# Lecture 03: Layering, Naming, and Filesystem Design

Principles of Computer Systems

Fall 2019

Stanford University

Computer Science Department

Instructors: Chris Gregg and

Phil Levis

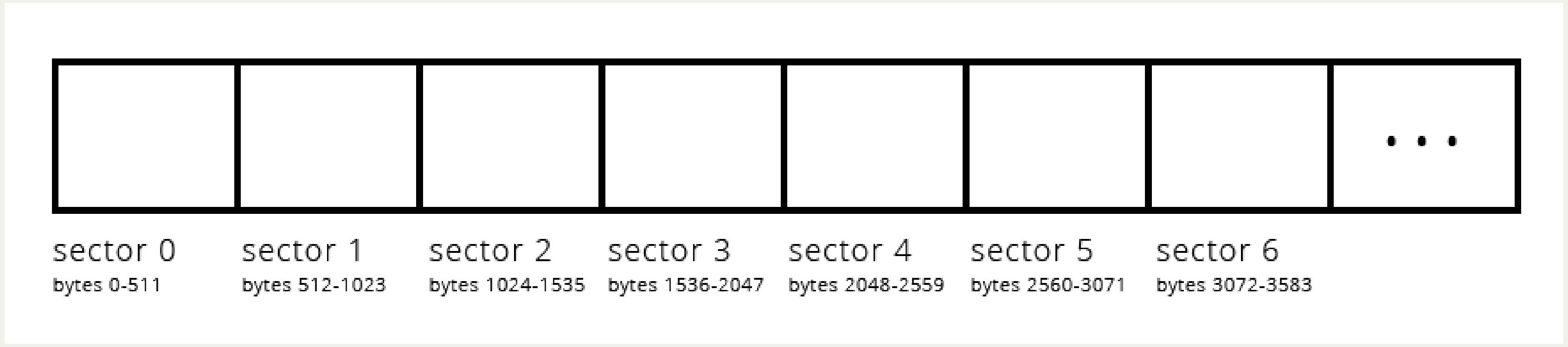
PDF of this presentation

## Lecture 03: Layering, Naming, and Filesystem Design

- Today, we are going to start discussing the *Unix version 6 file system*.
  - This is a relatively old file system (c. 1975), but it is open source, and was well-designed. It is simple and easy to understand (given that you take the time to understand it...)
  - Your second assignment is based on this file system
  - Modern file systems (particularly for Linux) are, in general, descendants of this file system, but they are more complex and geared towards high performance and fault tolerance. In other words -- they aren't the best file systems to learn from as your introduction to file systems. However, because of the beauty of open source, you can dig into the details of many modern file systems (e.g., the ext4 file system, which is the most common Linux file system right now)
  - So, for example, when we say that *a sector is 512 bytes*, know that this is for the Unix v6 file system, and not a general rule.
  - Some key takeaways from studying this file system:
    - You're seeing a part of computing history
    - You're investigating a good, thorough engineering design
    - You're learning details related to a particular file system, but with principles that are used in modern operating systems, too.
    - This is not the only way to create a file system!
      - In fact, it has some problems (which we will discuss)

## Lecture 03: Layering, Naming, and Filesystem Design

- Just like RAM, hard drives (or, more likely these days, *solid state drives*) provide us with a contiguous stretch of memory where we can store information.
- Information in RAM is *byte-addressable*: even if you're only trying to store a boolean (1 bit), you need to read an entire byte (8 bits) to retrieve that boolean from memory, and if you want to flip the boolean, you need to write the entire byte back to memory.
- A similar concept exists in the world of hard drives. Hard drives are divided into *sectors* (we'll assume 512 bytes), and are *sector-addressable*: you must read or write entire sectors, even if you're only interested in a portion of each.
- Sectors are often 512 bytes in size, but not always. The size is determined by the physical drive and might be 1024 or 2048 bytes, or even some larger power of two if the drive is optimized to store a small number of large files (e.g. high definition videos for youtube.com)
- Conceptually, a hard drive might be viewed like this:



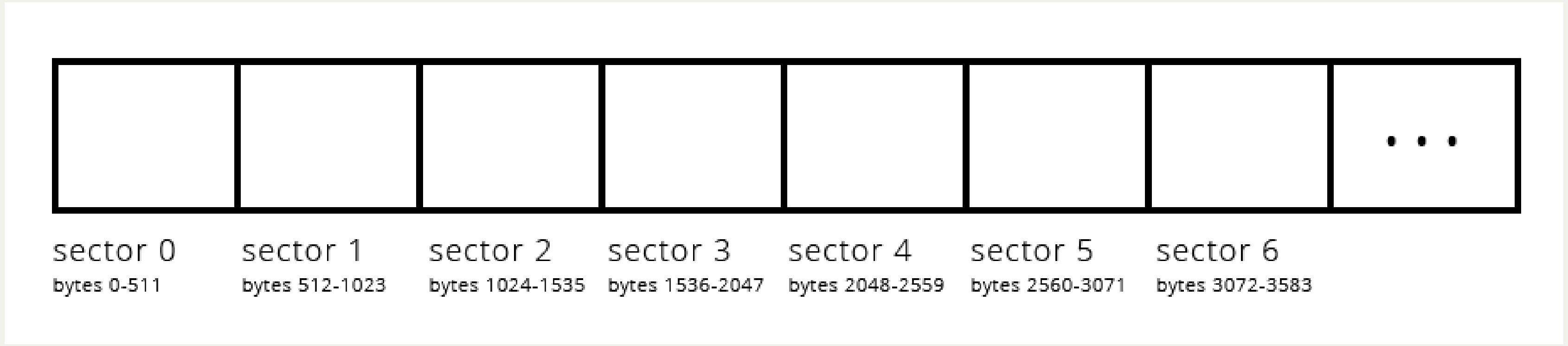
*Thanks to Ryan Eberhardt for the illustrations and the text used in these slides, and to Ryan and Jerry Cain for the content.*

## Lecture 03: Layering, Naming, and Filesystem Design

- The drive itself exports an API—a **hardware** API—that allows us to read a sector into main memory, or update an entire sector with a new payload.
- In the interest of simplicity, speed, and reliability, the API is intentionally small, and might export a hardware equivalent of the C++ class presented right below.

```cpp
class Drive {
public:
    size_t getNumSectors() const;
    void readSector(size_t num, unsigned char data[]) const;
    void writeSector(size_t num, const unsigned char data[]);
};
```

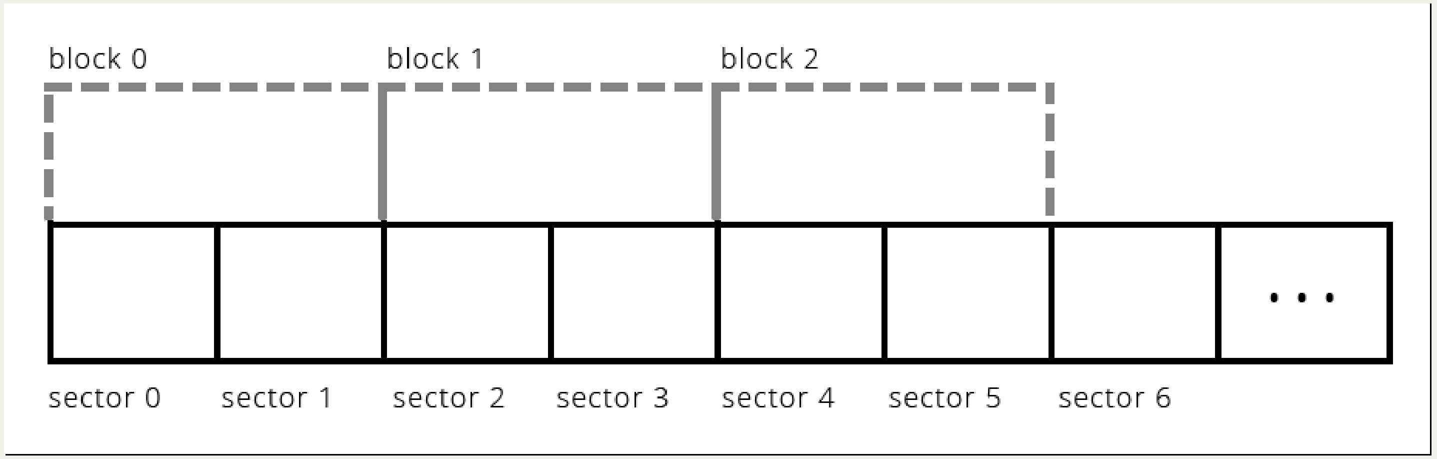
- This is what the hardware presents us with, and this small amount is all you really need to know in order to start designing basic filesystems. As filesystem designers, we need to figure out a way to take this primitive system and use it to store a user's files.



## Lecture 03: Layering, Naming, and Filesystem Design

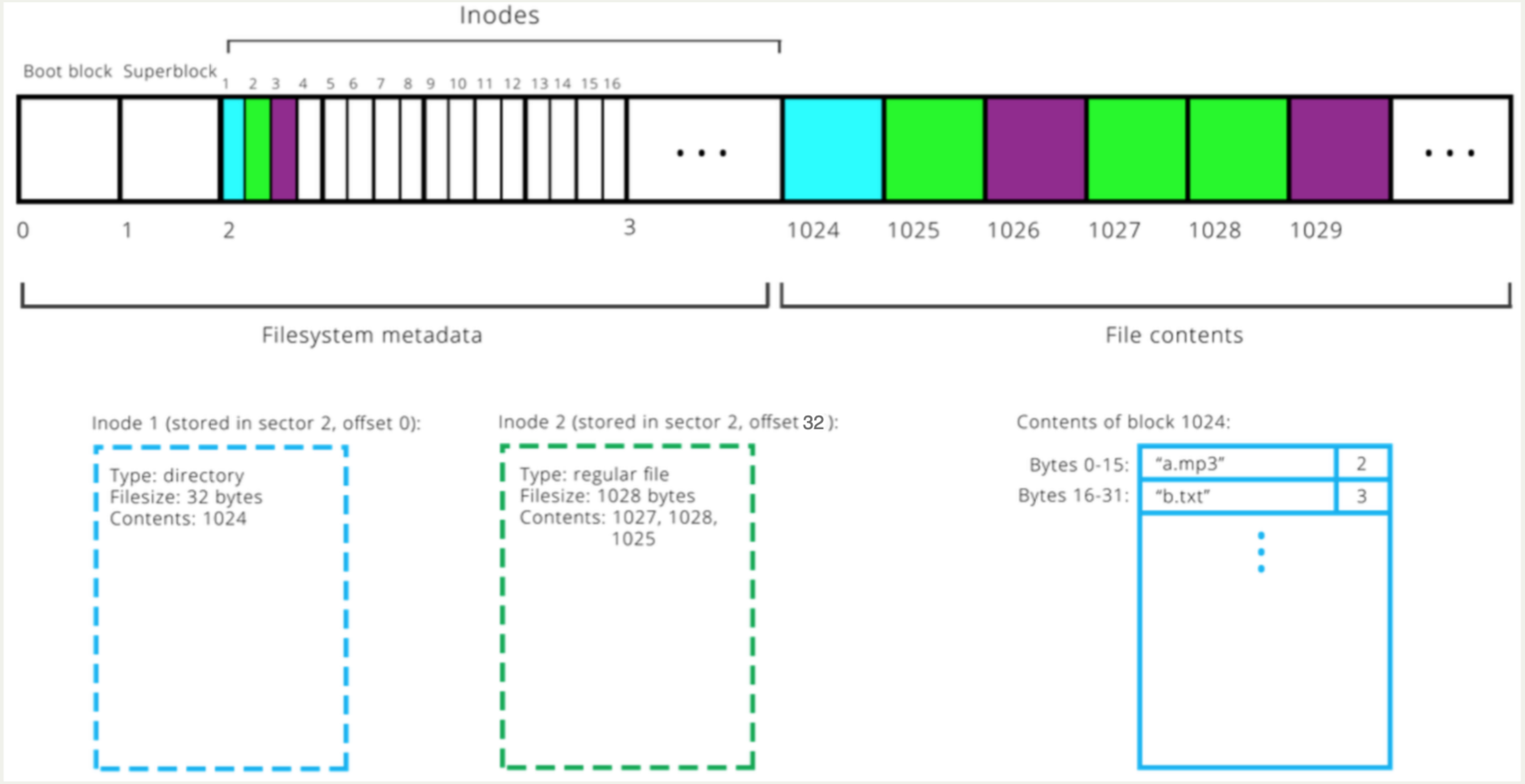
- Throughout the lecture, you may hear me use the term **block** instead of **sector**.
  - Sectors are the physical storage units on the hard drive.
  - The filesystem, however, generally frames its operations in terms of *blocks* (which are each comprised of one or more sectors).
  - If the filesystem goes with a block size of 1024 (as below), then when it accesses the filesystem, it will only read or write from the disk in 1024-byte chunks. Reading one block—which can be thought of as a software abstraction over sectors—would be framed in terms of two neighboring sector reads.
  - If the block abstraction defines the block size to be the same as the sector size (as the Unix v6 filesystem does), then the terms blocks and sectors can be used interchangeably (and the rest of this slide deck will do precisely that).

