Crash Recovery

Problem Solving Lab for CS111

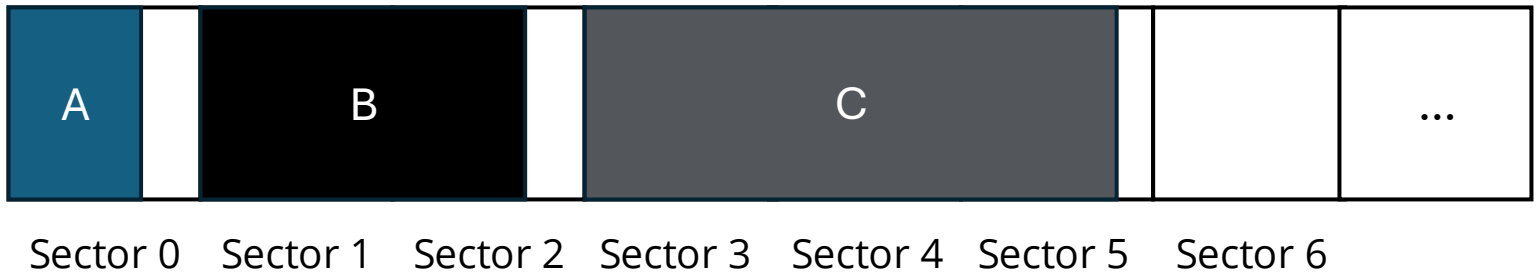
Stanford CS111ACE, Spring 2026

Slides adapted from Trip Master's slides

A Quick Review

Strategy 1: Contiguous Allocation

Main idea: Store the file contiguously



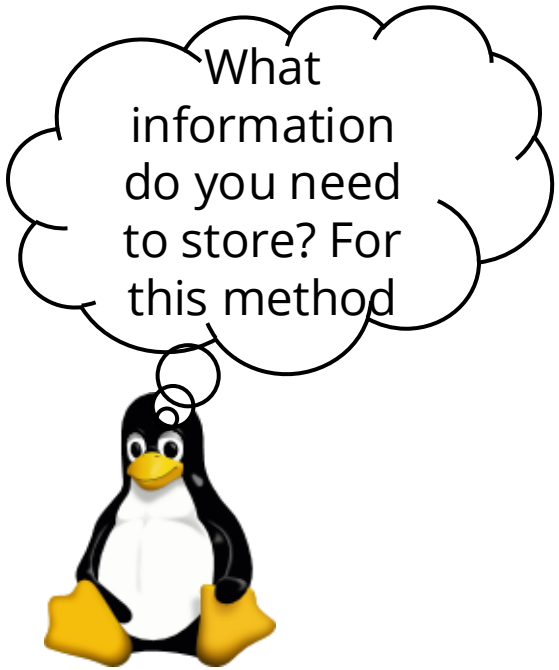
Pros:

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

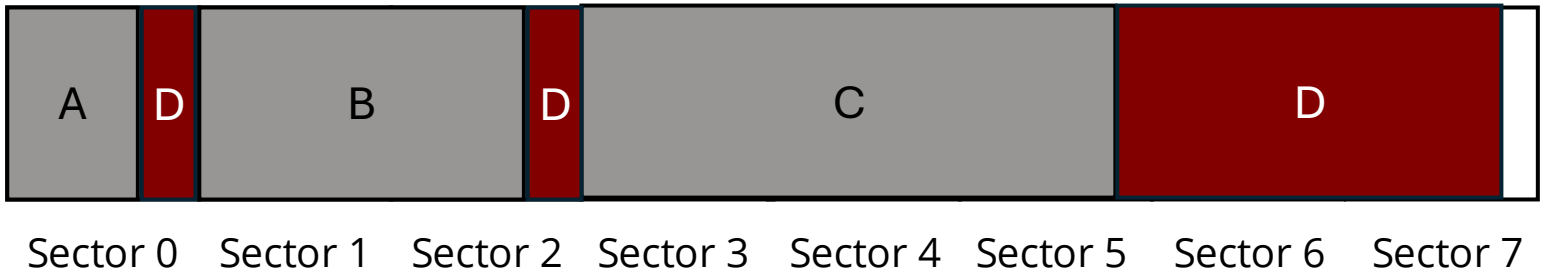
Task A: Let's try to store a 400-byte file

