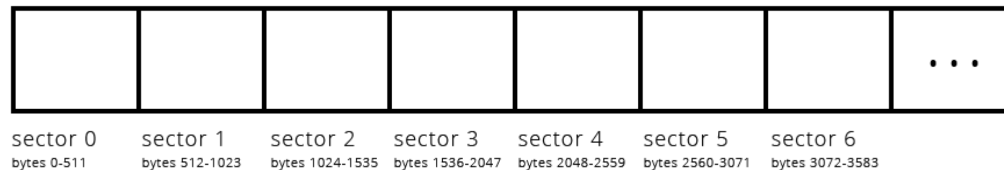
# Lecture 03: Layering, Naming, and Filesystem Design

- 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 all of the surrounding byte back to memory.
- The similar idea applies in the world of hard drives. Hard drives are divided into *sectors* (we'll assume 512 bytes for these drawing), and are *sector-addressable*: you must read or write an entire sector in full, even if you're only interested in a portion of it.
- Sectors are often our assumed 512 bytes in size, but it's not required. The size is determined by the physical drive and might be 1024 bytes, 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:



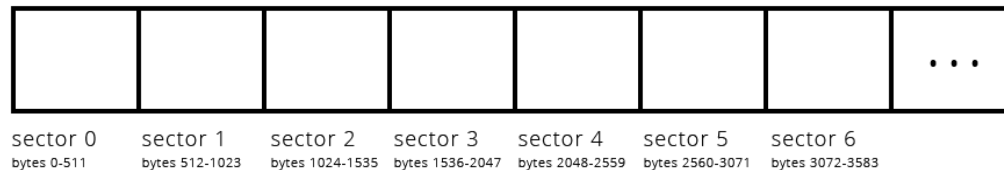

*Thanks to Ryan Eberhardt for the illustrations and the text used in these slides.*

# Lecture 03: Layering, Naming, and Filesystem Design

- The drive itself exports an API–a **hardware** API–that allows us to read a single 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.

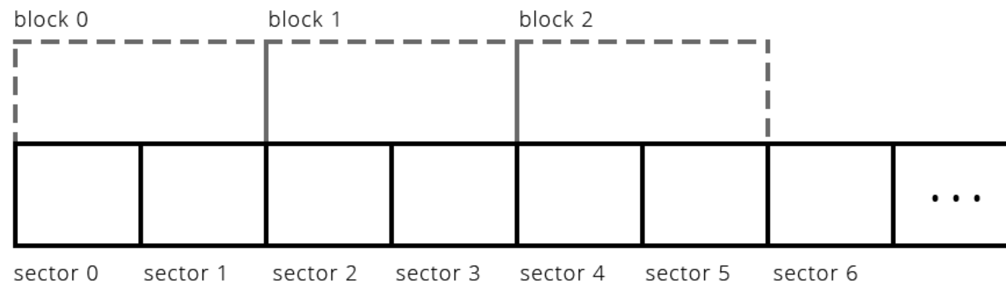
```
class Drive {
public:
    size_t getNumSectors() const;
    void readSector(size_t num, unsigned char data[]) const;
    void writeSector(size_t num, const unsigned char data[]);
};
```

- 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.