Example: the block size could be defined as two sectors



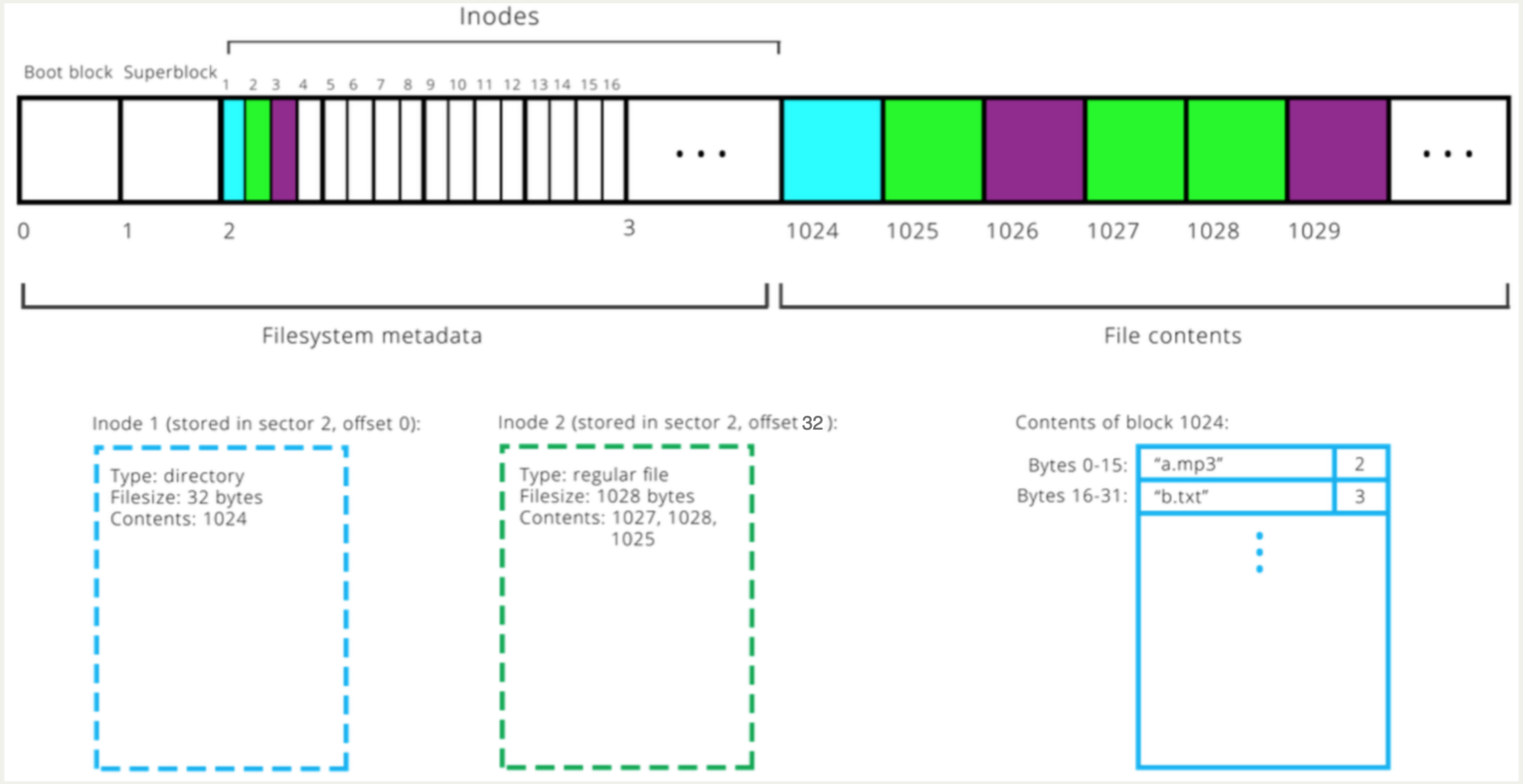
## Lecture 03: Layering, Naming, and Filesystem Design

- The diagram below shows how raw hardware could be leveraged to support filesystems as we're familiar with them. There's a lot going on in the diagram below, so we'll use the next several slides to dissect it and dig into the details.



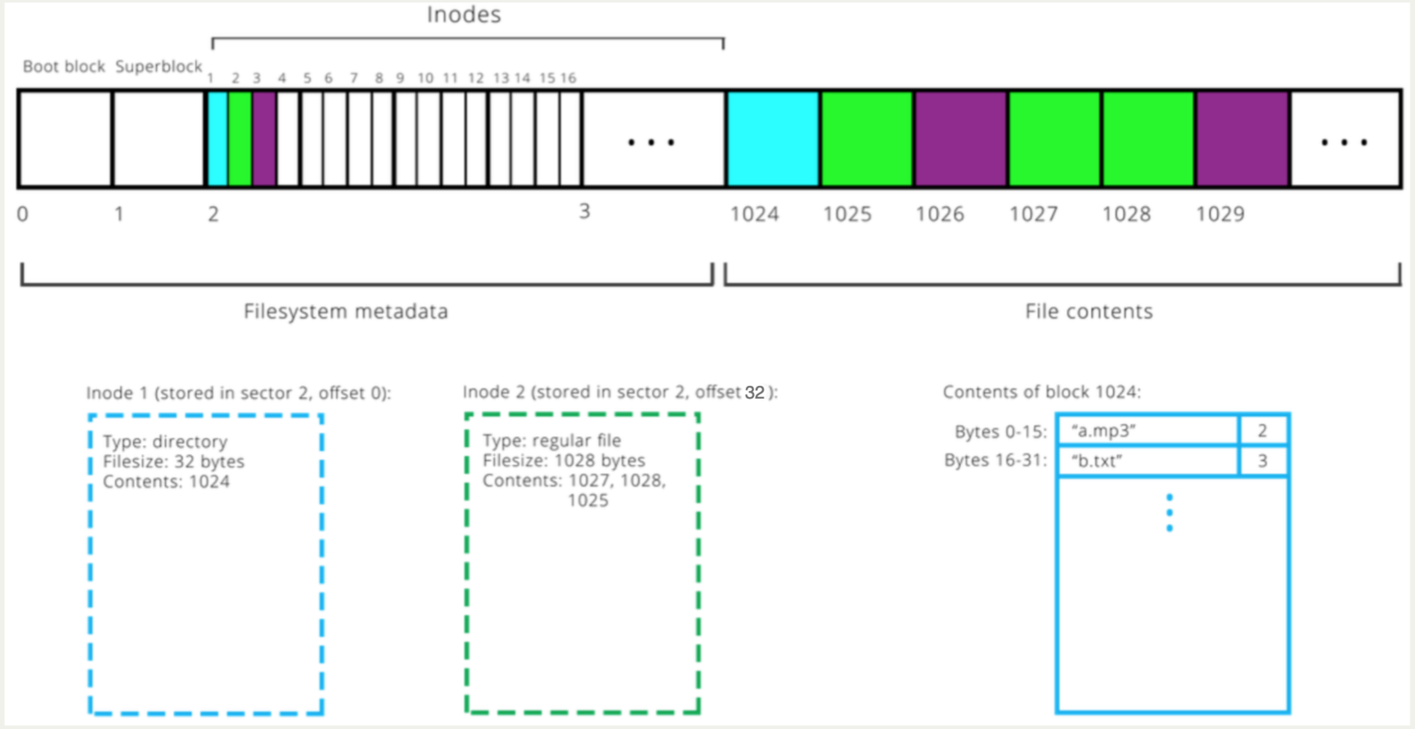
## Lecture 03: Layering, Naming, and Filesystem Design

- Filesystem metadata
  - The first block is the boot block, which typically contains information about the hard drive itself. It's so named because its contents are generally tapped when booting—i.e. restarting—the operating system.
  - The second block is the superblock, which contains information about the filesystem imposing itself onto the hardware.



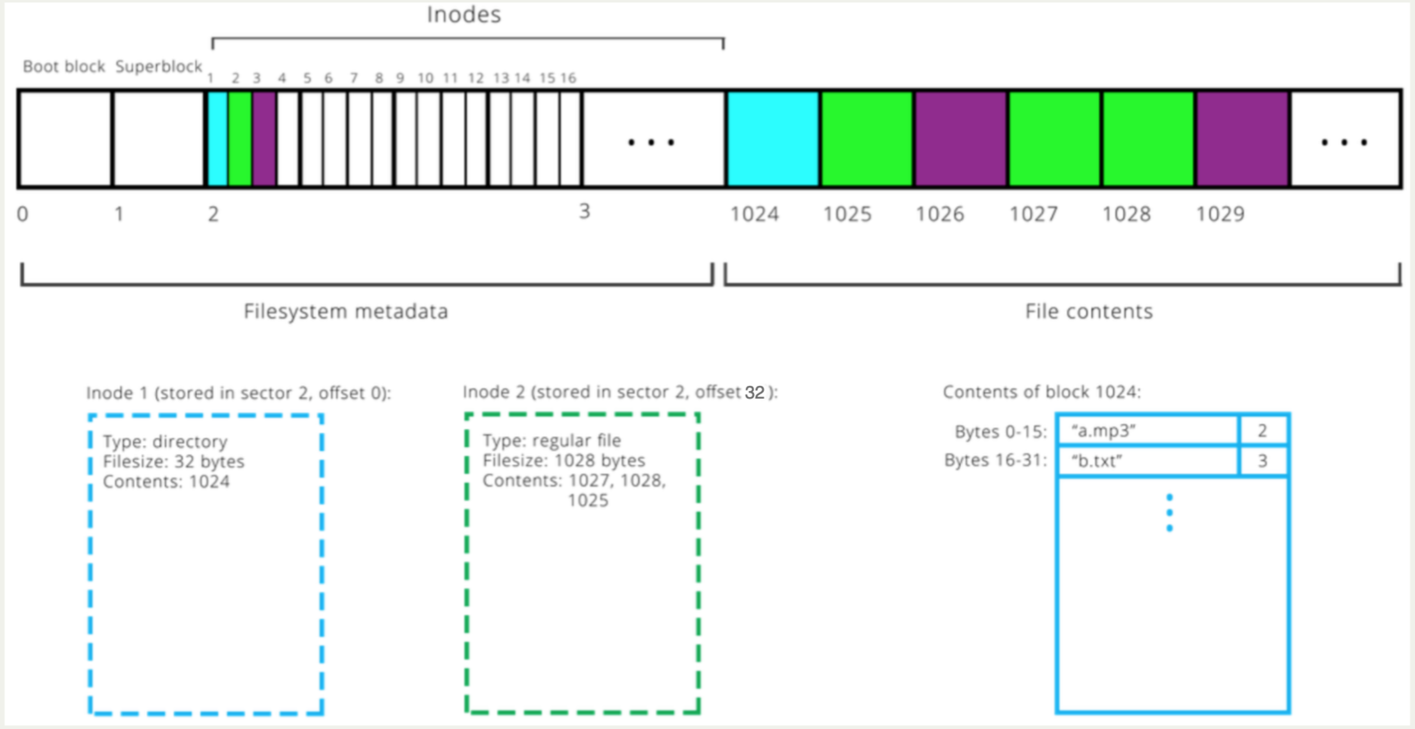
## Lecture 03: Layering, Naming, and Filesystem Design

- Filesystem metadata, continued
  - The rest of the metadata region stores the inode table, which at the highest level stores information about each file stored somewhere within the filesystem.
  - The diagram below makes the metadata region look much larger than it really is. In practice, at most 10% of the entire drive is set aside for metadata storage. The rest is used to store file payload.
  - If you took CS 107, you should be having flashbacks to the *heap allocator* assignment (sorry!). The disk memory is utilized to store both metadata *and* actual file data.



