CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 02: Filesystem Design

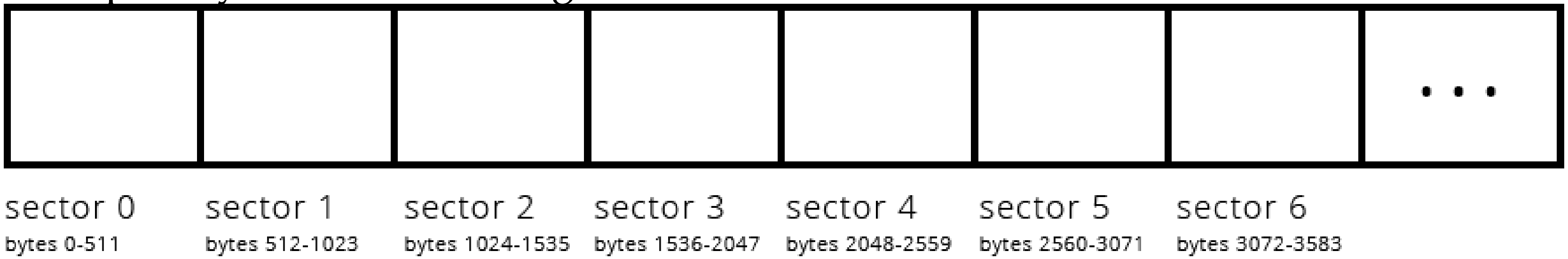
- Today, we are going to start discussing the Unix version 6 file system.
  - This is a relatively old file system (c. 1975), but it's open source and is well-designed. Its design is relatively easy to understand.
  - Your second assignment is a variation on this file system.
  - Modern file systems (particularly for Linux) are adaptations of this file system, but they are more complicated and geared toward high performance and fault tolerance. In other words, they aren't the best examples to study unless you're already familiar with filesystem design.
  - However, you can dig into the details of many open-source modern file systems (e.g., the ext4 file system, which is the most common Linux file system right now)
  - When we say that a "block is 512 bytes" in the introduction that follows, know that this is for the Unix version 6 file system, and not the block size for all filesystems.
  - Some key takeaways from studying this file system:
    - You're studying a part of computing history.
    - You're analyzing a professional-grade abstraction that strongly influenced the construction of subsequent filesystems.
    - 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 design a filesystem.

Thanks to Chris Gregg for providing this lovely overview motivating the study of this particular filesystem!

Lecture 02: 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.
- Hard drives are similar, except the default unit of memory is typically much larger. Hard drives are divided into sectors (we'll assume 512 byte sectors) and are **sector-addressable**: you must read or write an entire sector, even if you're only interested in a portion of it.
- 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:



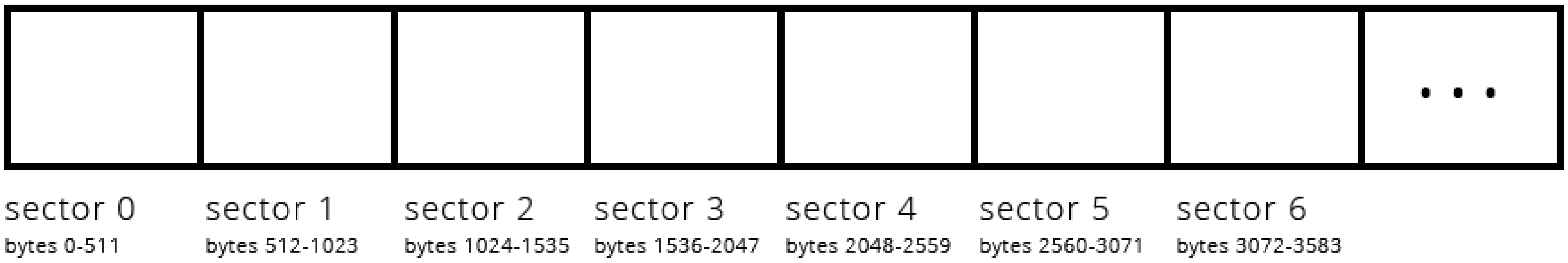
