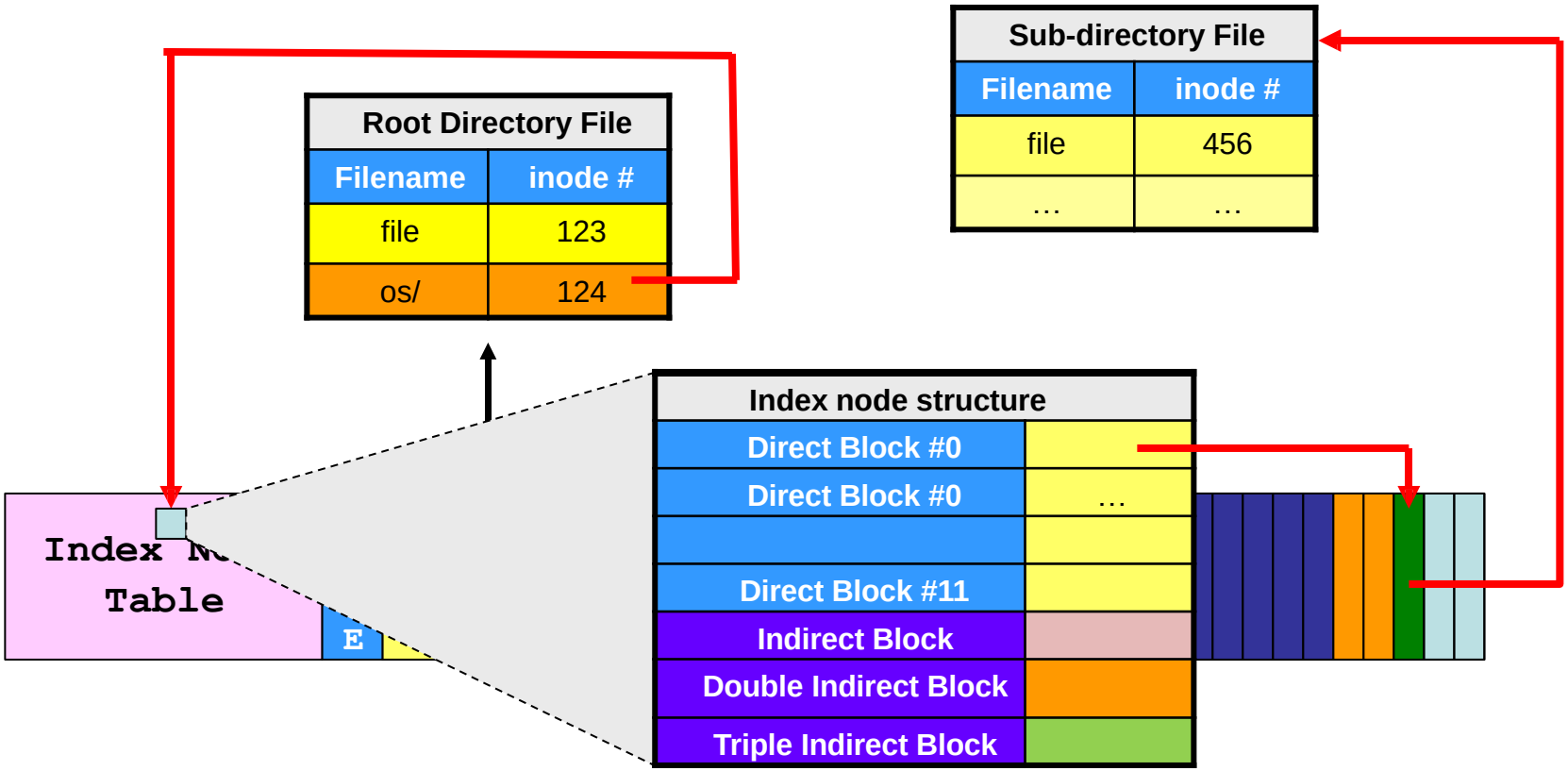
Traversing directory structure…

- Work together with the layout
  - Let's still take index-node allocation as an example…
  - E.g.: "`/file`"