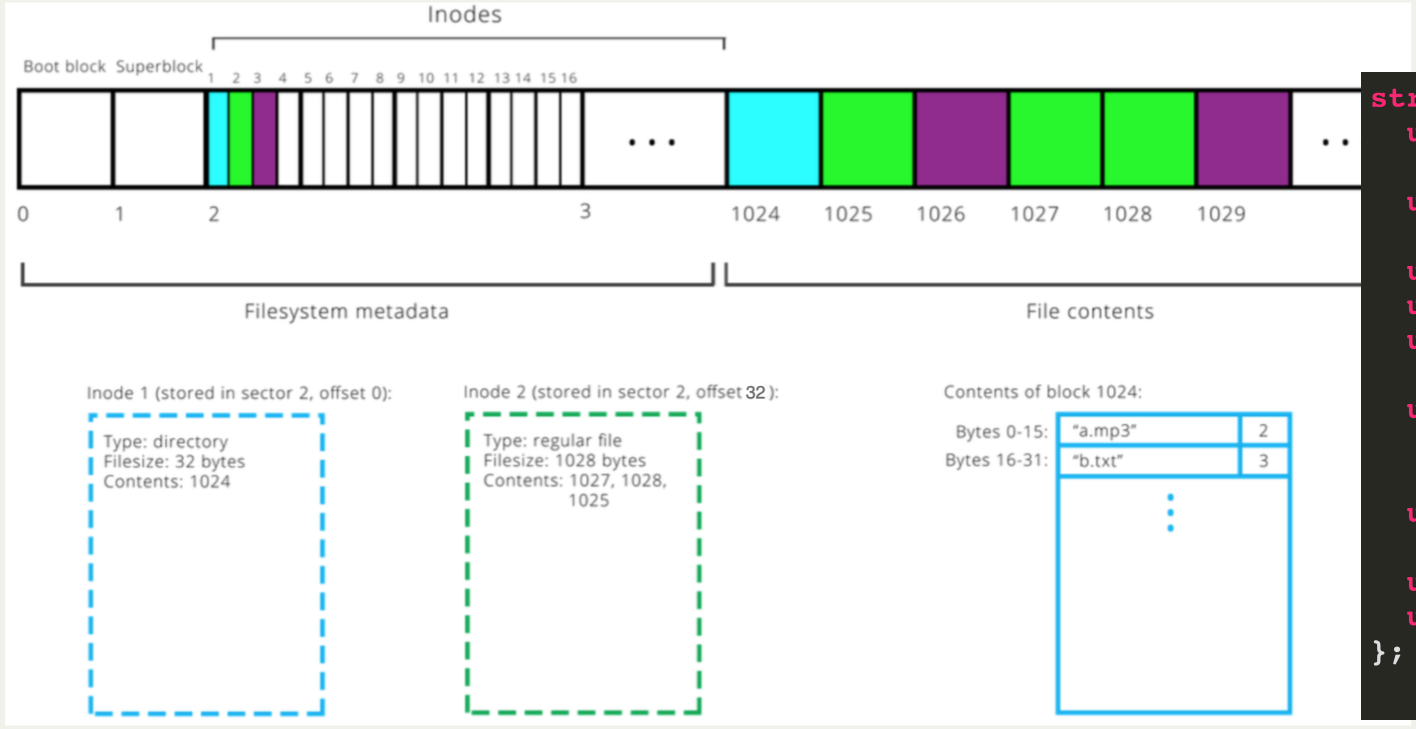
## Lecture 03: Layering, Naming, and Filesystem Design

- File contents
  - File payloads are stored in quantums of 512 bytes (or whatever the block size is).
  - When a file isn't a multiple of 512 bytes, then the final block is a partial. The portion of that final block that contains meaningful payload is easily determined from the file size.
  - The diagram below includes illustrations for a 32 byte (blue) and a 1028 (or 2 * 512 + 4) (green) byte file (as well as a purple file, which does not have an associated outline below), so each enlists some block to store a partial.



## Lecture 03: Layering, Naming, and Filesystem Design

- The inode
  - We need to track which blocks are used to store the payload of a file.
    - Blocks 1025, 1027, and 1028 are part of the same file, but you only know visually because they're the same color in the diagram.
  - **inodes** are data structures that store metainfo about a single file. Stored within an inode are items like file owner, file permissions, creation times, and, most importantly for our purposes, file type, file size, and the sequence of blocks enlisted to store payload.



Inode 1 (stored in sector 2, offset 0):
Type: directory
Filesize: 32 bytes
Contents: 1024

Inode 2 (stored in sector 2, offset 32 ):
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

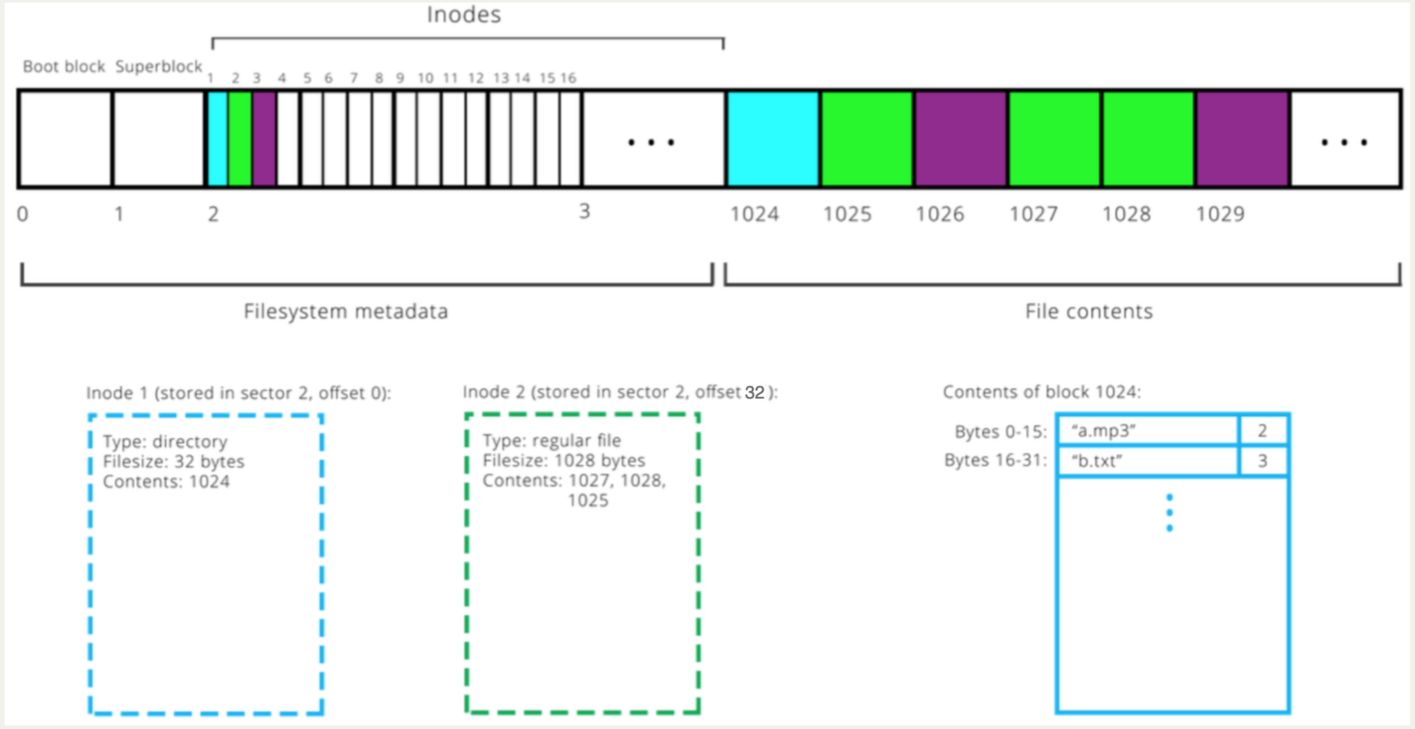
Contents of block 1024:

| | | |
|---|---|---|
| Bytes 0-15: | "a.mp3" | 2 |
| Bytes 16-31: | "b.txt" | 3 |

```
struct inode {
  uint16_t   i_mode;      // bit vector of file
                          //   type and permissions
  uint8_t    i_nlink;     // number of references
                          //   to file
  uint8_t    i_uid;       // owner
  uint8_t    i_gid;       // group of owner
  uint8_t    i_size0;     // most significant byte
                          //  of size
  uint16_t   i_size1;     // lower two bytes of size
                          //  (size is encoded in a
                          //  three-byte number)
  uint16_t   i_addr[8];   // device addresses
                          //   constituting file
  uint16_t   i_atime[2];  // access time
  uint16_t   i_mtime[2];  // modify time
};
```