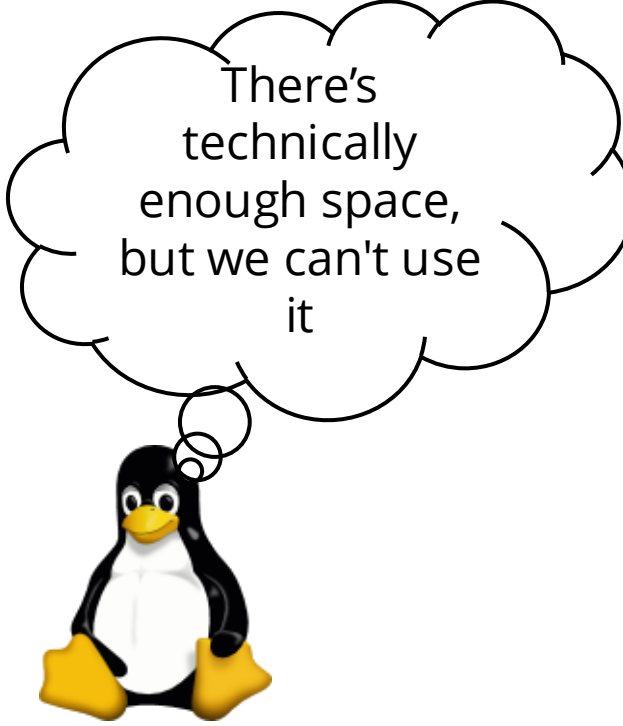
Task B: Let's try to store a 700-byte file

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

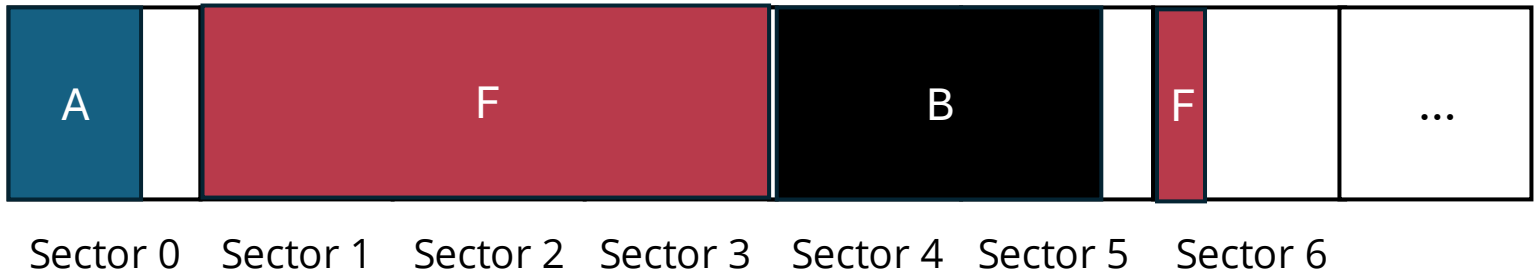
Task D: Let's try to store a 1400-byte file

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

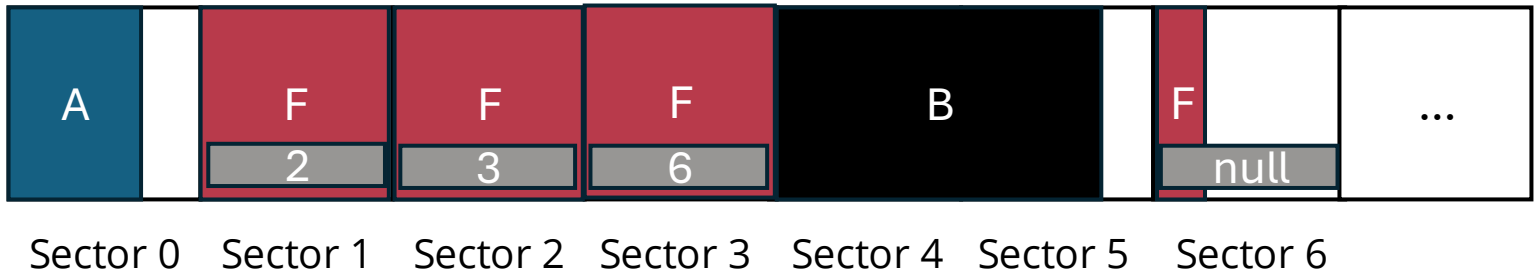
Task E: Delete file C

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

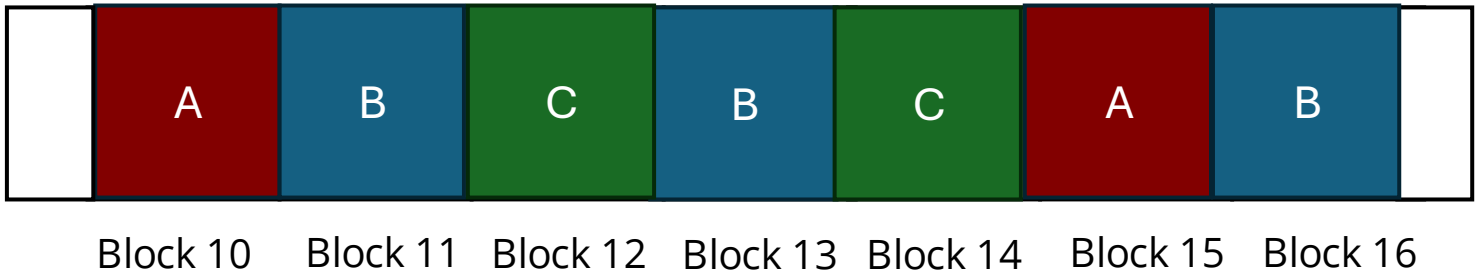
Task B: Let's try to store a 700-byte file