Thanks to Ryan Eberhardt for the illustrations and most of the text used in these slides.

Lecture 02: 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 on the drive 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, void *data) const;
    void writeSector(size_t num, const void *data);
};
```

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



Lecture 02: Filesystem Design

- What do each of these code snippets effectively do?

```
Drive d;
char data[512];
d.readSector(5, data);
```

```
Drive d;
char data[512];
bzero(data, sizeof(data));
d.writeSector(12, data);
```

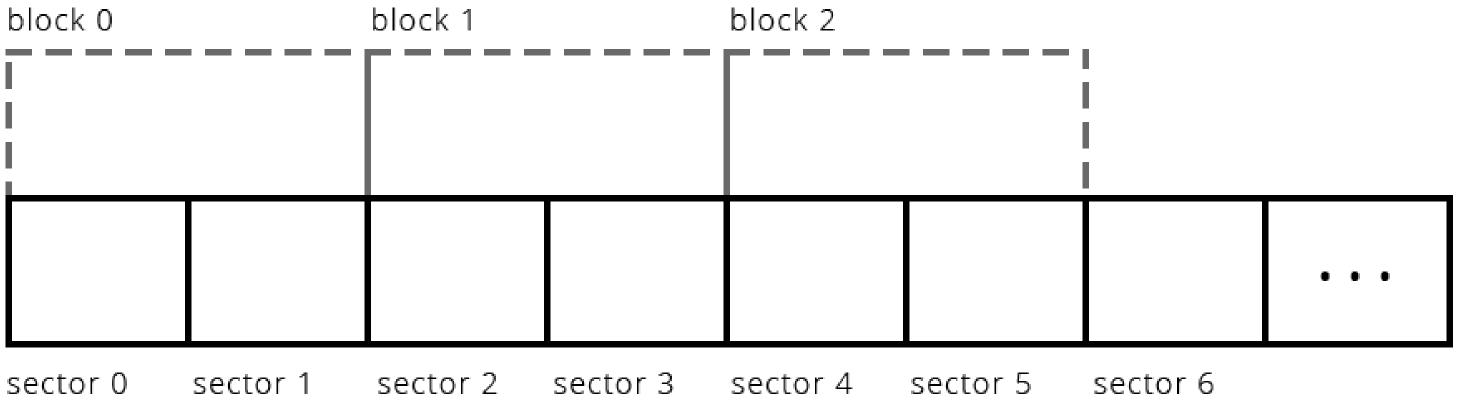
```
Drive d;
int numbers[512/sizeof(int)];
d.readSector(100, numbers);
numbers[0] = 110;
d.writeSector(100, numbers);
```

```
struct inode { bool allocated = false; /* other fields */ };

Drive d;
inode inodes[512/sizeof(inode)];
d.readSector(2, inodes);
for (size_t i = 0; i < 512/sizeof(inode); i++) {
   if (!inodes[i].allocated) {
      inodes[i].allocated = true;
      d.writeSector(2, inodes);
      return i;
   }
}
return -1;
```

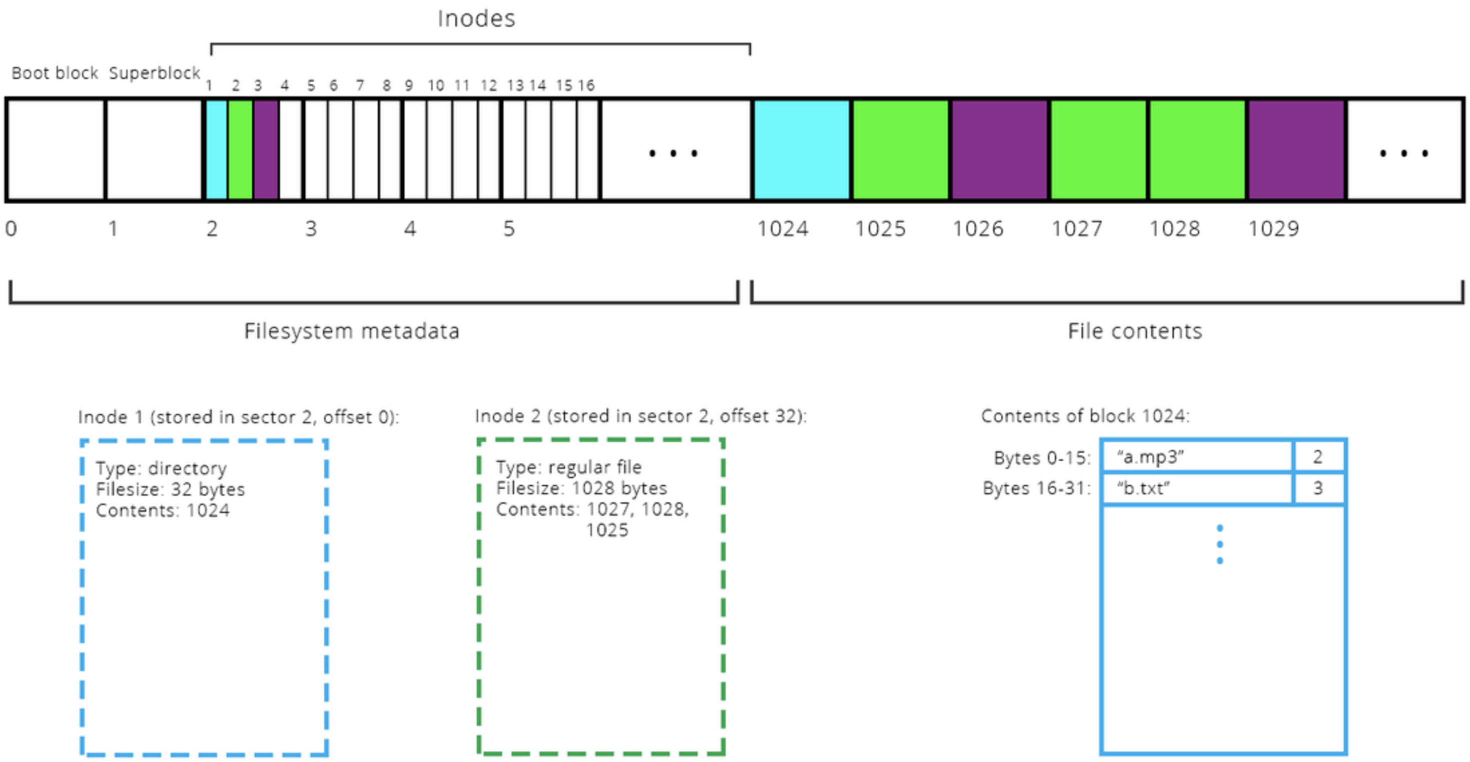
Lecture 02: Filesystem Design

- 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 has 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).



Lecture 02: 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.
    - I will provide a high level explanation of how the physical hardware of the drive is accessed and otherwise manipulated.
    - We'll dedicate live lecture time to go into the details.



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

Type: directory
Filesize: 32 bytes
Contents: 1024

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

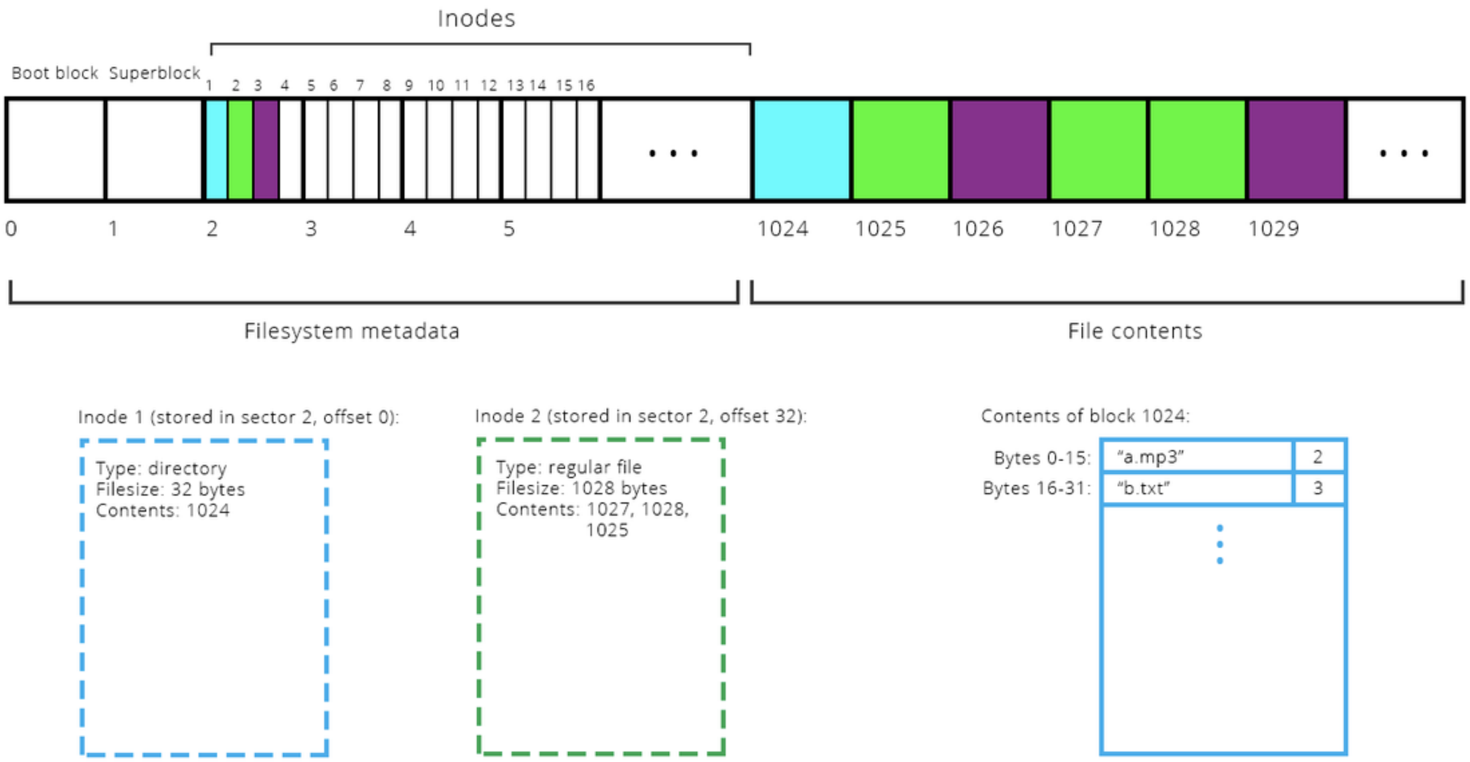
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

Lecture 02: 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 overlaying the hardware.