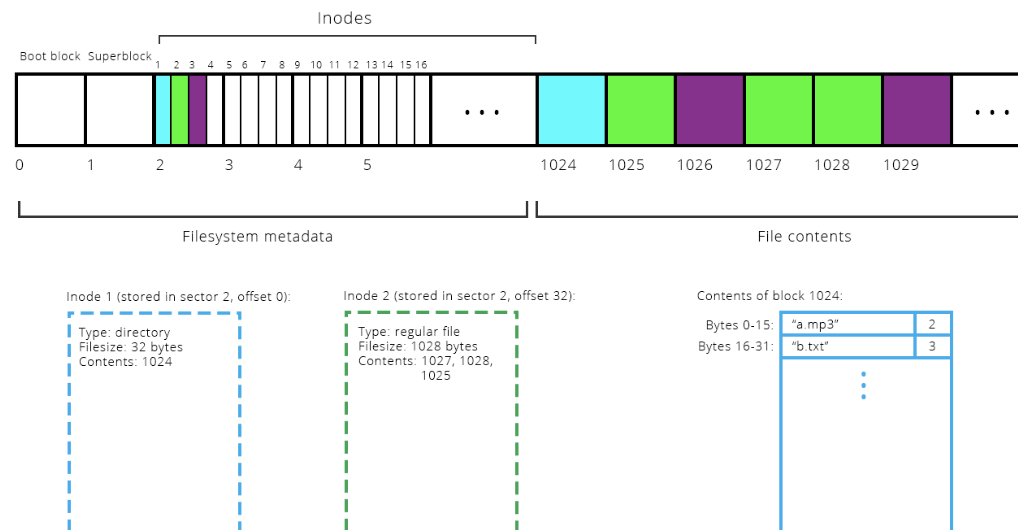
# Lecture 03: Layering, Naming, and Filesystem Design

- 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, 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 your secondary textbook does), then the terms blocks and sectors are used interchangeably. (The rest of this slide deck will do precisely that.)



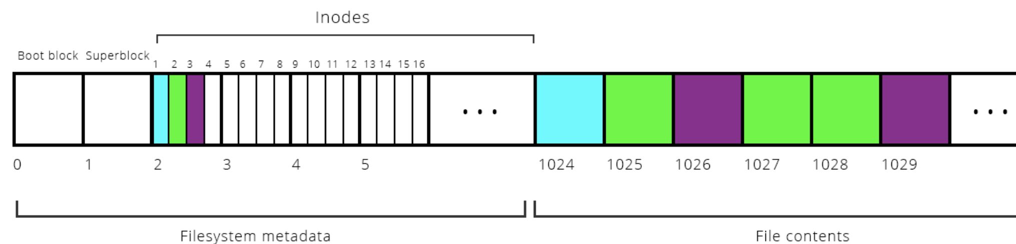
# Lecture 03: Layering, Naming, and Filesystem Design

- 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 let you know what's going on.



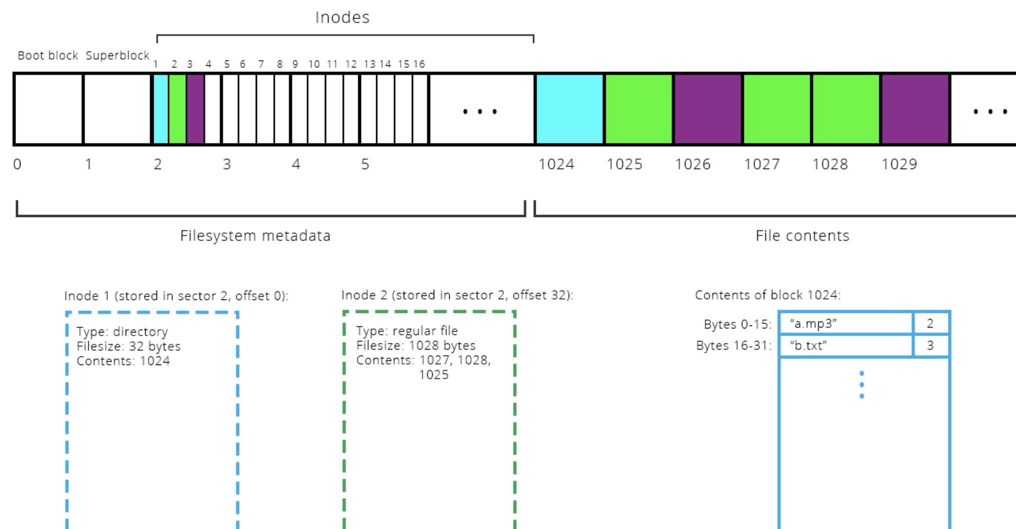
# Lecture 03: Layering, Naming, and Filesystem Design

- 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 superimposed onto the hardware.



