# I-Nodes, Super Block, and Boot Block

## Computer Operating Systems, Spring 2025


**Instructors:** Joel Ramirez, Travis McGaha

**Head TAs:** Ash Fujiyama, Emily Shen, Maya Huizar

**TAs:**

Ahmed Abdellah, Bo Sun, Joy Liu, Susan Zhang, Zihao Zhou, Akash Kaukuntla, Connor Cummings, Khush Gupta, Vedansh Goenka, Alexander Cho, Eric Zou, Kyrie Dowling, Vivi Li, Alicia Sun, Haoyun Qin, Rafael Sakamoto, Yousef AlRabiah, August Fu, Jonathan Hong, Sarah Zhang, Yu Cao

# Poll Everywhere

pollev.com/cis5480

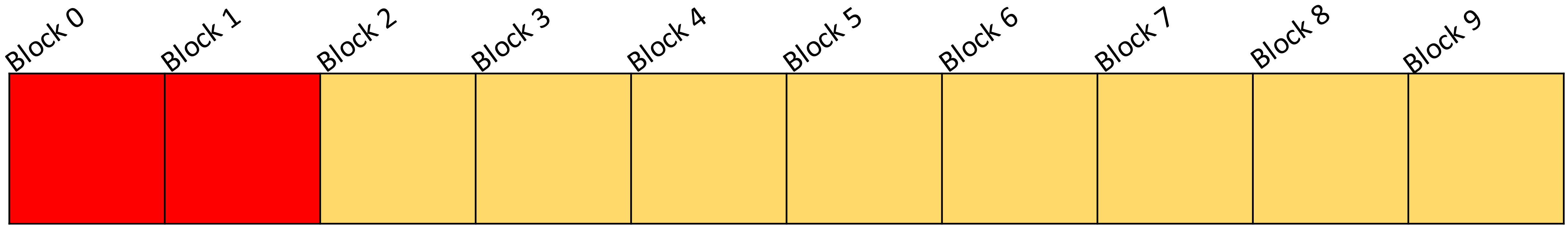
- Hope your Valentines day was nice! How is life? File systems is almost over! ☺

# Lecture Outline

- Linux Filesystem Implementation
  - Quick Review
  - Reserved Inodes & Root Inode
- File Paths
  - Absolute Paths & Relative Paths
- Resolving Absolute Paths
  - Directory Entries
- Accessing a Struct Dirent
- Putting It All Together
  - Bitmaps
  - Super Block
  - Boot Block

# Review: Filesystem and inodes

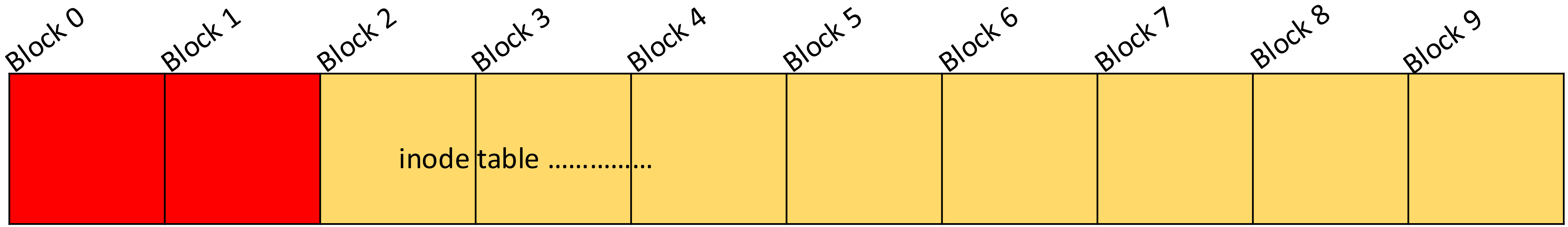
- The filesystem is composed of blocks that are at the smallest size, 512 bytes.
  - Today, filesystems blocks are much larger as we saw on Thursday.
  - On some, blocks reach up to 4096 bytes (4 KB).
- Linux Filesystem ext2, blocks smallest size is 1024 bytes.
- Physical Block Numbers start from 0. They correspond to ***physical blocks*** of memory on the memory device (e.g. hdd, ssd, sd cards, cds, floppy disk)




note: this is an attempt to present information as truthful as possible while trying to make it digestible

# Review: Filesystem and inodes

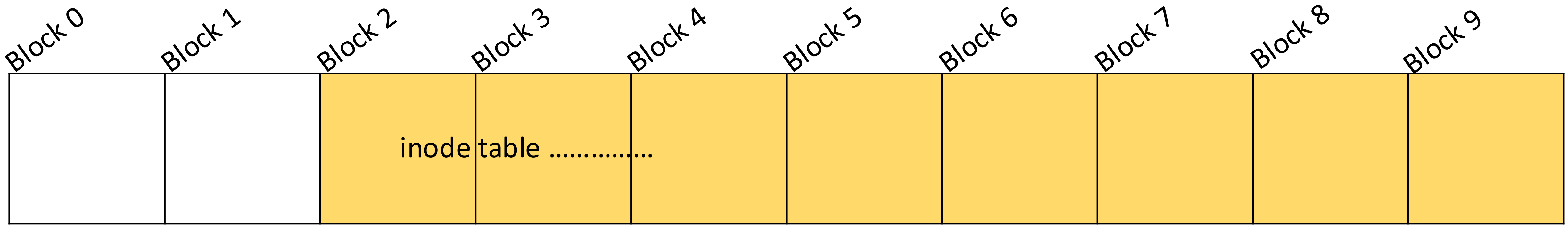
- From our perspective, the inode table starts on Block 2
  - Each block in the inode table is full of inodes, *even if the inode it does not refer to a file*
    - *(i.e. is unallocated)*
  - You can also think of the inode table as: `inode_array[]`
    - Where the `inode_array` is split up across blocks




note: this is an attempt to present information as truthful as possible while trying to make it digestible

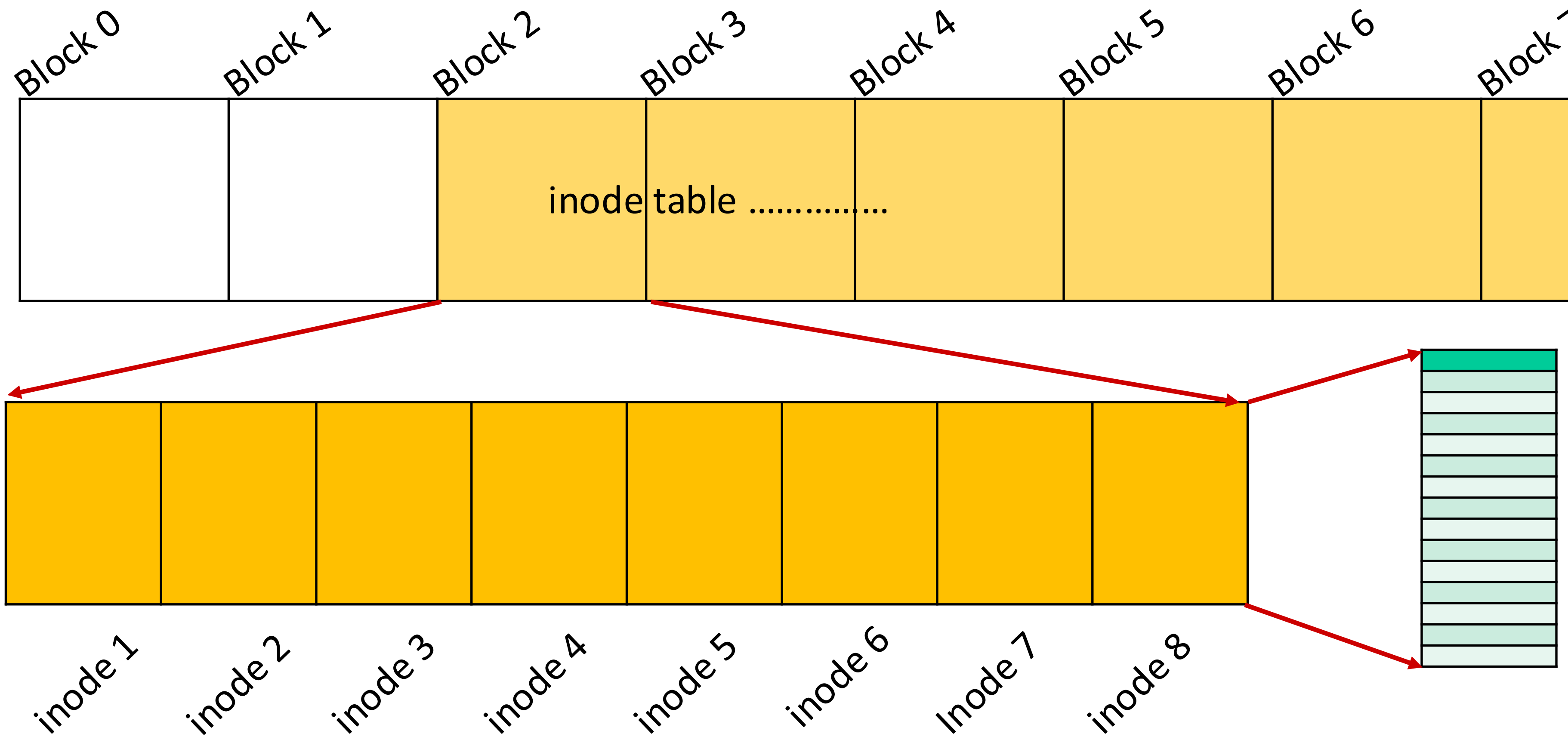
# Review: Filesystem and inodes

- From our perspective, the inode table starts on Block 2
  - Each block in the inode table is full of inodes, *even if the inode it does not refer to a file*
    - *(i.e. is unallocated)*
  - You can also think of the inode table as: `inode_array[]`
- Let's assume;
  - Block Size: 1024 Bytes
  - Inode Size: 128 bytes
    - Yes. They're that big in ext2.




note: this is an attempt to present information as truthful as possible while trying to make it digestible

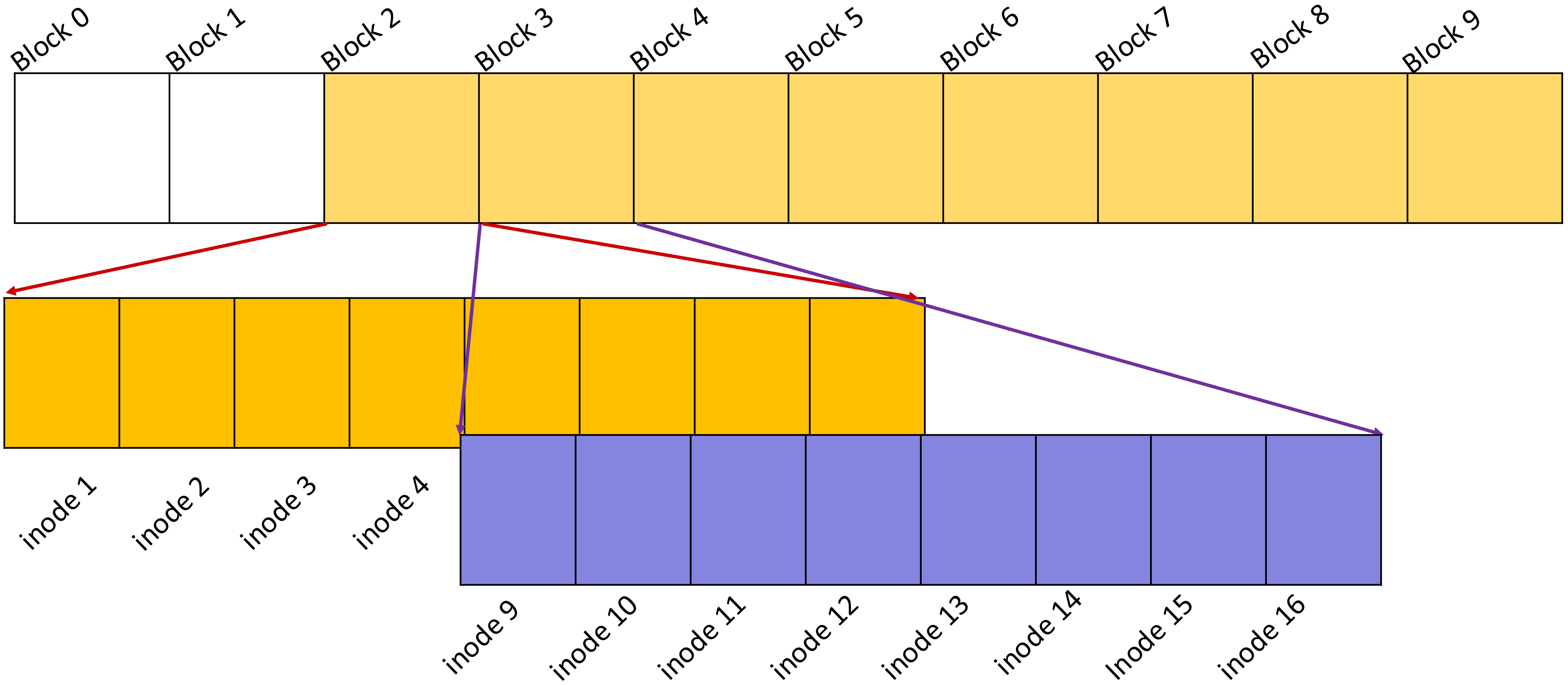
# Inodes at a glance: Physical Block 2




Each Inode contains
12 Physical Block Numbers
1 Singly Indirect Pointer
1 Doubly Indirect Pointer
1 Triple Indirect Pointer

*we say pointer, but they're physical block numbers.

