Filesystems

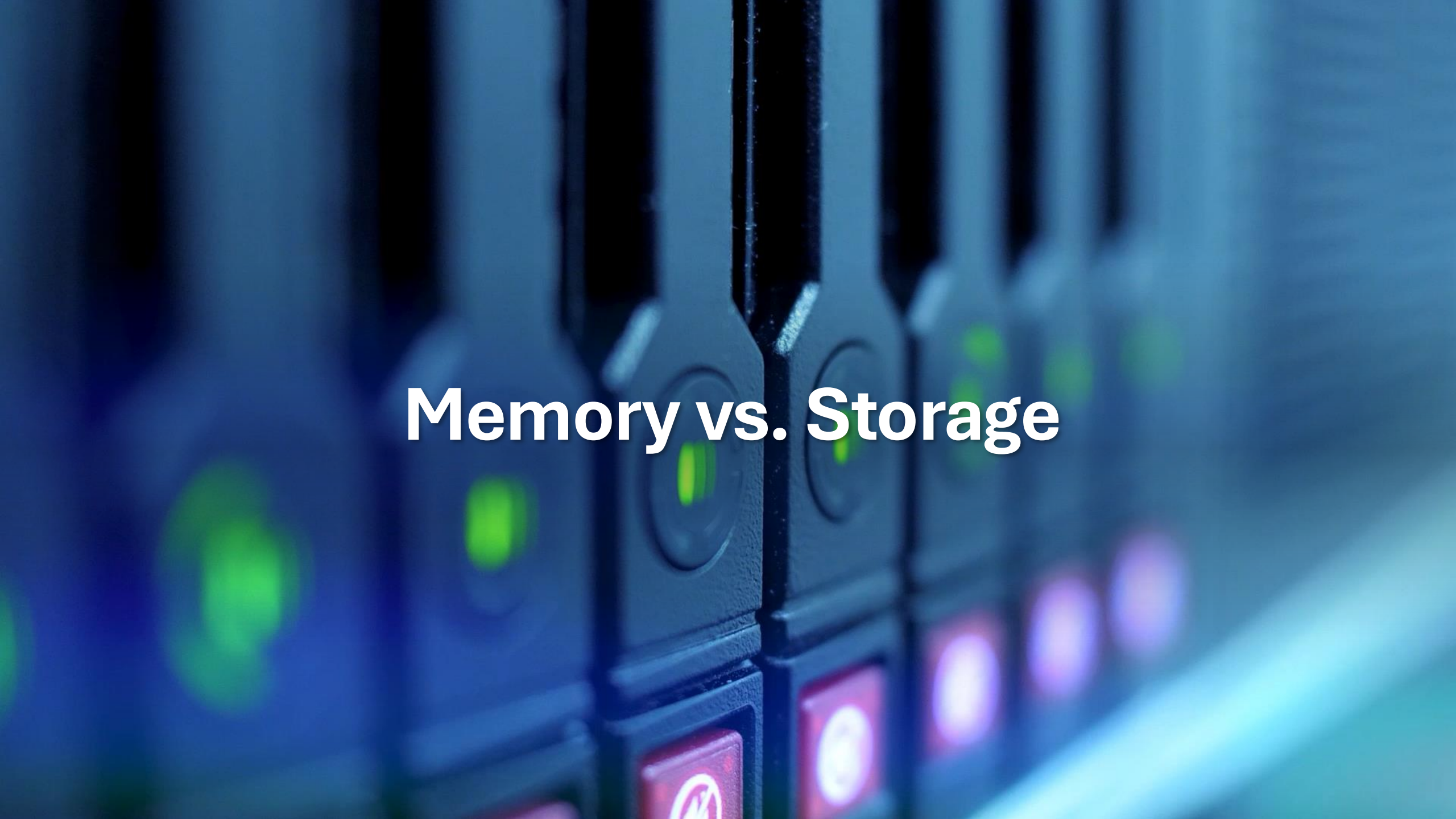
Problem Solving Lab for CS111

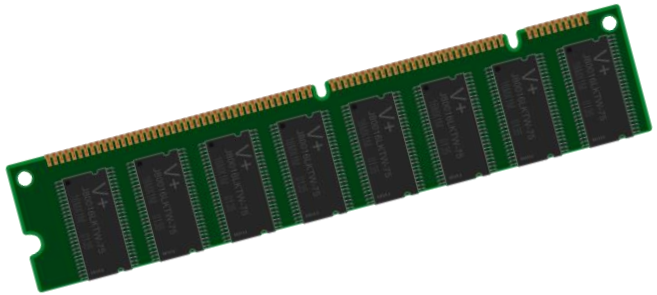
Stanford CS111ACE, Spring 2026

Slides adapted from Trip Master's slides

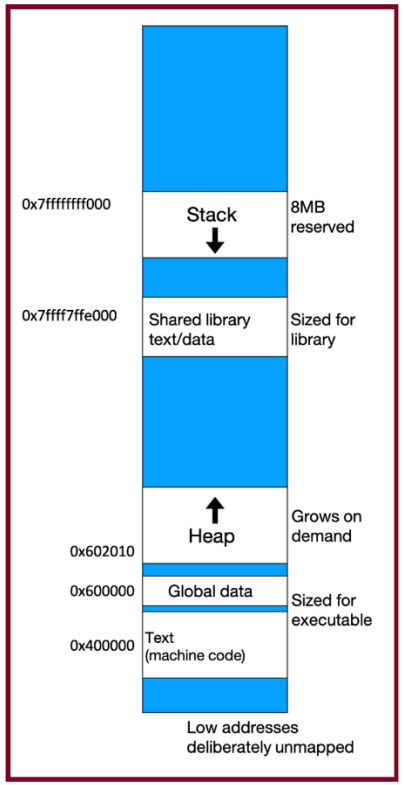
Roadmap

1. Memory vs. Storage
2. The Design of File Systems
3. BSD 4.3 and Indirect Addressing


Memory vs. Storage



- This is what “Memory” looks like
- “Volatile” – if you power off memory is wiped



Flashback to CS107: the stack and the heap are in Memory/RAM

- This is what storage looks like
- Persistent – if you power off what’s stored here is saved!
- **The fundamental question:** How does the operating system manage this?