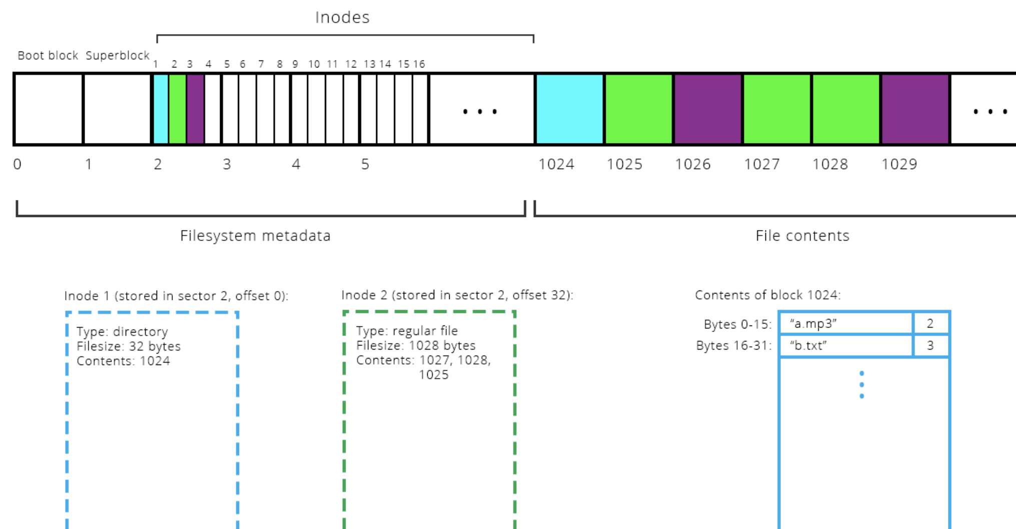
# Lecture 03: Layering, Naming, and Filesystem Design

- 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.



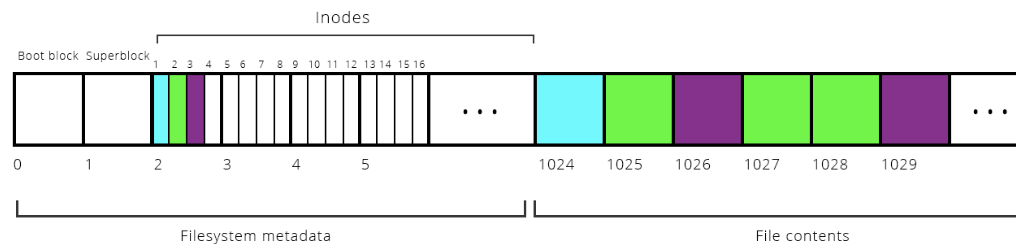
# Lecture 03: Layering, Naming, and Filesystem Design

- 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 its 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 (i.e. 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.




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

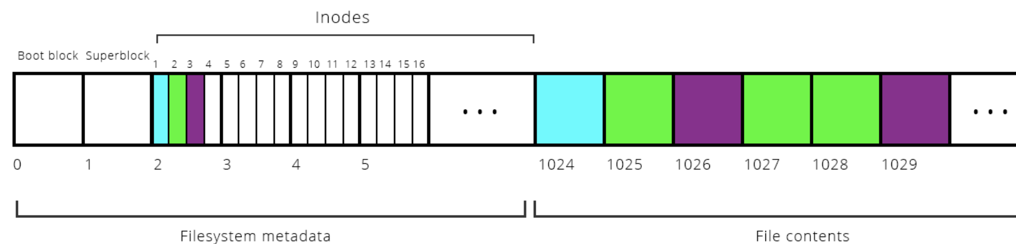
Inode 2 (stored in sector 2, offset 32):
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

# 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 three blocks are used to store the full payload. The first two are fully saturated, and the third will only store 1028 % 512, or 4, meaningful 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.