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

Type: directory
Filesize: 32 bytes
Contents: 1024

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

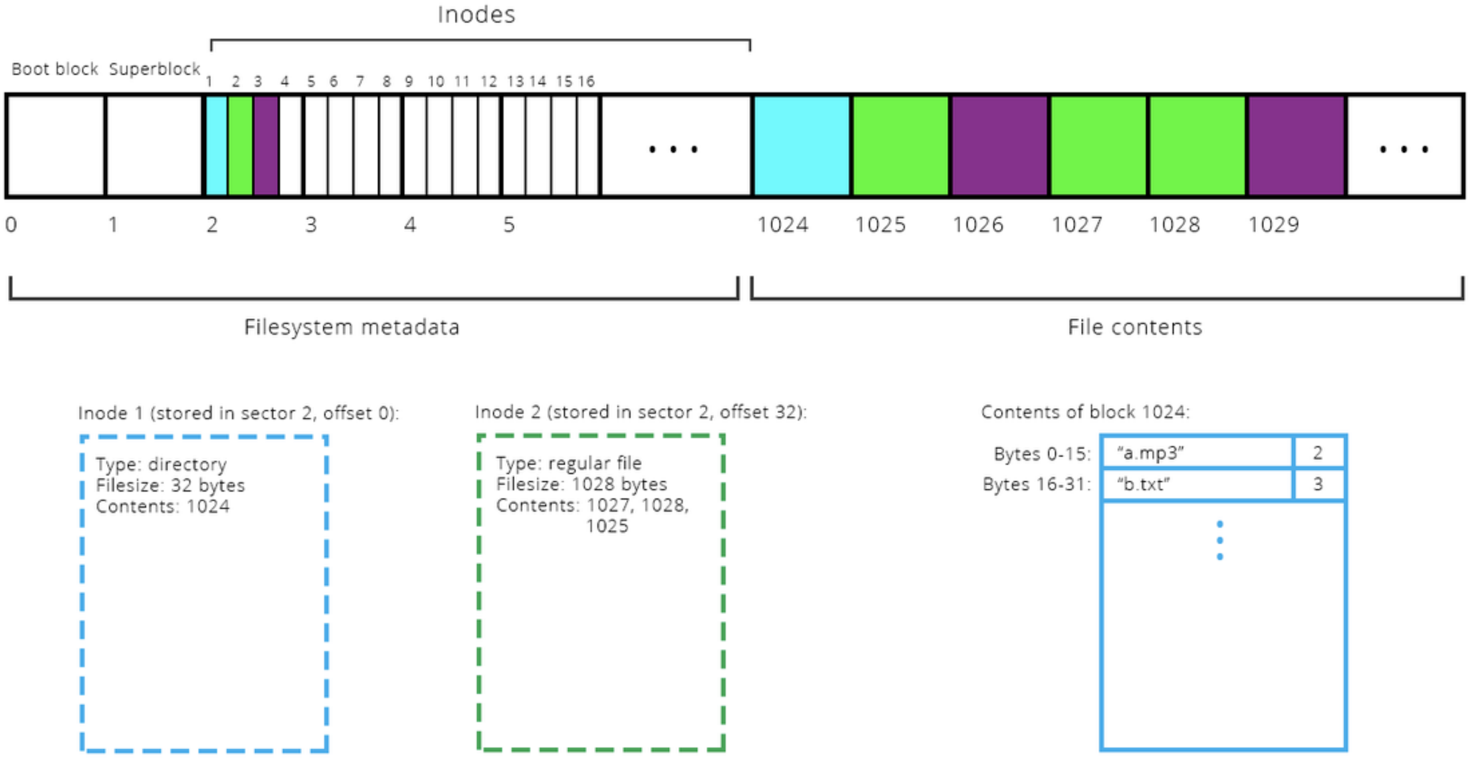
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

Lecture 02: 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, between 5 - 10% of the entire drive is set aside for metadata storage. The rest is used to store file payload.



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

Type: directory
Filesize: 32 bytes
Contents: 1024

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

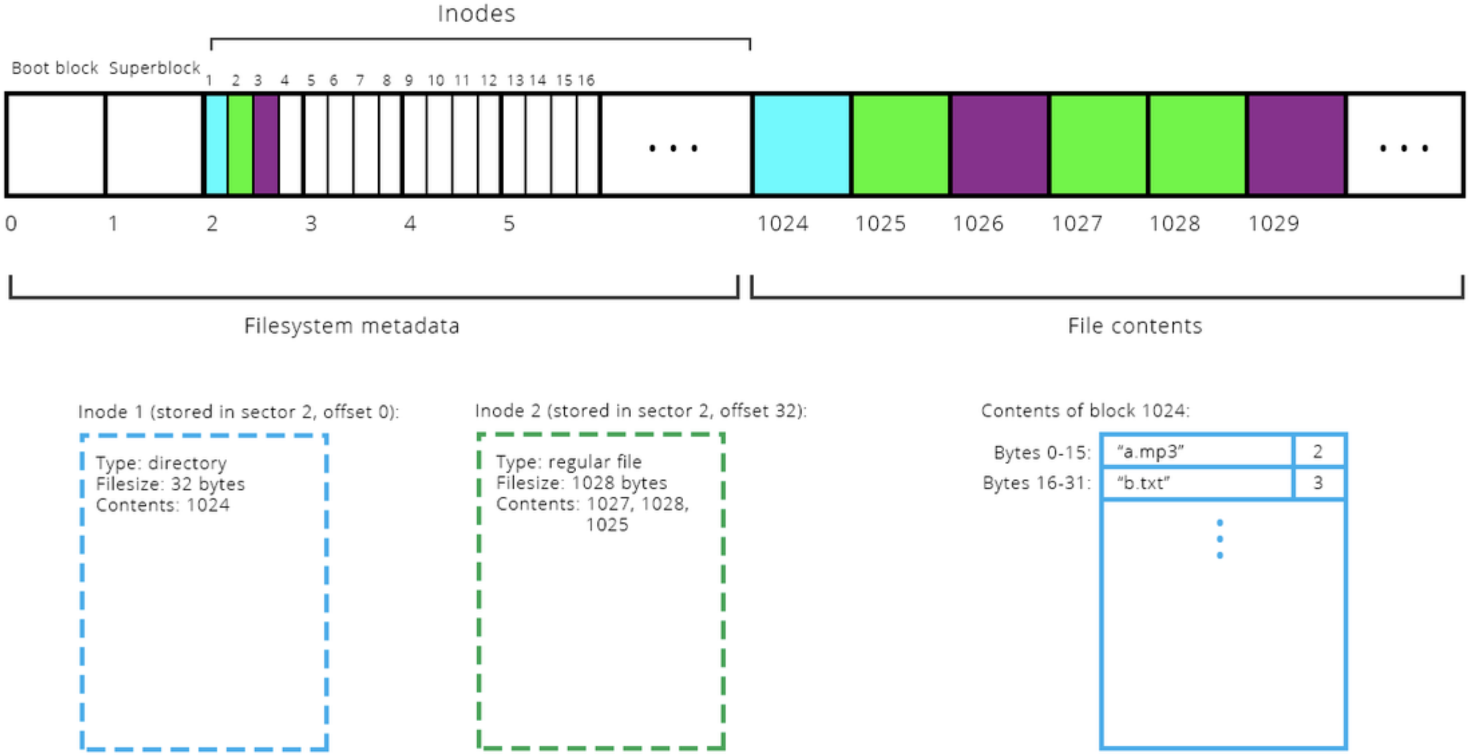
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

Lecture 02: 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.