Srigi
@srigi

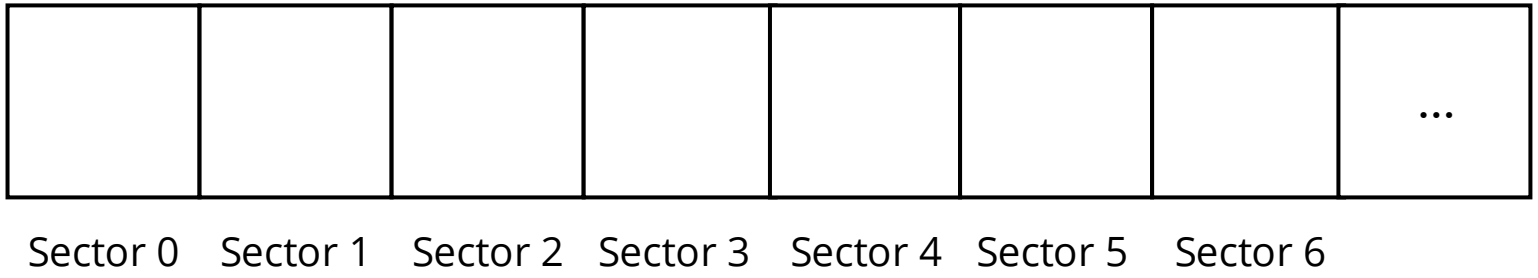
"Latency Numbers Every Programmer Should Know"

It is hard for humans to get the picture until you translate it to "human numbers":

1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years
OS virtualization reboot	4 s	423 years
SCSI command time-out	30 s	3000 years
Hardware virtualization reboot	40 s	4000 years
Physical system reboot	5 m	32 millenia

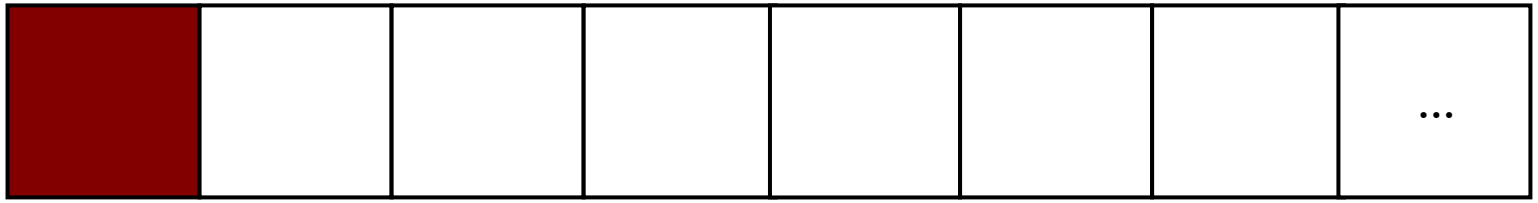
The Design of File Systems

How do we think about storage?



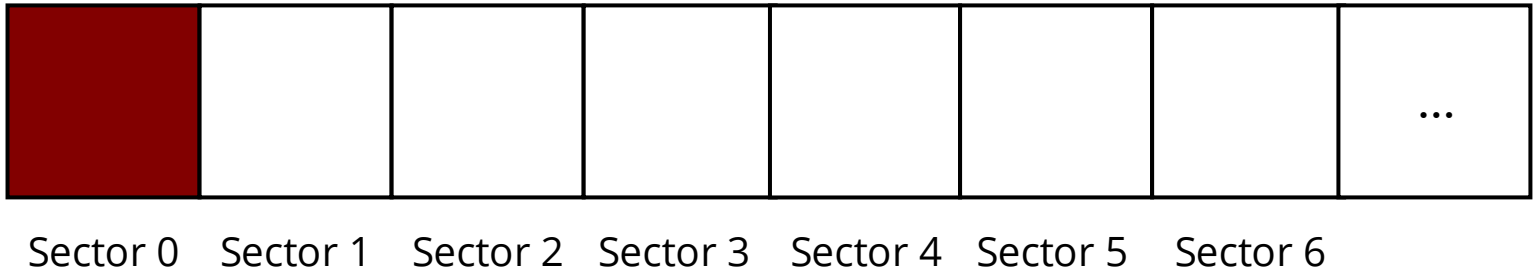
Our storage is divided into *sectors* or *blocks*

How do we think about storage?



A sector is fixed-size, let's say 512 bytes. In practice, they're larger

How do we think about storage?

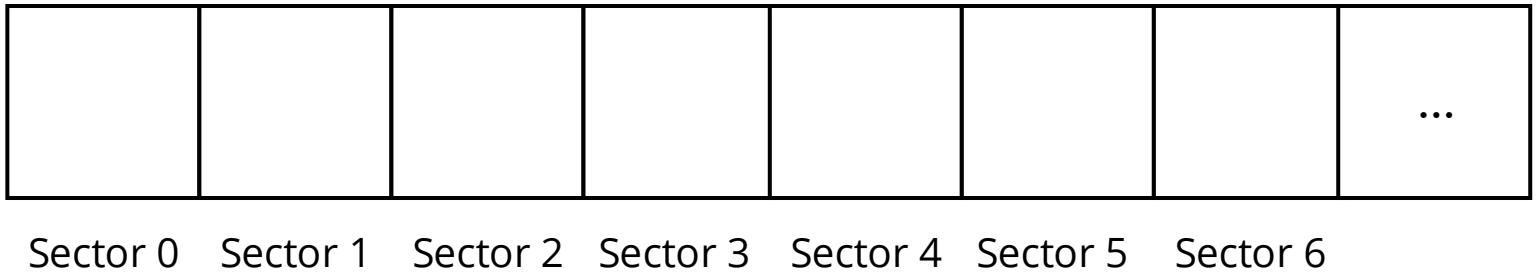


A sector is fixed-size, let's say 512 bytes. In practice, they're larger

We communicate with the disk in units of sectors, that is the granularity that we interact with storage

How do we store files? 🤔

- We need to store *information* (aka metadata) about each file
  - Metadata: size, owner, date created, etc,...
- We need to store the *contents* of each file
  - This is also known as the *payload*



Question: How do we store our metadata and contents given that our storage looks like this?

From lecture

- Remember that we saw four techniques
  - Contiguous allocation
  - Linked Files
  - Windows FAT (File Allocation Table)
  - Multi-level indexes (BSD 4.3 specifically)

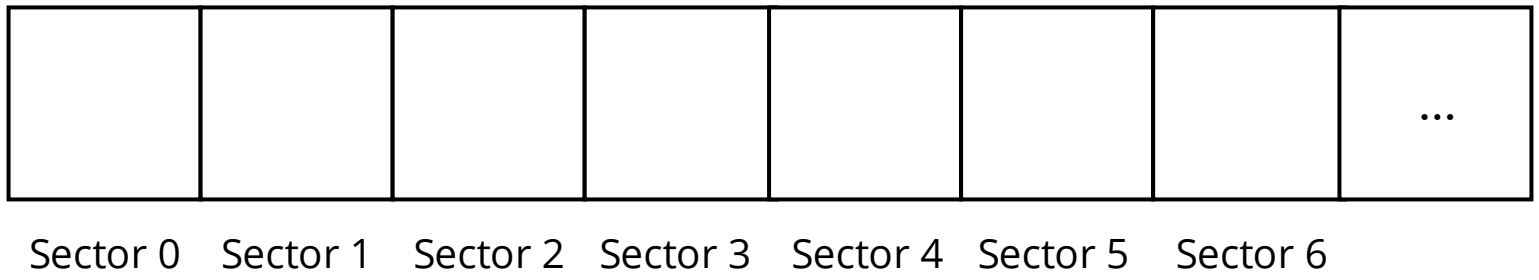
- Each of these have *tradeoffs* which we will review

Strategy 1: Contiguous Allocation

Main idea: Store the file contiguously

Pros:

- Extremely simple

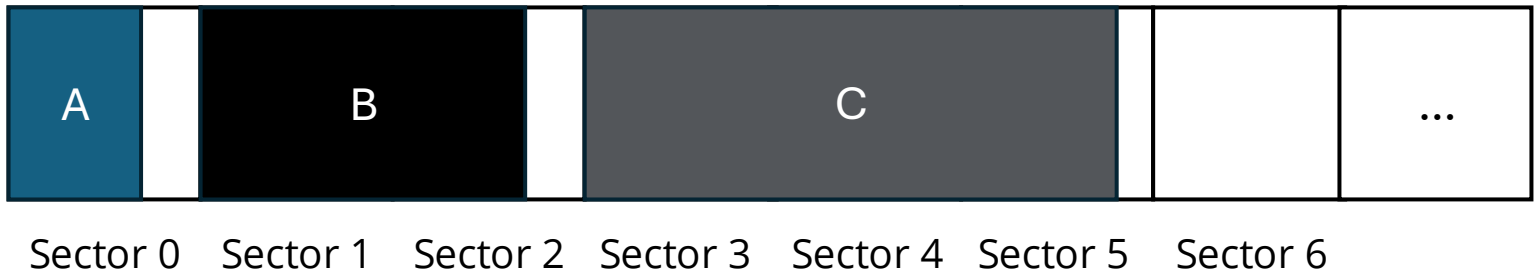


Strategy 1: Contiguous Allocation

Main idea: Store the file contiguously

Pros:

- Extremely simple



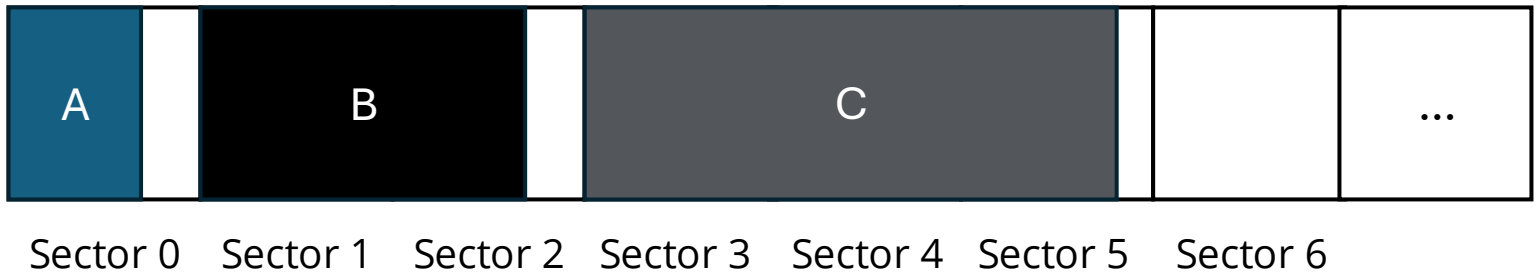
Task A: Let's try to store a 400-byte file

Task B: Let's try to store a 700-byte file

Task C: Let's try to store a 1500-byte file

Strategy 1: Contiguous Allocation

Main idea: Store the file contiguously



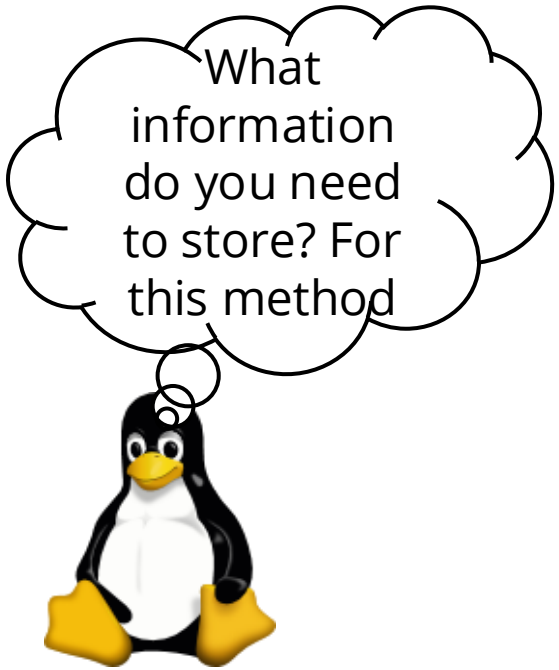
Pros:

- Extremely simple
- Need table of file-to-sector, and store file size

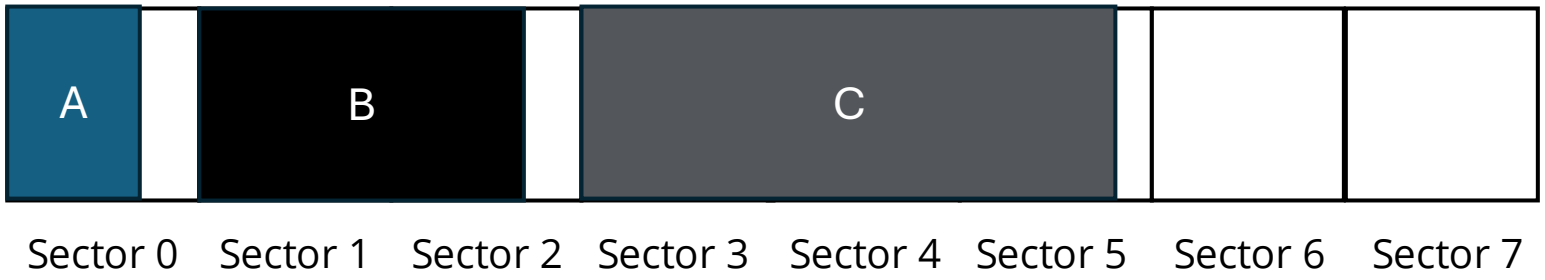
Task A: Let's try to store a 400-byte file

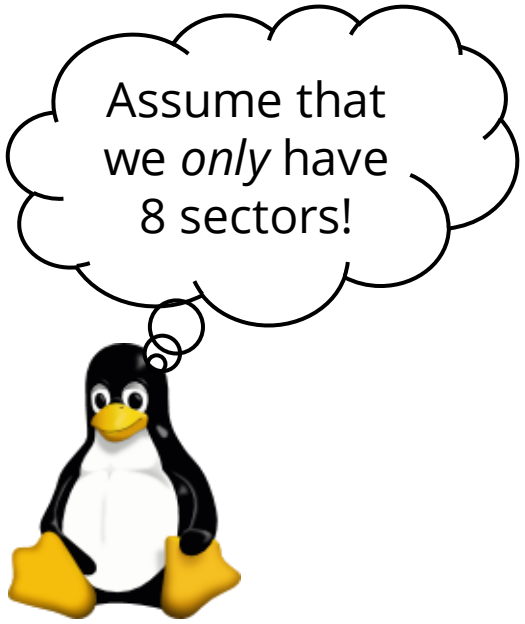
Task B: Let's try to store a 700-byte file