Inode 1 (stored in sector 2, offset 0):

Type: directory
Filesize: 32 bytes
Contents: 1024

Inode 2 (stored in sector 2, offset 32):

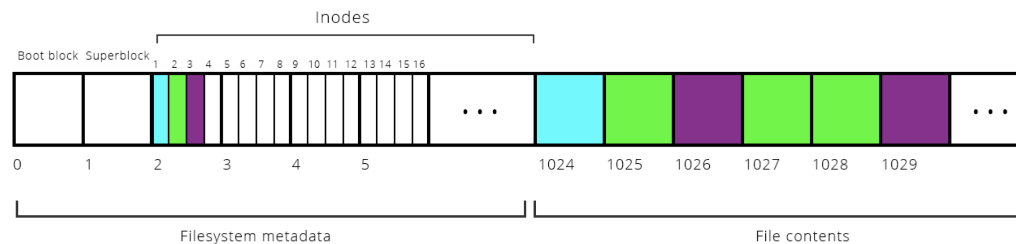
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

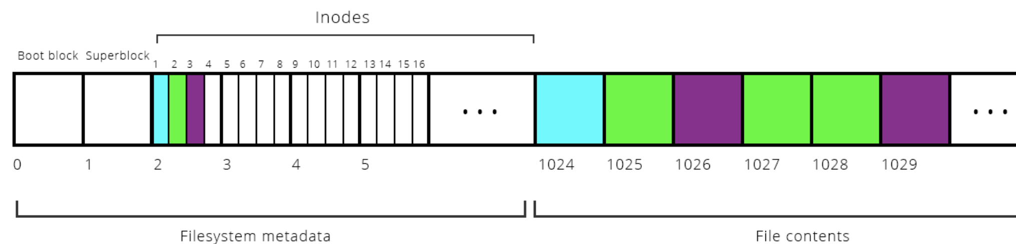
# Lecture 03: Layering, Naming, and Filesystem Design

- 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 byte.  Block 1025 might get reused to store those extra bytes!
  - 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.



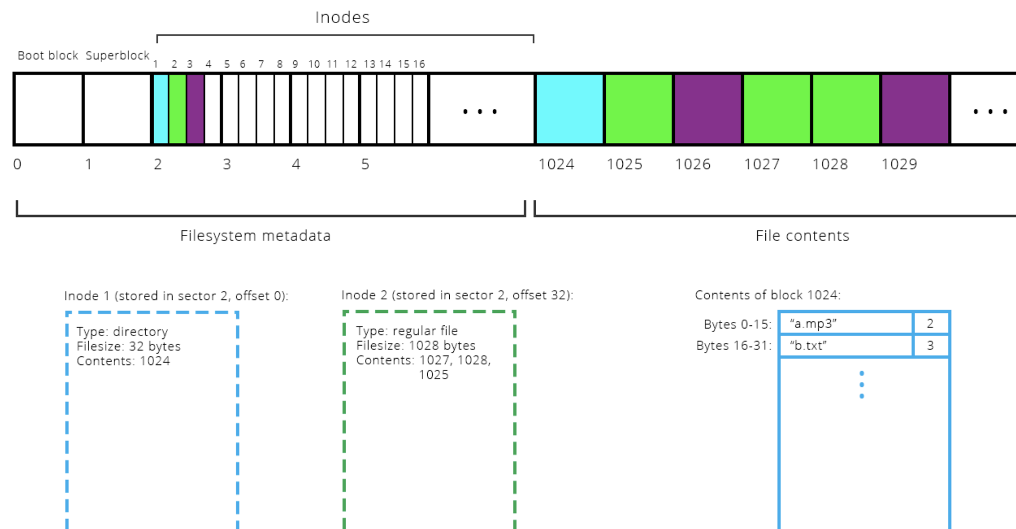
# Lecture 03: Layering, Naming, and Filesystem Design

- The inode, continued
  - A file's inodes tell us where we'll find its payload, but the inode itself also has to be stored on the drive. Realistically, where else could it persist between computer power cycles?
  - 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 cubbyhole.