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

Type: directory
Filesize: 32 bytes
Contents: 1024

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

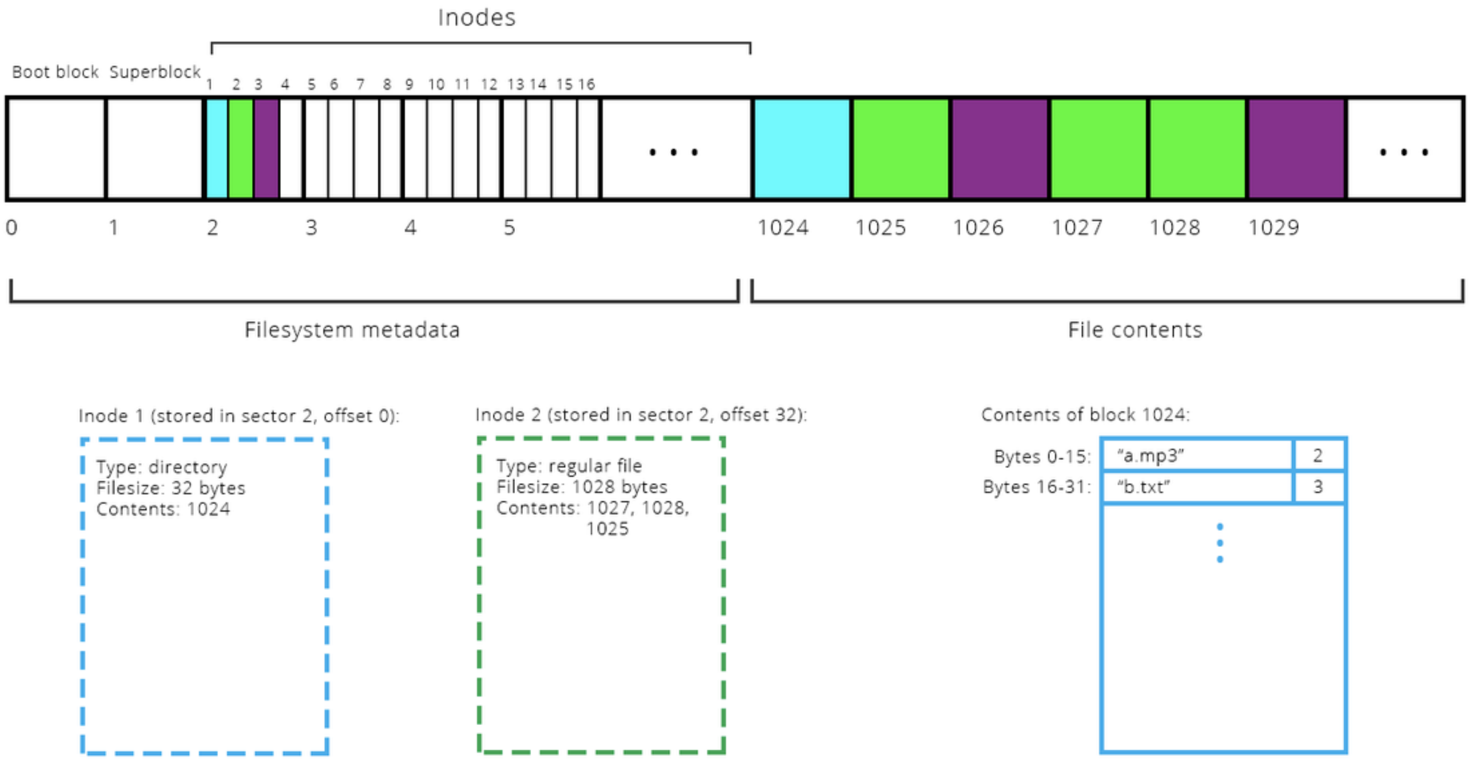
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

Lecture 02: 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, and you 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, 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):

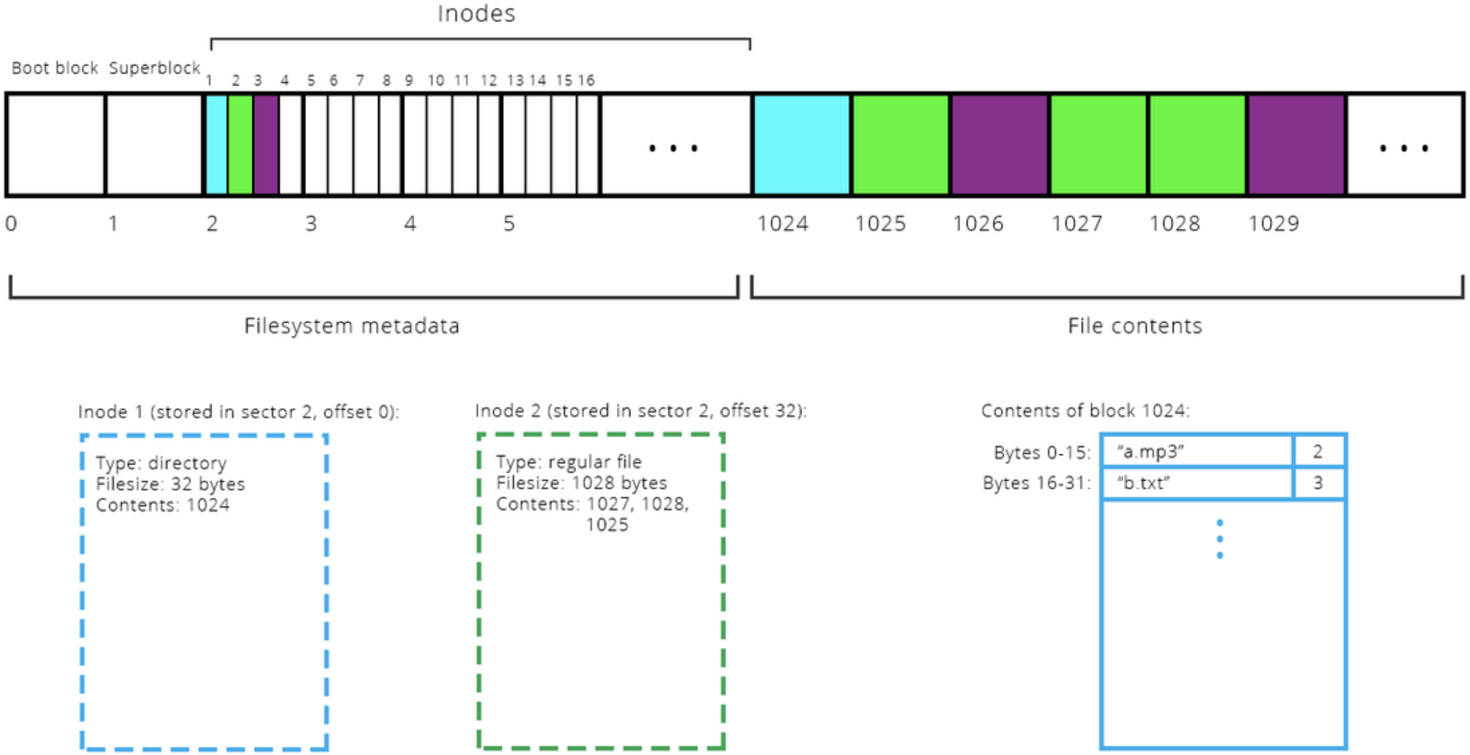
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028,
1025

Contents of block 1024:

Bytes 0-15:	“a.mp3”	2
Bytes 16-31:	“b.txt”	3