Task C: Let's try to store a 1500-byte file



Contiguous Allocation Con





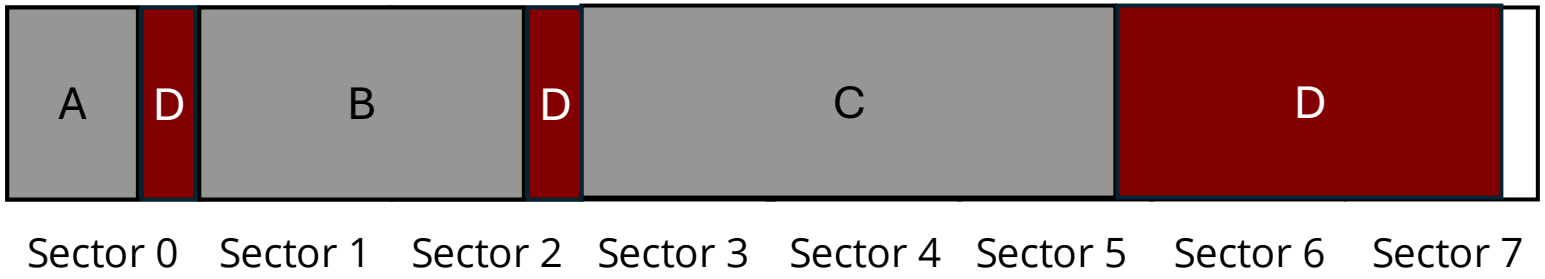
Task A: Let's try to store a 400-byte file

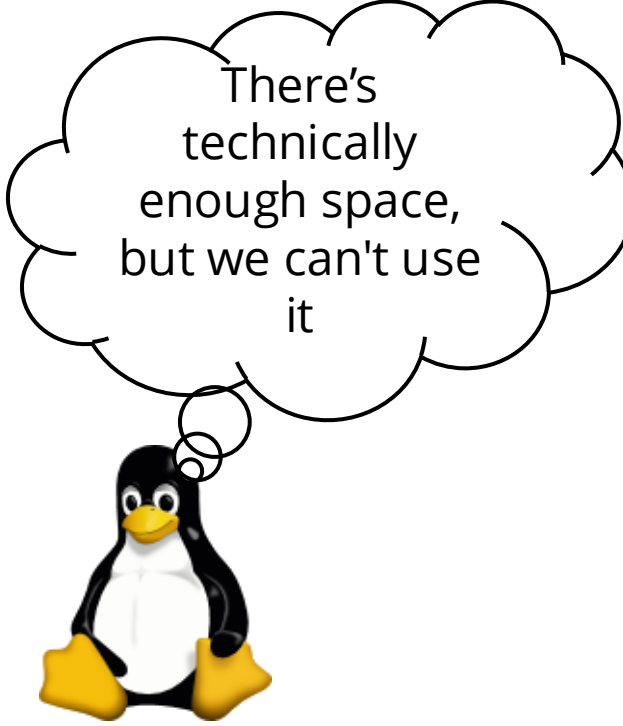
Task B: Let's try to store a 700-byte file

Task C: Let's try to store a 1500-byte file

Task D: Let's try to store a 1400-byte file

Contiguous Allocation Con





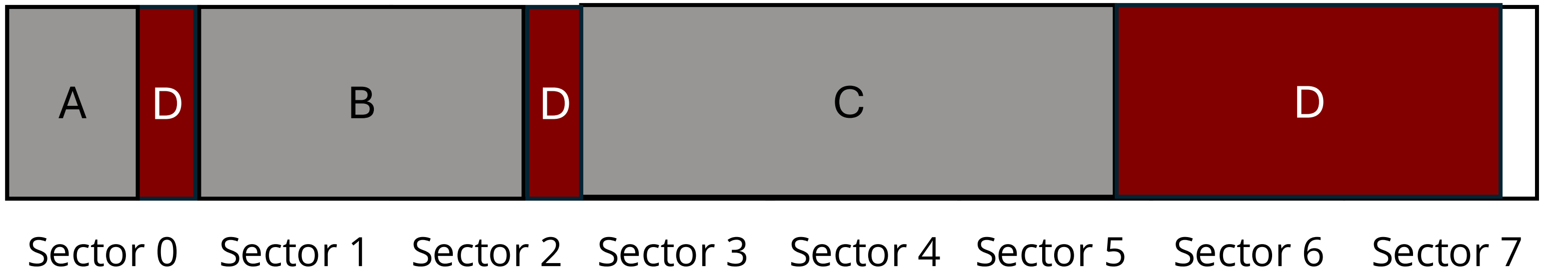
Task A: Let's try to store a 400-byte file

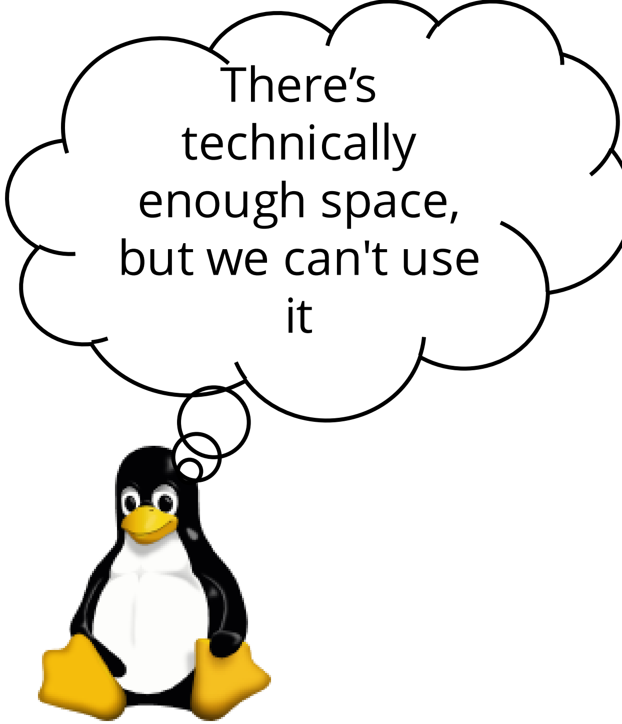
Task B: Let's try to store a 700-byte file

Task C: Let's try to store a 1500-byte file

Task D: Let's try to store a 1400-byte file

Contiguous Allocation Con





This is called *<u>external fragmentation</u>* – we can't allocate because the free space is in already in-use sectors!

Think-Pair-Share (3 min)

- Let's say I have *1000 512-byte sectors* on a disk that uses contiguous allocation (one file per sector). I then *write 1000 1-byte* files to disk.

- How many bytes in my disk are "unused" (meaning, not storing my files)? How many bytes are "free" (meaning, available to be written to)?
  - $\left(1000 \text{ sectors } * 512 \frac{Bytes}{Sector}\right) - \left(1000 \text{ files } * 1 \frac{byte}{file}\right) = 511{,}000 \text{ bytes}$
  - 0 bytes free

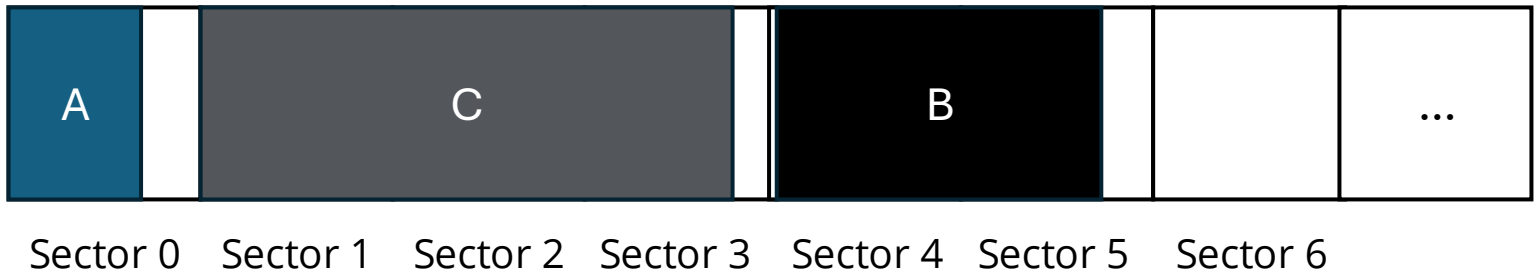
What questions can I answer?