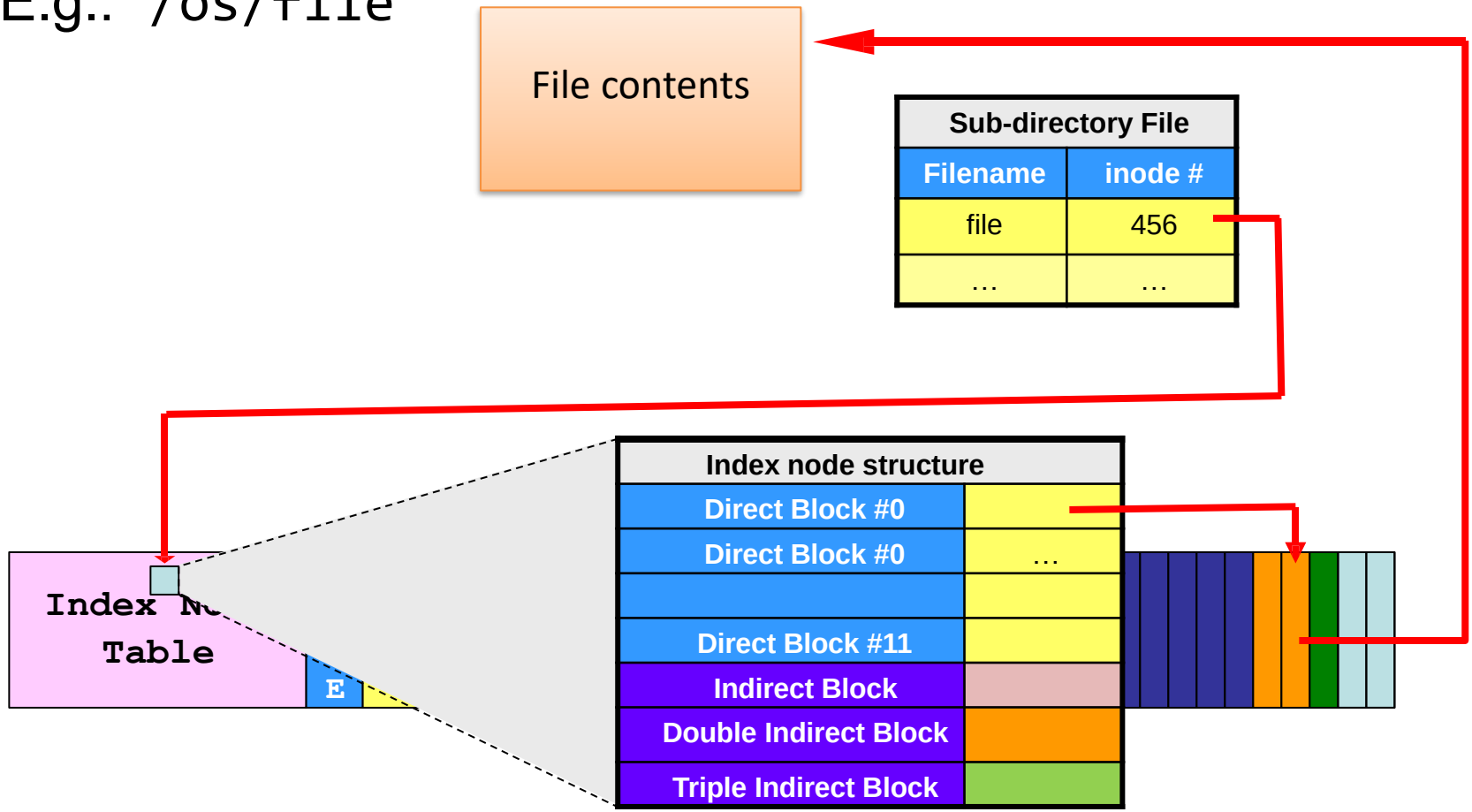
Traversing directory structure…

- Work together with the layout
  - Let's still take index-node allocation as an example…
  - E.g.: "`/os/file`"



Traversing directory structure…

- Work together with the layout
  - Let's still take index-node allocation as an example…
  - E.g.: "`/os/file`"