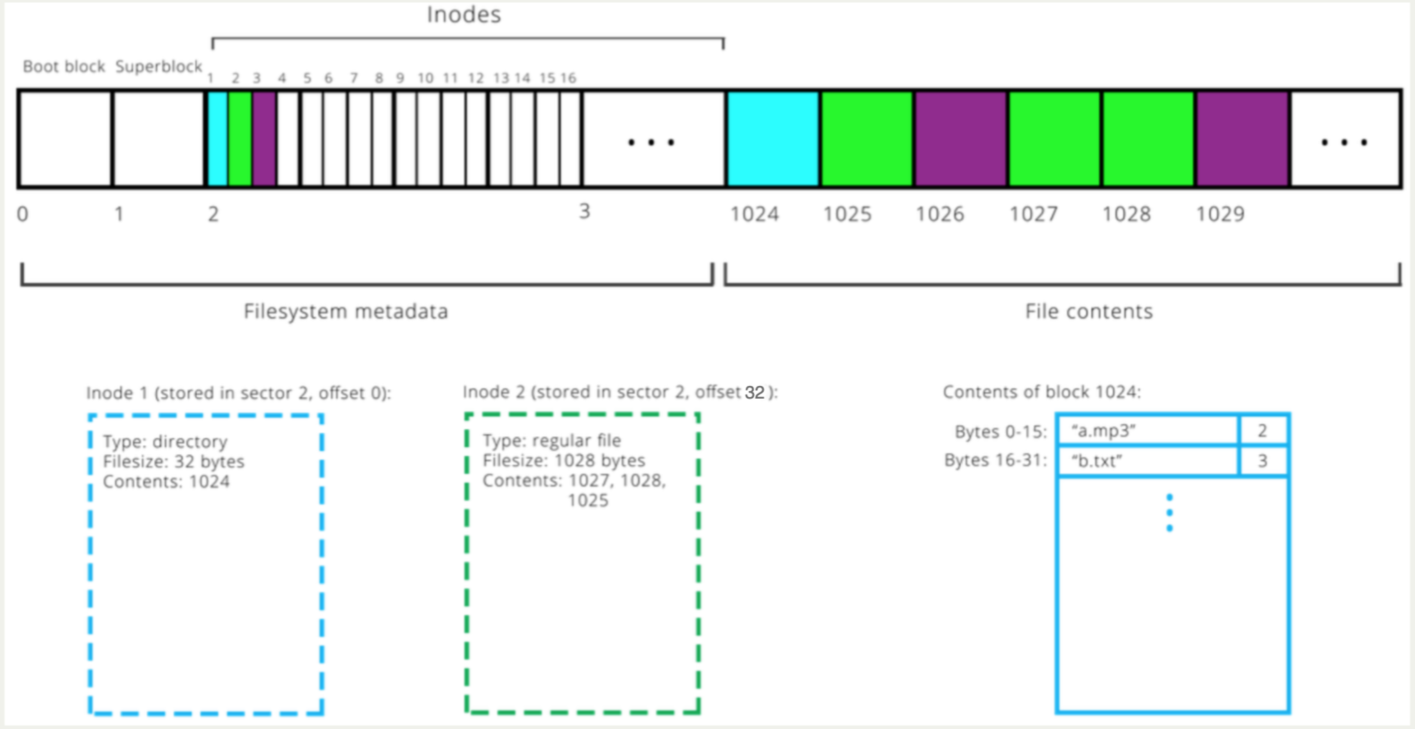
## Lecture 03: Layering, Naming, and Filesystem Design

- The inode, continued
  - Look at the contents of inode 2, outlined in green.
  - The file size is 1028 bytes. That means we need three blocks to store the payload. The first two will be saturated with meaningful payload, and the third will only store 1028 % 512, or 4, meaningful payload bytes.
  - The block nums are listed as 1027, 1028, and 1025, in that order. Bytes 0-511 reside within block 1027, bytes 512-1023 within block 1028, bytes 1024-1027 at the front of block 1025.



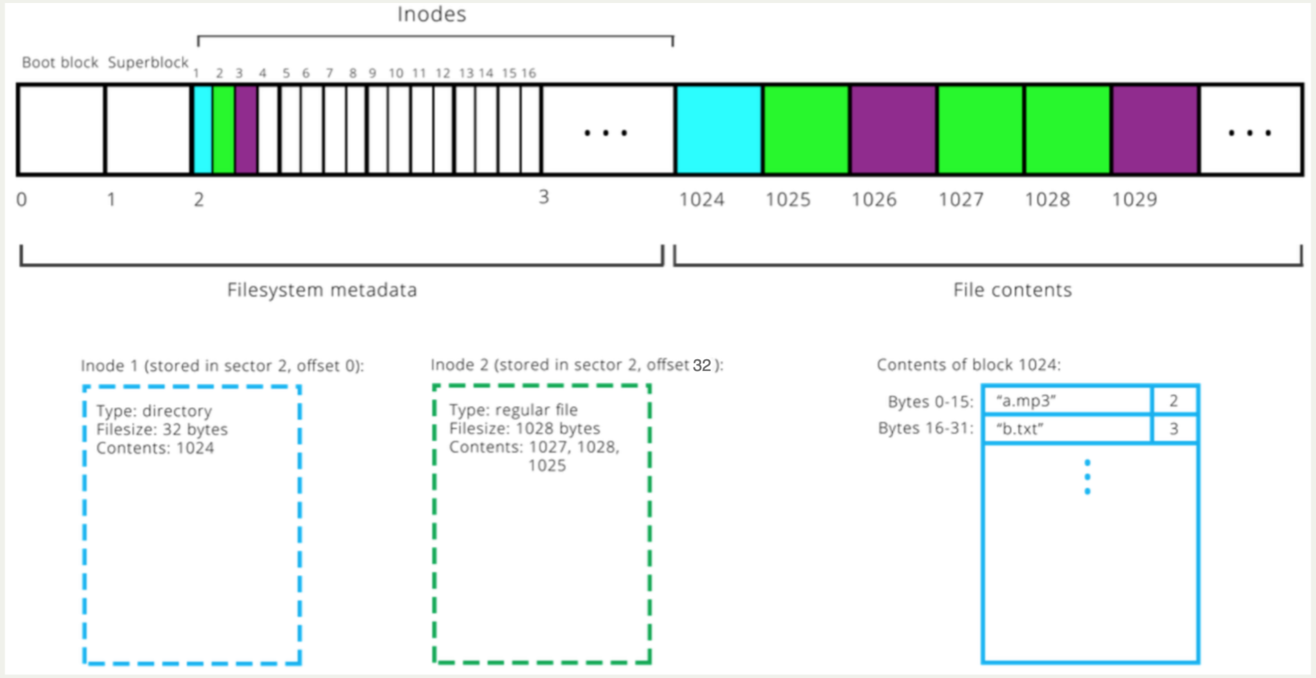
## Lecture 03: Layering, Naming, and Filesystem Design

- The inode, continued
  - The blocks used to store payload are not necessarily contiguous or in sorted order. You see exactly this scenario with file linked to inode 2. Perhaps the file was originally 1024 bytes, block 1025 was freed when another file was deleted, and then the first file was edited to include four more bytes of payload and then saved.
  - Some file systems, particularly those with large block sizes, might work to make use of the 508 bytes of block 1025 that aren't being used. Most, however, don't bother.



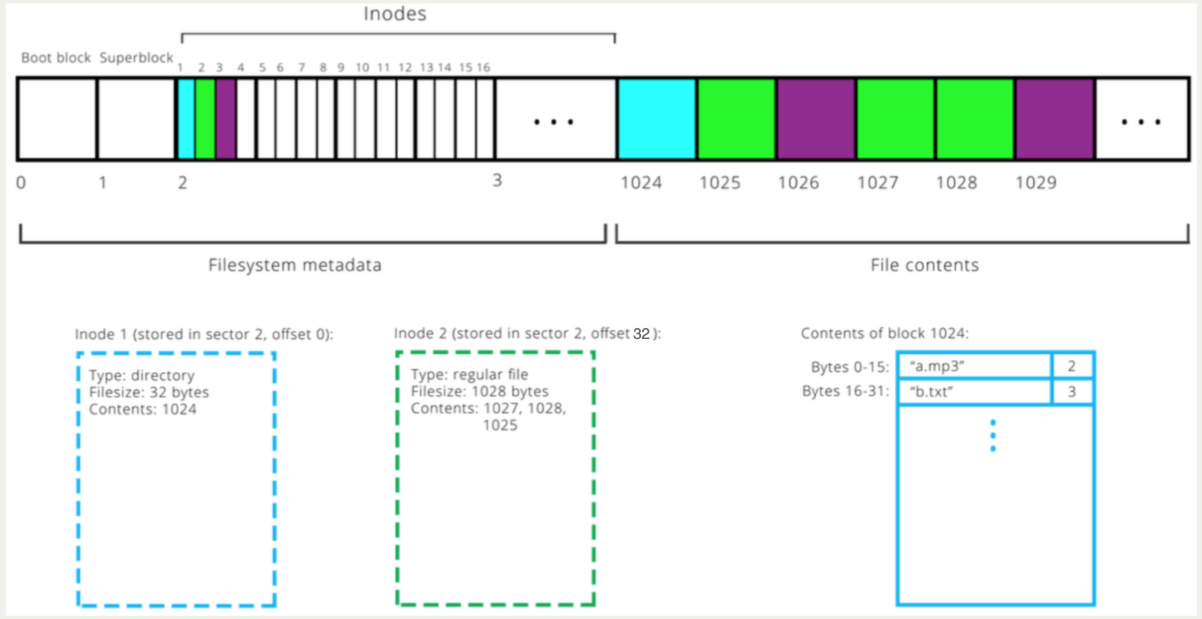
## Lecture 03: Layering, Naming, and Filesystem Design

- The inode, continued
  - A file's inodes tell us where we'll find its payload, but the inode also has to be stored on the drive as well.
  - A series of blocks comprise the inode table, which in our diagram stretches from block 2 through block 1023.
  - Because inodes are small—only 32 bytes in the case of the UnixV6 file system—each block within the inode table can store 16 inodes side by side, like the books of a 16-volume encyclopedia in a single bookshelf.



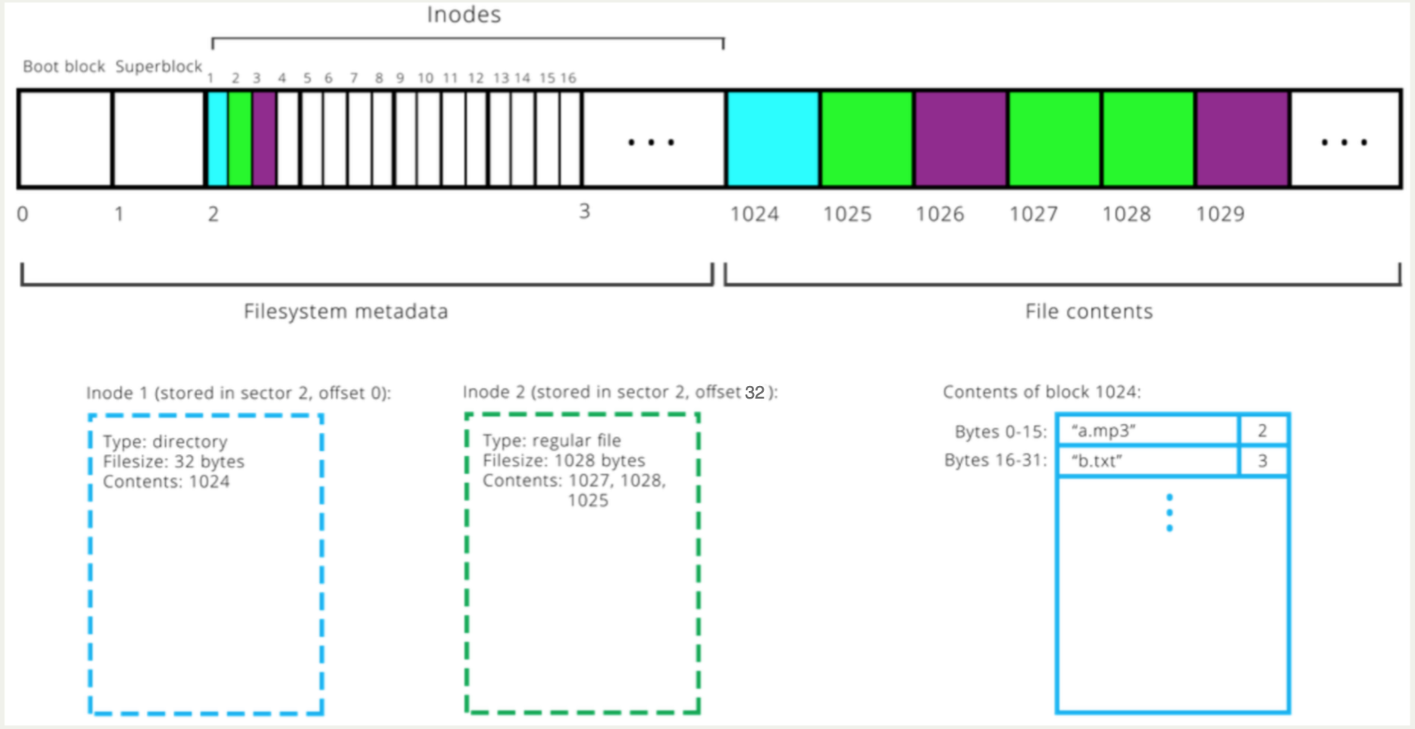
## Lecture 03: Layering, Naming, and Filesystem Design

- The inode, continued
  - As humans, if we needed to remember the inode number of every file on our system, we'd be sad. "Hey, I just put the roster spreadsheet into the shared Dropbox folder, at `7088881/521004/957121/4270046/37984/320459/1008443/5021000/2235666/154718`." 😱 (in reality, you would just say "I put the roster at 154718", but that would not represent any directory hierarchy.
  - Instead, we rely on filenames and a hierarchy of named directories to organize our files, and we prefer those names—e.g. `/usr/class/cs110/WWW/index.html`—to seemingly magic numbers that incidentally identify where the corresponding inodes sit in the inode table.