	⋮	

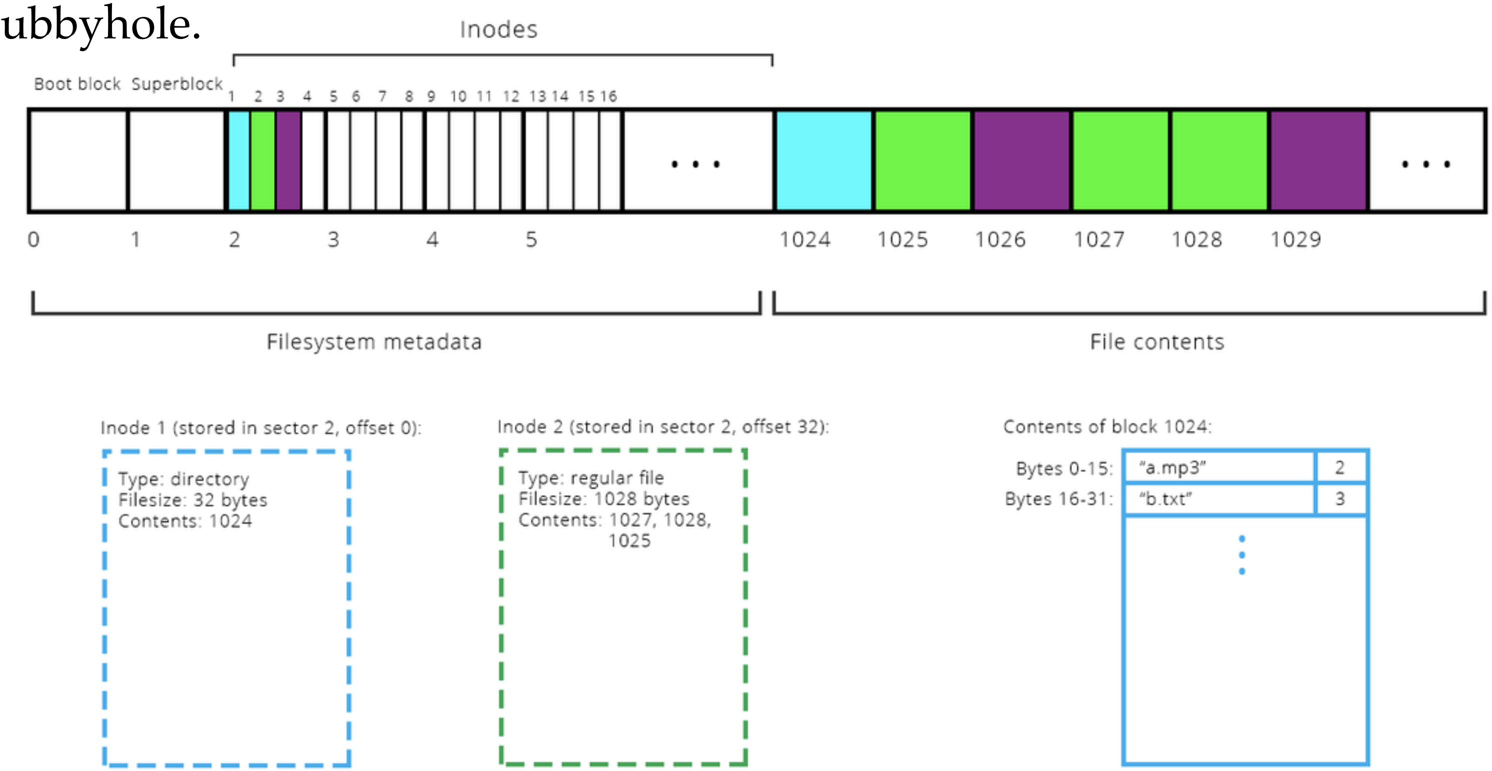
Lecture 02: Filesystem Design

- The inode, continued
  - Look at the contents of inode 2, outlined in green.
    - The file size is 1028 bytes, so three blocks are needed to store everything. The first two are saturated, but the third stores 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.



Lecture 02: Filesystem Design

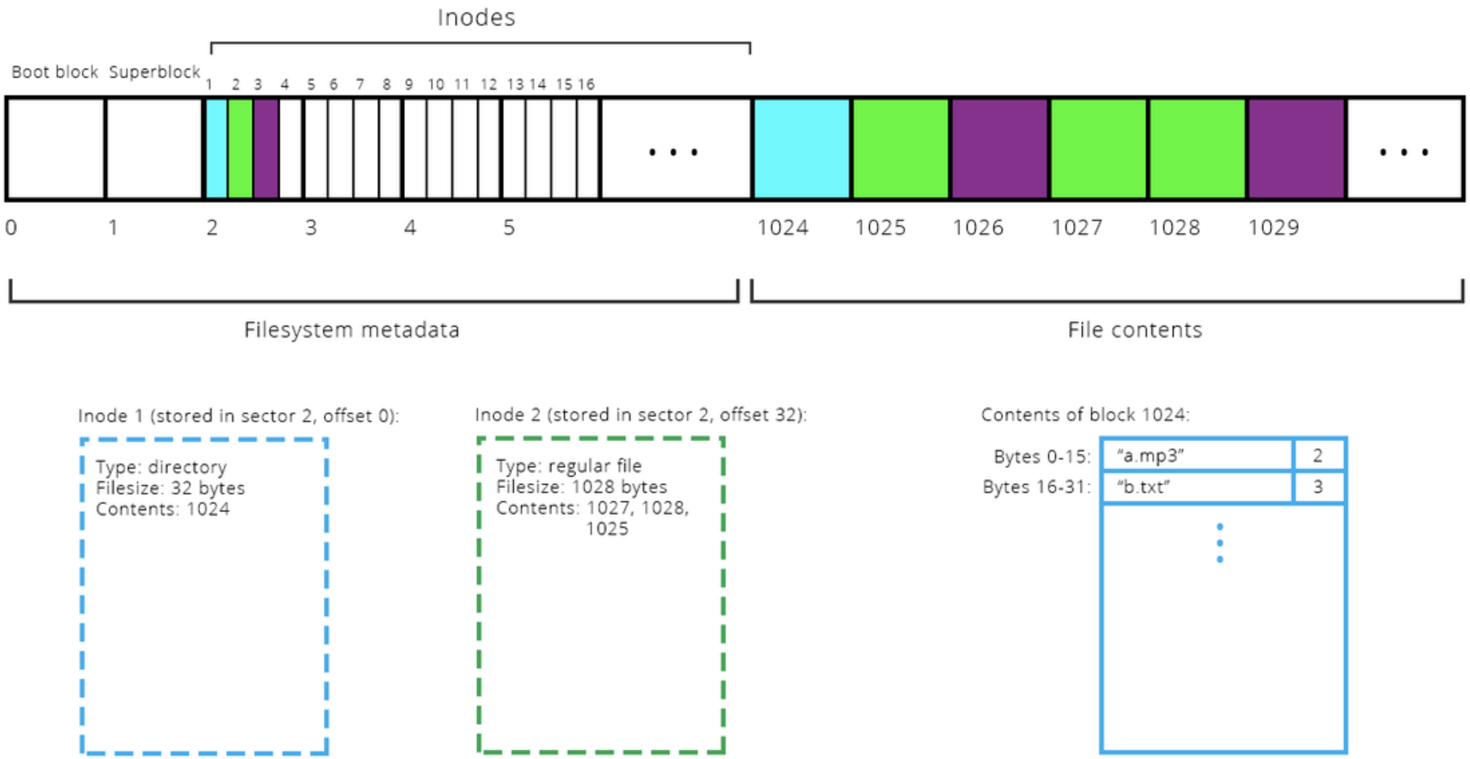
- The inode, continued
  - A file's inodes tell us where we'll find its payload, but the inode itself must reside on the the drive. (Where else could it persist between computer power cycles?)
