Lecture 03: Layering, Naming, and Filesystem Design

Principles of Computer Systems

Spring 2019

Stanford University

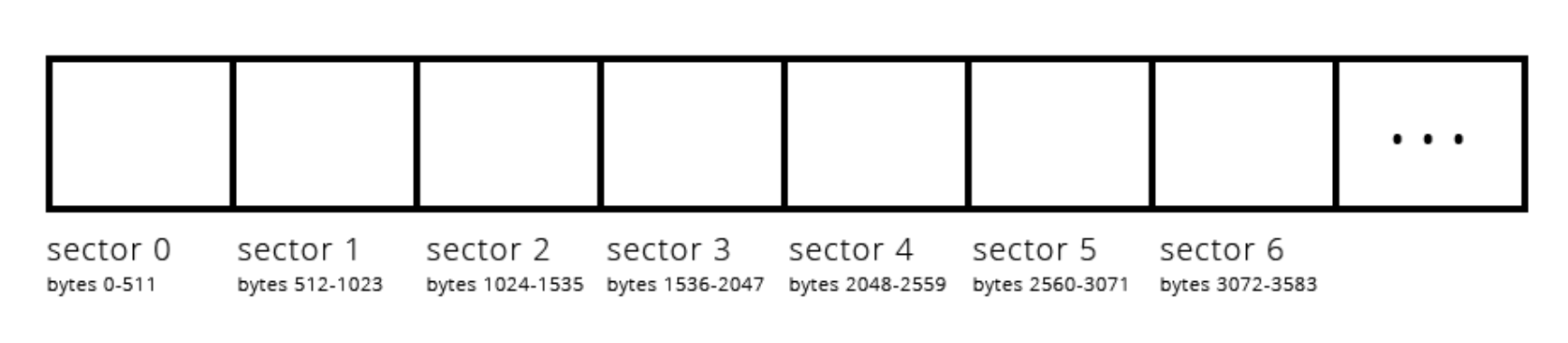
Computer Science Department

Lecturer: Chris Gregg

PDF of this presentation

Lecture 03: Layering, Naming, and Filesystem Design

- Just like RAM, hard 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

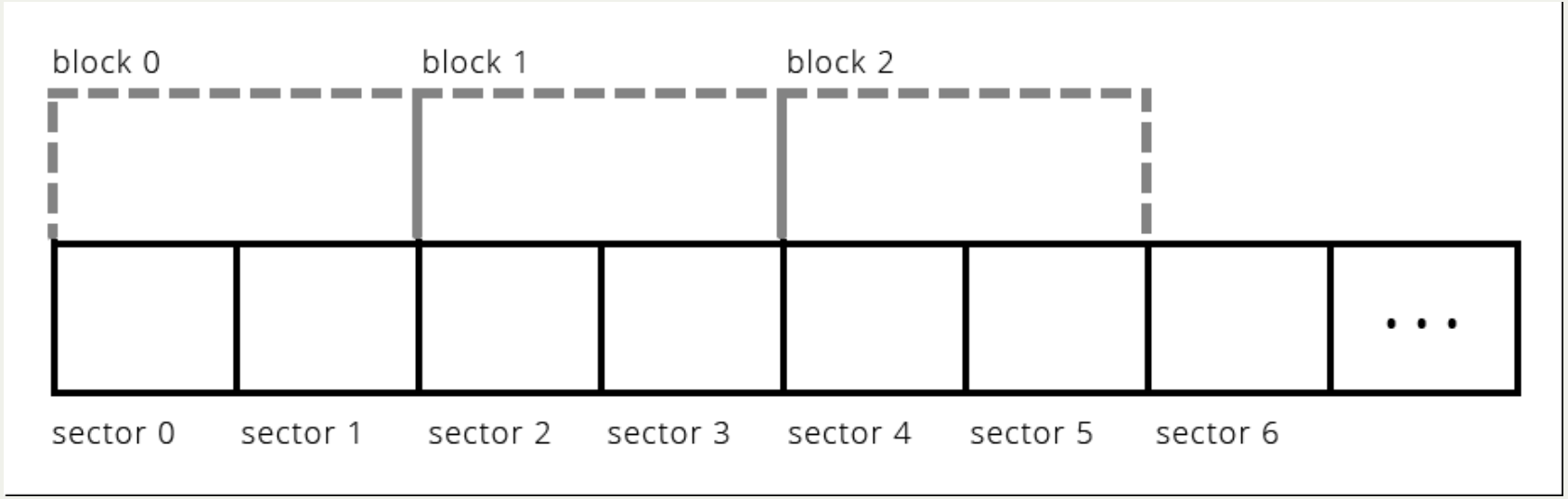
- The drive itself exports an API—a **hardware** API—that allows us to read a sector into main memory, or update an entire sector with a new payload.
- In the interest of simplicity, speed, and reliability, the API is intentionally small, and might export a hardware equivalent of the C++ class presented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

```
1 class Drive {
2 public:
3     size_t getNumSectors() const;
4     void readSector(size_t num, unsigned char data[]) const;
5     void writeSector(size_t num, const unsigned char data[]);
6 };
```

- 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.

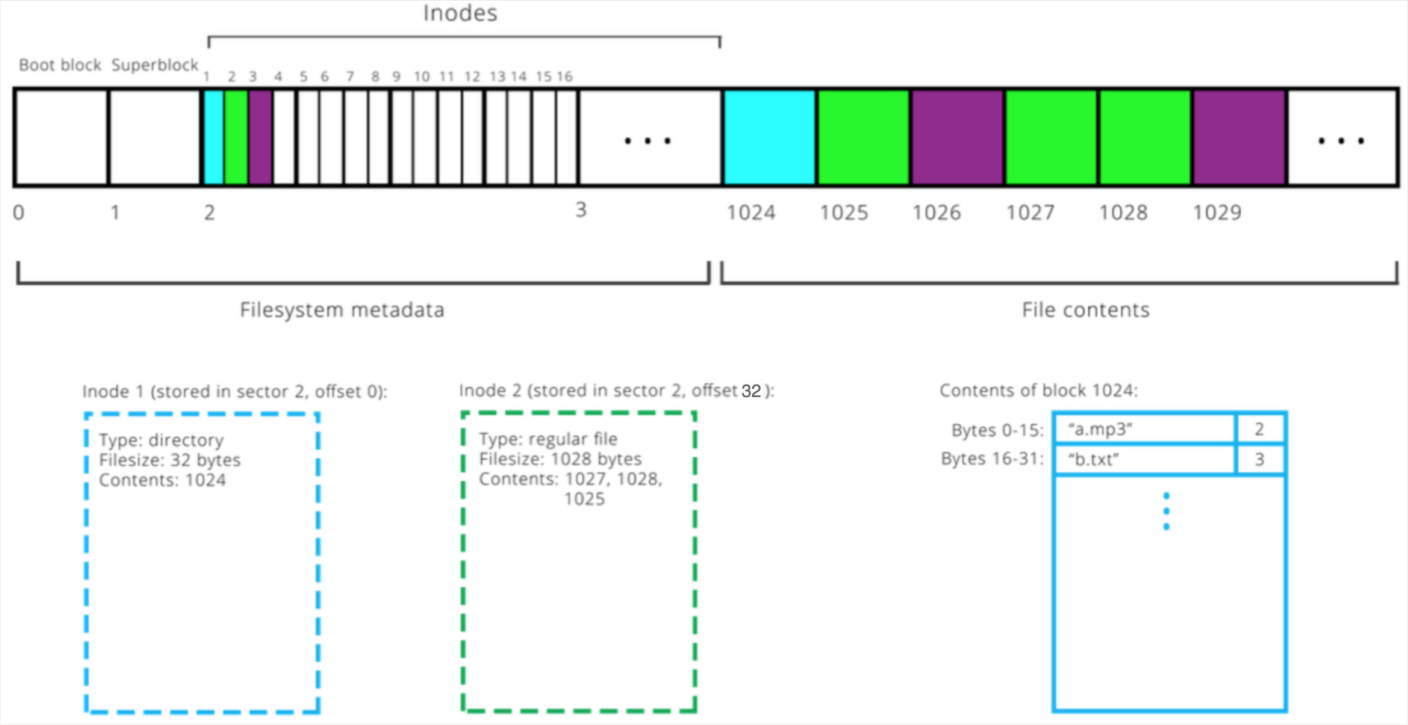
Lecture 03: Layering, Naming, and Filesystem Design

- 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, 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 discussed in the textbook does), then the terms blocks and sectors can be used interchangeably (and the rest of this slide deck will do precisely that).