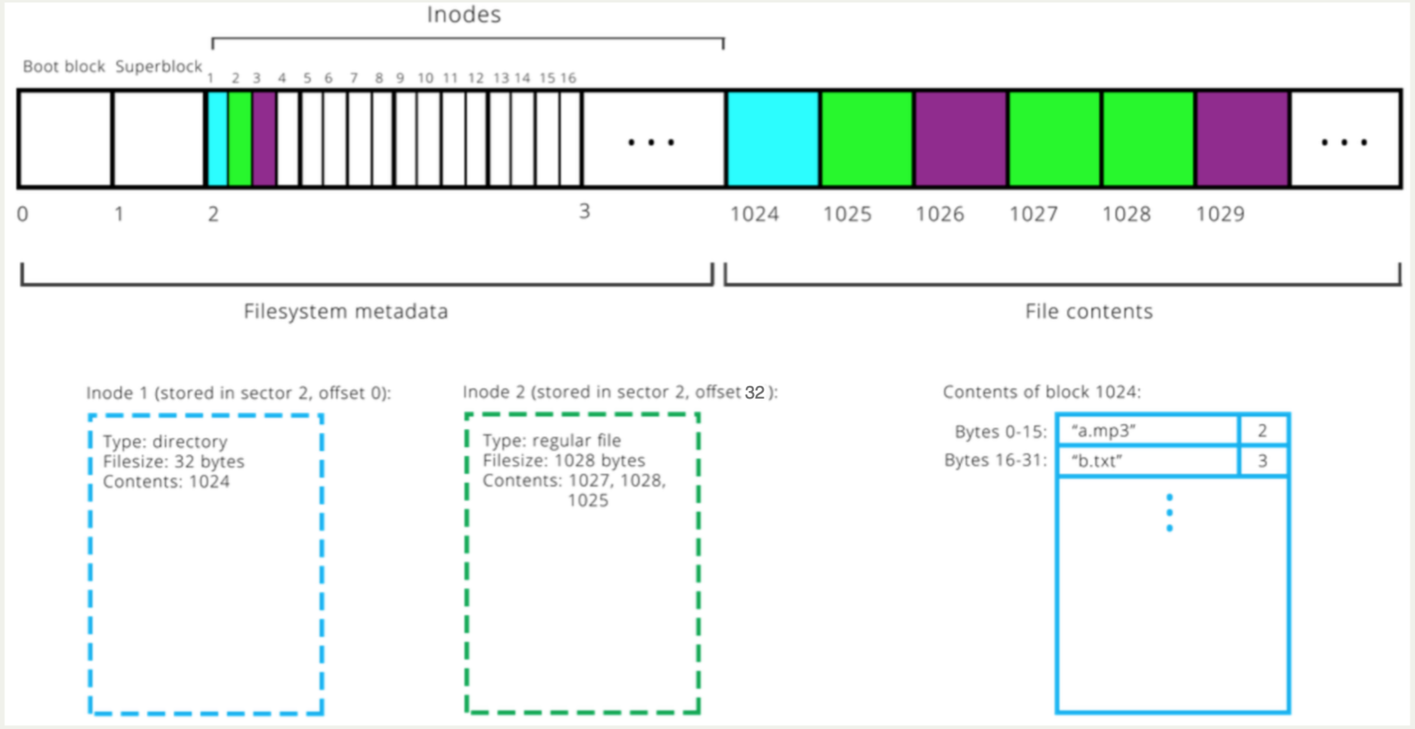
## Lecture 03: Layering, Naming, and Filesystem Design

- The inode, continued
    - We could wedge a filename field inside each inode. But that won't work, for two reasons.
        - Inodes are small, but filenames are long. Our Assignment 1 solution resides in a file named **`/usr/class/cs110/staff/master_repos/assign1/imdb.cc`**. At 51 characters, the name wouldn't fit in an inode even if the inode stored nothing else.
        - Linearly searching an inode table for a named file would be unacceptably slow. My own laptop has about two million files, so the inode table is at least that big, probably much bigger.



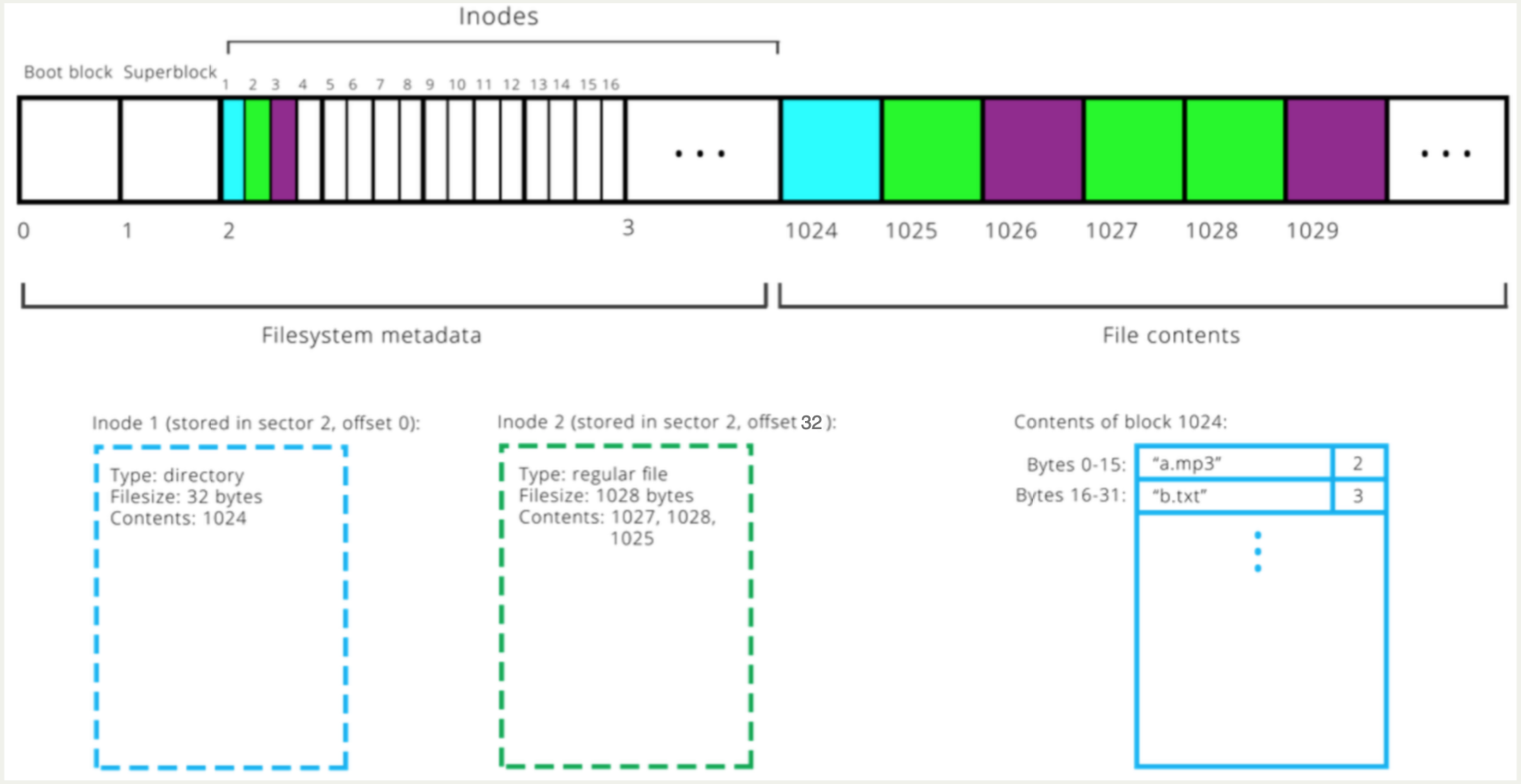
## Lecture 03: Layering, Naming, and Filesystem Design

- Introducing the directory file type
  - The solution is to introduce **directory** as a new file type. You may be surprised to find that this requires almost no changes to our existing scheme; we can layer directories on top of the file abstraction we already have. In almost all filesystems, directories are just files, the same as any other file (with the exception that they are marked as directories by the file type field in the inode). For Unix V6, the file payload is a series of 16-byte slivers that form a table mapping names to inode numbers.
  - Incidentally, you cannot look inside directory files explicitly, as the OS hides that information from you.



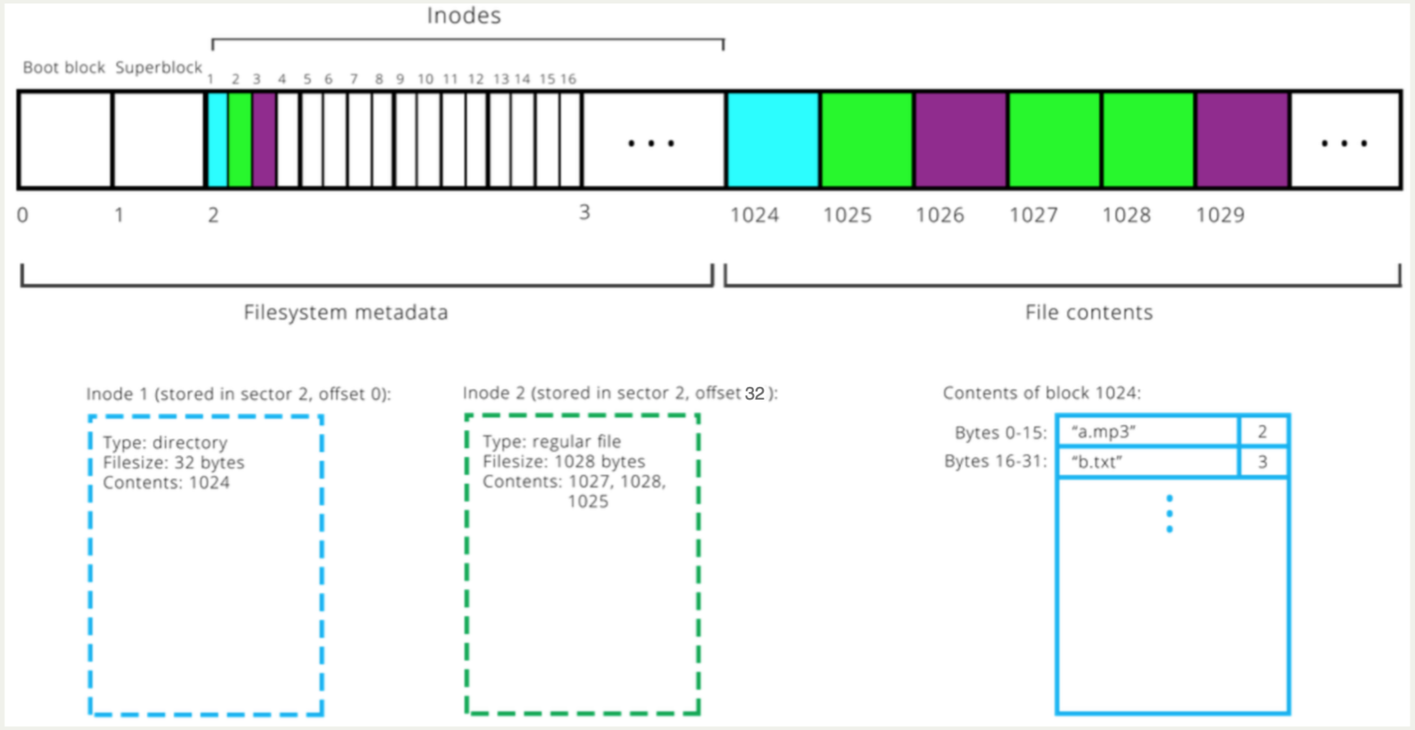
## Lecture 03: Layering, Naming, and Filesystem Design