  - A series of blocks comprise the inode table, which in our diagram stretches from block 2 through block 1023.
  - 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.



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

Lecture 02: Filesystem Design

- The inode, continued
  - 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.
  - If we needed to remember the inode number of every file on our system, we'd be sad.



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

Type: directory
Filesize: 32 bytes
Contents: 1024

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

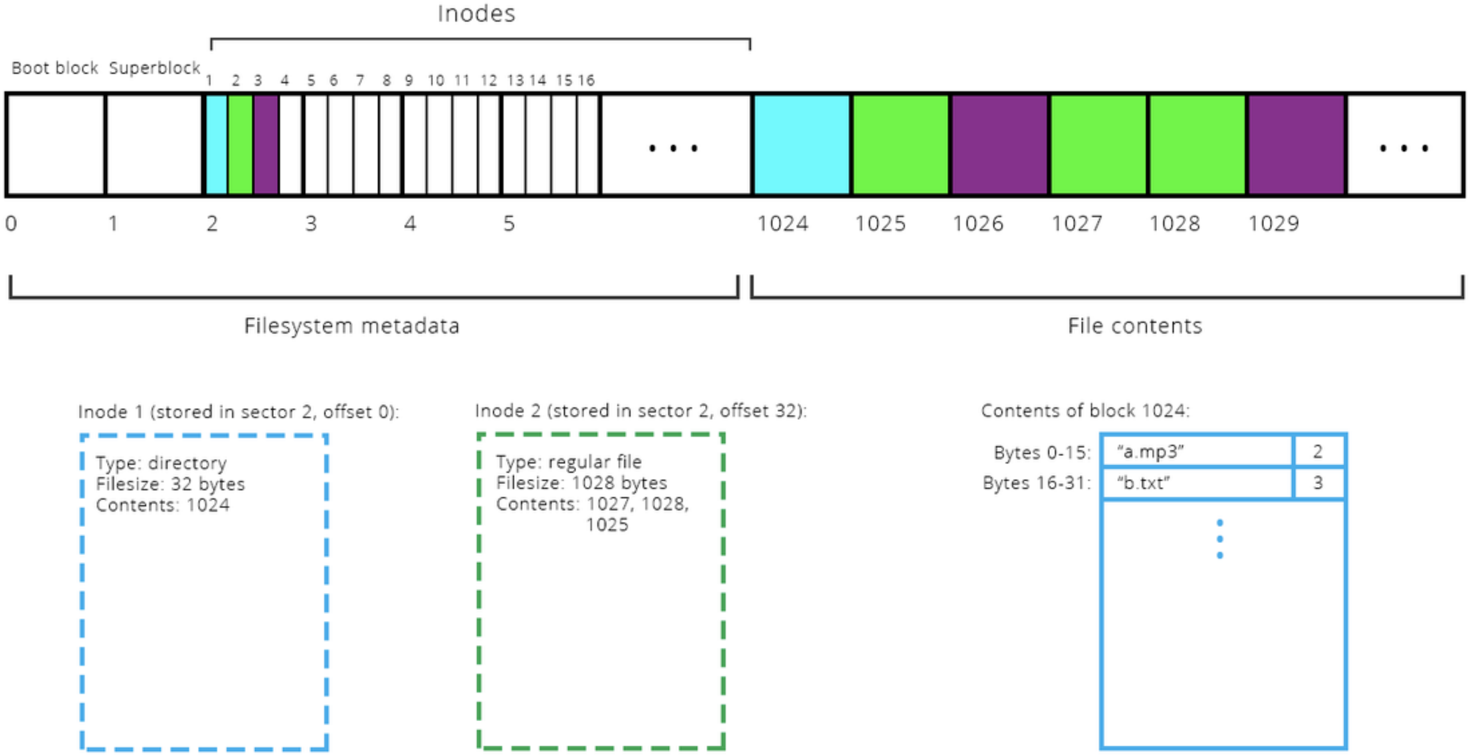
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

Lecture 02: Filesystem Design

- The inode, continued
  - We could wedge a filename field inside each inode. But that a bad idea, because:
    - Inodes are small, but filenames are long. My **`assign1`** 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.
    - 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.