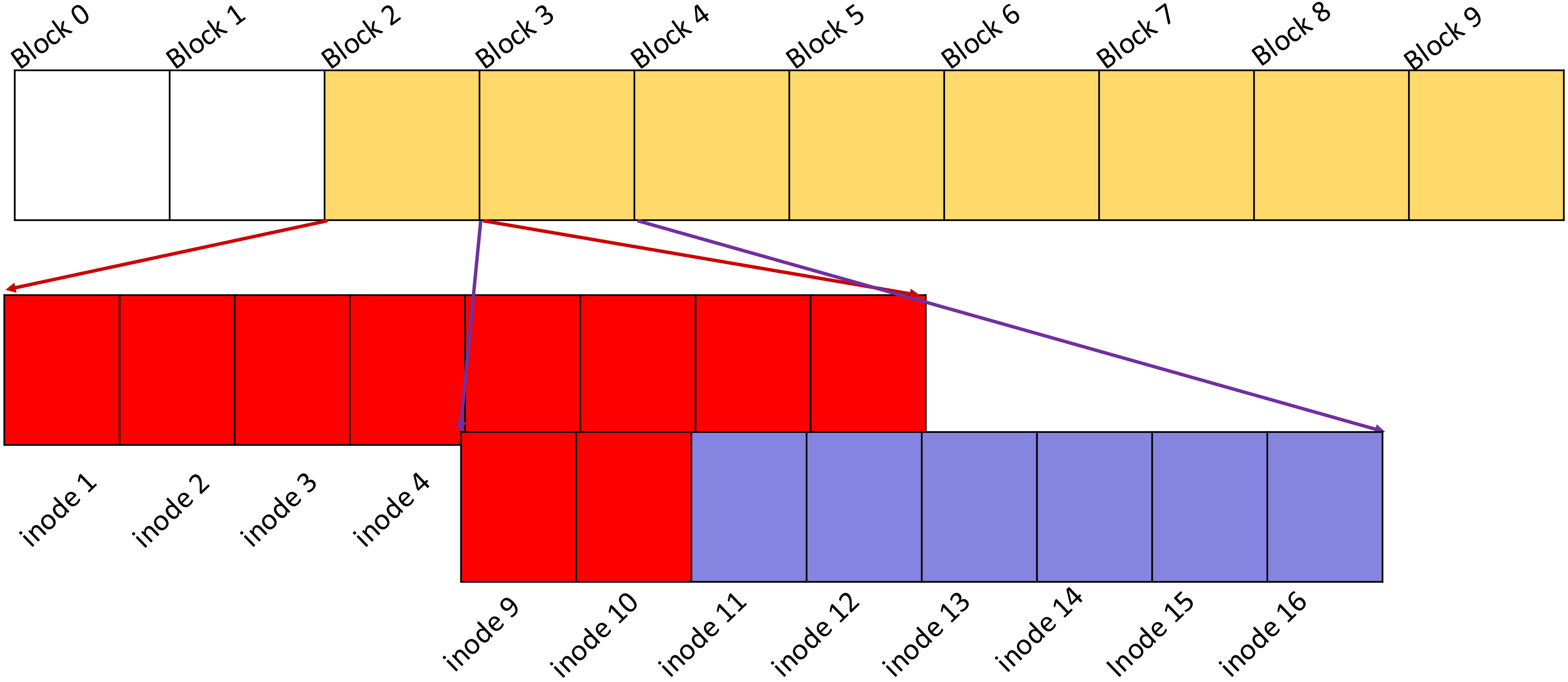
# Inodes at a glance: Physical Block 2



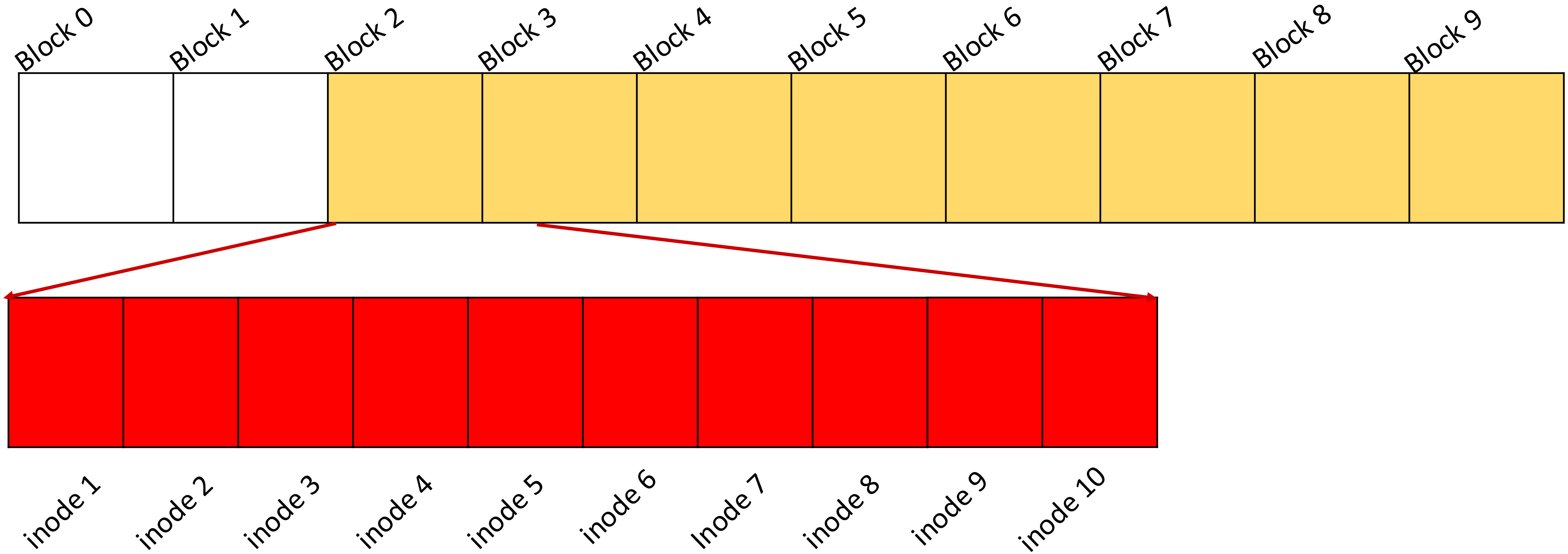
# Lecture Outline

- Linux Filesystem Implementation
  - Reserved Inodes
  - Root Inode
- File Paths
  - Absolute Paths & Relative Paths
- Resolving Absolute Paths
  - Directory Entries
- Accessing a Struct Dirent
- Putting It All Together
  - Bitmaps
  - Super Block
  - Boot Block

# Inodes at a glance: Reserved Inodes

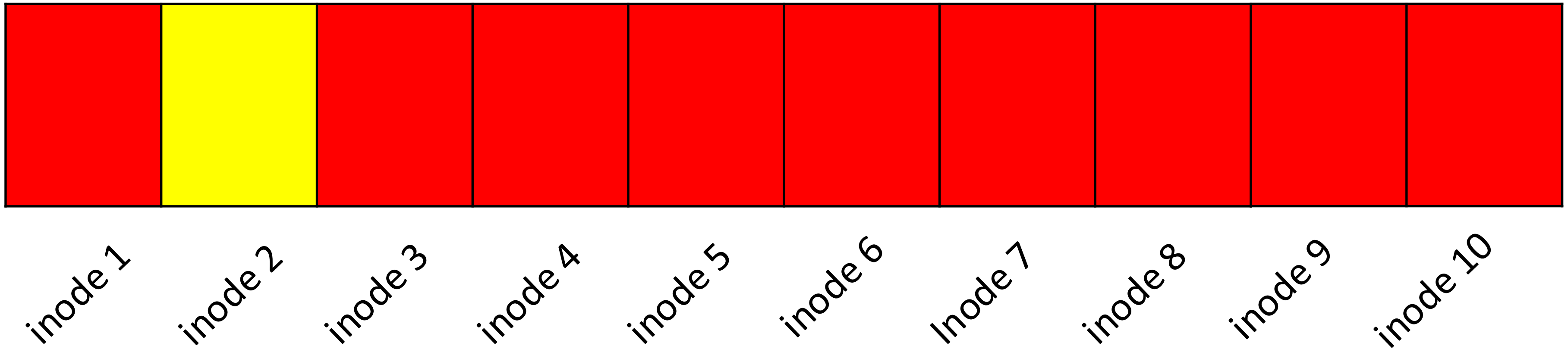


# Inodes at a glance: Reserved Inodes



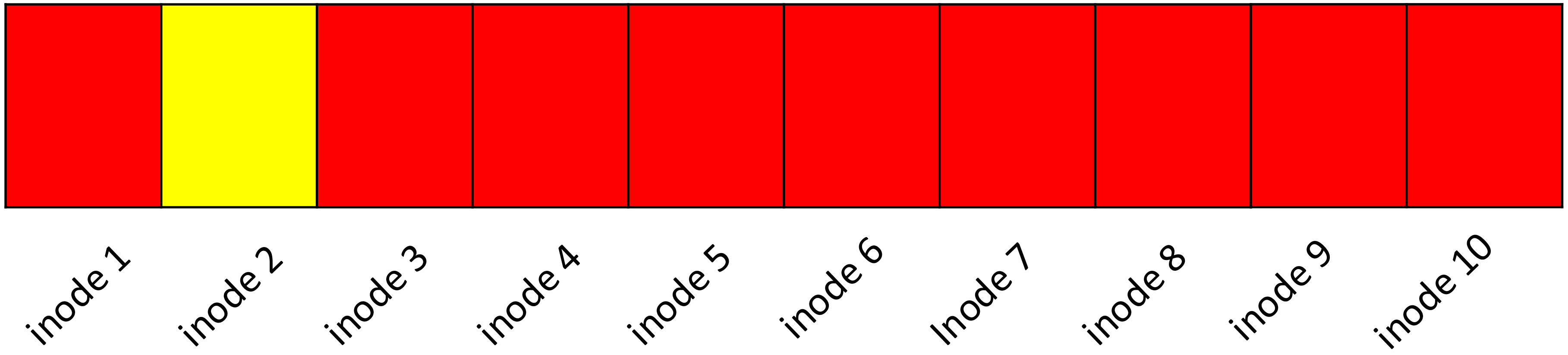

***There are up to 10 inodes that are reserved for special purposes, although many are unused and were never implemented for their use case.***

# The Most Important Special Inode: Inode 2




- Inode 2
  - Is the “root” directory of the filesystem (i.e. '/' )
  - Does not contain the physical file information for the root directly
  - The block numbers within the Inode tell us ***where*** to expect the data to be.

# The Most Important Special Inode: Inode 2




- Inode 2
  - Is the "root" directory of the filesystem (i.e. '/' )
  - `/Users/joelrmrz/Documents/file.txt`

The corresponding inode for this directory is inode 2.

**Wait, so where is the rest of "`Users/joelrmrz/Documents/file.txt`" ?**

# Lecture Outline

- Linux Filesystem Implementation
  - Quick Review
  - Reserved Inodes & Root Inode
- File Paths
  - Absolute Paths & Relative Paths
- Resolving Absolute Paths
  - Directory Entries
- Accessing a Struct Dirent
- Putting It All Together
  - Bitmaps
  - Super Block
  - Boot Block

# Resolving Paths; Looking for a file

- Let’s say I want to find: **/Users/joelrmrz/Documents/file.txt**
  - How would you usually go about it?
- You’d probably search for the Users directory first,
  - then **joelrmrz**,
  - then **Documents,**
  - **and finally, file.txt.**
- **The way absolute pathnames are resolved is very similar, albiet, with a bit more technical details. ☺**

# Absolute Paths

- **/Users/joelrmrz/Documents/file.txt**
  - Is an example of an absolute path, where the entire path, starting from root, /, is given.
- Not an absolute path: file.txt
  - Doesn't tell us ***where in the file system*** we can find a file with that name unless we traverse the whole thing.

```
/root
├── dir_one
│   ├── file.txt
│   └── subdir_one
├── dir_two
│   ├── file.txt
│   └── subdir_two
├── dir_three
│   ├── file.txt
│   └── subdir_three
│
```

Which `file.txt` are we referring to?