- Introducing the directory file type, continued
  - Have a look at the contents of block 1024, i.e. the contents of file with inumber 1, in the diagram below. This directory contains two files, so its total file size is 32; the first 16 bytes form the first row of the table (14 bytes for the filename, 2 for the inumber), and the second 16 bytes form the second row of the table. When we are looking for a file in the directory, we search this table for the corresponding inumber.



## Lecture 03: Layering, Naming, and Filesystem Design

- Introducing the directory file type, continued
  - What does the file lookup process look like, then? Consider a file at **`/usr/class/cs110/example.txt`**. First, we find the inode for the file **`/`** (which always has inumber 1. See here about why it is 1 and not 0). We search inode 1's payload for the token **`usr`** and its companion inumber. Let's say it's at inode 5. Then, we get inode 5's contents (which is another directory) and search for the token **`class`** in the same way. From there, we look up the token **`cs110`** and then **`example.txt`**. This will (finally) be an inode that designates a file, not a directory.



## Lecture 03: Layering, Naming, and Filesystem Design: Hard Links and Soft Links

- Directory entries hold file/directory names, and corresponding inumbers.
  - Could we have two file names refer to the same actual file? Yes!
  - When a path is resolved for a file, the final inumber will refer to a file, which we can then read. If another path *also* refers to that same file, there isn't any issue. It is just another way to get to the file in question.
  - The file system has to keep track of how many times the file is referenced in this way, because removing a file (using **`rm filename`** in Unix) only removes the reference, not the actual file itself.
- The **`struct inode`** contains a field, **`i_nlink`**, which keeps track of the number of links to a file. A file is removed from the disk only when this reference count becomes 0, and when no process is using the file (i.e., has it open). This means, for instance, that **`rm filename`** can even remove a file when a program has it open, and the program still has access to the file (because it hasn't been removed from the disk). This is not true for many other file systems (e.g., Windows)!

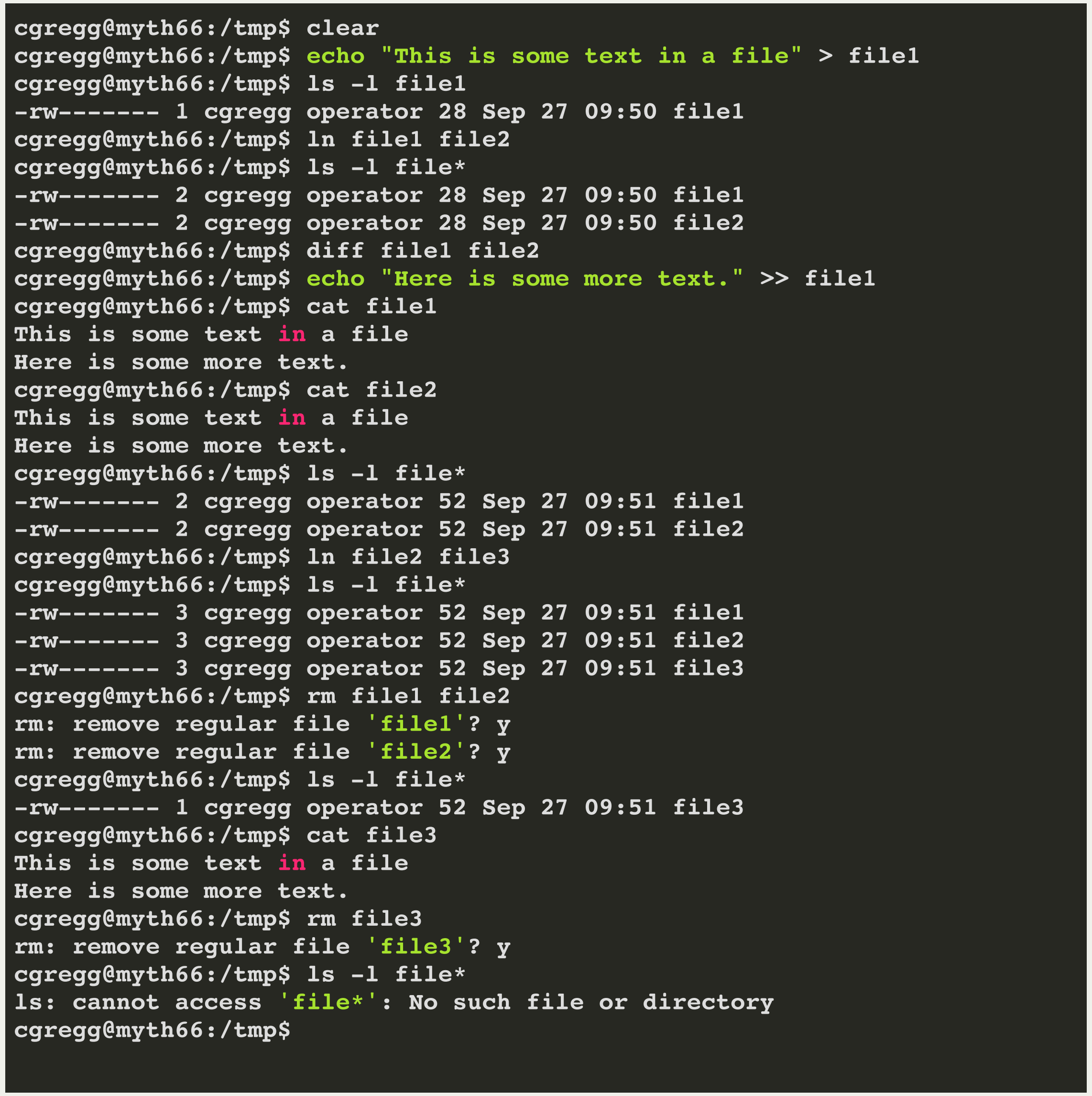
```
struct inode {
  ...
uint8_t   i_nlink;    // number of references
                       //   to file

  ...
};
```

- What we are describing here is called a *hard link*. All normal files in Unix (and Linux) are hard links, and two hard links are indistinguishable as far as the file they point to is concerned. In other words, there is no "real" filename, as both file names point to the same inode.

# Lecture 03: Layering, Naming, and Filesystem Design: Hard Links and Soft Links

Example of hard link creation, deletion, etc.:

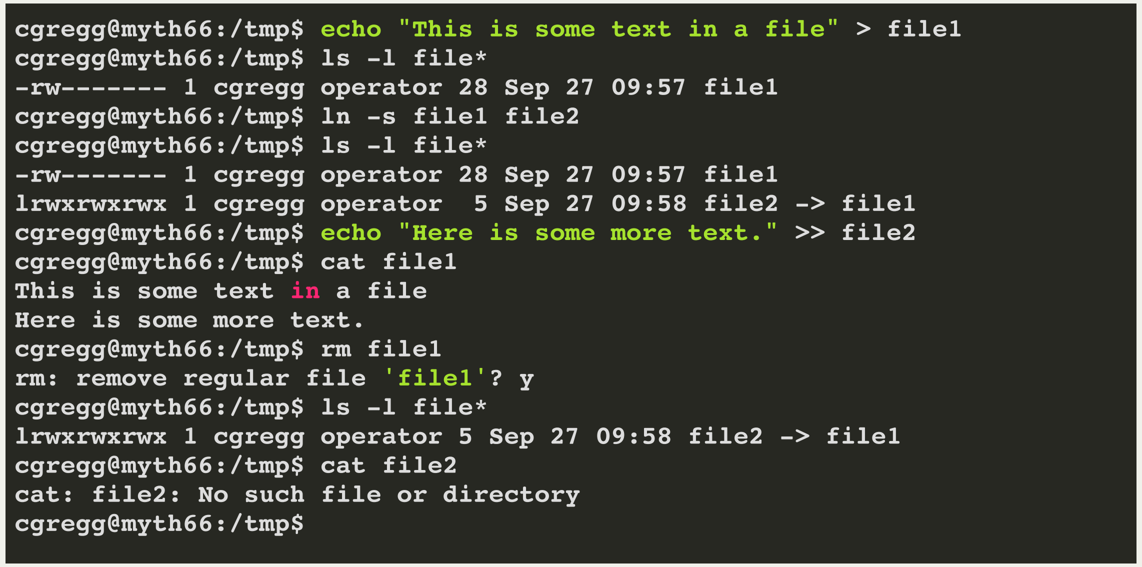


```
cgregg@myth66:/tmp$ clear
cgregg@myth66:/tmp$ echo "This is some text in a file" > file1
cgregg@myth66:/tmp$ ls -l file1
-rw------- 1 cgregg operator 28 Sep 27 09:50 file1
cgregg@myth66:/tmp$ ln file1 file2
cgregg@myth66:/tmp$ ls -l file*
-rw------- 2 cgregg operator 28 Sep 27 09:50 file1
-rw------- 2 cgregg operator 28 Sep 27 09:50 file2
cgregg@myth66:/tmp$ diff file1 file2
cgregg@myth66:/tmp$ echo "Here is some more text." >> file1
cgregg@myth66:/tmp$ cat file1
This is some text in a file
Here is some more text.
cgregg@myth66:/tmp$ cat file2
This is some text in a file
Here is some more text.
cgregg@myth66:/tmp$ ls -l file*
-rw------- 2 cgregg operator 52 Sep 27 09:51 file1
-rw------- 2 cgregg operator 52 Sep 27 09:51 file2
cgregg@myth66:/tmp$ ln file2 file3
cgregg@myth66:/tmp$ ls -l file*
-rw------- 3 cgregg operator 52 Sep 27 09:51 file1
-rw------- 3 cgregg operator 52 Sep 27 09:51 file2
-rw------- 3 cgregg operator 52 Sep 27 09:51 file3
cgregg@myth66:/tmp$ rm file1 file2
rm: remove regular file 'file1'? y
rm: remove regular file 'file2'? y
cgregg@myth66:/tmp$ ls -l file*
-rw------- 1 cgregg operator 52 Sep 27 09:51 file3
cgregg@myth66:/tmp$ cat file3
This is some text in a file
Here is some more text.
cgregg@myth66:/tmp$ rm file3
rm: remove regular file 'file3'? y
cgregg@myth66:/tmp$ ls -l file*
ls: cannot access 'file*': No such file or directory
cgregg@myth66:/tmp$
```

- In Unix, you can create a link using the **`ln`** command.
- Notice that the reference count for the file (the number after the permissions) goes up each time we create a hard link.
- Even if we delete one of the files, the other filenames that refer to the same file will remain (and the reference count goes down)
- Because there is only one actual file, changing the contents of the file through any of the hard links changes the file contents for all of the filename links (again, there is only one file!)