File System Layout

File system information
and partitioning

Storage layout

- What are stored on disk?
  - Root directory, index nodes/FAT, data blocks, free space information…
  - Others?
    - E.g., How do we know where the root directory is?
    - Where is the first inode?
  - File system information

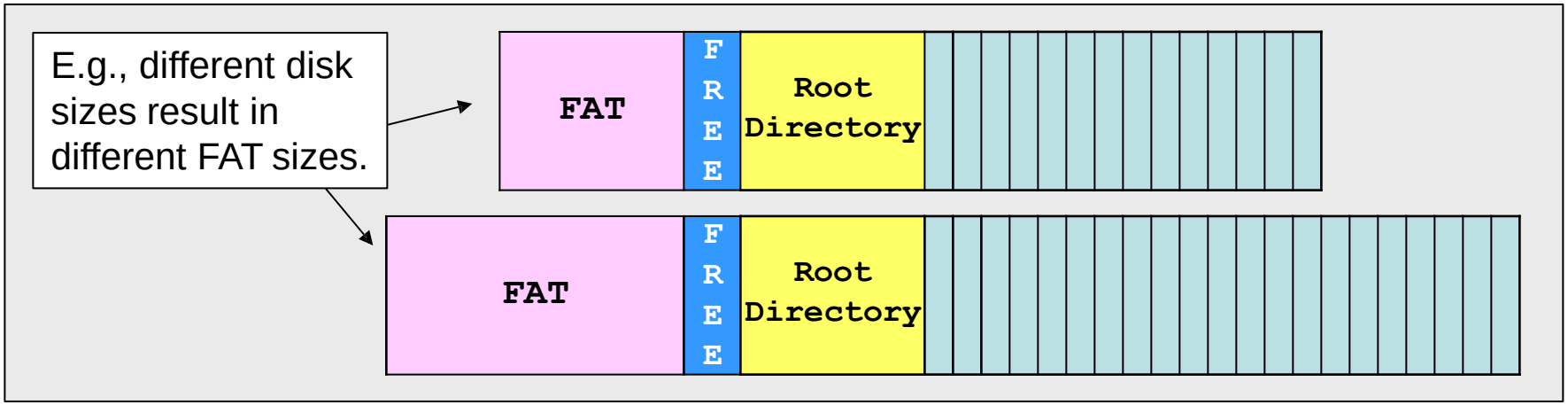
File System Information

- It is a set of important, FS-specific data…

Examples of FS-Specific Data
How large is a block?
How many allocated blocks are there?
How many free blocks are there?
Where is the root directory?
Where is the allocation information, e.g., FAT & inode table?
How large is the allocation information?