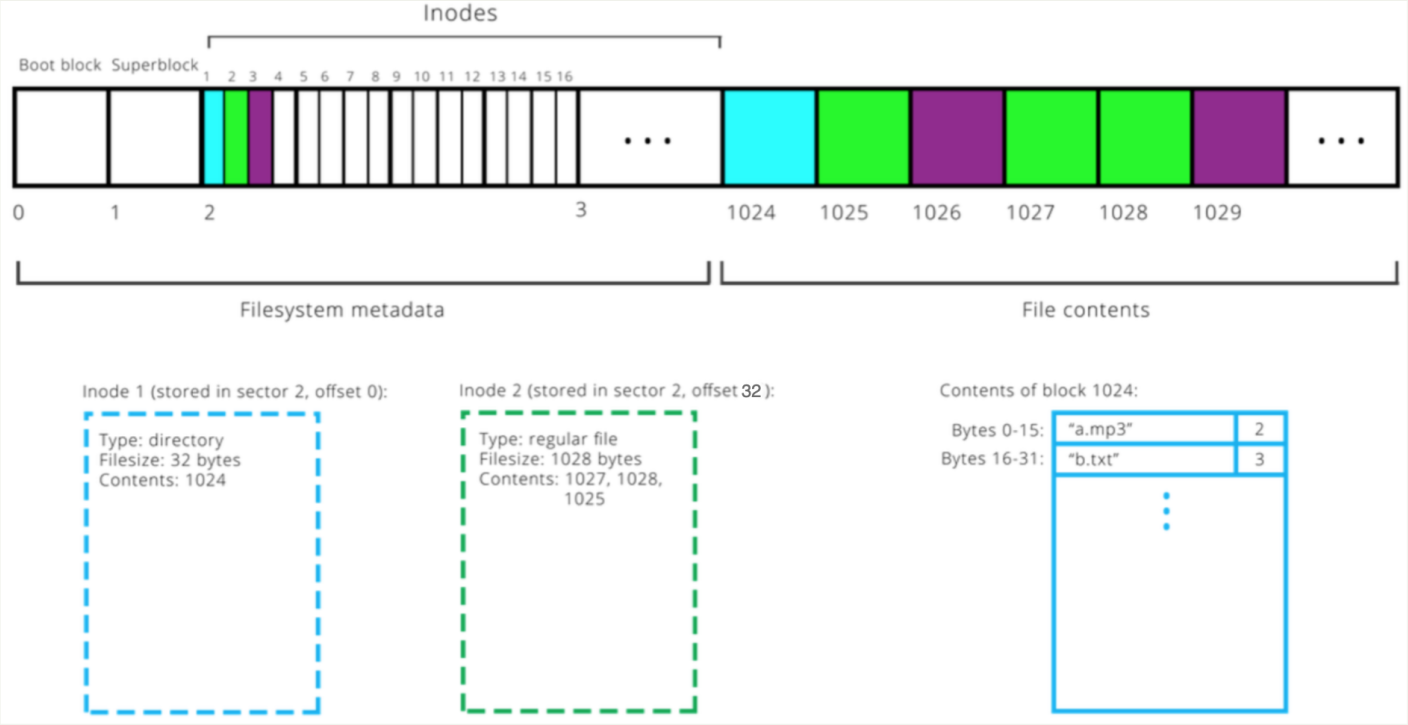
Lecture 03: Layering, Naming, and Filesystem Design

- 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.



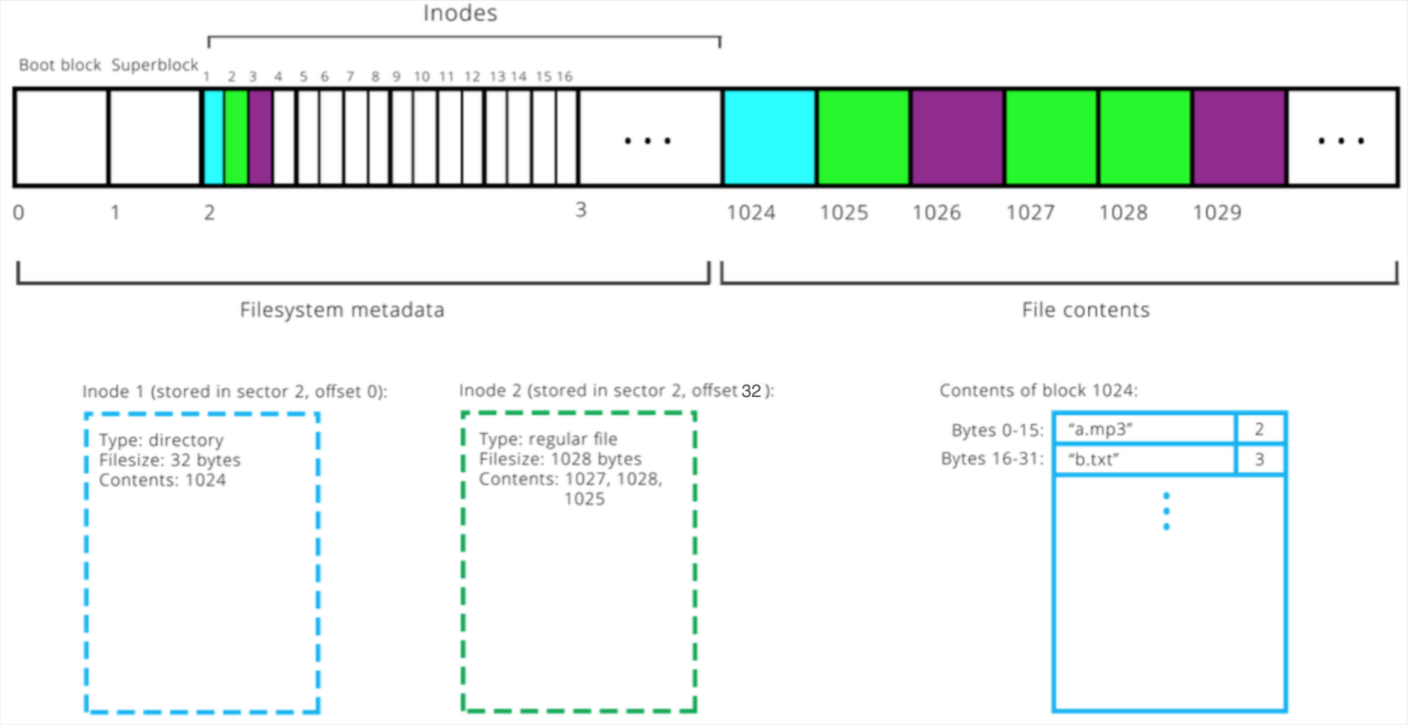
Lecture 03: Layering, Naming, and Filesystem Design

- 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.