No clue.

# Relative Paths

- **./file.txt**
  - Is an example of a relative path, where the ***starting directory*** is relative to the "."
  - Recall that a "." is a self reference to a directory.
- ./file.txt
  - Does tell us ***where in the file system*** we can find it, namely, in the directory we are 'in'.

```
/root
├── dir_one
│   ├── file.txt
│   └── subdir_one
├── dir_two
│   ├── file.txt
│   └── subdir_two
├── dir_three
│   ├── file.txt
│   └── subdir_three
│
```

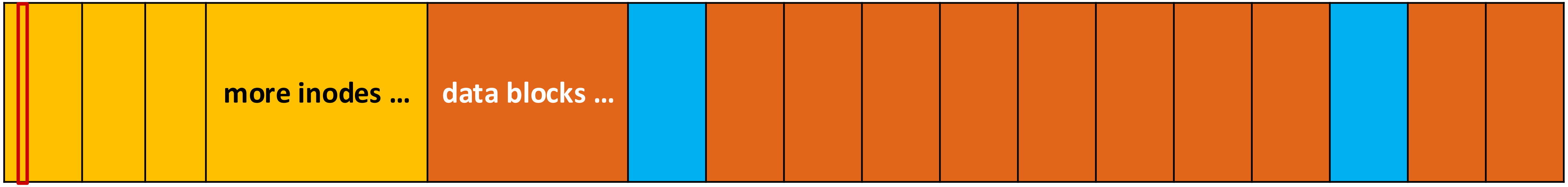
Which `./file.txt` are we referring to?

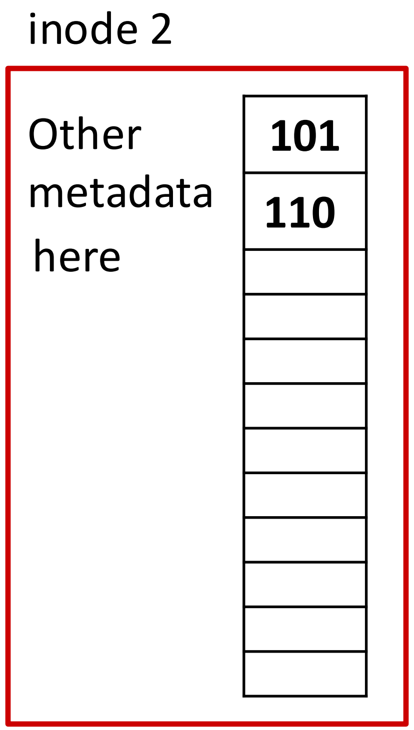
If we are "in" `dir_one`,
then we are referring to `/root/dir_one/file.txt`
And so forth.

# Lecture Outline

- Linux Filesystem Implementation
  - Quick Review
  - Reserved Inodes & Root Inode
- File Paths
  - Absolute Paths & Relative Paths
- Resolving Absolute Paths
  - Directory Entries
- Accessing a Struct Dirent
- Putting It All Together
  - Bitmaps
  - Super Block
  - Boot Block

# Walkthrough: Resolving an Absolute Pathname




Let's say I want to find: **/Users/joelrmrz/Documents/file.txt**

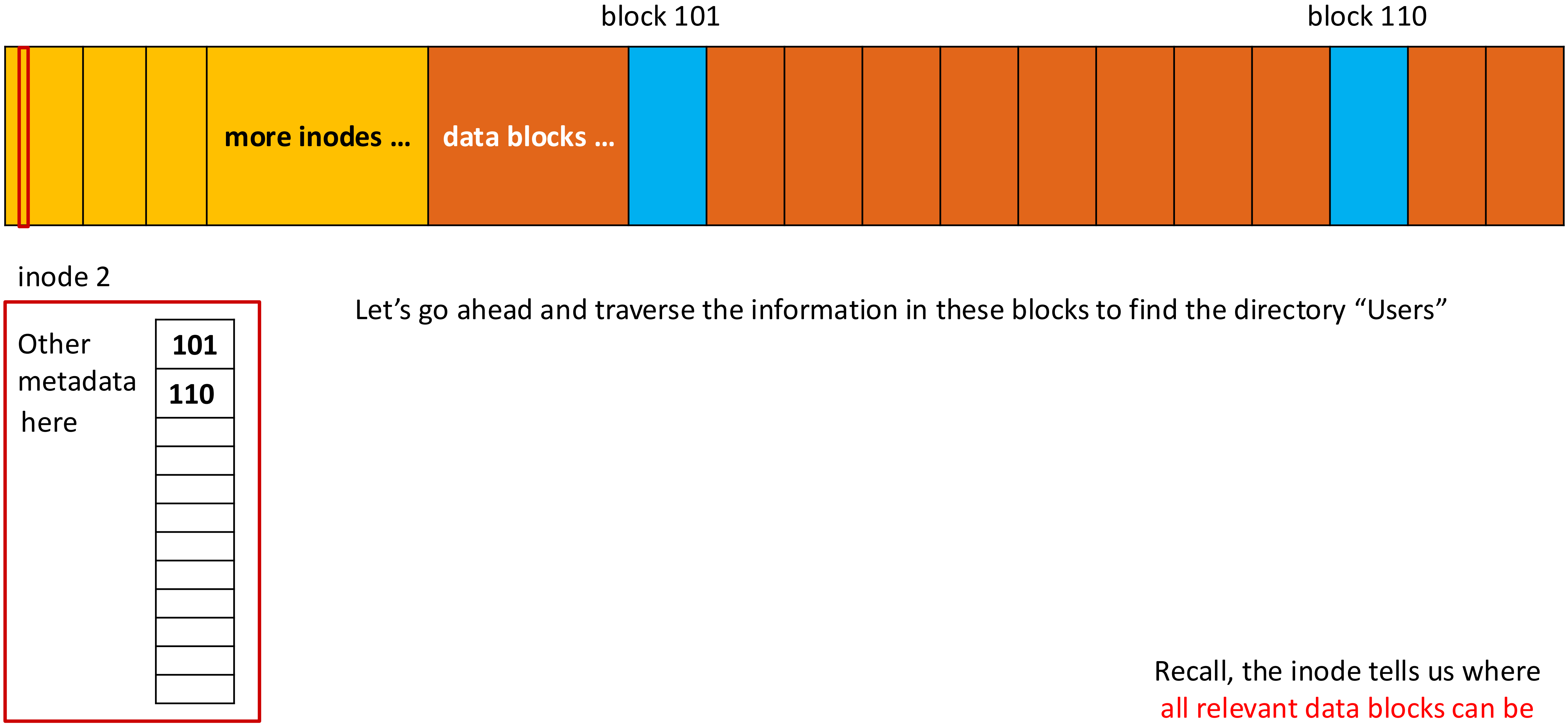
**First:** we need to look for the "Users" directory starting from the root directory. But, ***where*** exactly is the ***root directory***
***(i.e. what blocks contain the actual data for the root directory?)***

The inode tells us where! Blocks 101 and 110!

assume we only need direct block numbers

Recall, the inode tells us where all relevant data blocks can be found ***for a directory/file!***

# Walkthrough: Resolving an Absolute Pathname



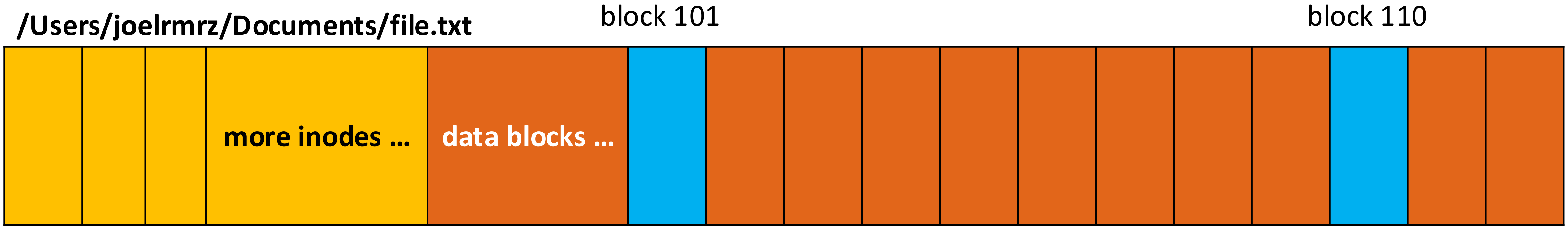

Let’s go ahead and traverse the information in these blocks to find the directory “Users”

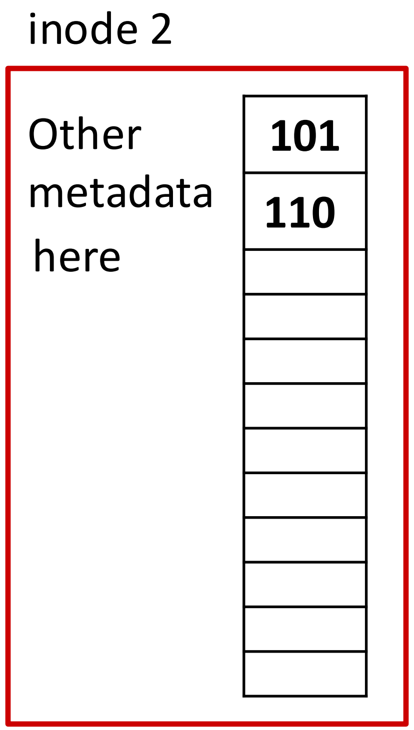
assume we only need direct block numbers

Recall, the inode tells us where all relevant data blocks can be found ***for the directory/file!***

# Walkthrough: Resolving an Absolute Pathname

**/Users/joelrmrz/Documents/file.txt**

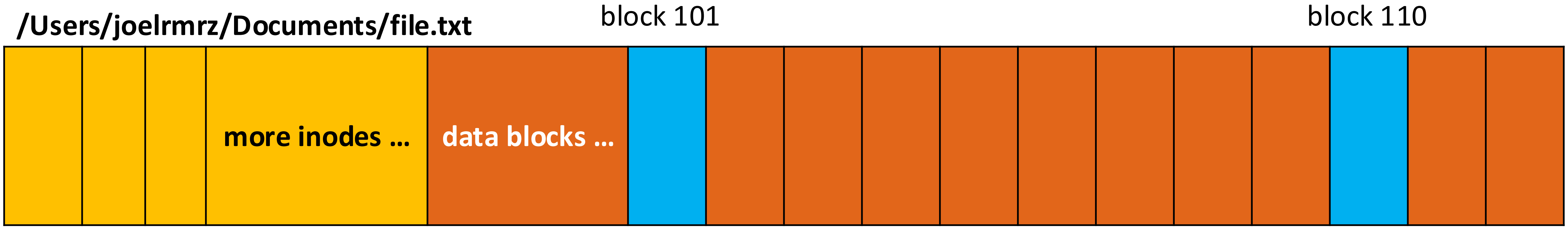


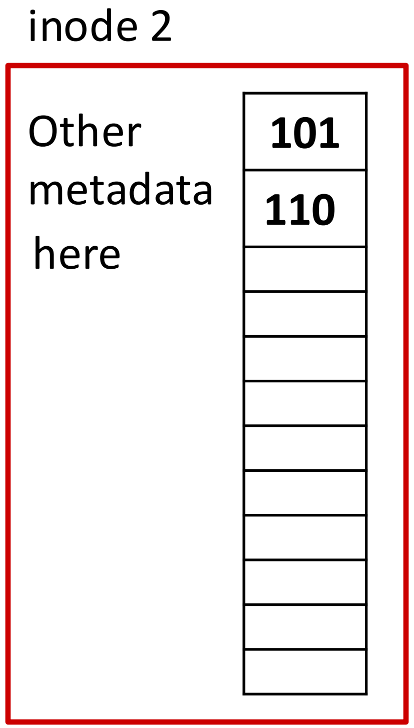




- Blocks 101 & 110 hold the information of directory “/”
  - These are the files/directories we would find ‘*inside*’ the root directory.
  - We need to look for the ***directory entry*** “`Users`” within the data blocks for the root directory.

# Walkthrough: Resolving an Absolute Pathname

**/Users/joelrmrz/Documents/file.txt**




- The directory entries in the root directory.