## Lecture 03: Layering, Naming, and Filesystem Design: Hard Links and Soft Links

- In addition to hard links, the Unix filesystem has the ability to create *soft links*. A soft link is a special file that contains the *path* of another file, and has no reference to the inumber.
- Soft links can "break" in the sense that if the path they refer to is gone (e.g., the file is actually removed from the disk), then the link will no longer work.
- To create a soft link in Unix, use the **`-s`** flag with **`ln`**.
- Example:

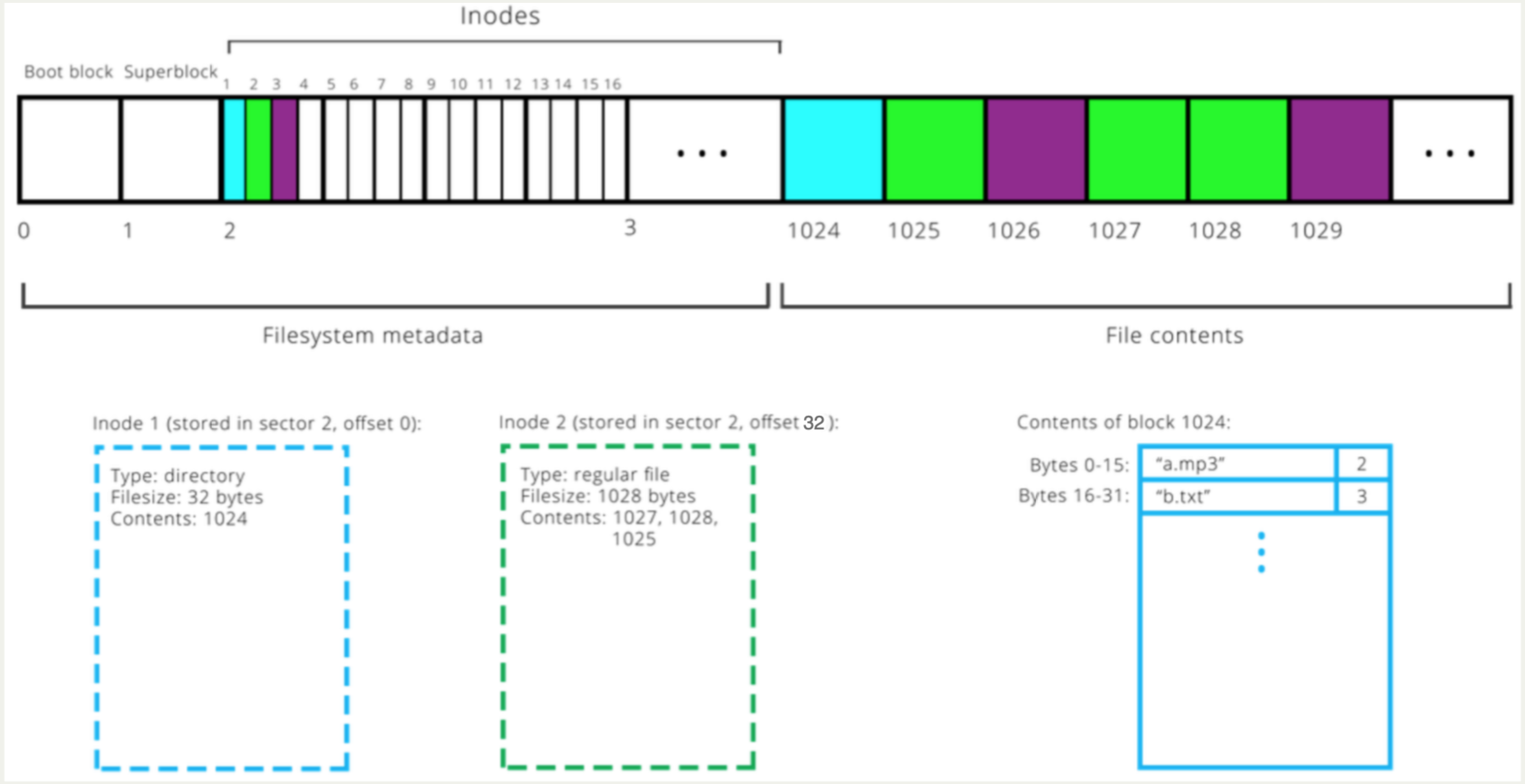


```
cgregg@myth66:/tmp$ echo "This is some text in a file" > file1
cgregg@myth66:/tmp$ ls -l file*
-rw------- 1 cgregg operator 28 Sep 27 09:57 file1
cgregg@myth66:/tmp$ ln -s file1 file2
cgregg@myth66:/tmp$ ls -l file*
-rw------- 1 cgregg operator 28 Sep 27 09:57 file1
lrwxrwxrwx 1 cgregg operator  5 Sep 27 09:58 file2 -> file1
cgregg@myth66:/tmp$ echo "Here is some more text." >> file2
cgregg@myth66:/tmp$ cat file1
This is some text in a file
Here is some more text.
cgregg@myth66:/tmp$ rm file1
rm: remove regular file 'file1'? y
cgregg@myth66:/tmp$ ls -l file*
lrwxrwxrwx 1 cgregg operator 5 Sep 27 09:58 file2 -> file1
cgregg@myth66:/tmp$ cat file2
cat: file2: No such file or directory
cgregg@myth66:/tmp$
```

- When we create a soft link, **`ls`** gives us the path to the original file
- But, the reference count for the original file remains unchanged
- Again, changing the contents of the file via either filename changes the file.
- If we delete the original file, the soft link breaks! The soft link remains, but the path it refers to is no longer valid. If we had removed the soft link before the file, the original file would still remain.

## Lecture 03: Layering, Naming, and Filesystem Design

- What about large files?
  - In the Unix V6 filesystem, inodes can only store a maximum of 8 block numbers. This presumably limits the total file size to 8 * 512 = 4096 bytes. That's way too small for any reasonably sized file.



## Lecture 03: Layering, Naming, and Filesystem Design

- What about large files? We have a solution!
  - To resolve this problem, we use a scheme called **indirect addressing**. Normally, the inode stores block numbers that directly identify payload blocks.
    - As an example, let's say the file is stored across blocks 2001-2008. The inode will store the numbers 2001-2008. We want to append to the file, but the inode can't store any more block numbers.
    - Instead, let's allocate a single block—let's say this is block 2050—and let's store the numbers 2001-2009 **in that block**. Then update the inode to store **only** block number 2050, and we set a flag specifying that we're using this indirect addressing scheme.
    - When we want to get the contents of the file, we check the inode and see this flag is set. We get the first block number, read that block, and then read the **actual** block numbers (storing file payload) from that block.
      - This is known as **singly**-indirect addressing.
      - We could store up to 8 singly indirect block numbers in an inode, and each can store 512 / 2 = 256 block numbers. This increases the maximum file size to 8 * 256 * 512 = 1,048,576 bytes = 1 MB (but see the next slide about why this is actually limited to 7 * 256 * 512 = 917,504 bytes).

## Lecture 03: Layering, Naming, and Filesystem Design

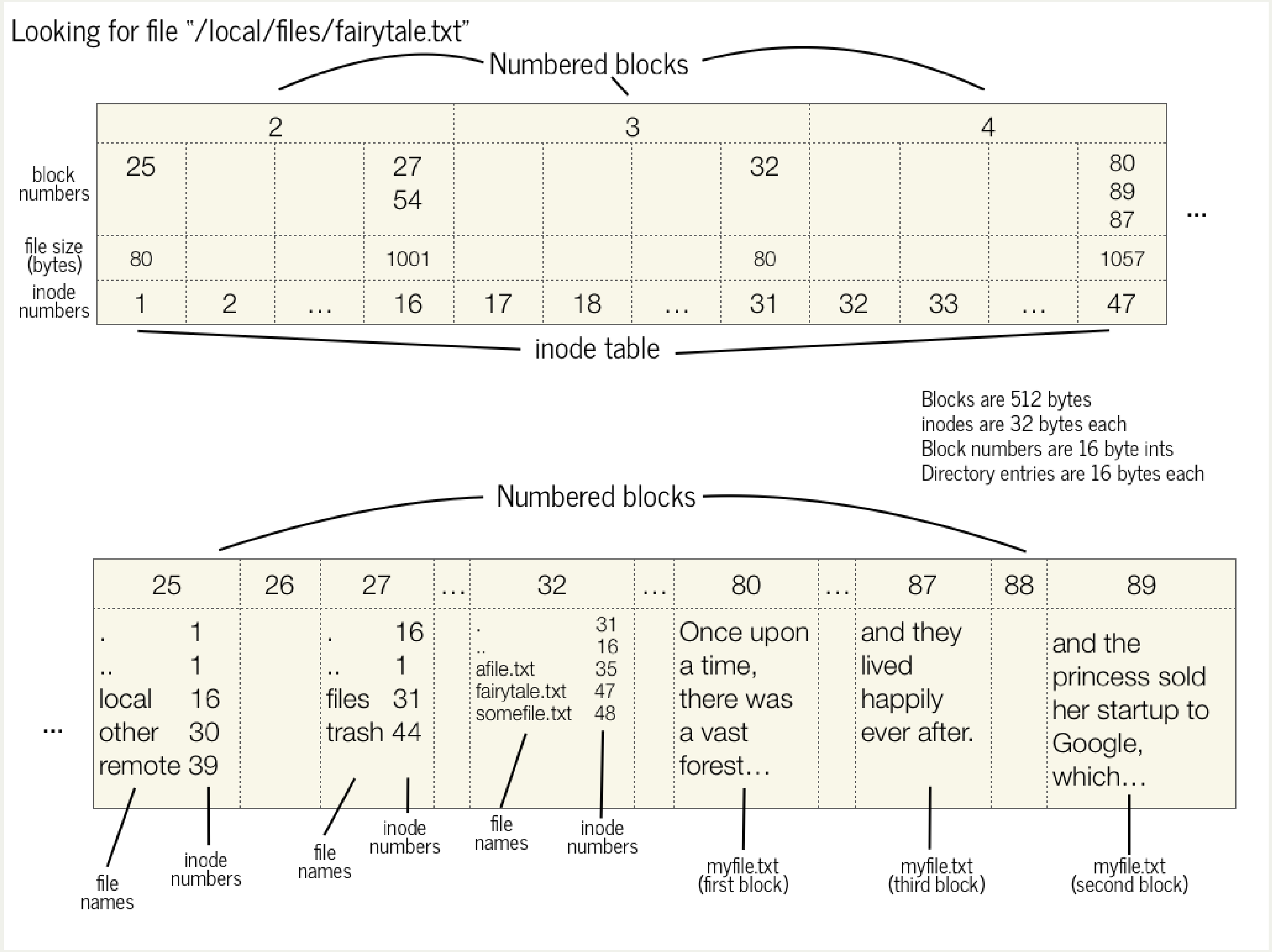
- What about *even larger* files? We have another solution!
  - 1MB is still not that big. To make the max file size even bigger, Unix V6 uses the 8th block number of the inode to store a **doubly indirect** block number.
    - In the inode, the first 7 block numbers store to singly indirect block numbers, but the last block number identifies to a block which itself stores singly-indirect block numbers.
    - The total number of singly indirect block numbers we can have is 7 + 256 = 263, so the maximum file size is 263 * 256 * 512 = 34,471,936 bytes = 34MB.
    - That's still not very large by today's standards, but remember we're referring to a file system design from 1975, when file system demands were lighter than they are today. In fact, because inumbers were only 16 bits, and block sizes were 512 bytes, the entire file system was limited to 32MB.
  - To summarize:
    - If a file is less than 512 * 8 = 4096 bytes, Unix V6 uses all 8 block numbers to point to 512 byte blocks, each of which has file data.
    - If a file is larger than 4096 bytes:
      - The first seven block numbers are indirectly addressed, leading to 7 * 256 * 512 = 917,504 bytes.
      - The eighth  block number, if needed, is doubly-indirectly addressed, leading to an additional 256 * 256 * 512 = 33,554,432 bytes, meaning that the largest file can be 34,471,936 bytes, or around 34MB.