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

Type: directory
Filesize: 32 bytes
Contents: 1024

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

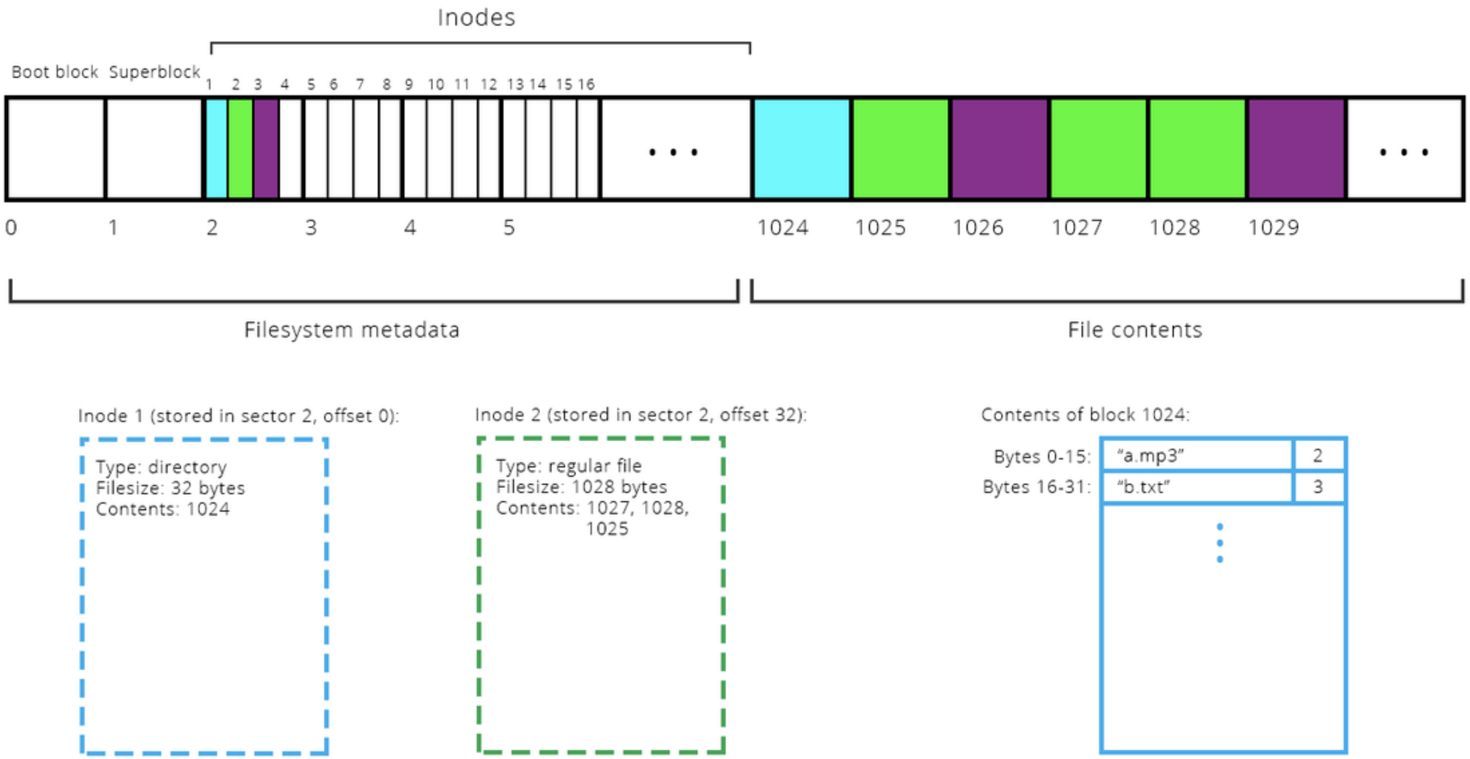
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

Lecture 02: 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, (with the exception that they are marked as directories by the file type field in the inode). The file payload is the accumulation of 16-byte slivers that form a table mapping names 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):

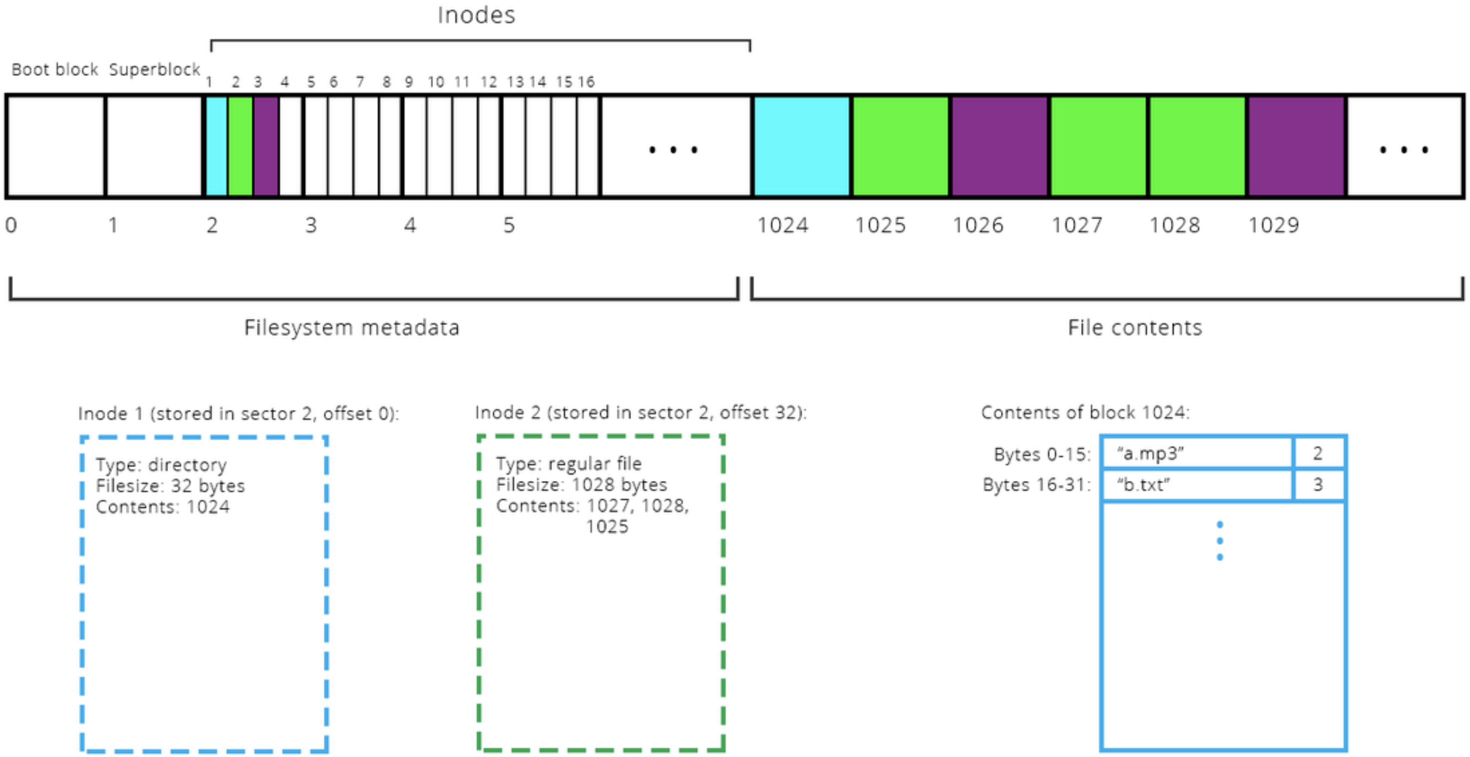
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

Lecture 02: Filesystem Design

- Introducing the directory file type
  - 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 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):

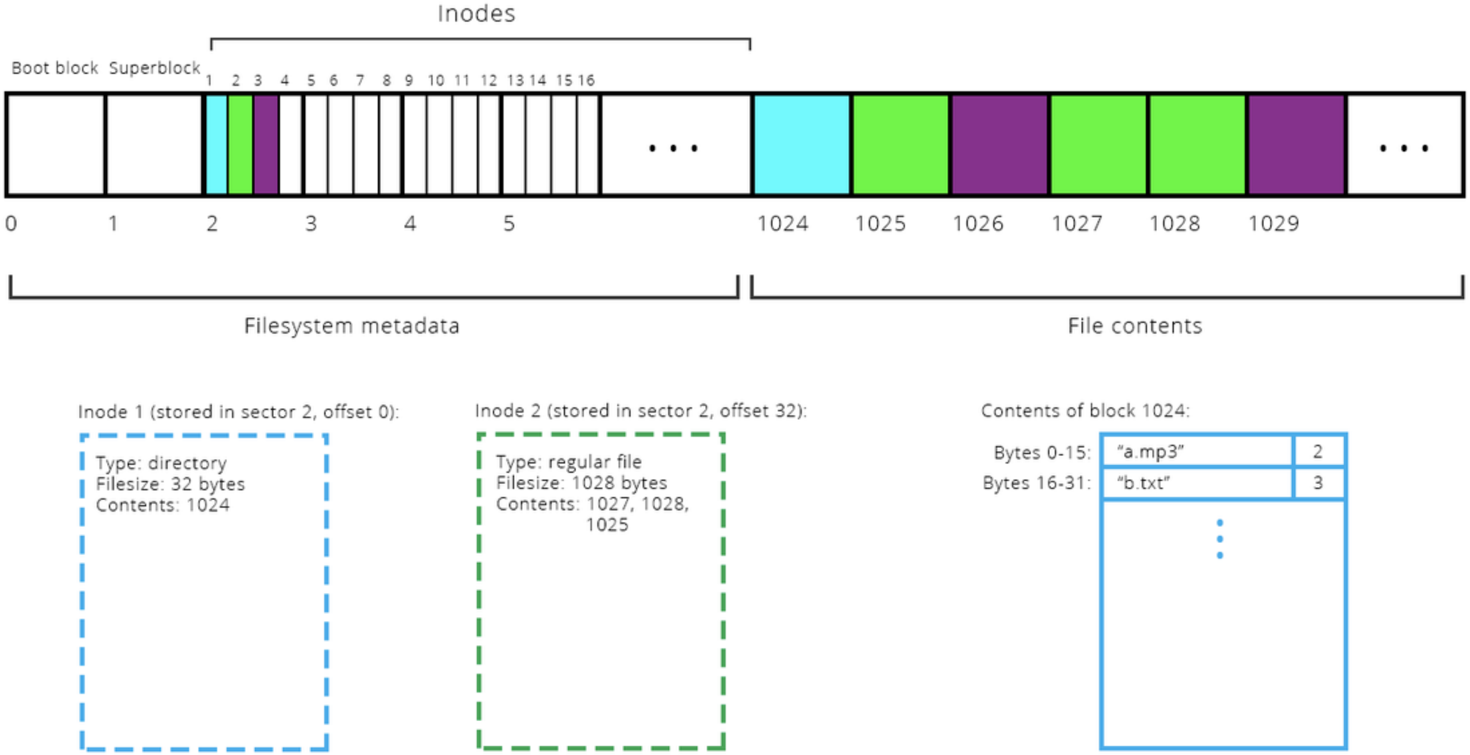
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

Lecture 02: Filesystem Design

- Introducing the directory file type
  - 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 by design is always associated with 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`**.