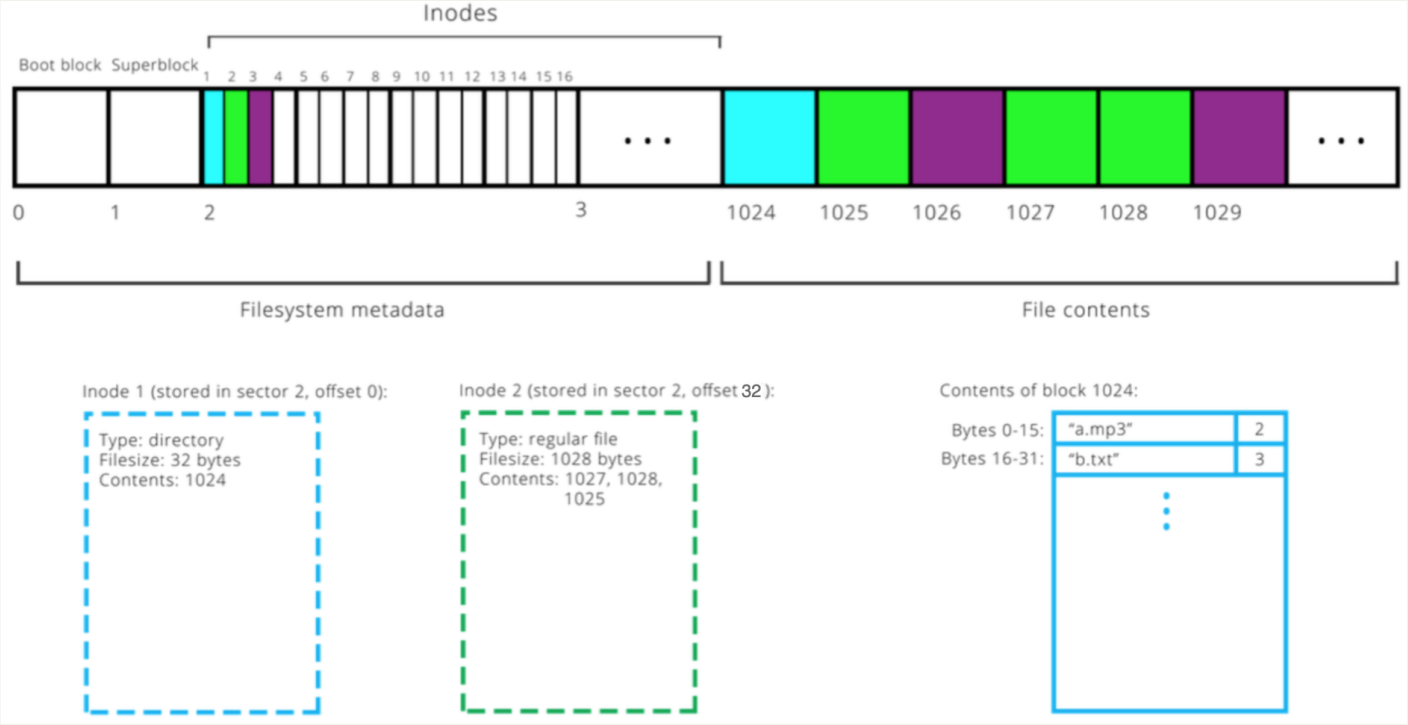
Lecture 03: Layering, Naming, and Filesystem Design

- 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.



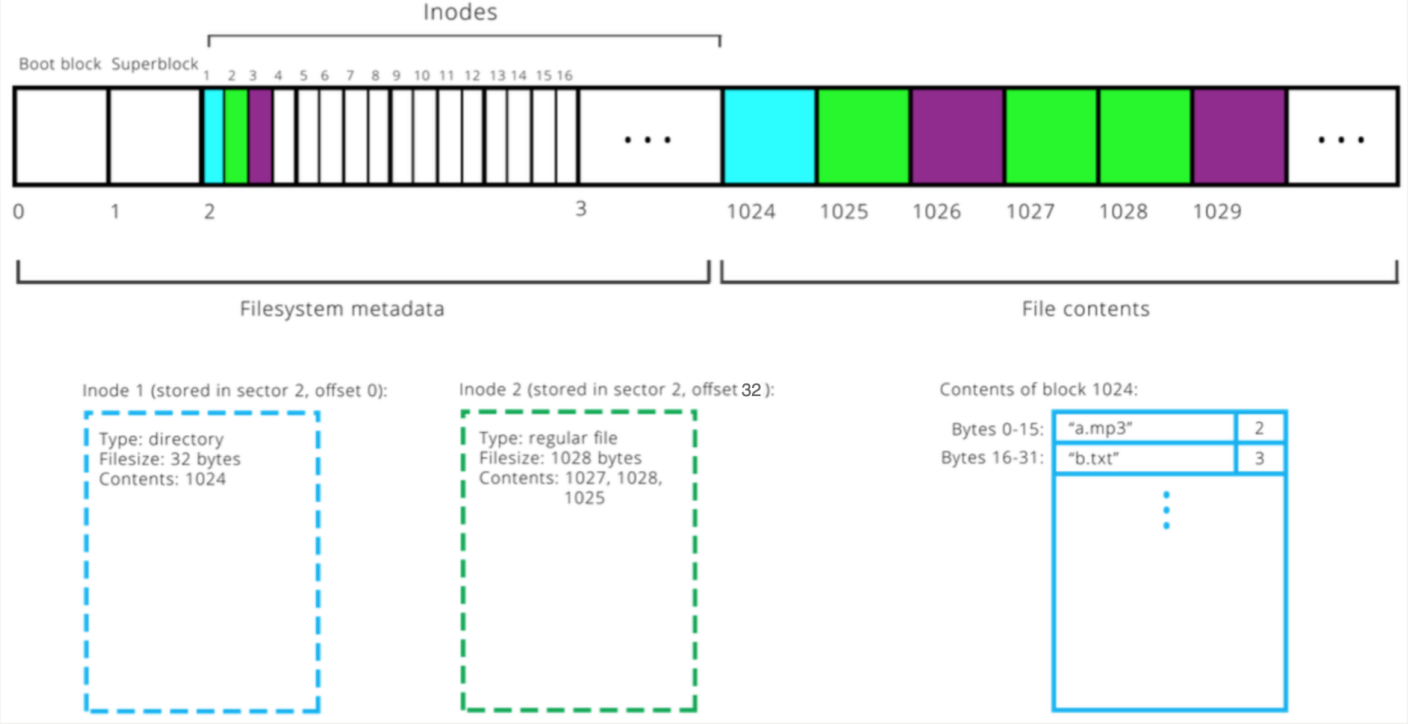
Lecture 03: Layering, Naming, and Filesystem Design

- 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 and a 1028 (or 2 * 512 + 4) byte file, so each enlists some block to store a partial.



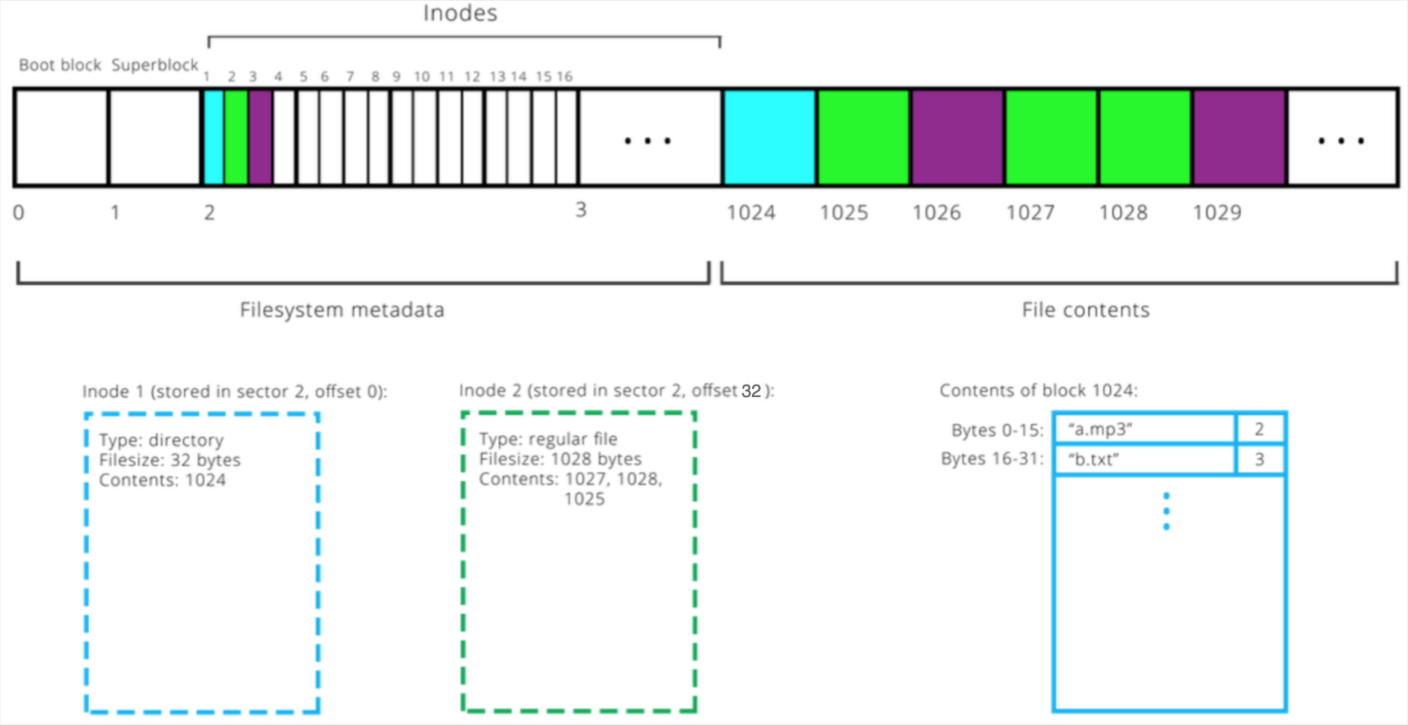
Lecture 03: Layering, Naming, and Filesystem Design