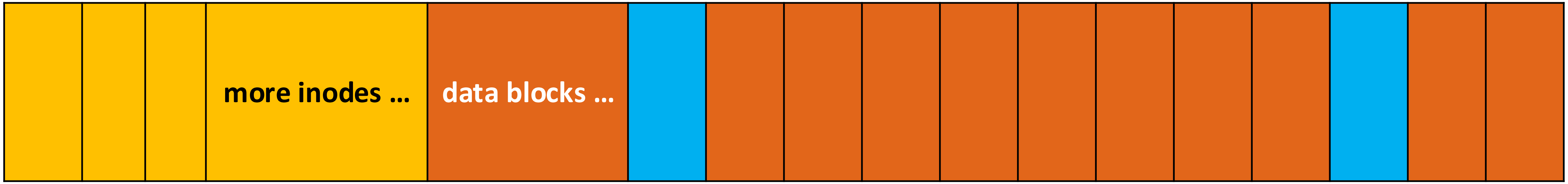
```
/root
├── bin
├── usr
├── USERS
├── .
├── ..
│
├── tmp
└── dev
```

These are the “entries” within the root directory. Shorted to the name “dirent” for directory entry.

# A Necessary Evil: Directory Entries

**/Users/joelrmrz/Documents/file.txt**

block 101 block 110

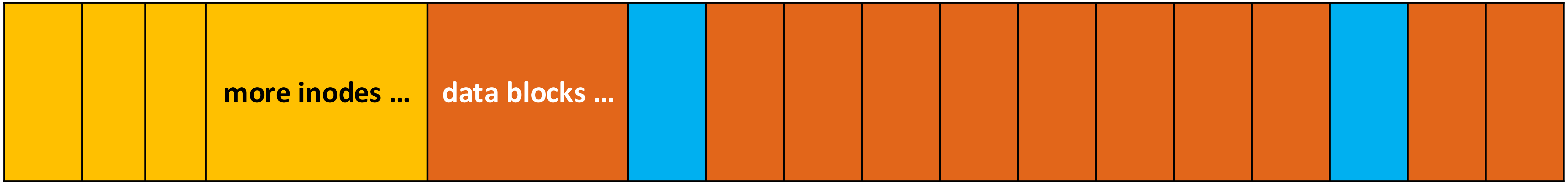



- Blocks 101 & 110 hold the information of directory "/"
  - These are the files/directories we would find '*inside*' the root directory.
  - We need to look for the ***directory entry*** "`Users`" within the data blocks for the root directory.
- Directory Entries
  - Are structs that ***represent*** the files/subdirectories within a directory.

```
struct dirent {
    ino_t d_ino;        /* File inode number */
    char d_name[];      /* Null terminated name of file */
    uint8_t d_type;     /* Indicator of file type ONLY IN ext2, 3, 4 */
}
```

# A Necessary Evil: Directory Entries

**/Users/joelrmrz/Documents/file.txt**



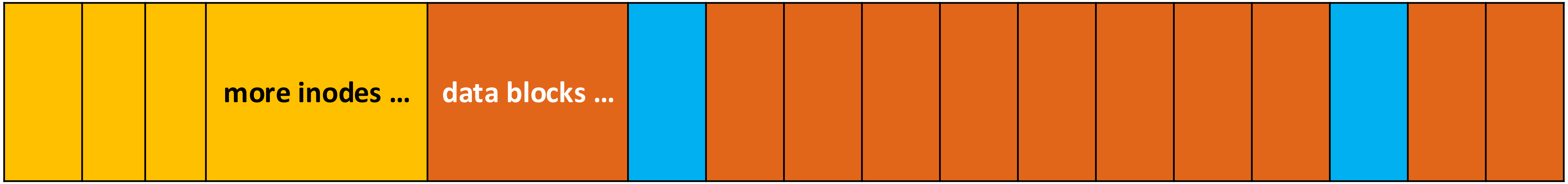

- Directory Entries
  - Are structs that ***represent*** the files/subdirectories within a directory.

```
struct dirent {
    ino_t d_ino;        /* File inode number */
    char d_name[];      /* Null terminated name of file */
    uint8_t d_type;     /* Indicator of file type ONLY IN ext2, 3, 4 */
}
```

- If we come across **`d_name` == "Users"**, we know where to look for "joelrmrz" now
  - namely, in the data blocks for the corresponding inode for this entry, **`d_ino`**.

# A Necessary Evil: Directory Entries

**/Users/joelrmrz/Documents/file.txt**



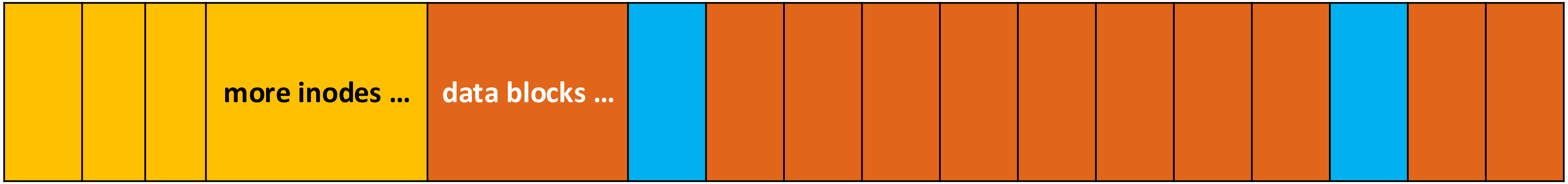

- Directory Entries
  - Are structs that ***represent*** the files/subdirectories within a directory.

```
struct dirent {
    ino_t d_ino;         /* File inode number */
    char d_name[];       /* Null terminated name of file */
    uint8_t d_type;      /* Indicator of file type ONLY IN ext2, 3, 4 */
}
```

- d_type
  - Not defined in ALL file systems but is useful in telling us the ***type*** of file we are referring to.
  - Directory, Regular File, Buffer File (pipe), and much more.

# A Necessary Evil: Directory Entries

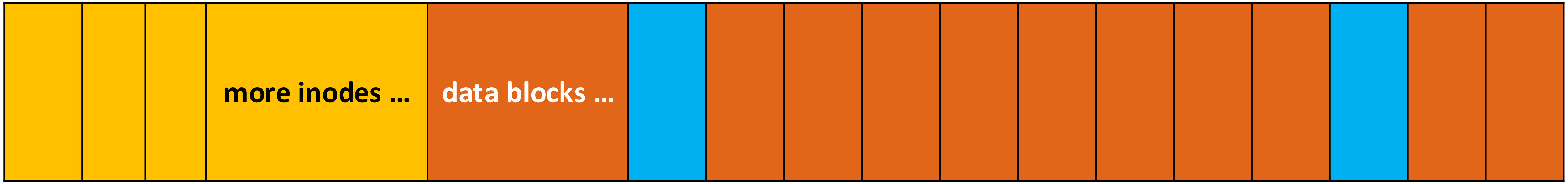
**/Users/joelrmrz/Documents/file.txt**

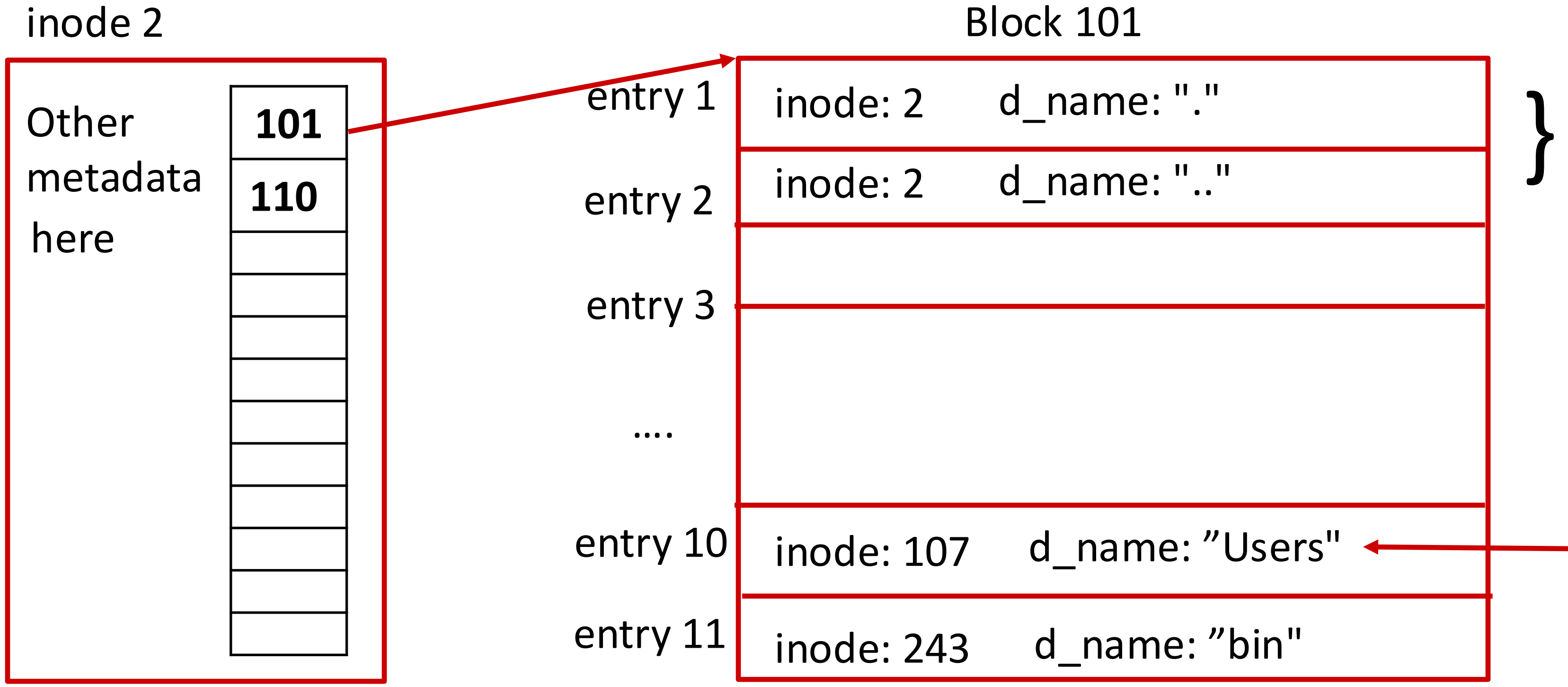



- Directory Entries
  - These ***exist within the file system itself***; the inode tells us the data blocks that contain these.
  - *You know where to start looking,* **inode 2.**
  - **You can not just start reading the values in a directory via "open"; if only life were that simple.**
    - **More on that later….**

# Walkthrough: Resolving an Absolute Pathname

**/Users/joelrmrz/Documents/file.txt**




Yup, these have their own directory entries.

We are in the root directory, so "..” just refers to the root again as it has no parent.

Awesome! Here it is!
But, ***where*** is Inode 101?

Poll Everywhere

pollev.com/cis5480

# *Which physical block contains inode 107?*

**Block 101**

| | | |
|---|---|---|
| entry 0 | inode: 2 | d_name: "." |
| entry 1 | inode: 2 | d_name: ".." |
| entry 2 | | |
| …. | | |
| entry 10 | inode: 107 | d_name: ”Users" |
| entry 11 | inode: 243 | d_name: ”bin" |

Assume:

- The inode table starts at Block 2
- A block is 1024 bytes
- Each inode is 128 bytes

**First ask yourself:**
**How would I find Inode 8? What about inode 15?**

We'll give you around 10 minutes to figure it out…really try to find out!

You'll have to do similar math to find which physical block FAT entries are in, you can't just index normally. You need to load the block first.

reminder: we start counting inodes from 1 ***(there is no inode 0)***

Poll Everywhere

pollev.com/cis5480

# *Which physical block contains inode 107?*

Left blank for any work you'd like to do here:

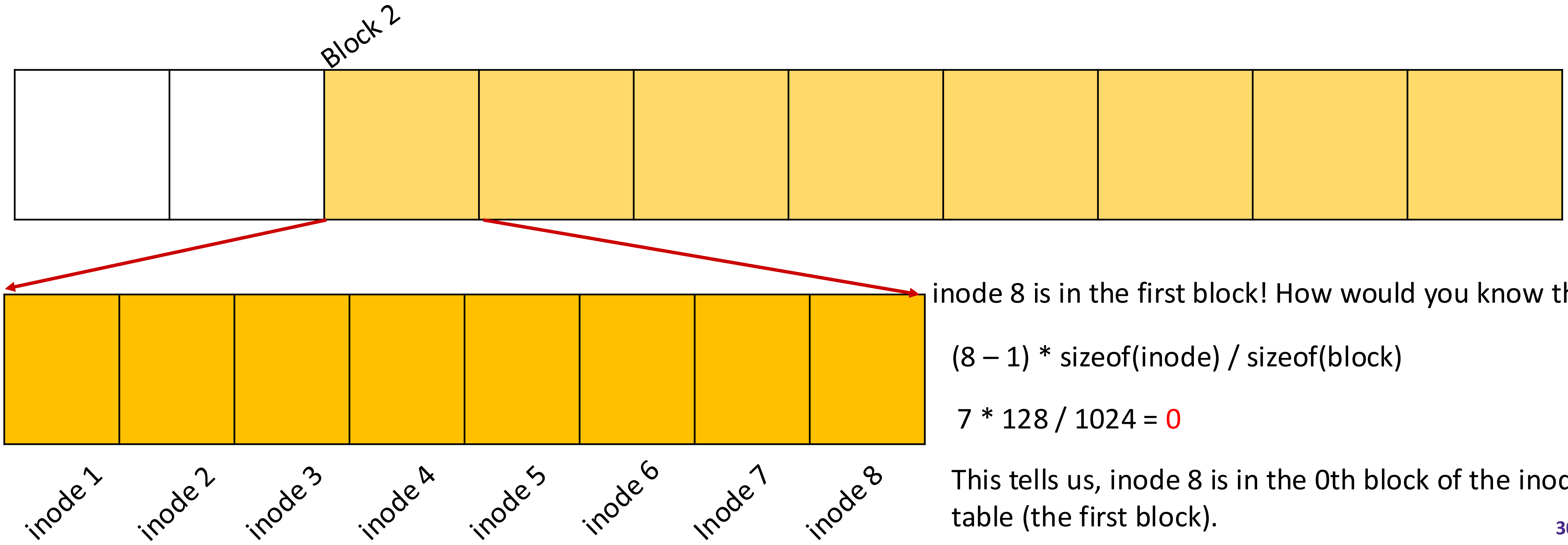
Poll Everywhere

pollev.com/cis5480

# *Which physical block contains inode 107?*

**First ask yourself:**

**How would I find inode 8?**




inode 8 is in the first block! How would you know this?