Strategy 2: Linked Files

Main idea: Store the file across various sectors

Pros:

- Less fragmentation



Task A: Let's try to store a 400-byte file

Task C: Let's try to store a 1500-byte file

Task B: Let's try to store a 700-byte file

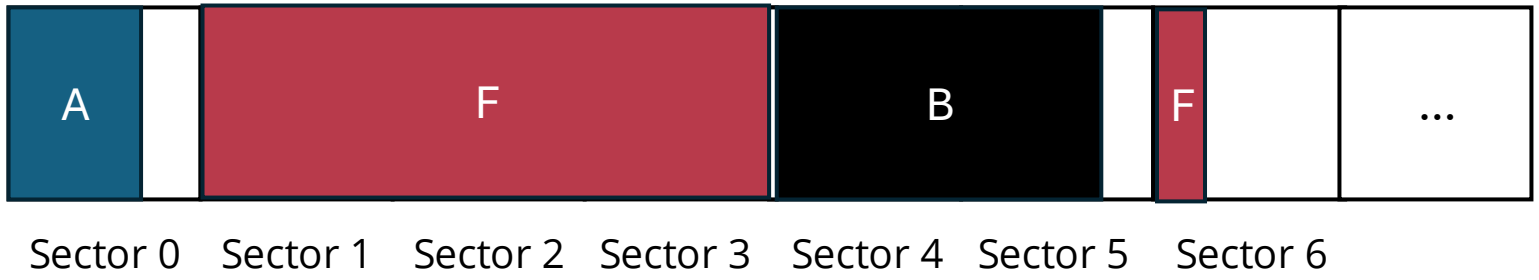
Task D: Delete file C

Strategy 2: Linked Files

Main idea: Store the file across various sectors

Pros:

- Less fragmentation



Task C: Let's try to store a 1500-byte file

Task B: Let's try to store a 700-byte file

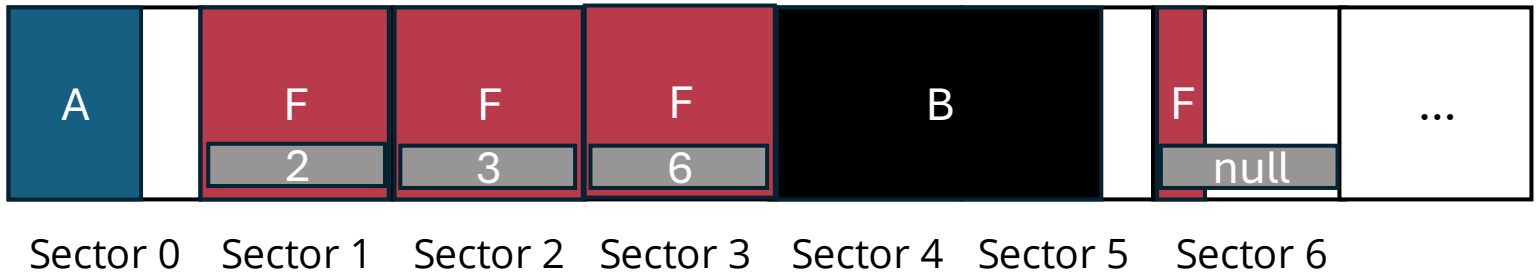
Task E: Delete file C

Task F: Let’s try to store a 1600-byte file

Strategy 2: Nothing is free

Cons:

- Need to store location to next sector in within sector
- More *seeks*
- Can’t easily jump to arbitrary locations in file



Task C: Let's try to store a 1500-byte file

Task B: Let's try to store a 700-byte file

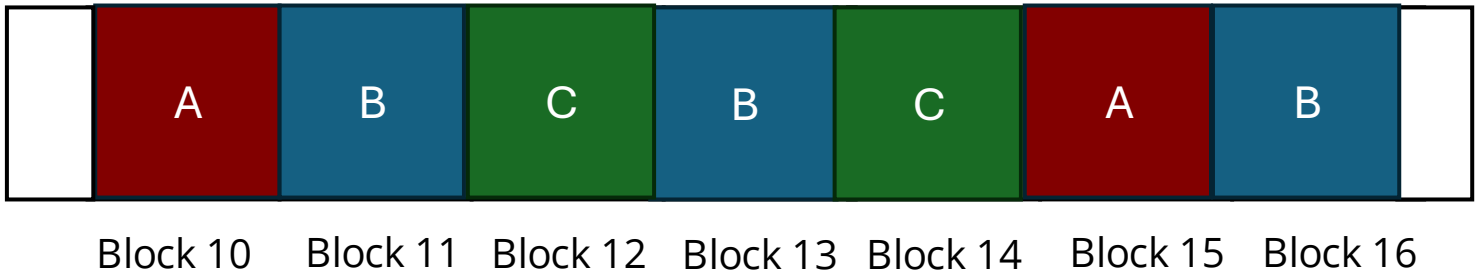
Task E: Delete file C

Task F: Let’s try to store a 1600-byte file

What questions can I answer?

Strategy 3: Windows FAT

Main idea: Store the links in one table in *memory*. Hence the name **F**ile **A**llocation **T**able



9	...
10	15
11	13
12	14
13	16
14	END
15	END
16	END
17	...

Pros:

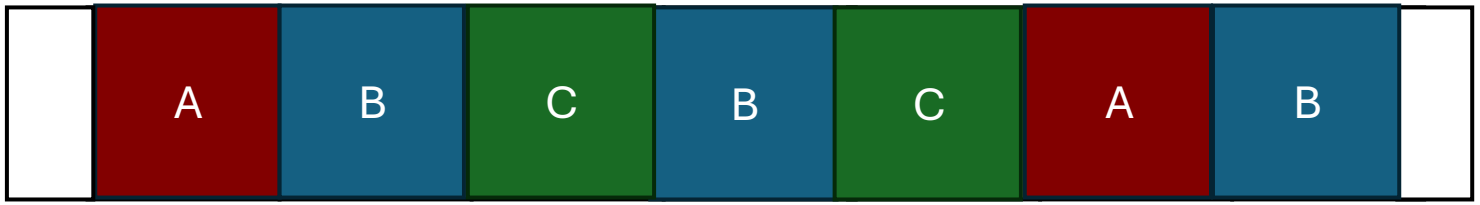
- Blocks now have

Think-Pair-Share: Why is random access faster for Windows FAT compared to the linked files filesystem?

Strategy 3: Persistence

Cons:

- You need to store the table in memory, what if the table is very large?
- You need to store it in persistent memory
- You're still jumping through a table to get the index of your next block!