- 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 because they're the same color in the diagram.
  - **inodes** are 32-byte 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.



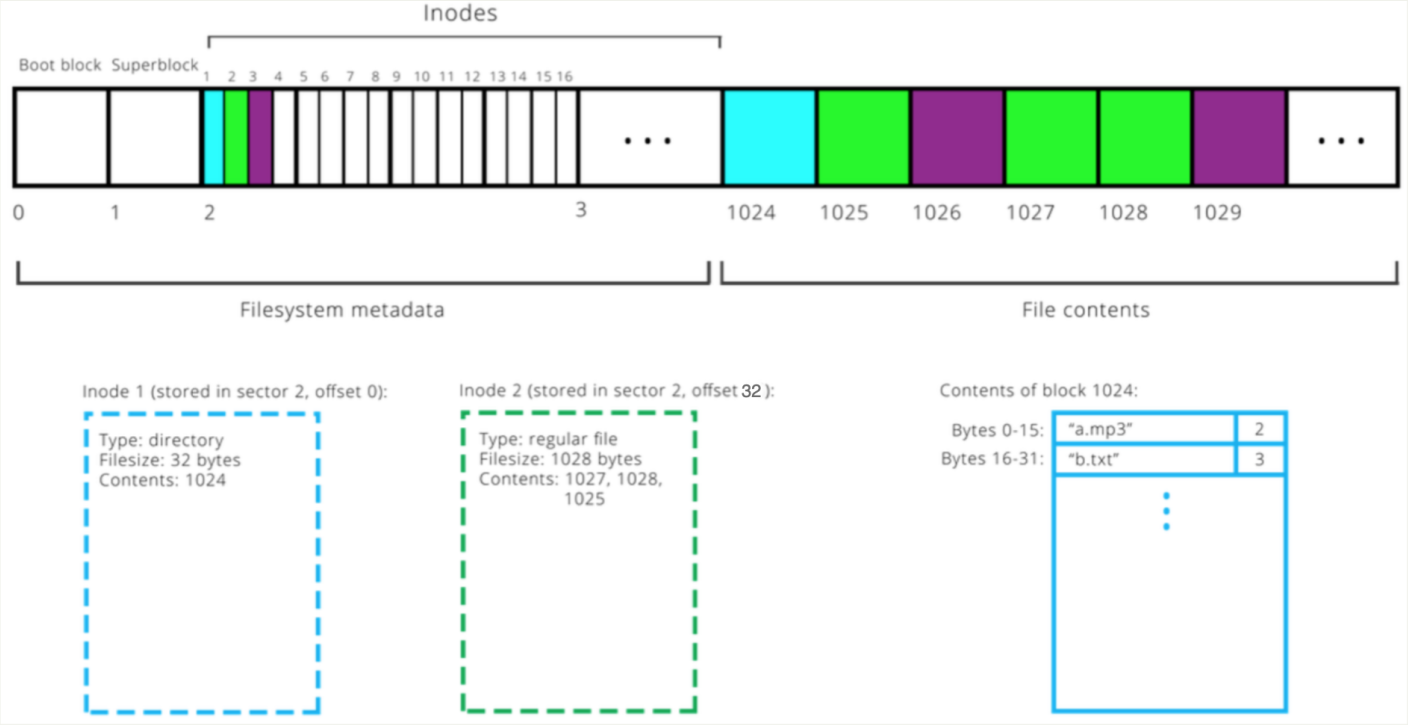
Lecture 03: Layering, Naming, and Filesystem Design

- 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.



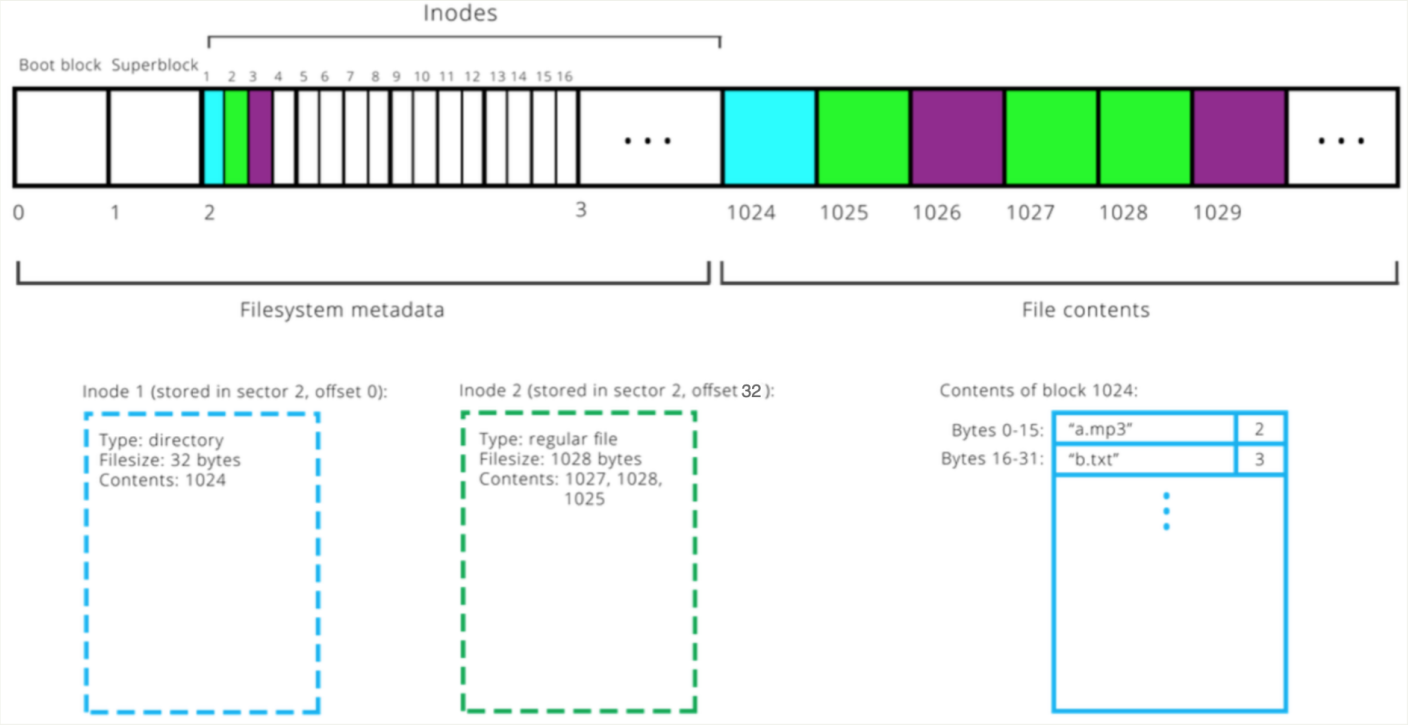
Lecture 03: Layering, Naming, and Filesystem Design

- 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.