(8 – 1) * sizeof(inode) / sizeof(block)

7 * 128 / 1024 = 0

This tells us, inode 8 is in the 0th block of the inode table (the first block).

Poll Everywhere

pollev.com/cis5480

# *Which physical block contains inode 107?*

**First ask yourself:**

**How would I find inode 8?**

```
#define INODE_TABLE_BLOCK_START 2
```

How to find the physical block number that contains the inode?

```
block_number = INODE_TABLE_BLOCK_START + ( (inode_num – 1) * sizeof(inode) )/sizeof(block)

block_number = 2 + (8 – 1) * 128 / 1024

block_number = 2 + (7 * 128)/1024

block_number = 2 + 0

block_number = 2
```

Why is this zero?
Integer Division

Note: Need to do the multiplication first in software, integer division will make the quotient zero.

Poll Everywhere

pollev.com/cis5480

# *Which physical block contains inode 107?*

**First ask yourself:**

**What about inode 15?**

```
#define INODE_TABLE_BLOCK_START 2
```

How to find the physical block number that contains the inode?

```
block_number = INODE_TABLE_BLOCK_START + ((inode_num – 1) * sizeof(inode))/sizeof(block)

block_number = 2 + ((15 – 1) * 128)/1024

block_number = 2 + (14 * 128) / 1024

block_number = 2 + 1

block_number = 3
```

Why is this one?
Integer Division

```
#define INODES_PER_BLOCK sizeof(BLOCK)/sizeof(inode)
```

You could also do: `INODE_TABLE_BLOCK_START + ((inode_num – 1) / INODES_PER_BLOCK)`

Poll Everywhere

pollev.com/cis5480

# *Which physical block contains inode 107?*

**First ask yourself:**

**What about inode 15?**

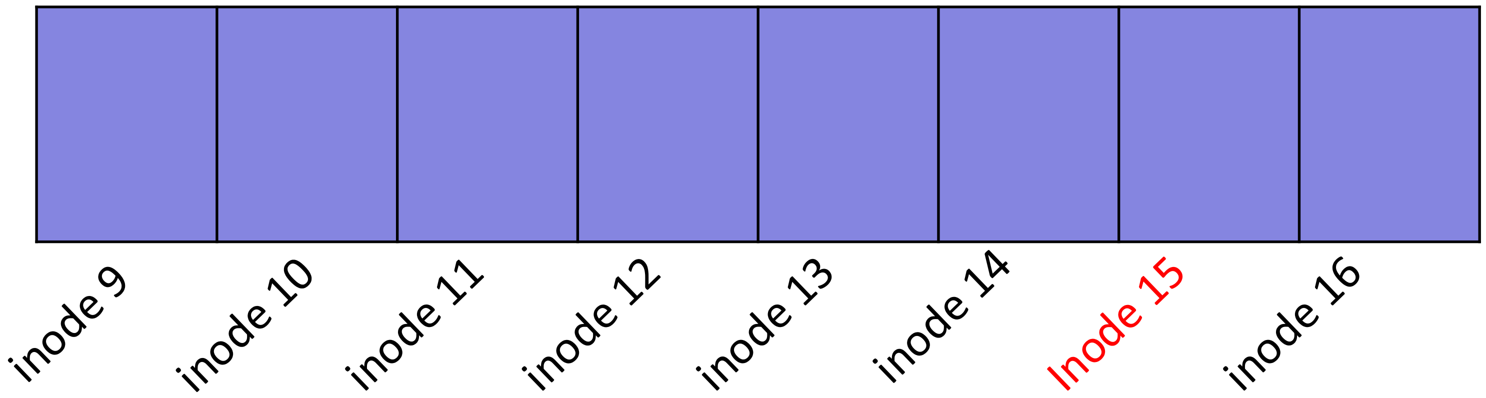
```
#define INODE_TABLE_BLOCK_START 2
```

How to find the physical block number that contains the inode?

```
block_number = INODE_TABLE_BLOCK_START + ((inode_num – 1) * sizeof(inode))/sizeof(block)
```

**`block_number = 3`**

The Second Block of the Inode Table (Physical Block 3)



# Poll Everywhere

pollev.com/cis5480

## *Which physical block contains inode 107?*

```
block_number = 2 + (inode_num – 1) * sizeof(inode) /sizeof(block)

block_number = 2 + ((107 – 1) * 128) / 1024

block_number = 2 + 13

block_number = 15
```

Poll Everywhere

pollev.com/cis5480

# *What is the index of the inode 107 relative to the block it is in?*

**Block 101**

| | | |
|---|---|---|
| entry 0 | inode: 2 | d_name: "." |
| entry 1 | inode: 2 | d_name: ".." |
| entry 2 | | |
| …. | | |
| entry 10 | inode: 107 | d_name: ”Users" |
| entry 11 | inode: 243 | d_name: ”bin" |

Assume:

- The inode table starts at Block 2
- A block is 1024 bytes
- Each inode is 128 bytes

**First ask yourself:**
**how would I find Inode 8? What about inode 15?**

reminder: we start counting inodes from 1 ***(there is no inode 0)***

Poll Everywhere

pollev.com/cis5480

## *What is the index of the inode 107 relative to the block it is in?*

Left blank for any work you'd like to do here:

## Poll Everywhere

pollev.com/cis5480

# *What is the index of the inode 107 relative to the block it is in?*

**First ask yourself:**

**What about inode 15?**

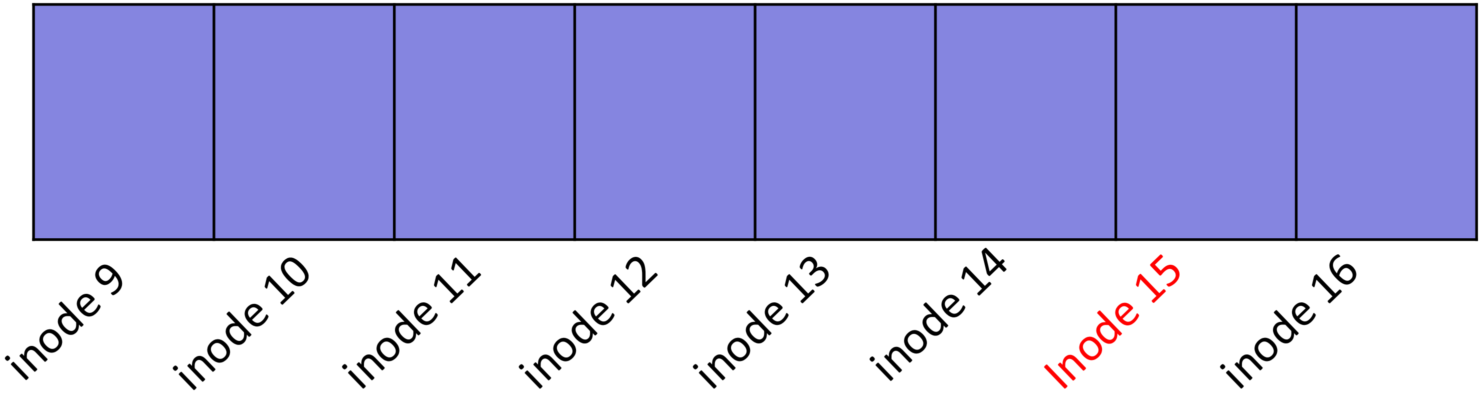
How would I be able to find the ***index*** of the inode relative to the physical block itself?

```
index = (inode_num – 1) % INODES_PER_BLOCK

index = (14) % 8 = 6
```

The Second Block of the Inode Table (Physical Block 3)

```
struct inode curr = inode_table_blocktwo[6];
```

psuedocode btw.



Poll Everywhere

pollev.com/cis5480

## So, where is inode 107?

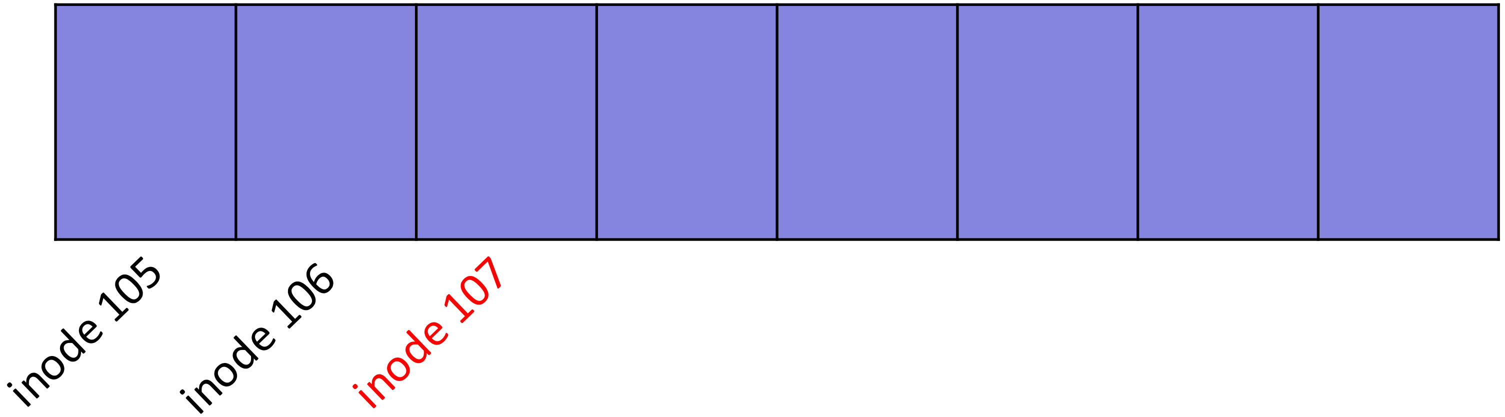
```
block_number = 2 + (inode_num – 1) * sizeof(inode) /sizeof(block)

block_number = 2 + ((107 – 1) * 128) / 1024

block_number = 2 + 13

block_number = 15

index = (inode_num – 1) % INODES_PER_BLOCK

index = (107 – 1) % 8

index = 106 % 8

index = 2
```