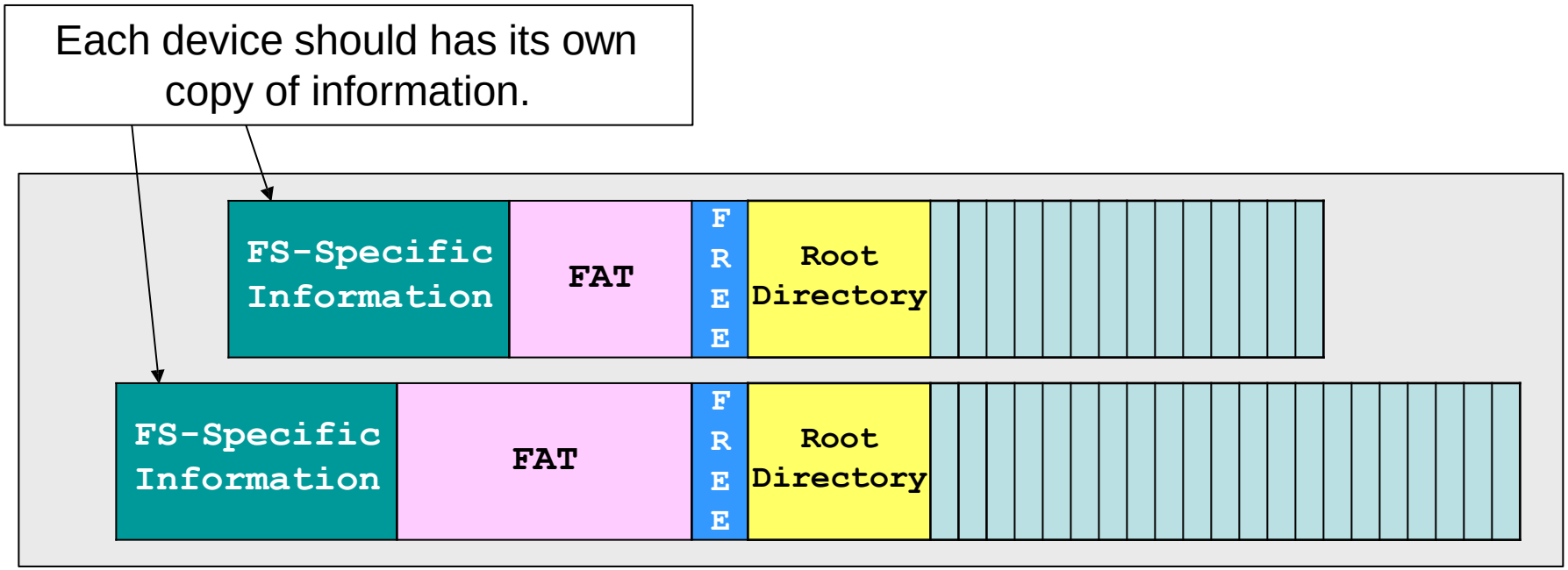
File System Information

- It is a set of important, FS-specific data…
  - Can we hardcode those information in the kernel code…
  - No!!! Because different storage devices have different needs.



File System Information

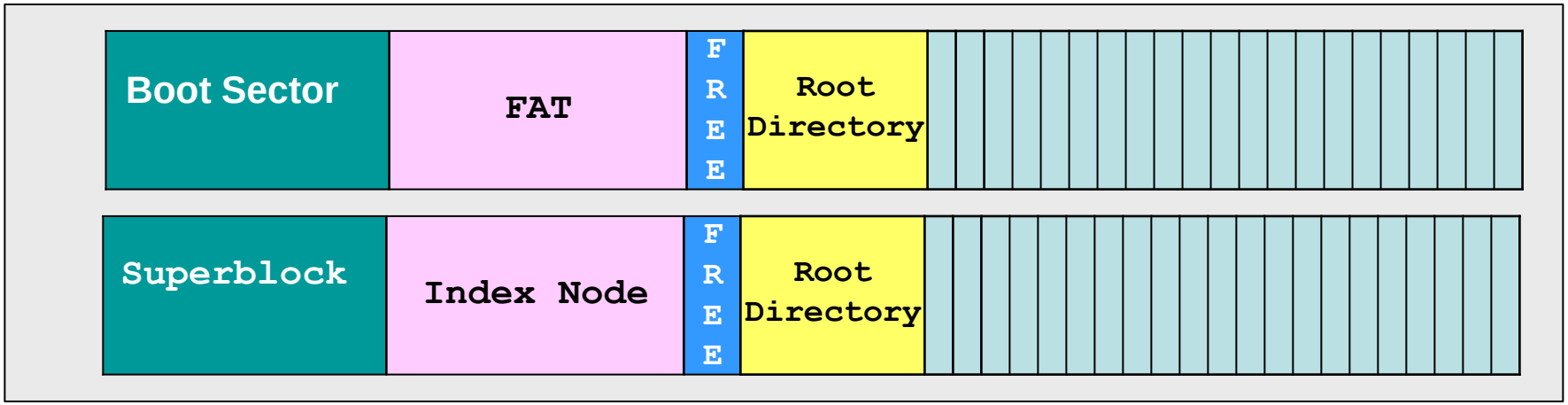
- It is a set of important, FS-specific data…
  - **Solution**: The workaround is to save those information on the device.