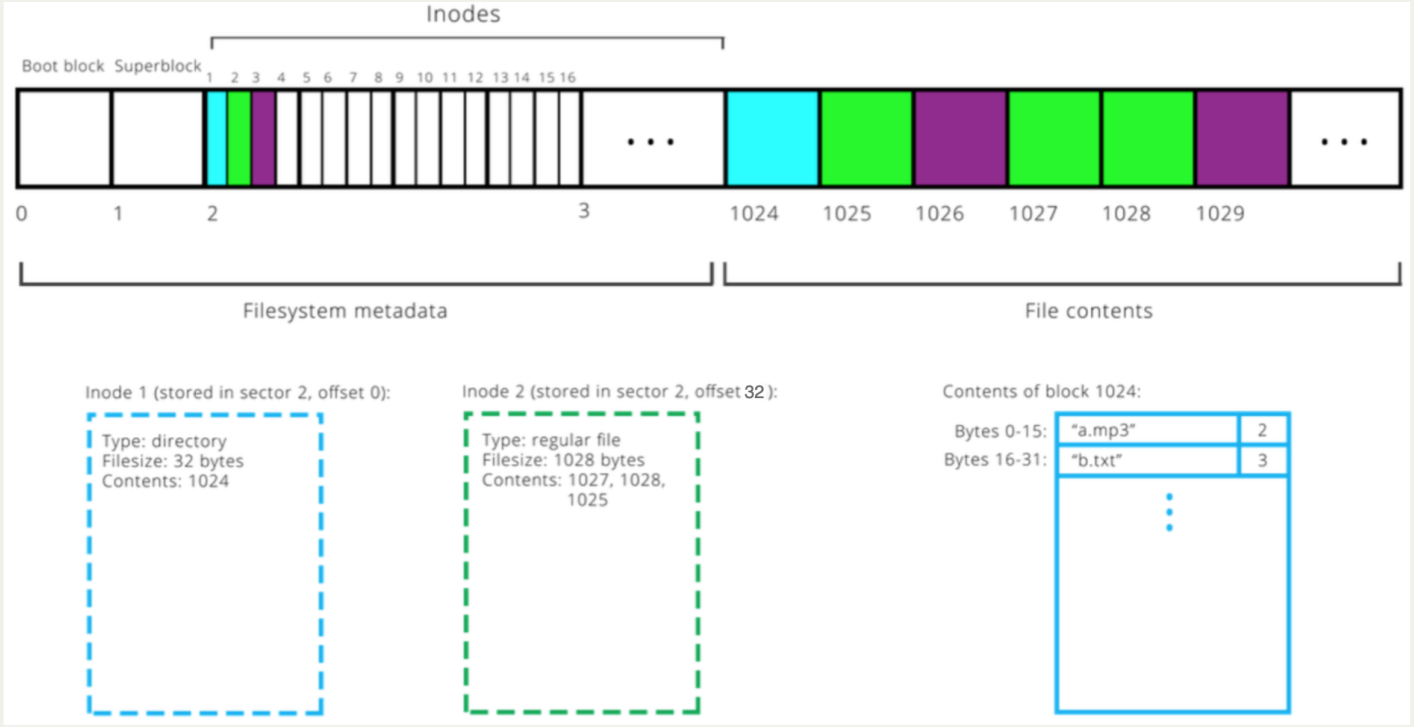
Lecture 03: Layering, Naming, and Filesystem Design

- 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—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.



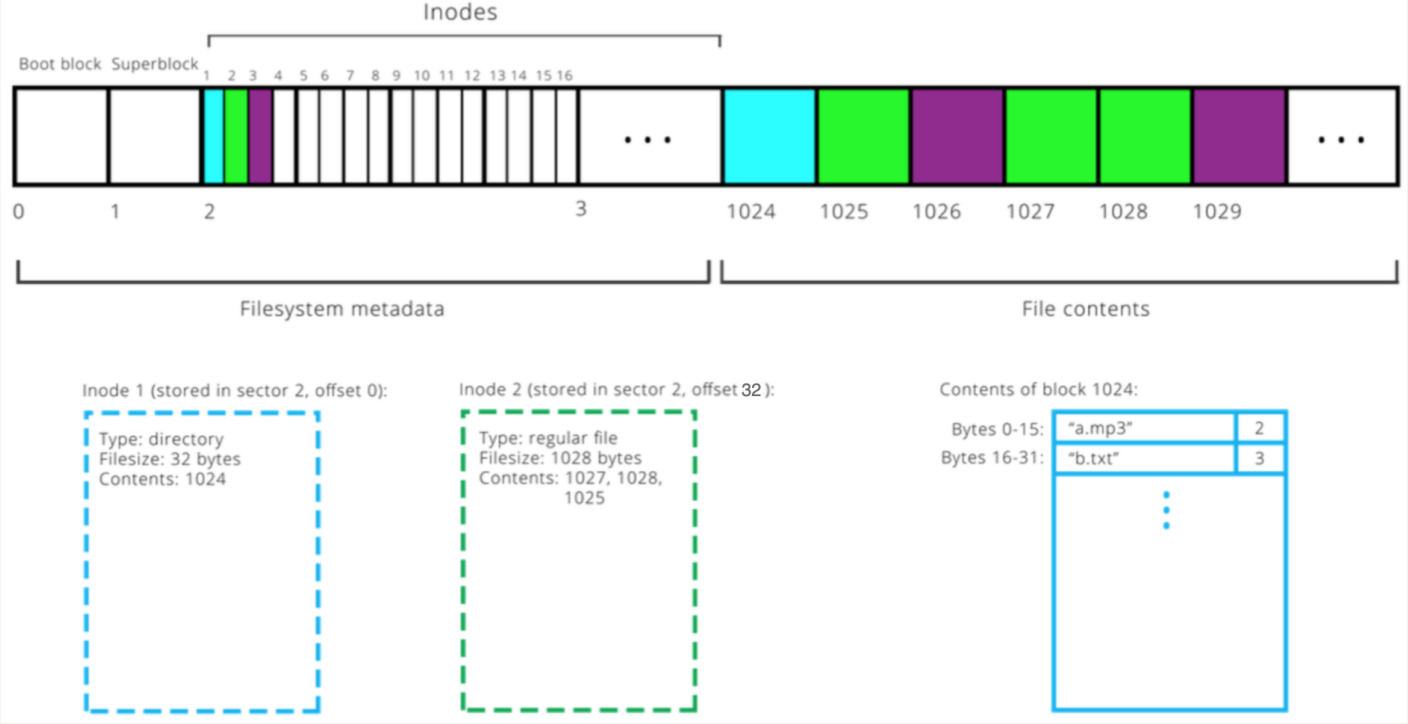
Lecture 03: Layering, Naming, and Filesystem Design

- 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`." 😱
  - 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.