Poll Everywhere pollev.com/cis5480

# *So, where is inode 107?*

`block_number = 15   index = 2`

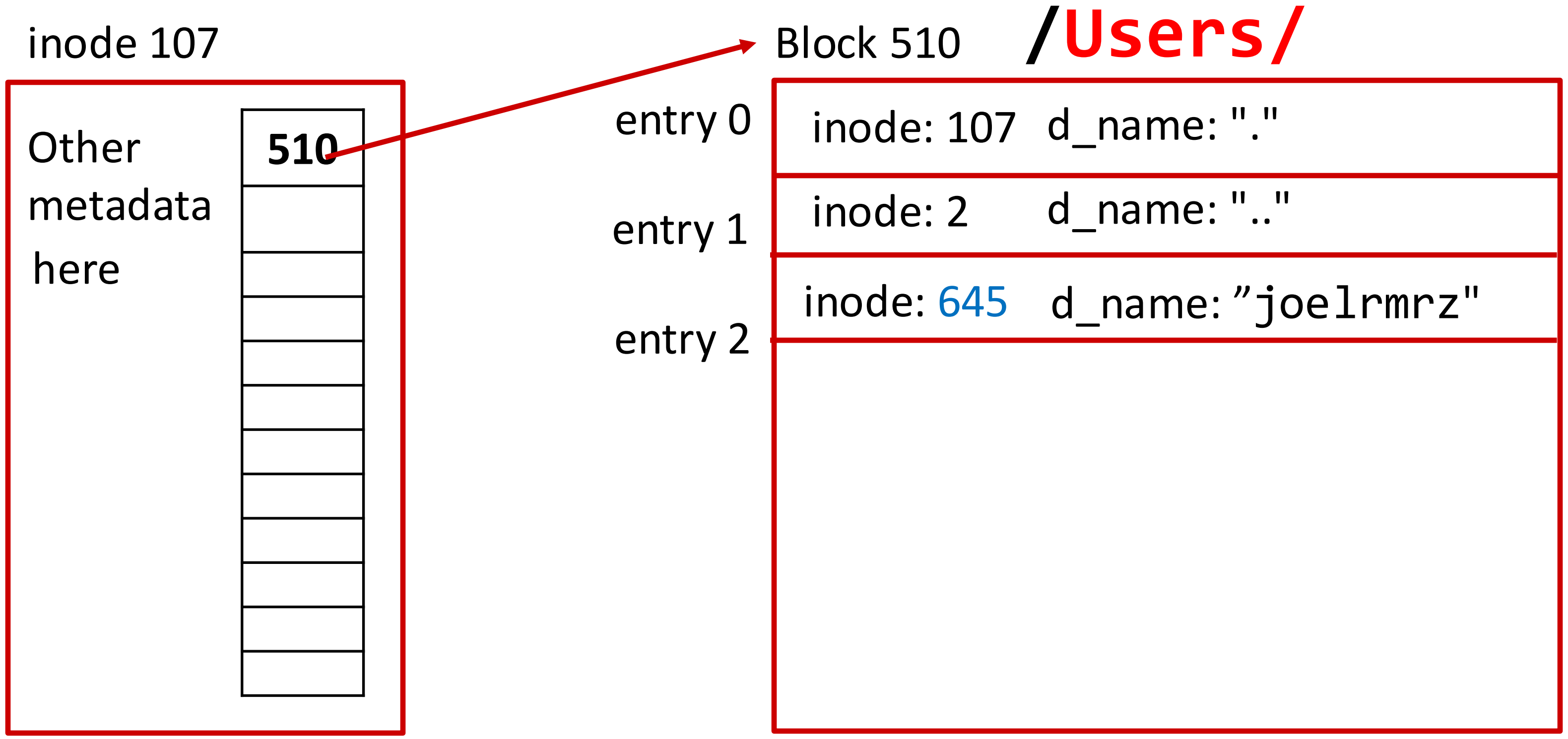
The InodeTableBlock[13] (Physical Block 15)




Finally, we're done!

# Directories in the Filesystem

- Now that we know where inode 107 is, let's look at its directory entries
  - Now, "." is the current directories inode and ".." is the root directory inode!
  - And there is only one user on my computer, so we only have 3 entries in this directory.
  - Now, we go to inode 645 and continue on doing the same thing.




We are currently "inside" the Users/ directory; or rather examining the entries for this directory.

note: there is no natural ordering to the dirents.

# Resolving Absolute Paths for *Regular Files*

- To open `/dir_one/dir_two/dir_three/file.txt`:
  1. Start from the root directory (inode 2).
  2. Check the data blocks associated with this inode.
  3. If it's not a directory, you've found your file! Otherwise, continue.
  4. Iterate through the directory entries.
  5. If d_name matches the target file or directory, follow its inode (d_inode).
  6. Repeat from step 2 with the new inode.
  7. If d_name is not found at any step, ***the path cannot be resolved.***
     - ***i.e. the file does not exist***

If directories are huge, this requires calculating singly, doubly, or even triple indirect blocks.

The way to resolve paths to directories is incredibly similar.

# Poll Everywhere

**pollev.com/cis5480**

To open `/dir_one/dir_two/dir_three/file.txt`, what is the minimum number of inodes that must be accessed to fully resolve the path and retrieve the data blocks of file.txt? Assume you have not accessed the root inode.

## Poll Everywhere

pollev.com/cis5480

To open `/dir_one/dir_two/dir_three/file.txt`, what is the minimum number of inodes that must be accessed to fully resolve the path and retrieve the data blocks of file.txt? Assume you have not accessed the root inode.

Let's walk through this:

- The first inode we need is inode 2, the root directory
- Next, we need the inode for dir_one
- Next, we need the inode for dir_two
- Next, we need the inode for dir_three
- Next, we need the inode for file.txt
  - This final inode contains the block numbers with the data which contains the ***actual file.txt.***

So, in total, 5 inodes!

# Poll Everywhere

pollev.com/cis5480

What is the minimum number of directory entries (struct dirents) that must be traversed to resolve the path `/dir_one/dir_two/dir_three/file.txt` and access the data blocks of file.txt in the best-case scenario? Assume you have not accessed the root inode.

# Poll Everywhere

pollev.com/cis5480

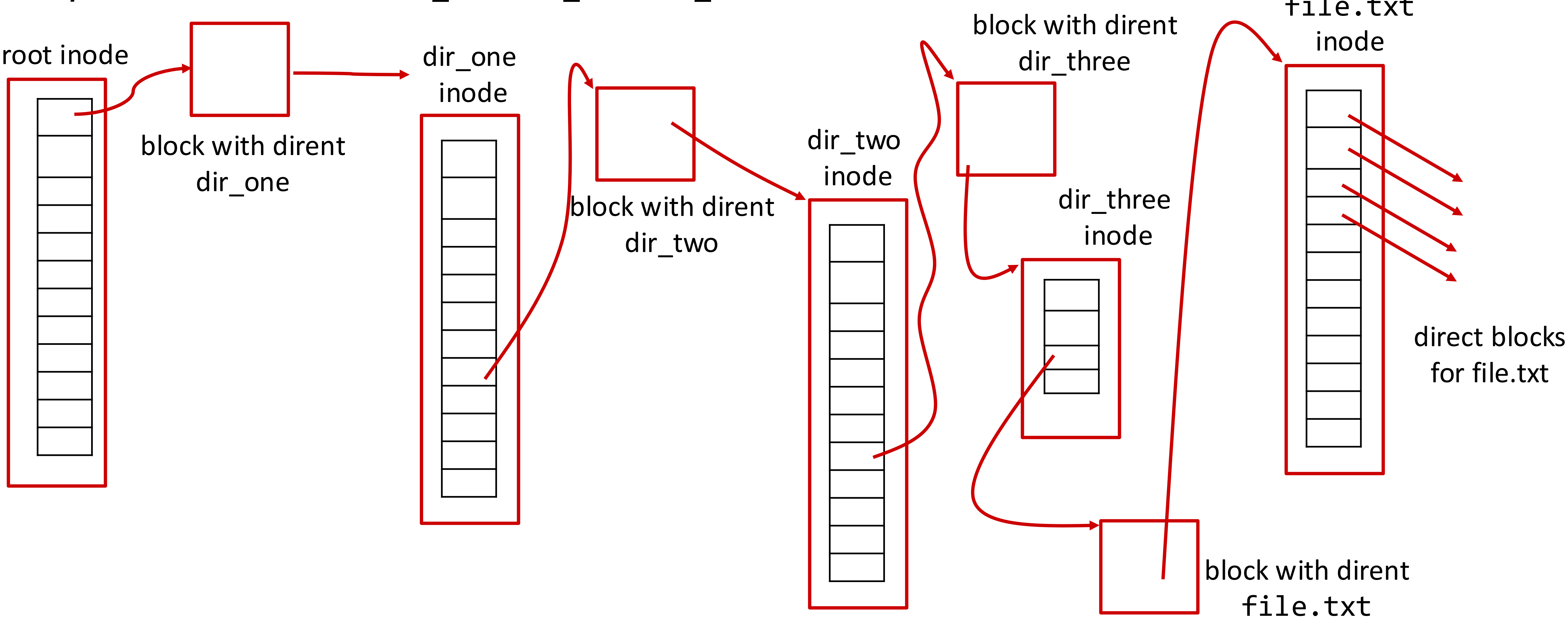
What is the minimum number of directory entries (struct dirent) that must be traversed to resolve the path `/dir_one/dir_two/dir_three/file.txt` and access the data blocks of file.txt in the best-case scenario? Assume you have not accessed the root inode.

Let's walk through this best case scendario:

- The first `struct dirent` we encounter within the data blocks of the root inode is
  - the directory entry for dir_one
- The second `struct dirent` we encounter in the data blocks of the dir_one inode is
  - the directory entry for dir_two
- The third `struct dirent` we encounter in the data blocks of the dir_two inode is
  - the directory entry for dir_three
- The fourth `struct dirent` we encounter in the data blocks of the dir_three inode is
  - the directory entry for file.txt
- And we are done!

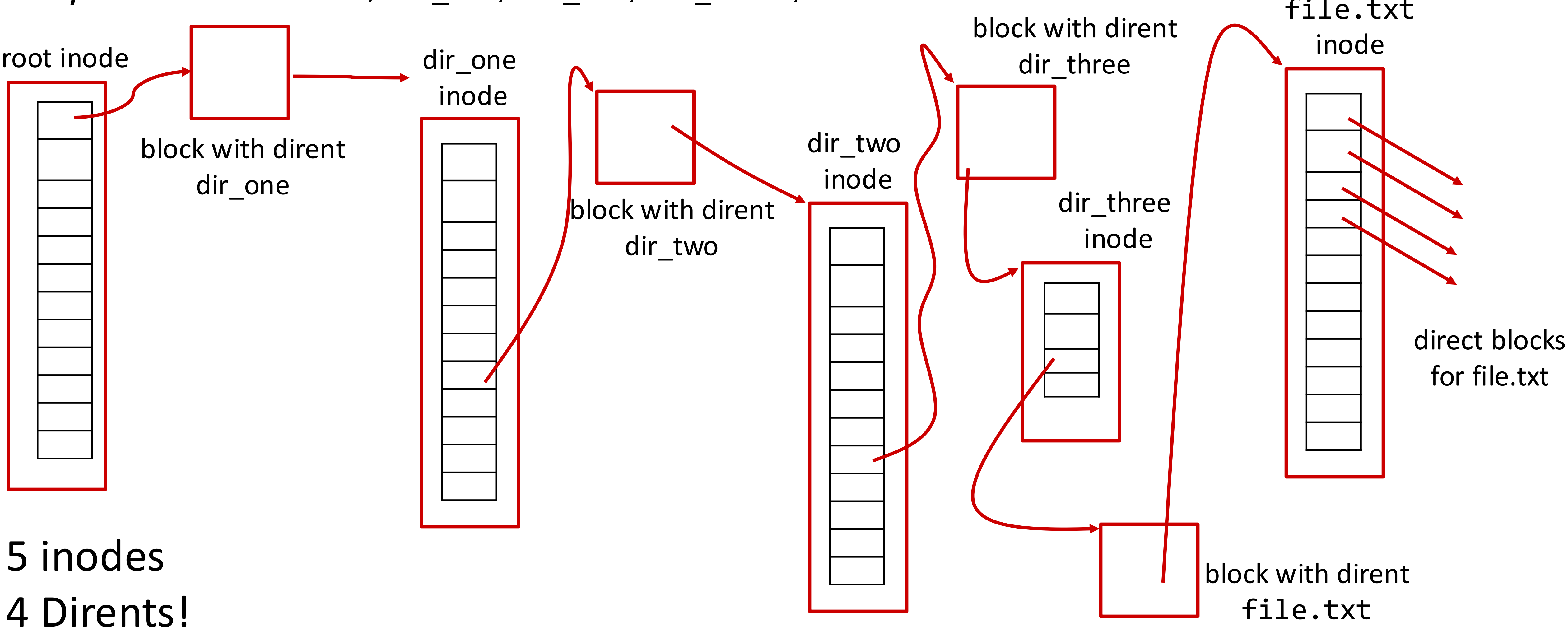
# A Possible Scenario for Resolving Paths