File System Information

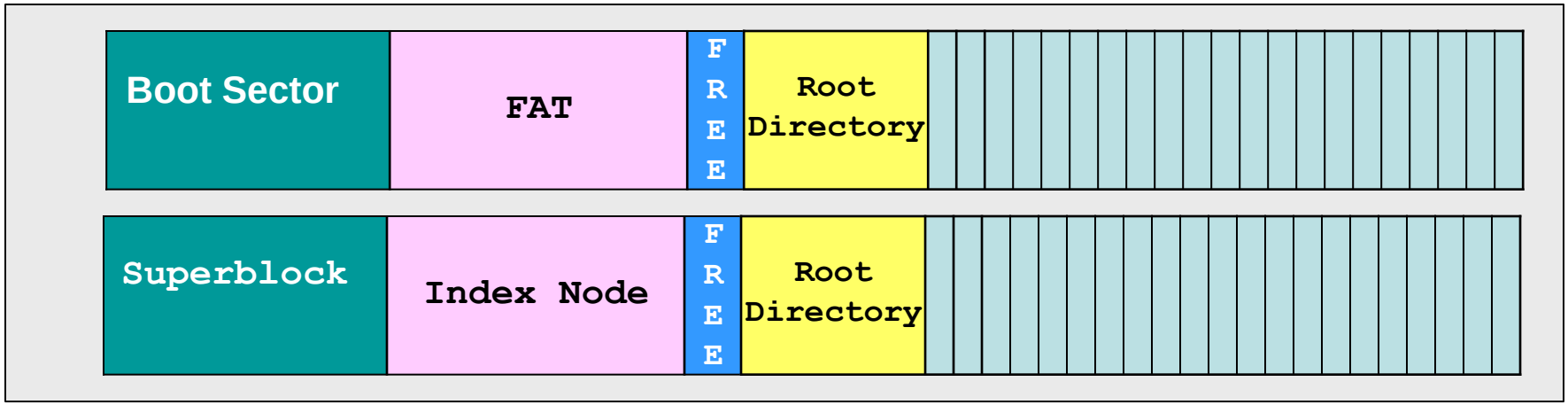
- It is a set of important, FS-specific data…
  - **Solution**: The workaround is to save those information on the device.