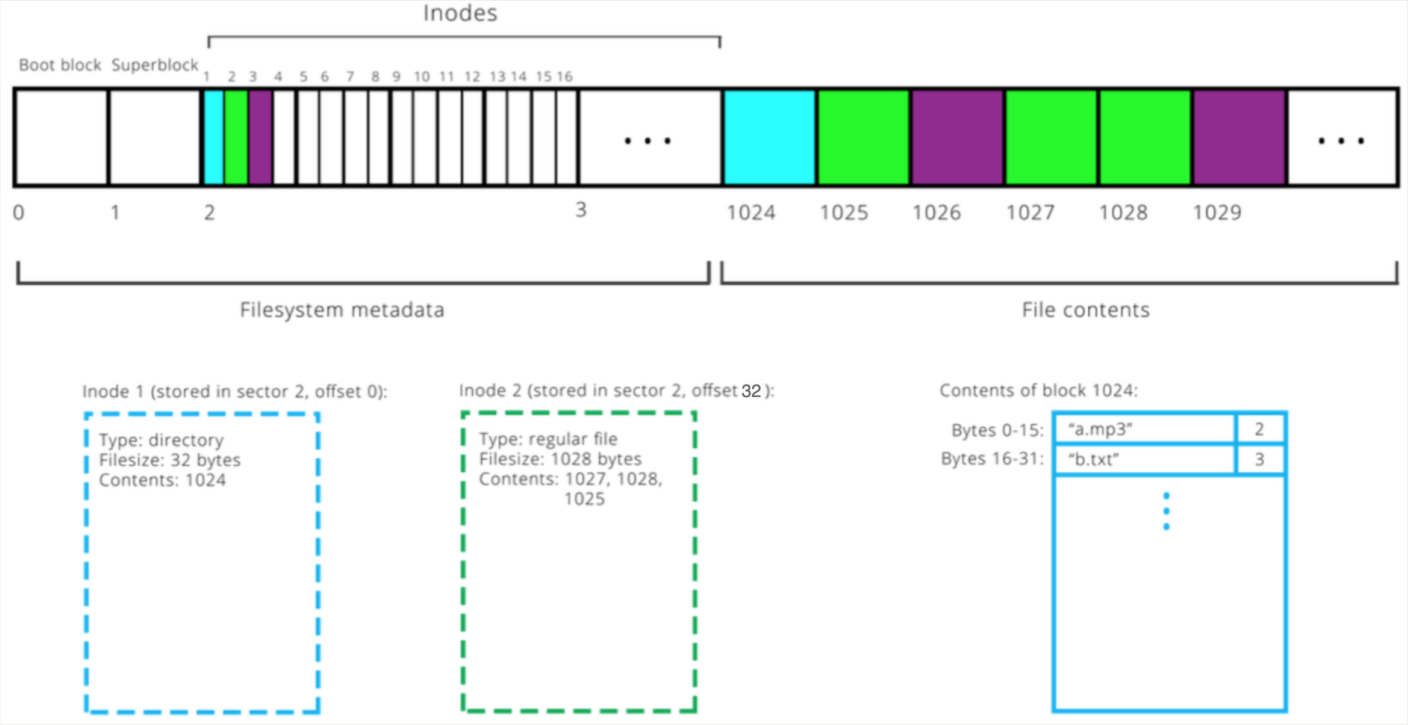
Lecture 03: Layering, Naming, and Filesystem Design

- 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.



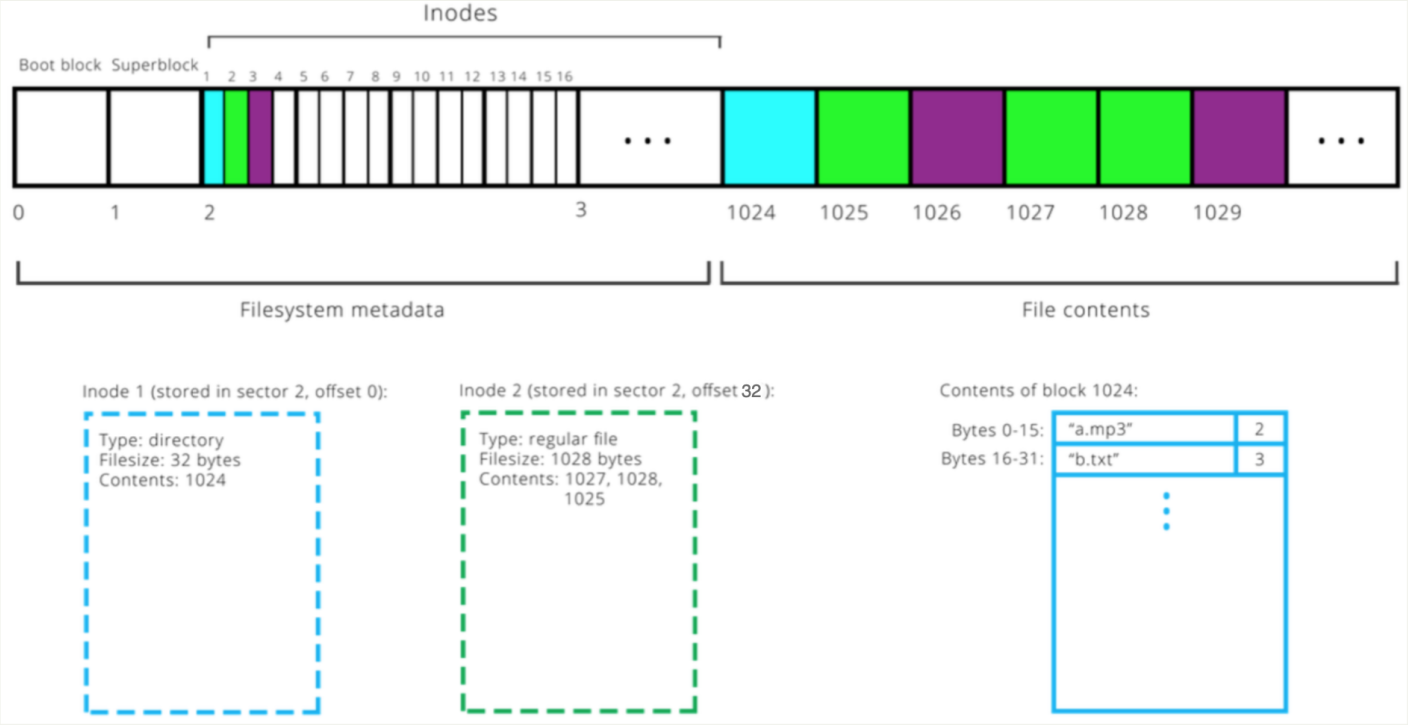
Lecture 03: Layering, Naming, and Filesystem Design

- 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). 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.



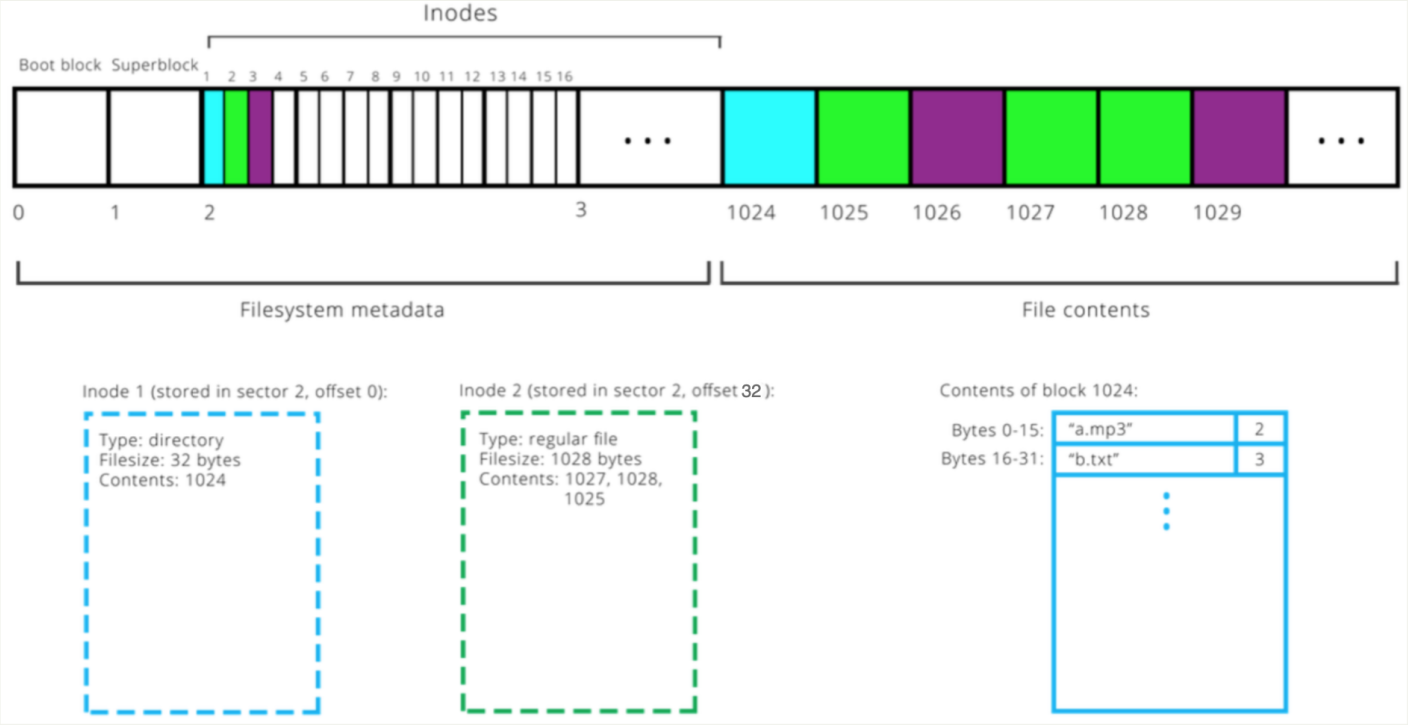
Lecture 03: Layering, Naming, and Filesystem Design

- 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.