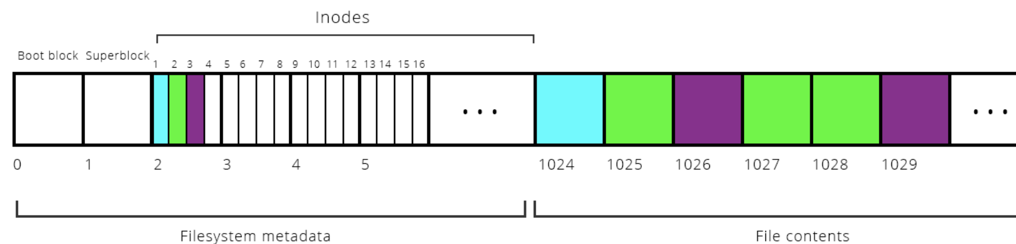
# Lecture 03: Layering, Naming, and Filesystem Design

- The inode, continued
  - If we needed to remember the inode number of every file on our system, we'd be sad.
  - 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. My 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 slow. My own laptop has about two million files, so the inode table is at least that big, probably much bigger.




Inode 1 (stored in sector 2, offset 0):

Type: directory
Filesize: 32 bytes
Contents: 1024

Inode 2 (stored in sector 2, offset 32):

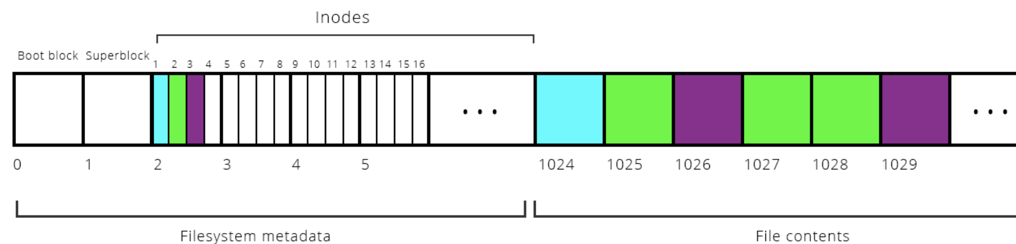
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

# Lecture 03: Layering, Naming, and Filesystem Design

- Introducing the directory file type
    - The solution is to introduce the **directory** as a new file type. You may be surprised to find that this requires almost no changes to our existing scheme, as we can layer directories atop 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 conames to inode numbers.




Inode 1 (stored in sector 2, offset 0):

Type: directory
Filesize: 32 bytes
Contents: 1024

Inode 2 (stored in sector 2, offset 32):

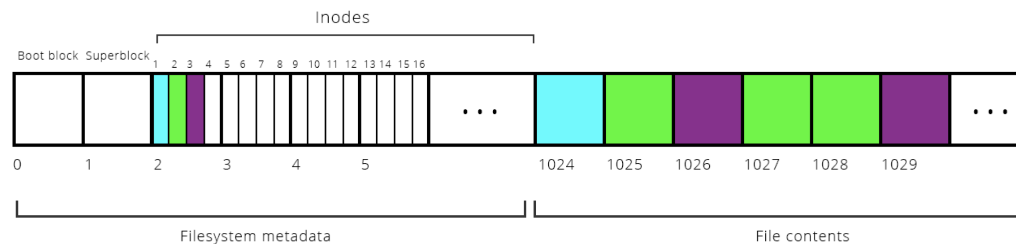
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

# 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 looking for a file in a directory, we're actually searching for that name and the inumber right next to it.