In FAT* & NTFS	**Boot Sector**
In Ext*	**Superblock**



Story so far…

- We talked about the file system layout
  - FAT and index node

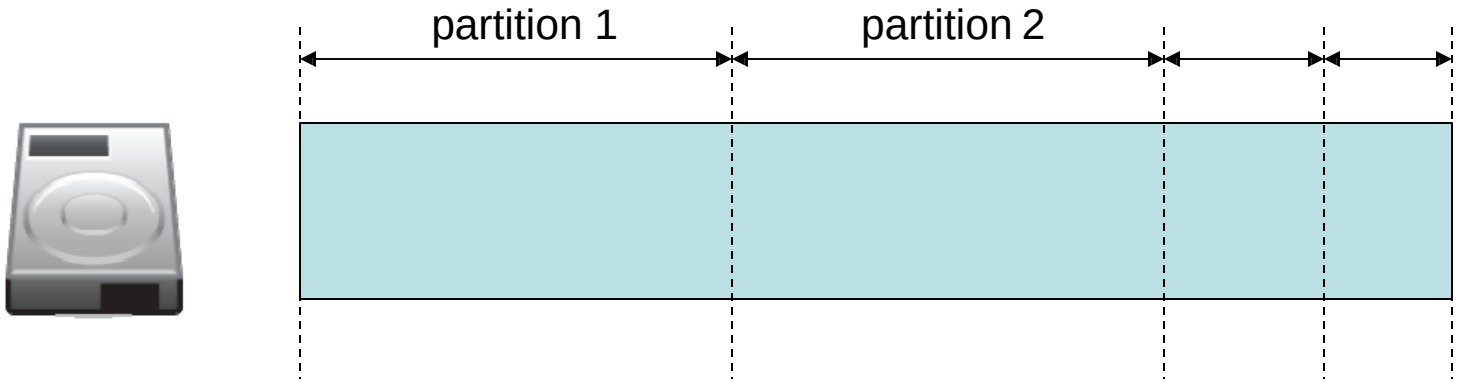


Only one file system can be stored in a disk? **No!**

What is the problem with a very large file system? **Large FAT**

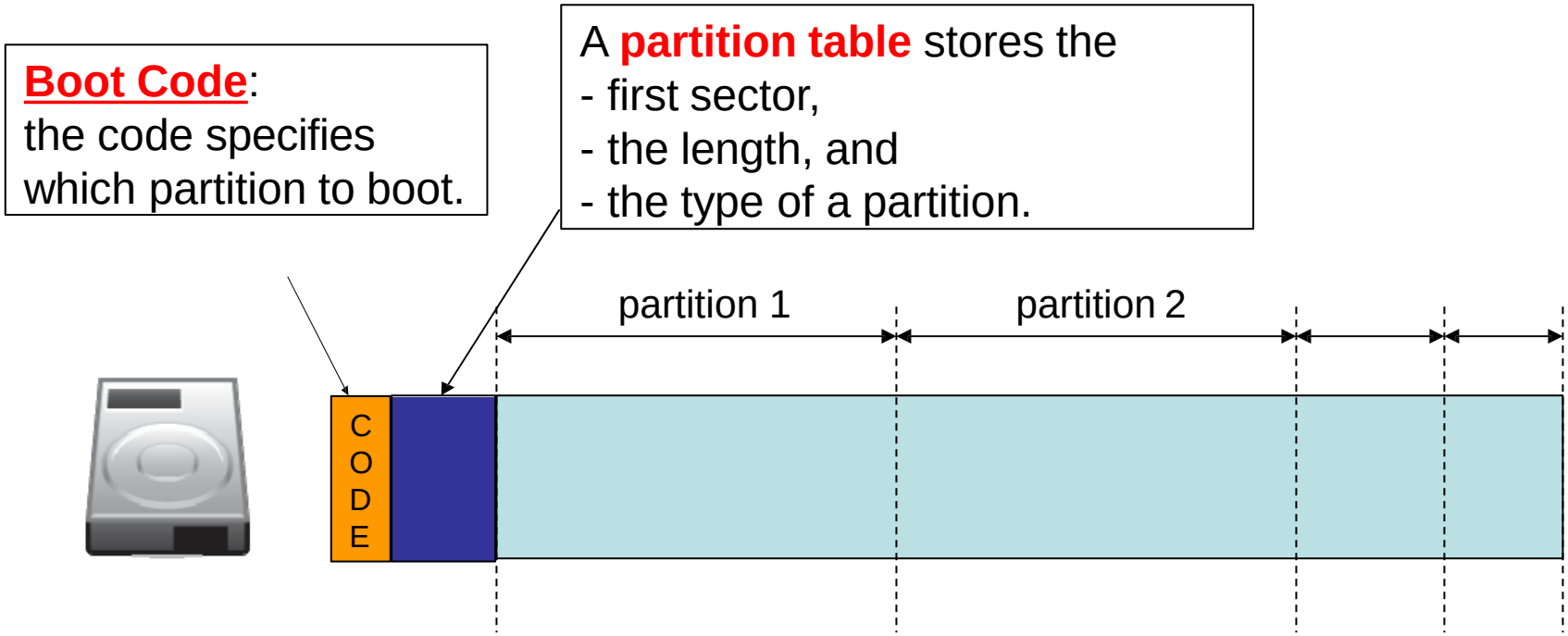
Disk partitions

- Partitioning is needed to
  - limit the file system size
  - support multiple file systems on a single disk