## Lecture 03: Layering, Naming, and Filesystem Design: Examples

- Given our UNIX v6 file system, let's take a look at three examples:
    1. We want to find a file called "/local/files/fairytale.txt", which is a small file.
    2. We want to read a file called "/medfile", which is a medium sized file (larger than 512 * 8 = 4096 bytes but smaller than 7 * 256 * 512 = 917,504 bytes)
    3. We want to read a file called "/bigfile", which is a large file (larger than 917,504 bytes but smaller than (7 * 256 * 512) + (256 * 256 * 512) = 34MB.

# Lecture 03: Layering, Naming, and Filesystem Design: Example 1: Find a file

We want to find a file called "/local/files/fairytale.txt", which is a small file.

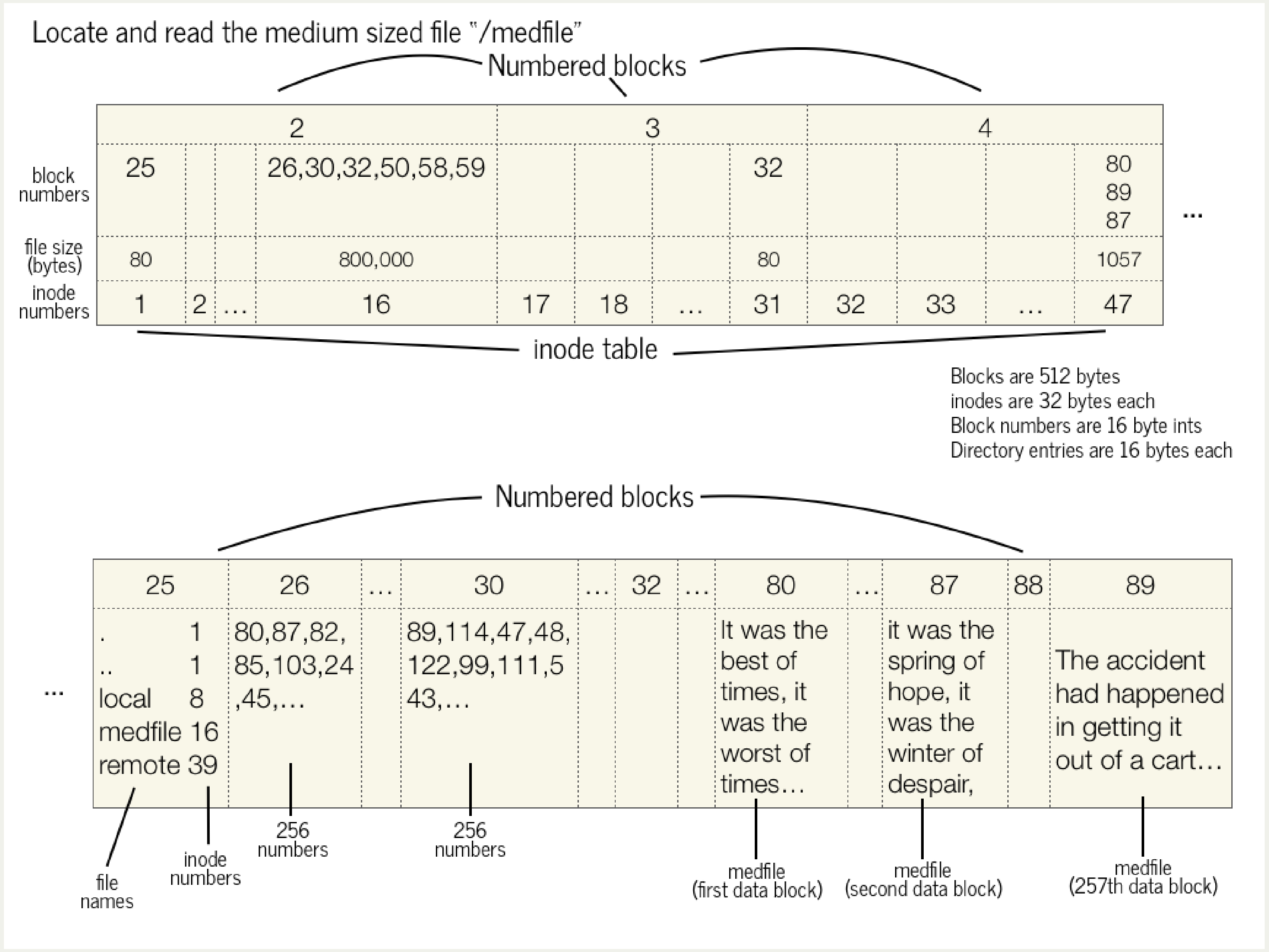


Steps:

1. Go to inode number 1 for the root directory. See that we need to go to block 25, and that it is a small file (80 bytes).
2. Read block 25, which has 16-byte directory entries. Look for "local" and see that it has inode number 16.
3. Go to inode number 16, and see that the directory is at blocks 27 and 54 (it is bigger than 512 bytes).
4. Read block 27 (and possibly 54) to find "files"
5. See that files has inode number 31.
6. Go to inode number 31, and then to block 32 to find the directory file and look for "fairytale.txt", which has inode number 47.
7. Go to inode number 47, and see that we have to read three blocks (in order): 80, 89, and 87.
8. Read 512 bytes from blocks 80, 89, and 87, to get the entire file.

## Lecture 03: Layering, Naming, and Filesystem Design: Example 1: Find a file

We want to find a file called "/medfile", which is a large-sized file.

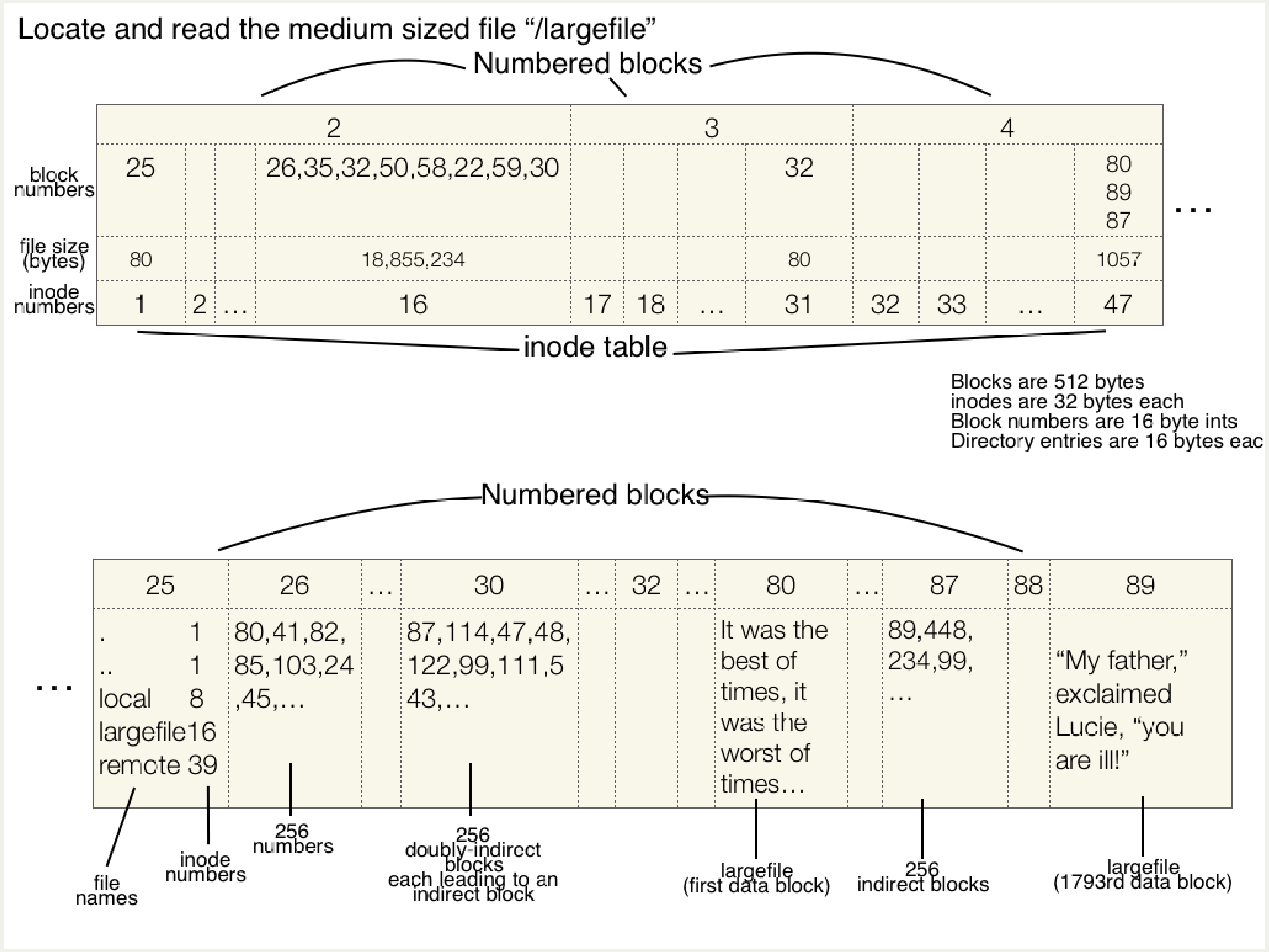


Steps:

1. Go to inode number 1 for the root directory. See that we need to go to block 25, and that it is a small file (80 bytes).
2. Read block 25, which has 16-byte directory entries. Look for "medfile" and see that it has inode number 16.
3. Go to inode number 16, and see that it is large (800,000 bytes).
4. Go to block 26, and start reading block numbers. For the first number, 80, go to block 80 and read the beginning of the file (the first 512 bytes). Then go to block 87 for the next 512 bytes, etc.
5. After 256 blocks, go to block 30, and follow the 256 block numbers to 89, 114, etc. to read the 257th-511th blocks of data.
6. Continue with all indirect blocks, 32, 50, 58, 59 to read all 800,000 bytes.

# Lecture 03: Layering, Naming, and Filesystem Design: Example 1: Find a file

We want to find a file called "/largefile", which is a very large file.



Steps:

1. Go to inode number 1 for the root directory. See that we need to go to block 25, and that it is a small file (80 bytes).
2. Read block 25, which has 16-byte directory entries. Look for "largefile" and see that it has inode number 16.
3. Go to inode number 16, and see that it is very large (18,855,234 bytes).
4. Go to block 26, and start reading block numbers. For the first number, 80, go to block 80 and read the beginning of the file (the first 512 bytes). Then go to block 41 for the next 512 bytes, etc.
5. After 256 blocks, go to block 35, repeat the process in step 4. Do this a total of 7 times, for blocks 26, 35, 32, 50, 58, and 59, reading 1792 blocks.
6. Go to block 30, which is a doubly-indirect block. From there, go to block 87, which is an indirect block. From there, go to block 89, which is the 1793rd block.