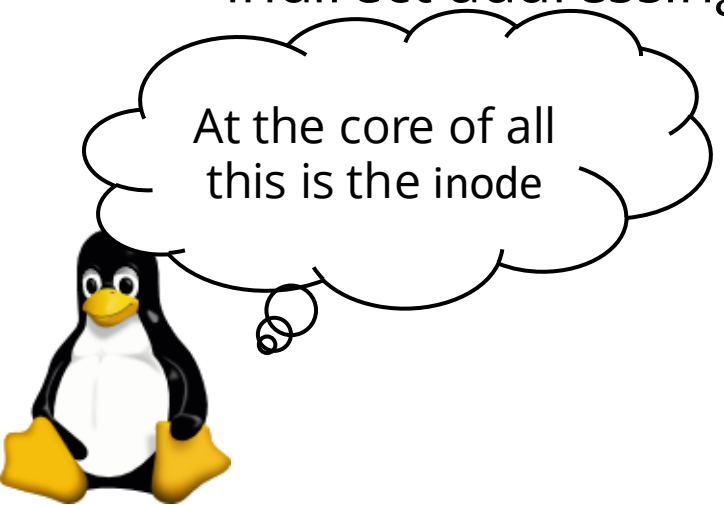
9	...
10	15
11	13
12	14
13	16
14	END
15	END
16	END
17	...

What questions can I answer?

Strategy 4: BSD 4.3 Filesystem (multi-level indexing)

Main idea:

- Store file sectors sequentially somewhere
- Store sector numbers in multiple ”levels”
  - Which necessitates indirect addressing




```
struct inode {
   uint16_t   di_mode;      // type and permissions
      uint16_t   di_nlink;     // # references

   uint32_t   di_uid;       // owner
   uint32_t   di_gid;       // group of owner

   uint64_t   di_size;      // file size in bytes

   uint32_t   di_atime;     // access time (seconds)
   uint32_t   di_atimensec;  // access time (nanoseconds)
   uint32_t   di_mtime;     // modify time (seconds)
   uint32_t   di_mtimensec;  // modify time (nanoseconds)
   uint32_t   di_ctime;     // inode change time (seconds)
   uint32_t   di_ctimensec;  // inode change time (nanoseconds)

   uint32_t   di_db[12];    // direct block pointers
   uint32_t   di_ib[3];     // indirect block pointers (single, double, triple)

   uint32_t   di_flags;     // status flags (immutable, append-only, etc.)
   uint32_t   di_blocks;    // # blocks actually allocated
   uint32_t   di_gen;       // generation number (for NFS)
   uint32_t   di_spare[4];   // reserved for future use
};
```

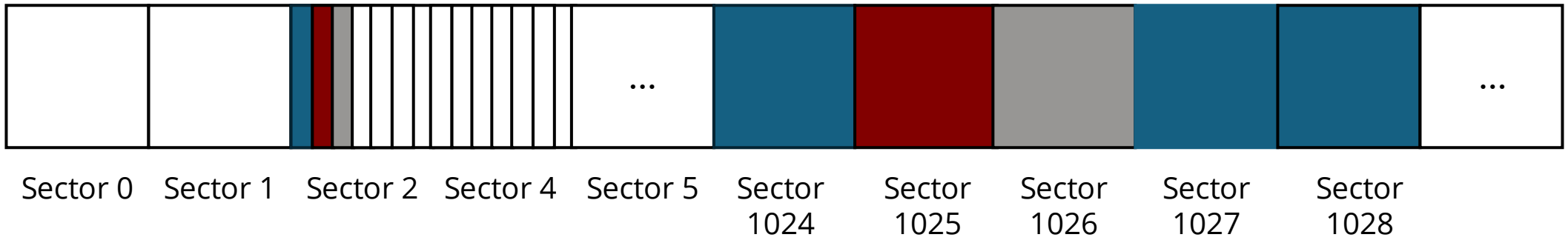
Check this out if you *really* want to

The inode

```
struct inode {
   uint16_t   di_mode;      // type and permissions
      uint16_t   di_nlink;     // # references

   uint32_t   di_uid;       // owner
   uint32_t   di_gid;       // group of owner

   uint64_t   di_size;      // file size in bytes

   uint32_t   di_atime;     // access time (seconds)
   uint32_t   di_atimensec;  // access time (nanoseconds)
   uint32_t   di_mtime;     // modify time (seconds)
   uint32_t   di_mtimensec;  // modify time (nanoseconds)
   uint32_t   di_ctime;     // inode change time (seconds)
   uint32_t   di_ctimensec;  // inode change time (nanoseconds)

   uint32_t   di_db[12];    // direct block pointers
   uint32_t   di_ib[3];     // indirect block pointers (single, double, triple)

   uint32_t   di_flags;     // status flags (immutable, append-only, etc.)
   uint32_t   di_blocks;    // # blocks actually allocated
   uint32_t   di_gen;       // generation number (for NFS)
   uint32_t   di_spare[4];   // reserved for future use
};
```

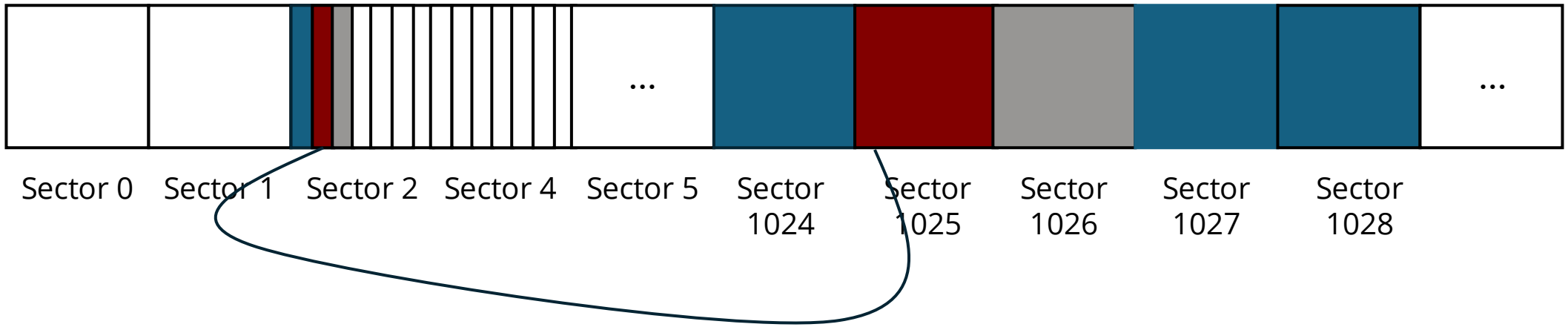
- The inode is at the core of the BSD 4.3 filesystem
- No filename – why?
  - duplication