`A` ***`possible`*** `scenario /dir_one/dir_two/dir_three/file.txt`



# Resolving Absolute Paths for *Regular Files*

A ***possible*** scenario /dir_one/dir_two/dir_three/file.txt



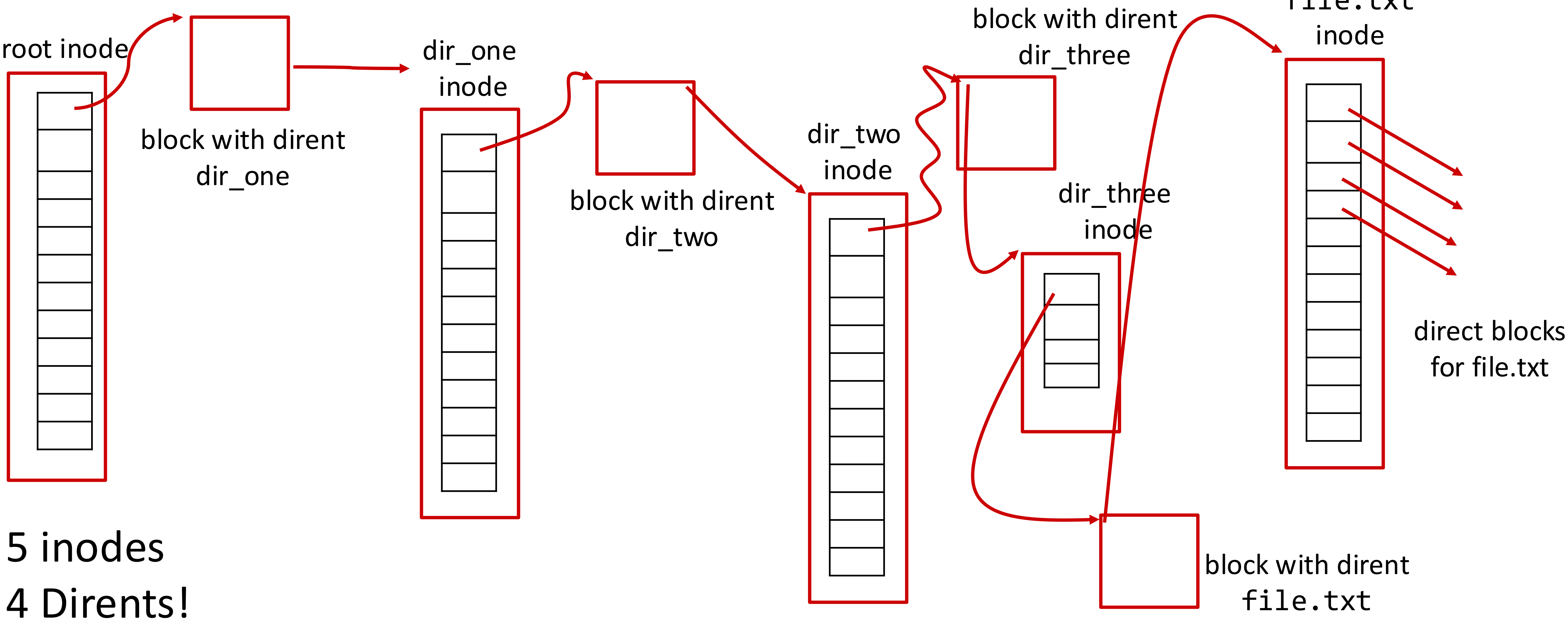

5 inodes

4 Dirents!

# Poll Everywhere

pollev.com/cis5480

A ***perfect*** scenario `/dir_one/dir_two/dir_three/file.txt`




5 inodes

4 Dirents!

# Lecture Outline

- Linux Filesystem Implementation
  - Quick Review
  - Reserved Inodes & Root Inode
- File Paths
  - Absolute Paths & Relative Paths
- Resolving Absolute Paths
  - Directory Entries
- Accessing a Struct Dirent
- Putting It All Together
  - Bitmaps
  - Super Block
  - Boot Block

# Accessing a Struct Dirent

```
#include <dirent.h>
```

```
DIR *opendir(const char *pathname);
```

- returns a "directory stream" which is just a maintained iterator over the entries in the directory.

```
struct dirent *readdir(DIR *dir_stream);
```

- returns a struct dirent that is allocated by the kernal for you (do not free it)
  - Returns the dirent the `dir_stream` is currently pointing to
- The directory entry remains valid until the next call to **`readdir()`** or **`closedir()`** on the same directory stream, `dir_stream`.

```
int closedir(DIR *dir_stream)
```

- Does what you think it does.

# Filtering and Sorting Dirents

```
int scandir(const char *restrict pathname,
                struct dirent ***restrict direntry_pointer_array,
                typeof(int (const struct dirent *)) *filter,
                typeof(int (const struct dirent **, const struct dirent **)) *generic_compare);
```

- The **`scandir()`** function reads the directory `pathname` (relative or absolute) and builds an array of pointers to directory entries using malloc.
  - It returns the number of entries in the array.
  - A pointer to the array of directory entries is stored in the location referenced by *`direntry_pointer_array`*
- The `filter` argument is a pointer to a user supplied subroutine to select which entries are to be included in the array.
  - Should return a non-zero value if the directory entry is to be included in the array.
  - If `filter` is null, then all the directory entries will be included.
- The `generic_compare` argument is a pointer to a user supplied subroutine which is passed to qsort(3) to sort the completed array. If this pointer is null, the array is not sorted.

This is here purely for completeness. And gives you a way to manually implement the "ls" command.

# Lecture Outline

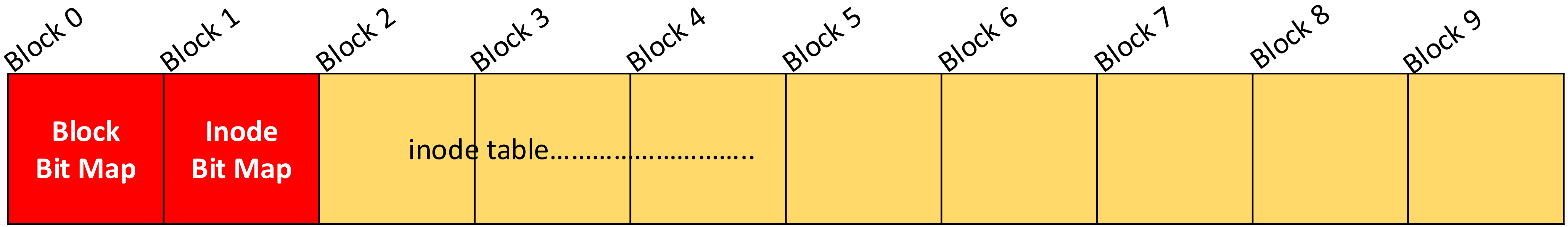
- Linux Filesystem Implementation
  - Quick Review
  - Reserved Inodes & Root Inode
- File Paths
  - Absolute Paths & Relative Paths
- Resolving Absolute Paths
  - Directory Entries
- Accessing a Struct Dirent
- Putting It All Together
  - Bitmaps
  - Super Block
  - Boot Block

# A more realistic rendition of the Linux Filesystem

- When the filesystem is formatted (set up), the entire inode table is created with a set number of inodes.
- Thus, the number of blocks that can hold file data is also limited and can be tracked.
- We need two things:
  - A bit map for the inodes to correspond to allocated and unallocated inodes.
  - A bit map for the blocks to correspond to allocated and unallocated blocks.
  - These bitmaps should take up at least one block each ***prior to the inode table.***

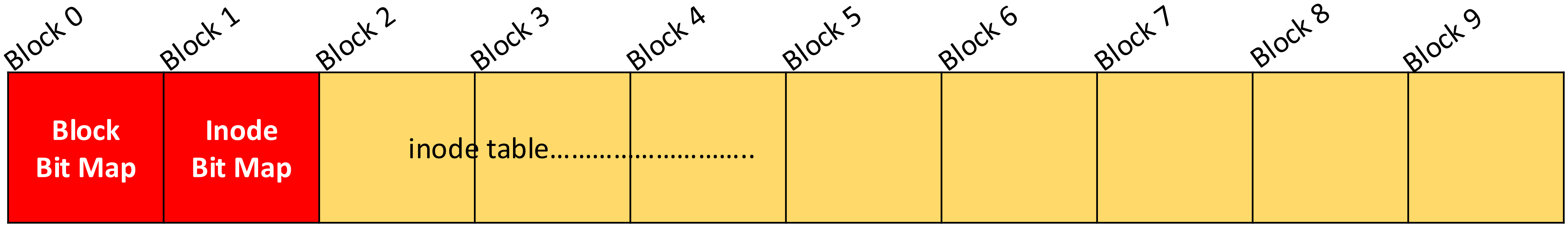
# A more realistic rendition of the Linux Filesystem

- We need two things:
  - A bit map for the inodes to correspond to allocated and unallocated inodes.
  - A bit map for the blocks to correspond to allocated and unallocated blocks.
  - These bitmaps take up at least one block each ***prior to the inode table.***



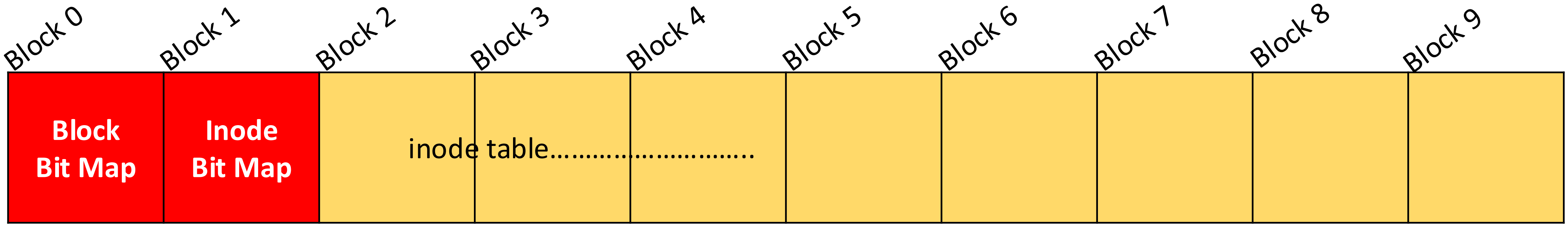
# A much more realistic rendition of the Linux Filesystem

- We have the bit maps now, what else do we need?
  - ***We don't have any information about the file system itself. Some questions we need answers to.***
    - How large are the blocks, how many inodes in the inode table, how many free blocks in total, how many blocks are allocated to data, how many blocks are hidden from the user (just for the os usage), what type of file system is this, what is the size of each inode, which is the first nonspecial inode, is the file system in a valid state, and much much more.



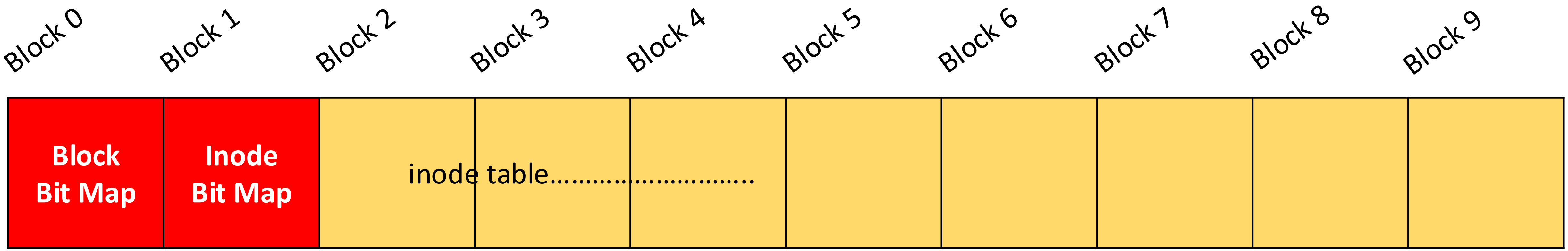
# A much more realistic rendition of the Linux Filesystem

- **The Super Block**
  - Is a block in the filesystem that contains metadata ***about*** the file system itself.
  - Used by the operating system to maintain the file system
    - Because it is so important, many copies of the super block are maintained within the file system.
    - Just incase the super block the kernel has becomes corrupted
    - Required overhead, as the superblock is written back to disk frequently.