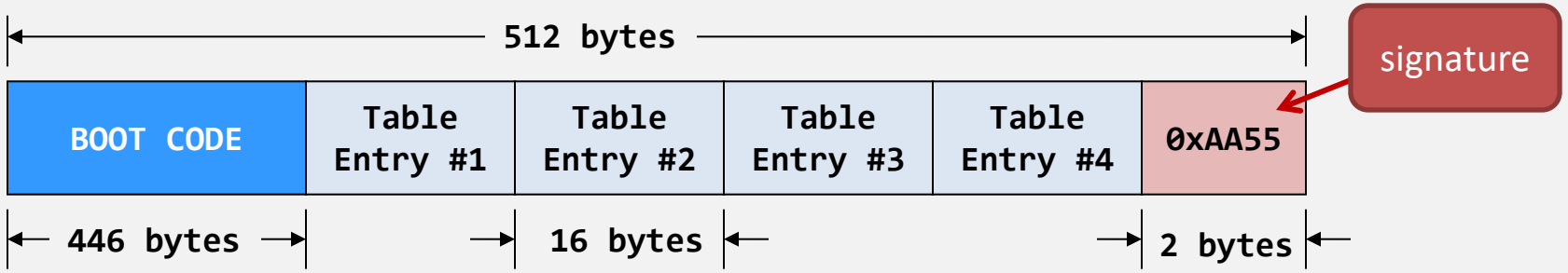


Disk partitions

- What is a disk partition?
  - A disk partition is a logical space…
    - A file system must be stored in a partition.
    - An operating system must be hosted in a partition.



Master boot record (MBR)…



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

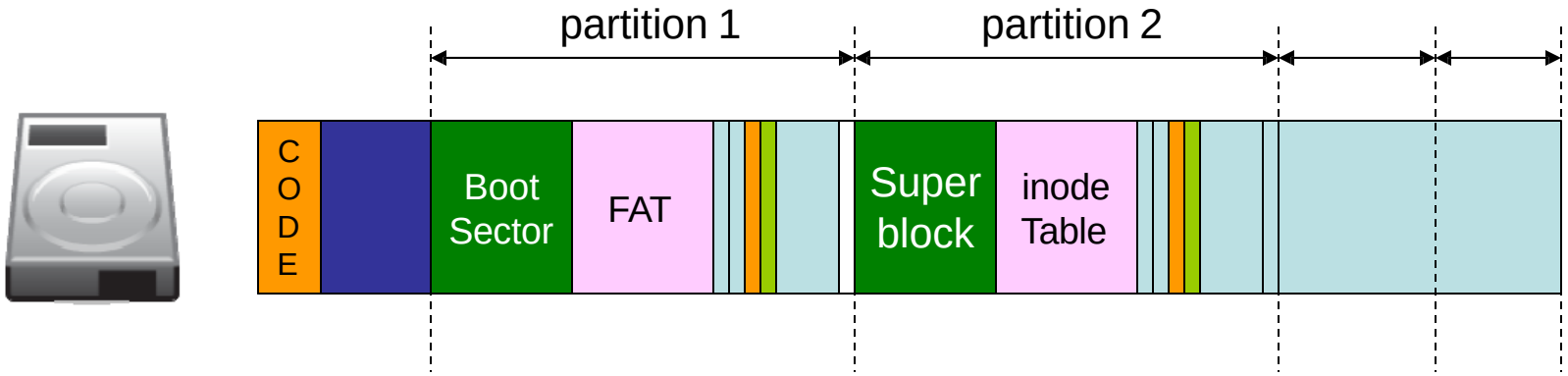
Partition Table Entry	
Bytes	**Description**
0-0	Bootable flag; 0x80 means bootable.
1-3	Starting CHS address
4-4	Partition type http://www.datarecovery.com/hexcodes.asp
5-7	Ending CHS address
8-11	Starting LBA address (measured in # of sectors)
12-15	Sizes in sectors

Disk partitions - summary

- Benefits of partitioning:
  - **Performance**
    - A smaller file system is more efficient!
      - Think about 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.

Final view of a disk storage space

- Final view of disk layout



- Now, do you know what is meant by “formatting” a disk?
  - Create and initialize a file system!
  - In Windows, we have “**`format.exe`**”.
  - In Linux, we have “**`mkfs.ext2`**”, “**`mkfs.ext3`**”, etc.

Summary of part2

- We have looked into many details about different file system layouts:
  - Contiguous allocation;
  - Linked list allocation; and
  - Index-node allocation.
- We also show the complete view of disk space
  - File system specific information & disk partition
- Linked list allocation and index-node allocation are the main streams but not the only way to implement modern file systems.

So far, we have learnt:

What are stored on disk

File: content + attributes
Directory: Directory file

How to access them?

File operations: `open(), read(), write()`
Directory lookup: Directory traversal

How are the files stored on disk?

File system layout: Contiguous/linked-list (FAT)/index-node allocation

Topics not covered:
Only the attributes of file name and locations are covered, how about other attributes? Free space management?

We'll look into some real implementations (FAT32 + EXT2/3/4)