The BSD 4.3 Filesystem

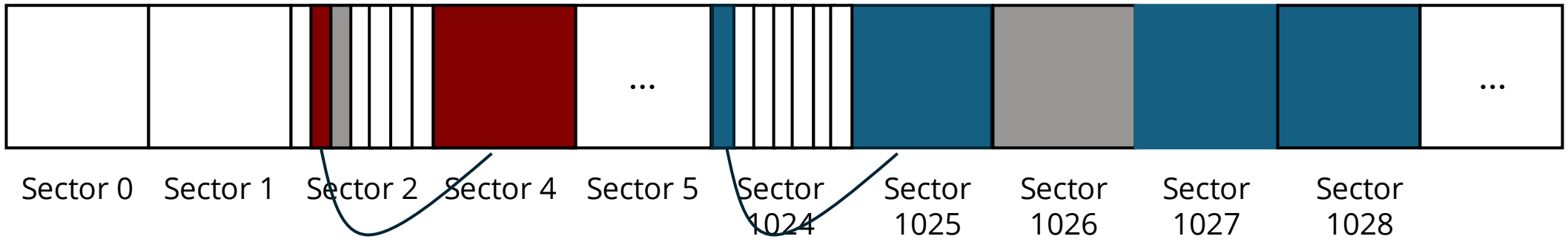


The BSD 4.3 Filesystem

This inode *contains information about where the payload for this file lives in storage*



The BSD 4.3 Filesystem

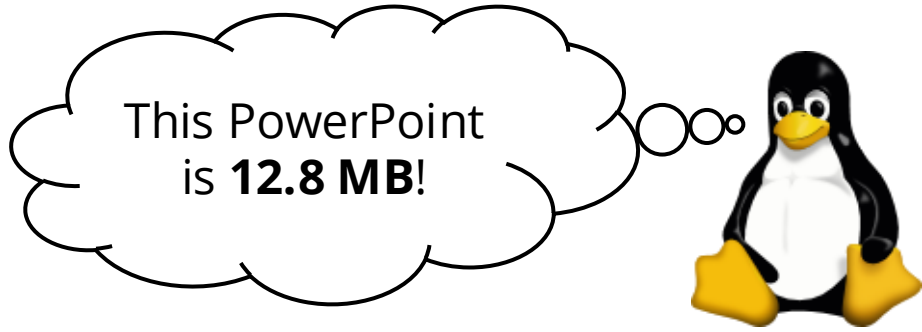


Eventually BSD FS changed so that the inodes are closer to where the data lives

Question: Max file size

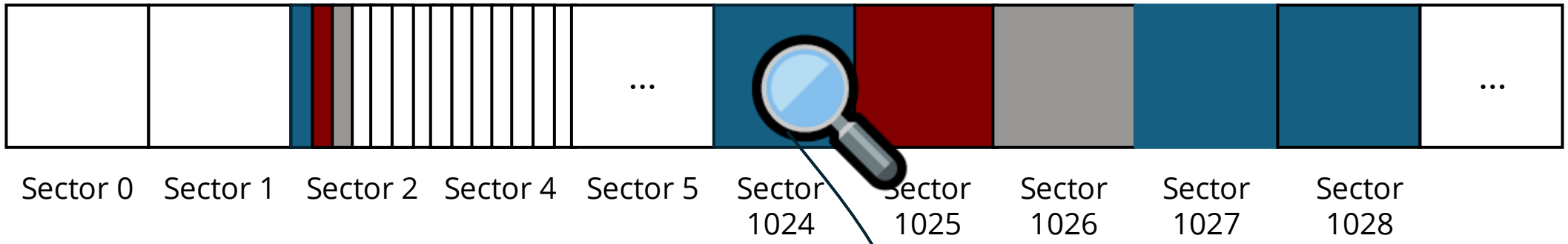
```
struct inode {
   uint16_t   di_mode;      // type and permissions
      uint16_t   di_nlink;     // # references

   uint32_t   di_uid;       // owner
   uint32_t   di_gid;       // group of owner

   uint64_t   di_size;      // file size in bytes

   uint32_t   di_atime;     // access time (seconds)
   uint32_t   di_atimensec;  // access time (nanoseconds)
   uint32_t   di_mtime;     // modify time (seconds)
   uint32_t   di_mtimensec;  // modify time (nanoseconds)
   uint32_t   di_ctime;     // inode change time (seconds)
   uint32_t   di_ctimensec;  // inode change time (nanoseconds)

   uint32_t   di_db[12];    // direct block pointers
   uint32_t   di_ib[3];     // indirect block pointers (single, double, triple)

   uint32_t   di_flags;     // status flags (immutable, append-only, etc.)
   uint32_t   di_blocks;    // # blocks actually allocated
   uint32_t   di_gen;       // generation number (for NFS)
   uint32_t   di_spare[4];   // reserved for future use
};
```

- i_addr[] stores all the block numbers that make up a file!
- What is the max file size for a file if we use i_addr to only store *payload blocks*?
  - 48 Kb



What questions can I answer?

Consider 2 types of blocks!



Payload blocks:

89 50 4E 47
0D 0A 1A 0A
00 00 00 0D
49 48 44 52

i_addr = [1024, 1027, 1028,..]

Consider 2 types of blocks!

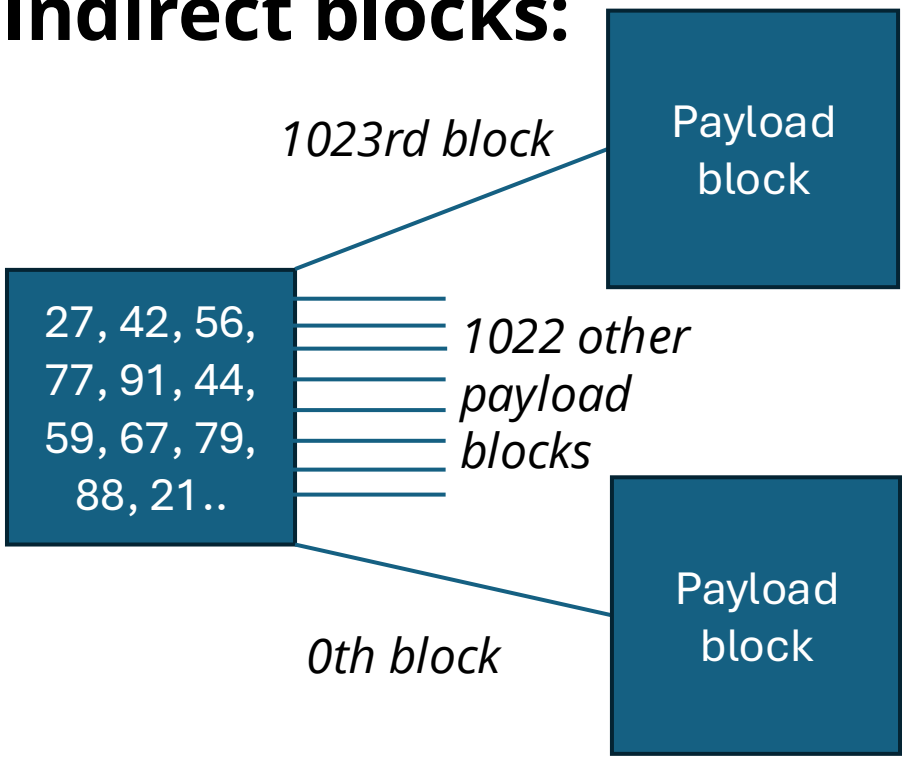
Indirect blocks:

27, 42, 56, 77, 91, 44, 59, 67, 79, 88, 21..