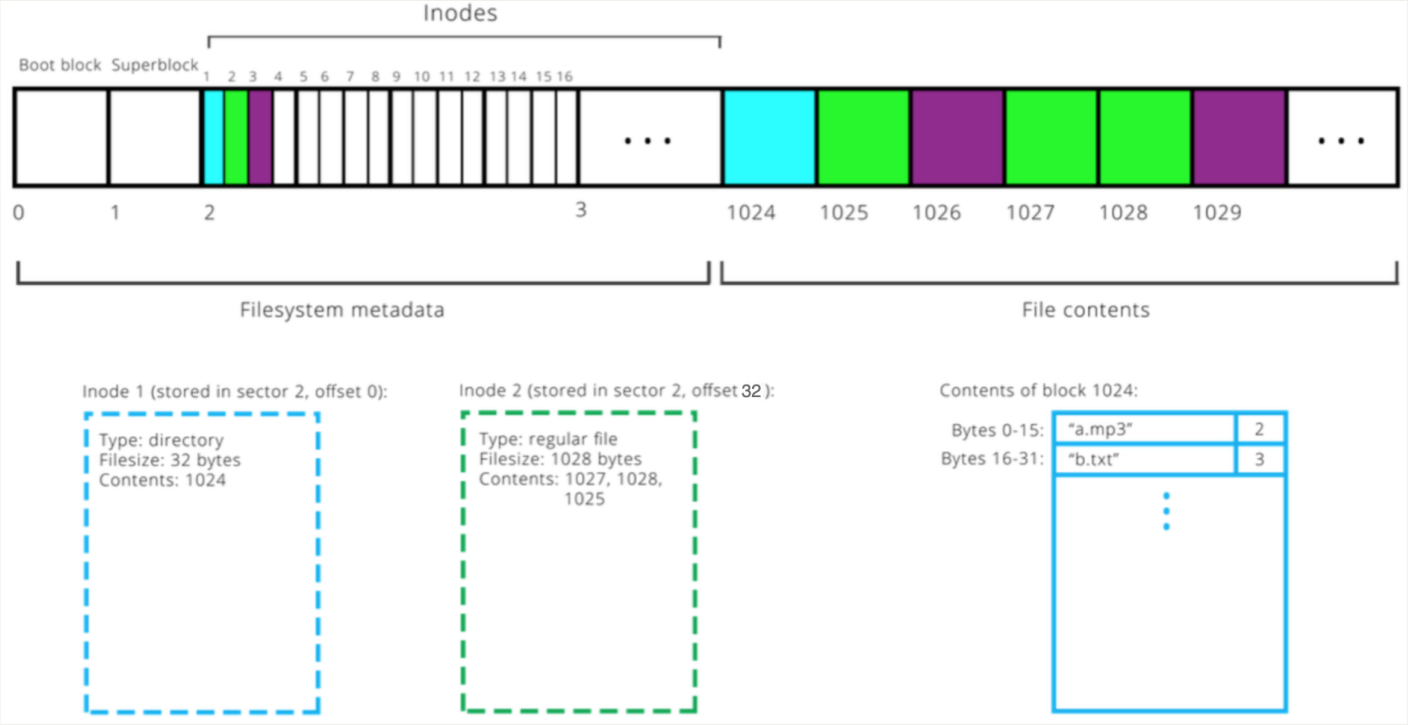
Lecture 03: Layering, Naming, and Filesystem Design

- 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). 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 and search for the token **`class`** in the same way. From there, we look up the token **`cs110`** and then **`example.txt`**.



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

- Introducing the directory file type
  - 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

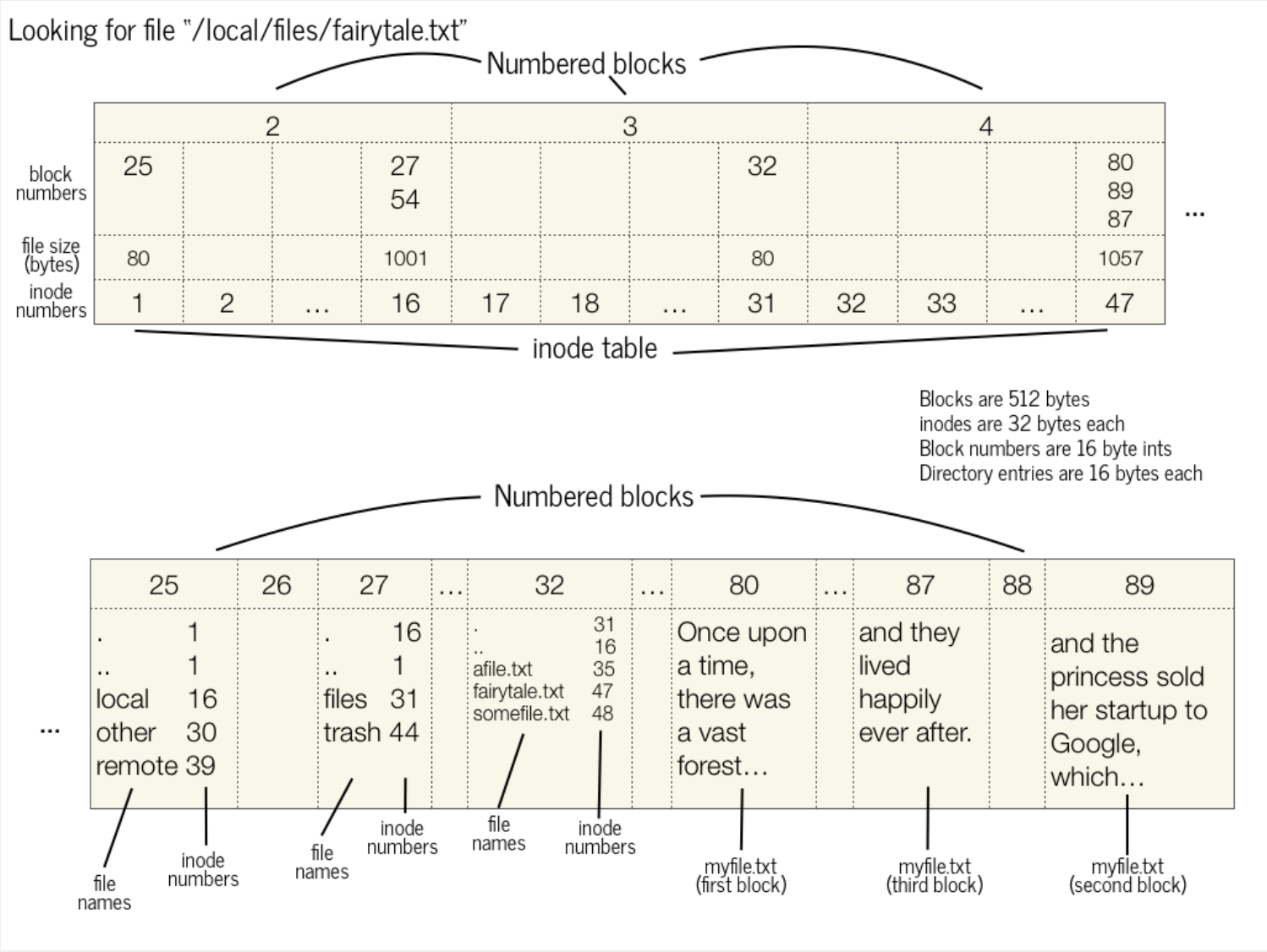
- What about *even larger* files? We have another solution!
  - 1MB is still not that big. To make the max file size even bigger, Unix V6 uses the 8th block number of the inode to store a **doubly indirect** block number.
    - In the inode, the first 7 block numbers store to singly indirect block numbers, 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.
  - 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.