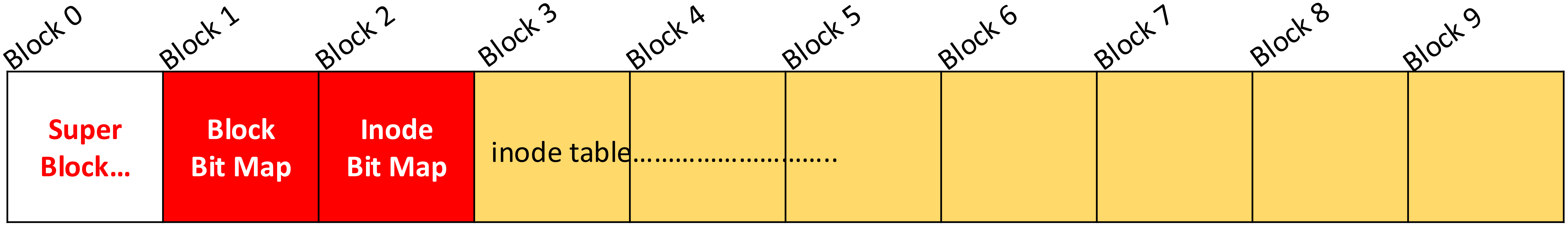
# A much more realistic rendition of the Linux Filesystem

- Linux ext2: Super Block Values
  - **inode_count, block_count, reserved_blocks for the kernel (why is this important?)**
  - **`free_inode_count, free_block_count, first_data_block, block_size,`**
  - **`filesystem_magic_number, error_no`**
  - **`first_real_inode`** **(version 0 set to 11, can be set to any in future versions)**
  - **`inode_table_start`**
  - **There are many many more; 1024 bytes worth of information.**



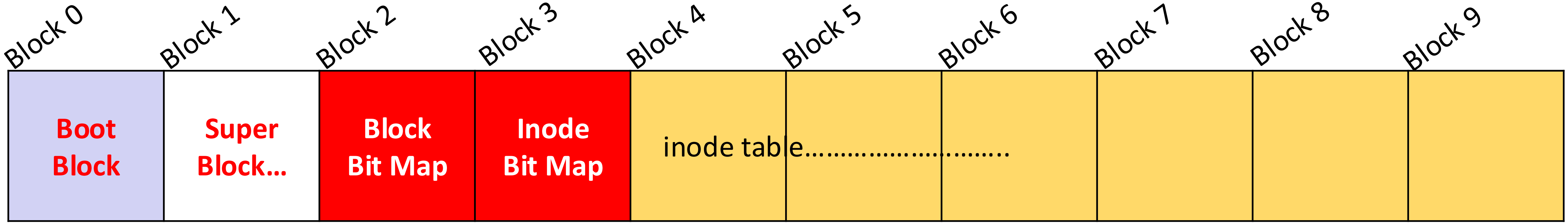
# A much more realistic rendition of the Linux Filesystem

- To make room for the Super Block, let's scoot everything over one block.
- ***Can you believe we're still missing one thing?***
  - ***Probably the most important piece…***



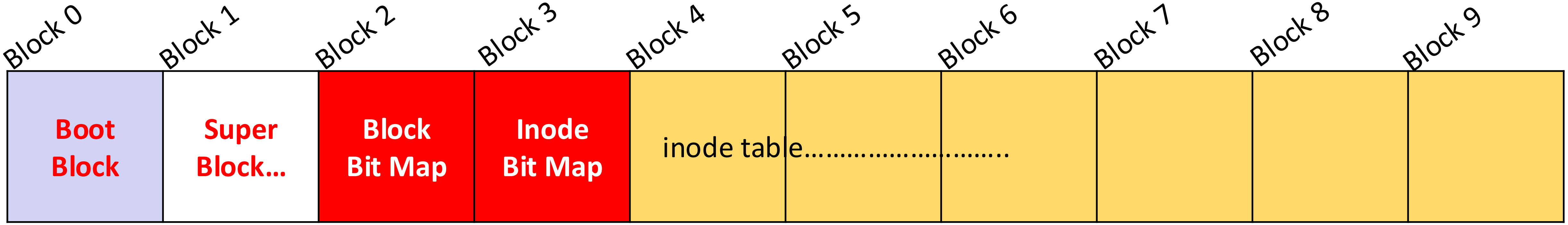
# A much more realistic rendition of the Linux Filesystem

- WHERE IS THE OPERATING SYSTEM/KERNAL?
  - It is ***not just going to appear on the computer.***
- ***The code for the operating system is stored within the filesystem itself!***
- It is stored at the start of the file system device and "only" takes up 1024 bytes; so let's scoot everything over.
- Ah, finally, a more realistic design.



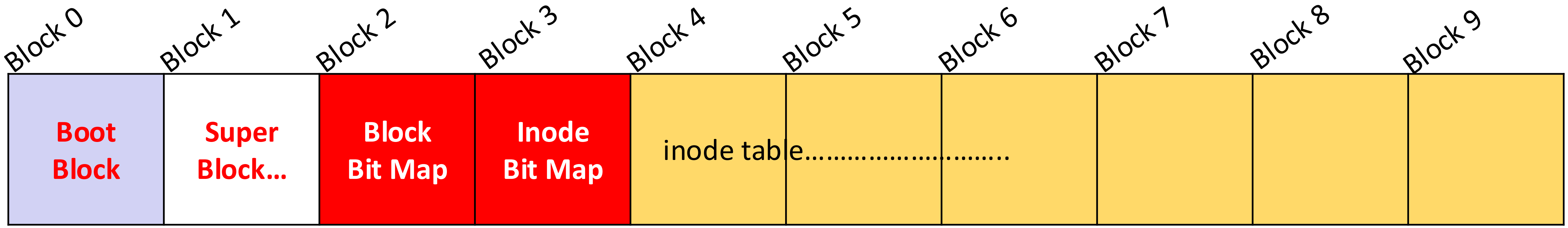
# A much more realistic rendition of the Linux Filesystem

- The boot block
  - Contains the initial segments of code necessary to bootstrap the operating system
    - (that is, so that the operating system can ***install itself***)
    - 1024 bytes might seem like a little bit, but really, the boot code is much smaller than this.
      - **512 bytes minimum needed for the boot block, *with 446 bytes dedicated to the actual bootloader itself. Talk about optimization.***
      - ***Why 512? (legacy support).***



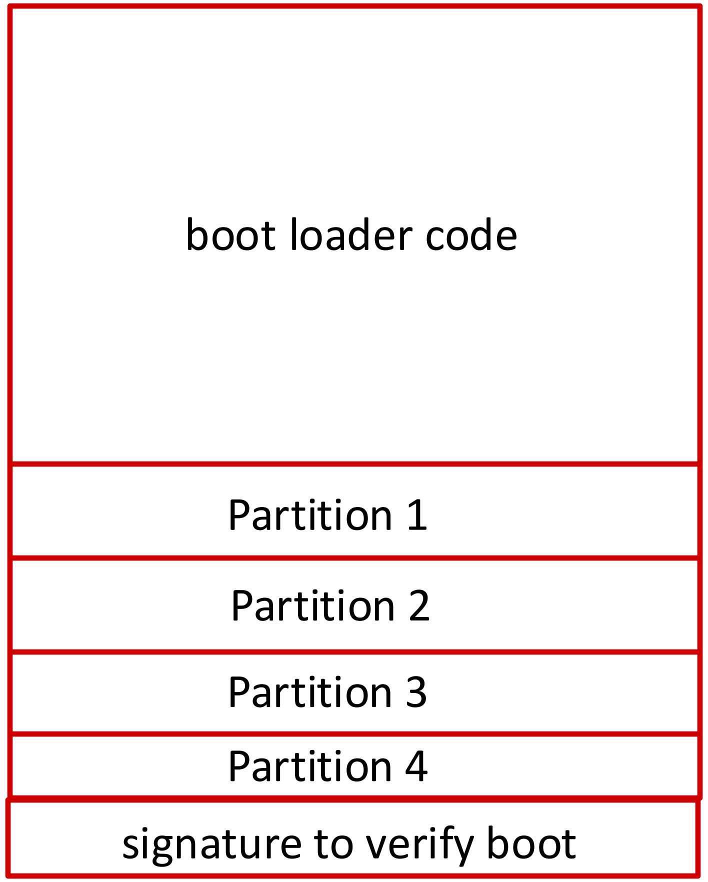
# A much more realistic rendition of the Linux Filesystem

- The boot block
  - 512 bytes minimum needed for the boot block, with 446 bytes dedicated to the actual bootloader itself. *Talk about optimization.*
  - The boot block also contains information not about the operating system itself, but also about the physical disk itself. It tells it (the cpu) ***where to find the rest of the OS code***. Because, the linux kernal is not going to fit in 446 bytes, that would be silly.



# A closer look at the boot block

- The boot block
  - 512 bytes minimum needed for the boot block, with 446 bytes dedicated to the actual bootloader itself. *Talk about optimization.*




The bootloader/pre-kernal code tells the cpu

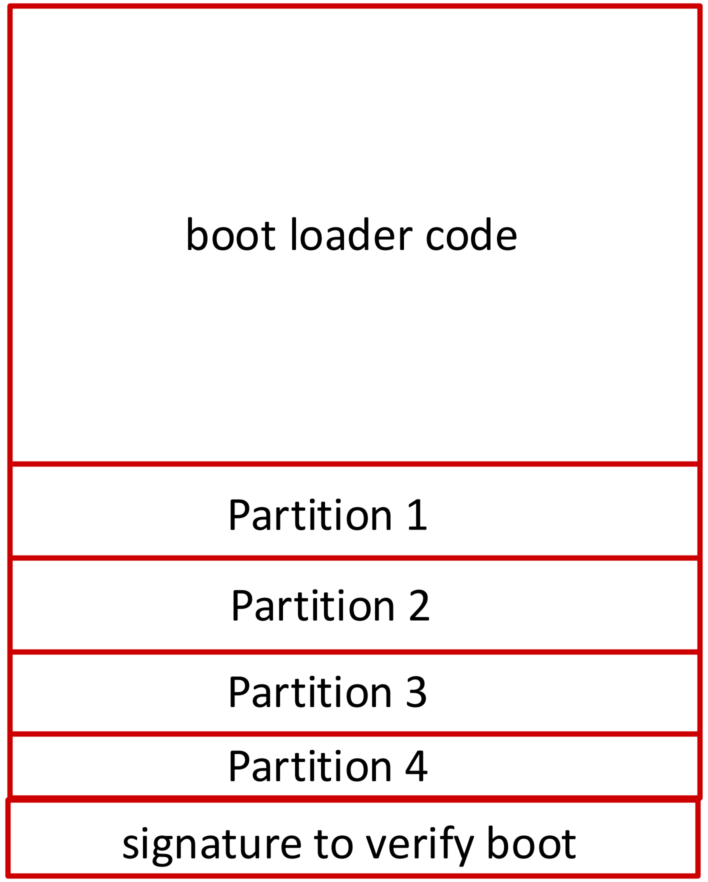
***What it should do to prepare itself to install the actual kernel.***

1. ***How to set up registers.***
2. ***How to set up memory allocation***
   A. ***(literally, a small stack and text segment, and others)***
3. ***Validate the file system***
4. ***Establish permissions (software execution permission)***
5. ***Examines the CPU to denote which type it is and what other hardware it can use.***
6. ***Tells it where to locate the rest of the kernel is within the filesystem.***

It is an extremely intricate processes that itself could be an entire class.

# A closer look at the boot block

- The boot block
  - 512 bytes minimum needed for the boot block, with 446 bytes dedicated to the actual bootloader itself. *Talk about optimization.*

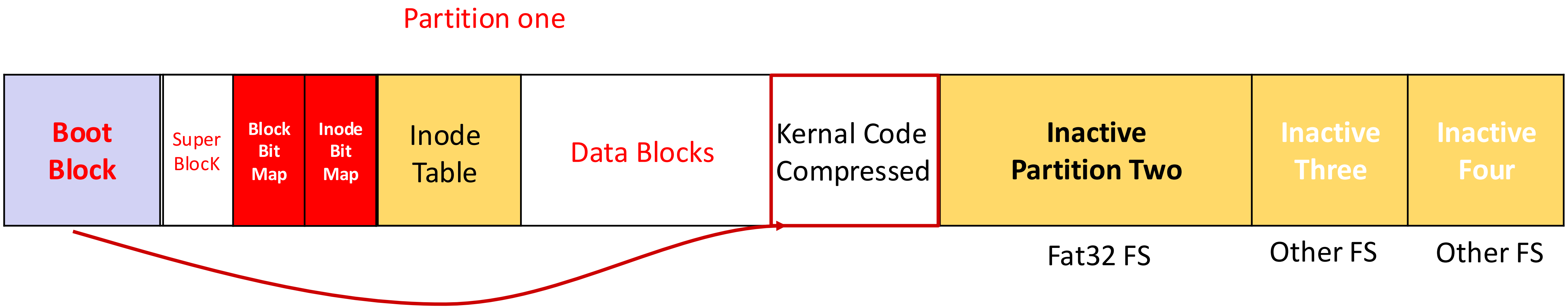



## What are these partitions?

They tell the boot loader “mini kernel” which portions of the file system denote which is ‘active’, that is, which contains the rest of the ***kernel code***. If you have both windows and macos on your machine, then you have two sperate partitions that are active and can be activated by you manually/by the firmware (BIOS).

They can also denote that you have other filesystems on the same disks (hence that it is partitioned).

# A real realistic rendition of the Linux Filesystem




You press the power button; what happens?

The hardware searches for attached storage devices (CDs, hard drives, flash drives, etc.) to find a **boot block**.

1. The **boot block** is always at the start of the device or contains a pointer to the actual location of the bootloader or kernel.
2. The **boot block** provides the CPU with its first non-firmware instructions, which are loaded into RAM and executed. This process sets up the system to install the full kernel, which is stored in the file system.
3. After setting up memory, initializing registers, and handling other necessary configurations, the CPU loads the kernel code from the location specified by the bootloader in the active partition.
4. The kernel then decompresses itself and loads into memory (RAM), either all at once or in parts.
5. It loads in the super block for the file system and configures everything (that it needs to).
6. And finally, you are running the operating system for the first time, and it spawns our first processes
   - A. **init** on linux, ***launchd*** on mac, ***sessionmanager*** on windows which are ***daemons***
7. Yeah, I know. ☺

# And that was file systems! Truly a miracle.

- Next up, Multithreading with Travis on Thursday!