- Each block number is 4-bytes
- 1024 block numbers fit into each sector, if it’s an indirect block

Consider 2 types of blocks!

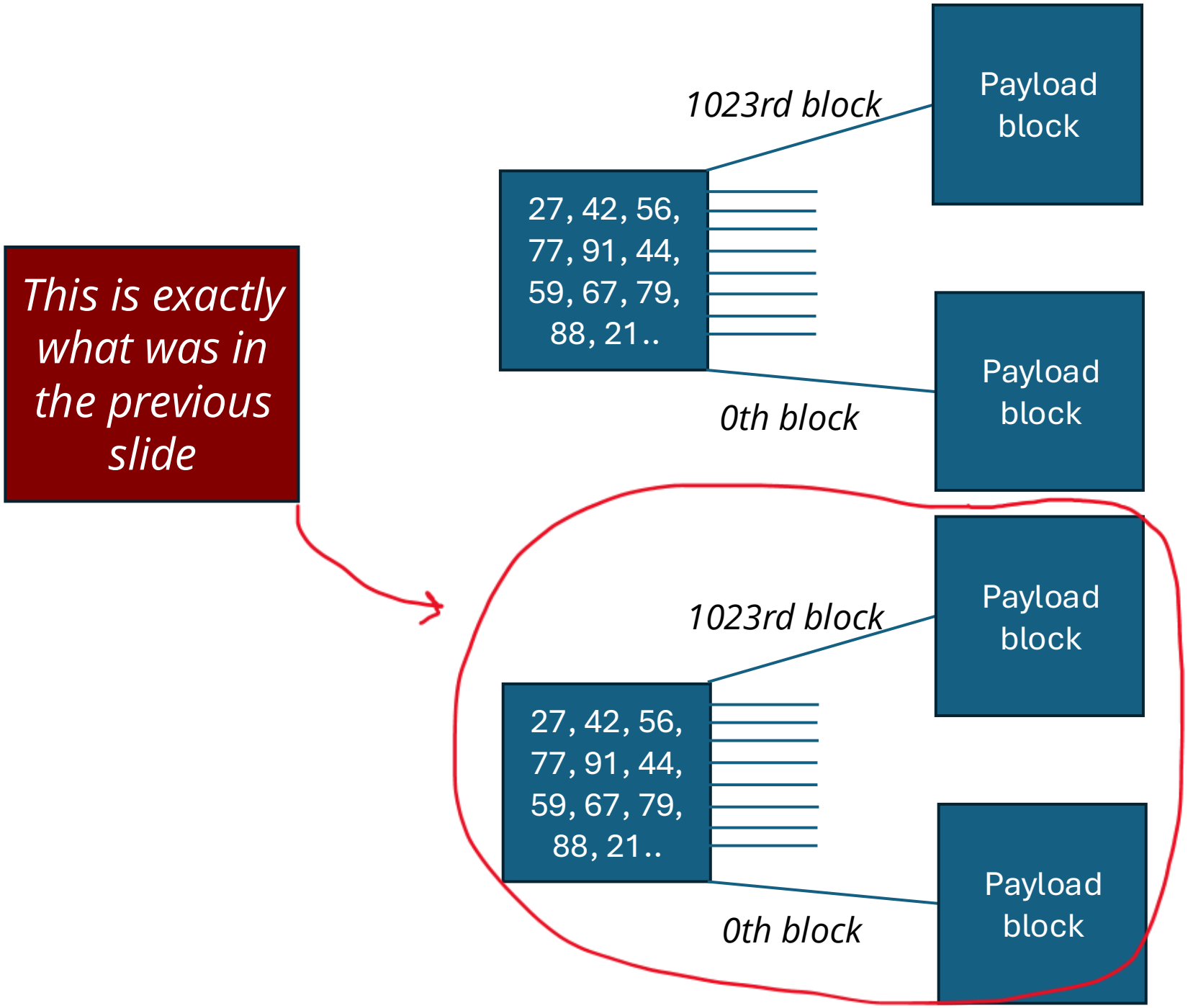
Indirect blocks:

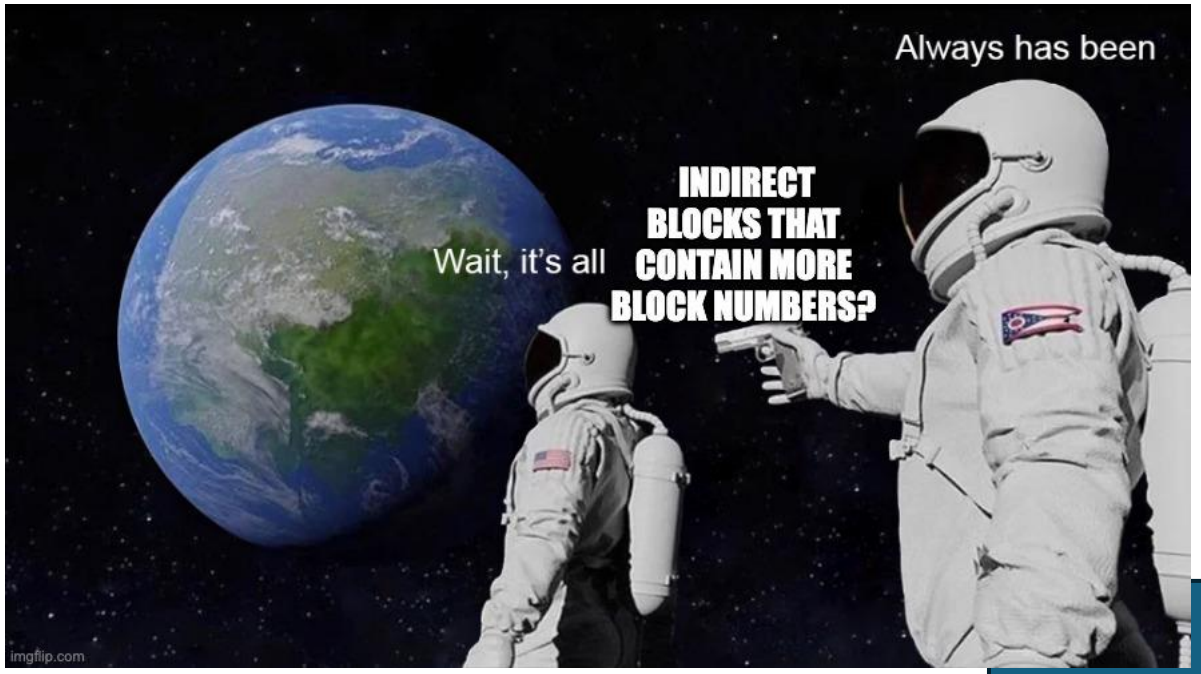


- Each block number is 4-bytes
- 1024 block numbers fit into each sector, if it's an indirect block

What questions can I answer?

Let’s take it one step further


This is exactly what was in the previous slide
1023rd block
Payload block
27, 42, 56, 77, 91, 44, 59, 67, 79, 88, 21..
Payload block
0th block
1023rd block
Payload block
27, 42, 56, 77, 91, 44, 59, 67, 79, 88, 21..
Payload block
0th block



- We can have a block that *<u>contains</u>* block numbers to *<u>singly-indirect blocks</u>*