Inode 1 (stored in sector 2, offset 0):

Type: directory
Filesize: 32 bytes
Contents: 1024

Inode 2 (stored in sector 2, offset 32):

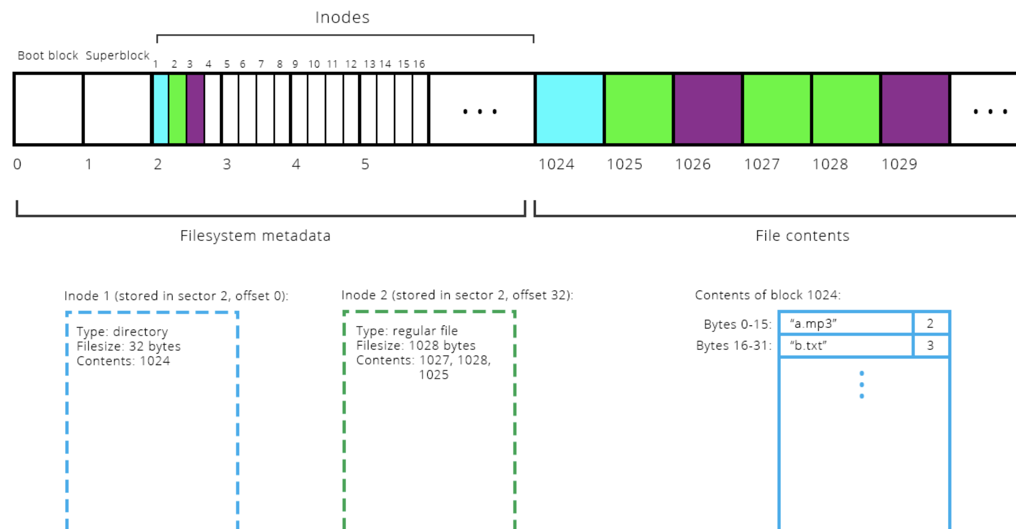
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

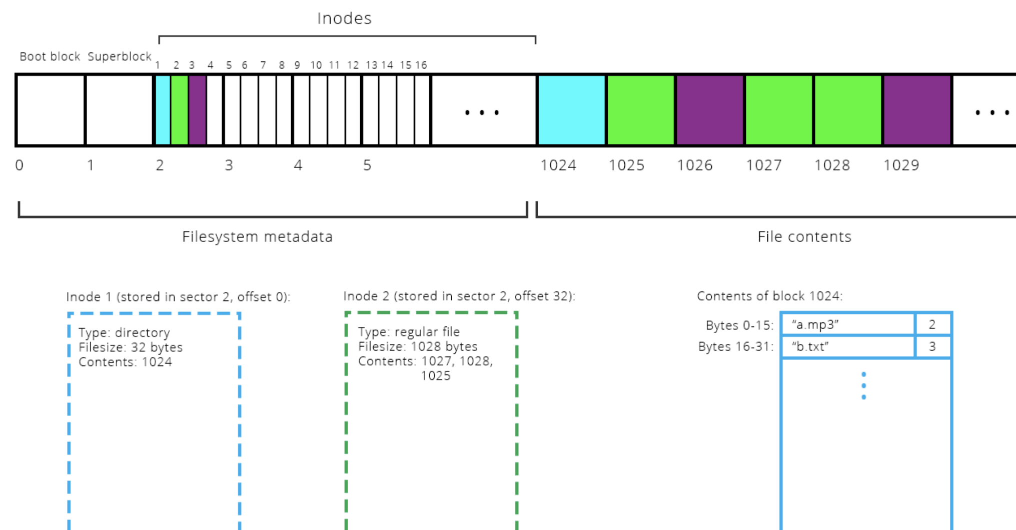
# Lecture 03: Layering, Naming, and Filesystem Design

- 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 (the one that's described as a case study in your textbook), inodes can only store a maximum of 8 block numbers. This presumably limits the total file size to 8 * 512 = 4096 bytes, but fortunately that's not really the case.



# 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 **direct** block numbers —ones storing true user payload—from that block.
            - This is known as **singly**-indirect addressing.
            - We can 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.

# Lecture 03: Layering, Naming, and Filesystem Design

- What about large files? We have a better solution!
  - That's 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.