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

Type: directory
Filesize: 32 bytes
Contents: 1024

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

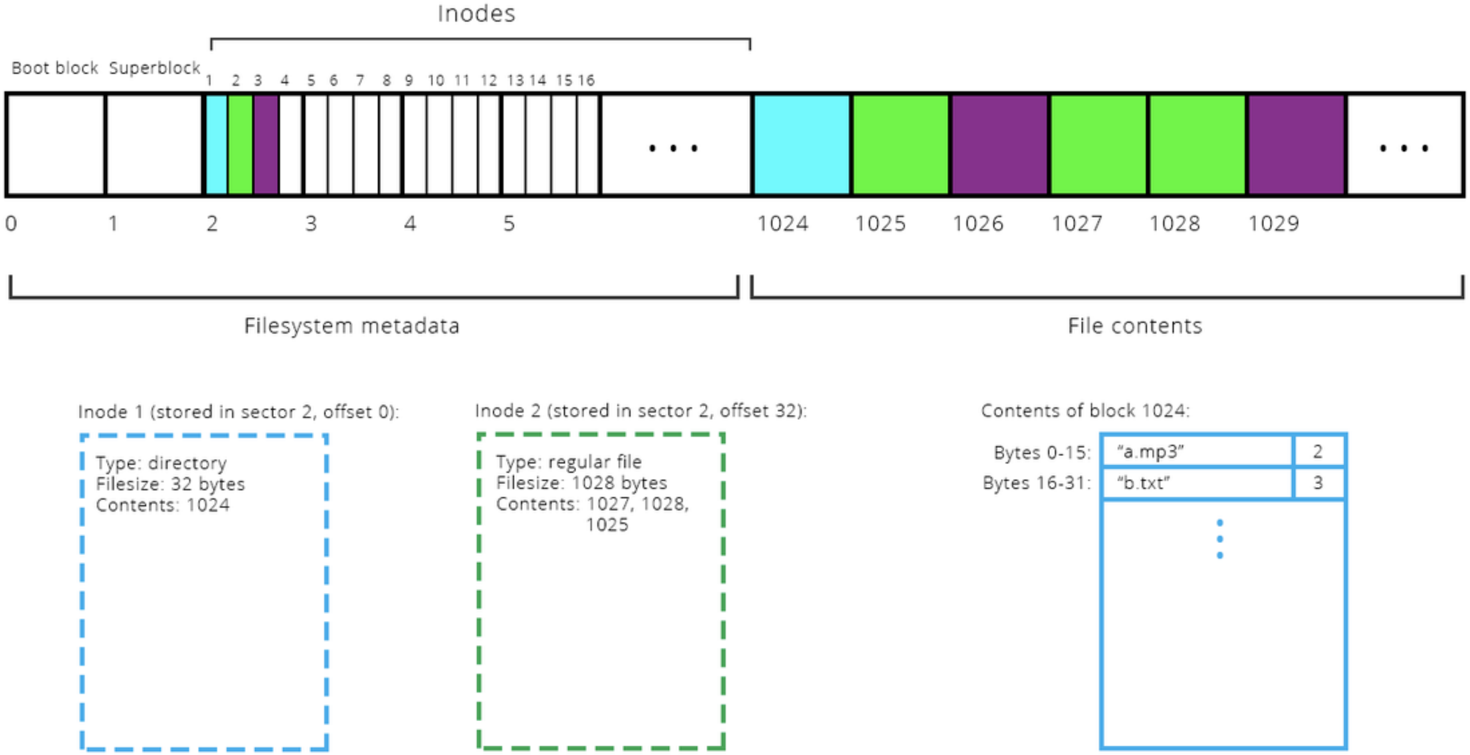
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

Contents of block 1024:

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

Lecture 02: Filesystem Design

- What about large files?
  - In the Unix V6 filesystem (the one that's described as a case study in your textbook), inodes 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.



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 02: 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.
        - 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.
    - How do we know when an inode relies on indirect addressing?
        - Simply examine the file size. If it's "big", then assume indirect addressing.
        - Optionally, include an extra bool (or even a single bitflag) in the inode.

Lecture 02: Filesystem Design

- What about large files? We have a solution!
    - 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.