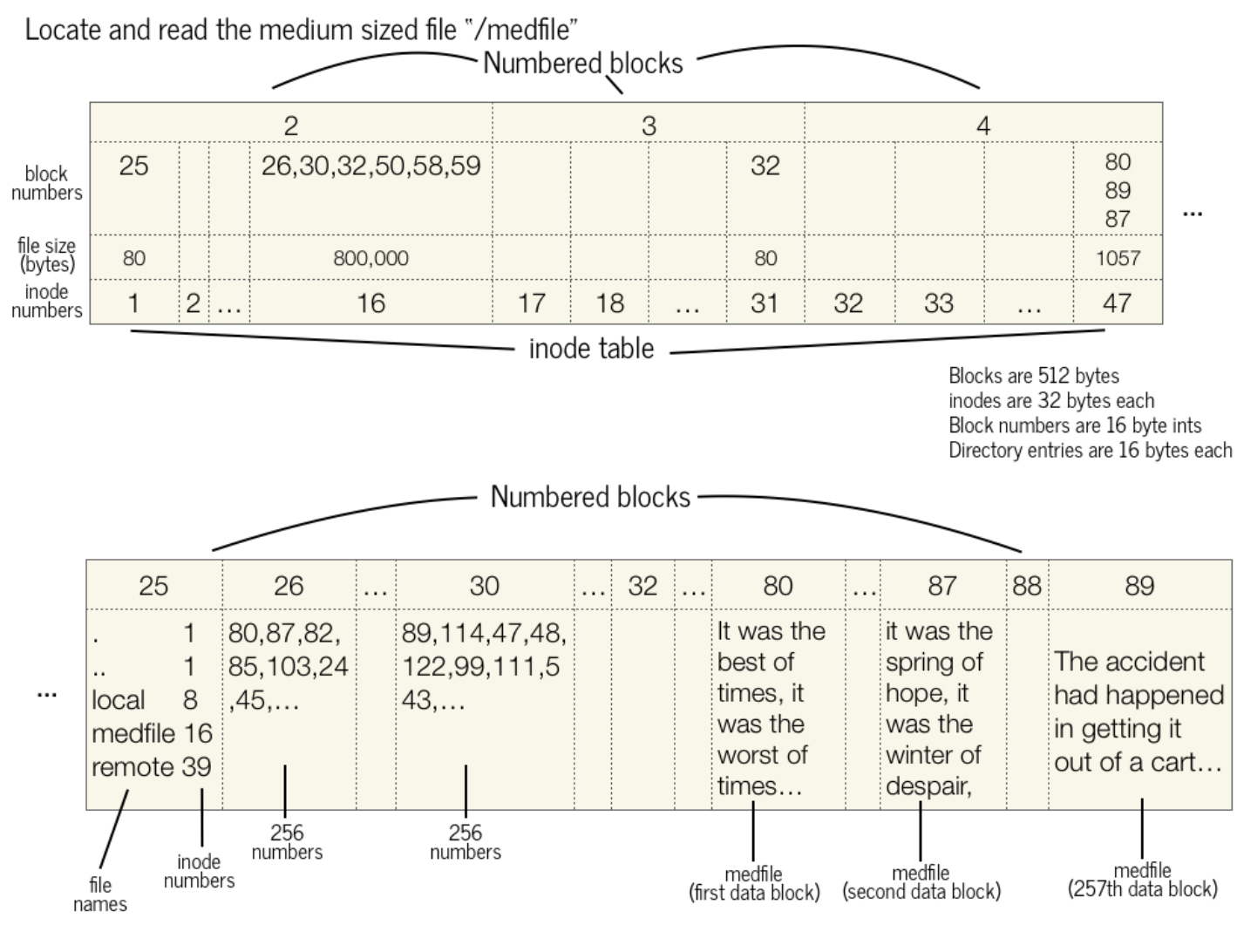


Steps:

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.

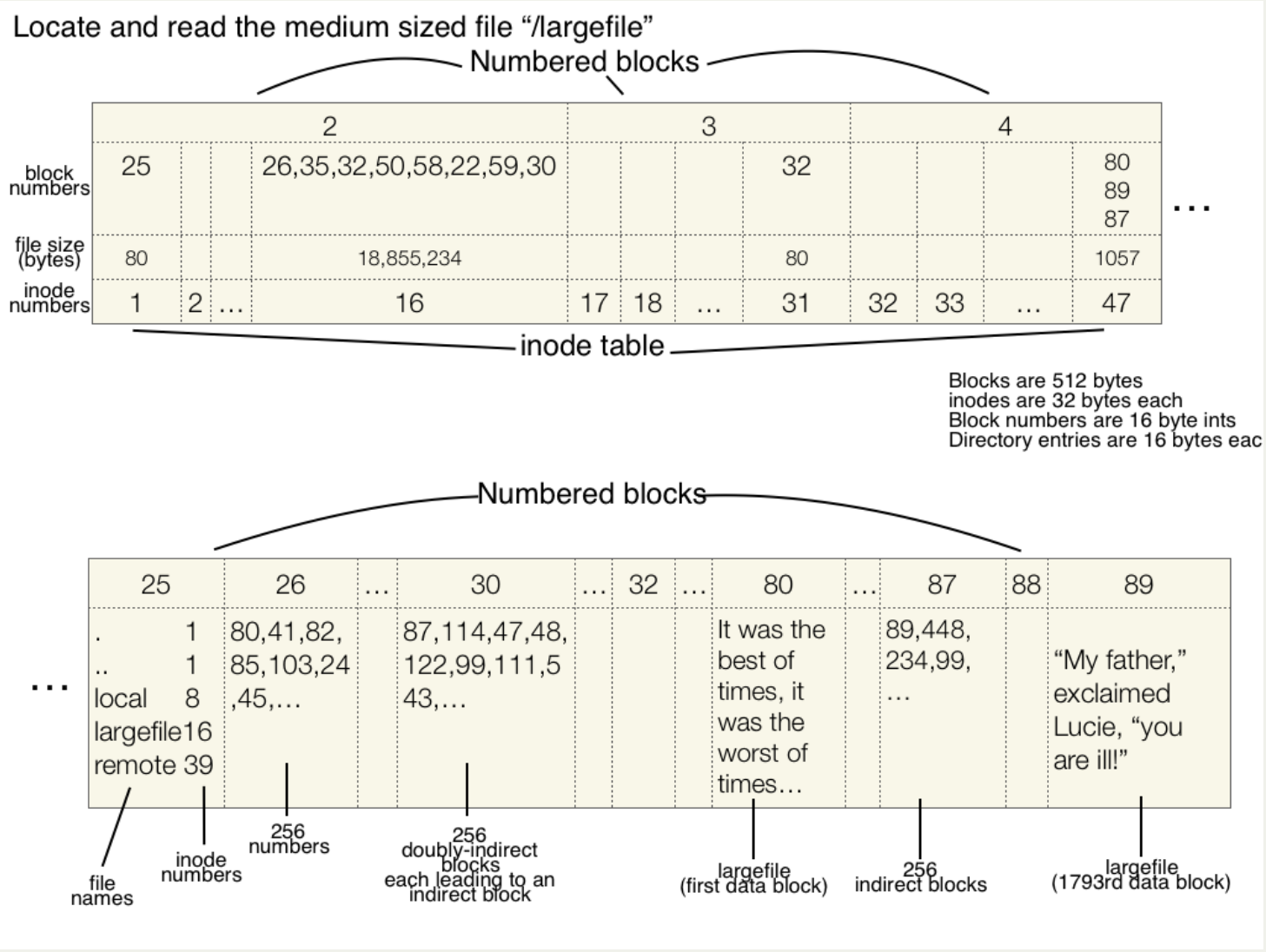


Steps:

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.