Task E: Delete file C

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

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:

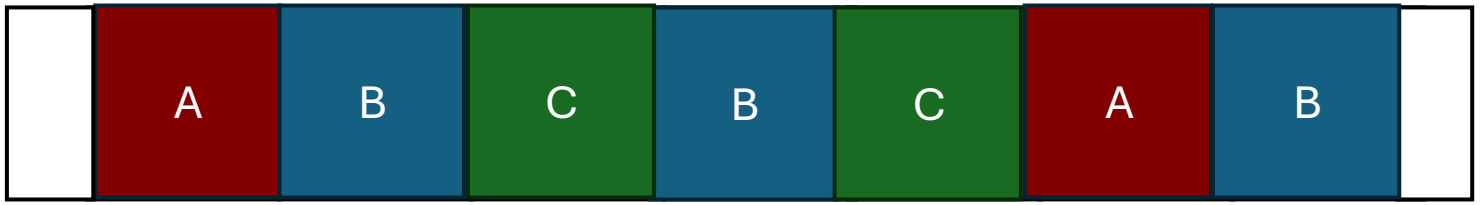
- Blocks now have

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

Strategy 3: Persistence

Cons:

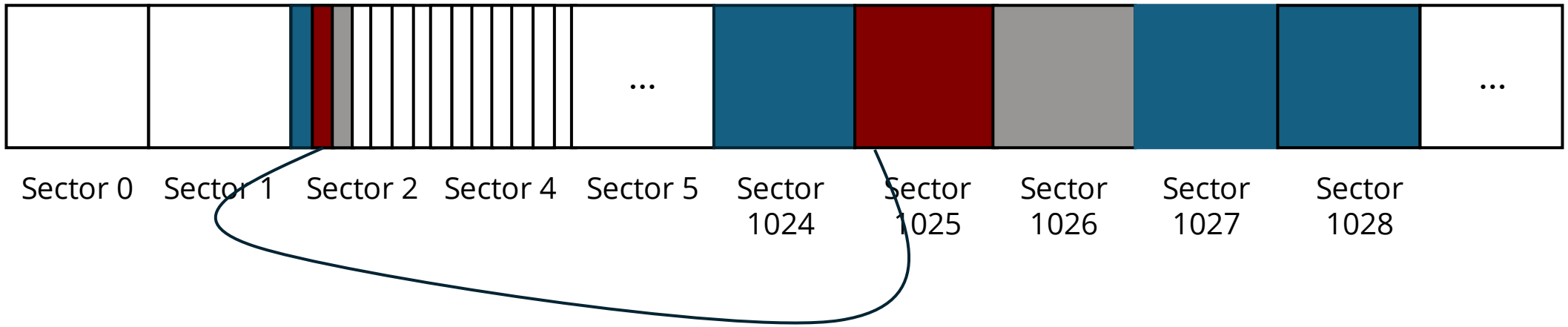
- You need to store the table in memory, what if the table is very large?
- You need to store it in persistent memory
- You're still jumping through a table to get the index of your next block!



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

The BSD 4.3 Filesystem

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



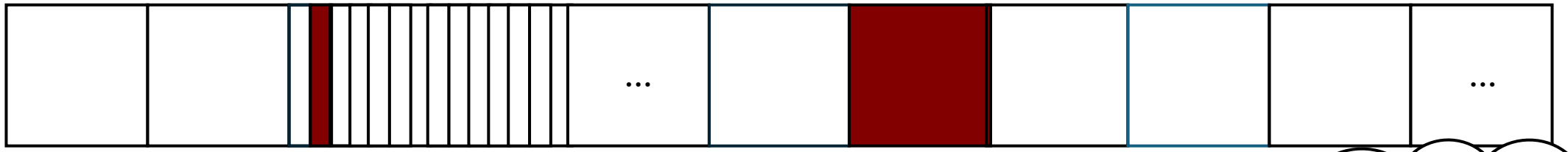
dirent

What is a dirent?

```
struct direntv6 {
   uint16_t d_inumber;
   char d_name[14];
};
```

- Stores the inode number associated with a directory
- Remember that inodes can represent files *or* directories
  - They use the i_mode flag to tell us this

Directories

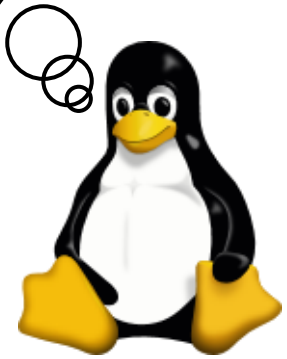


Sector 0 Sector 1 Sector 2 Sector 4 Sector 5 Sector 1024 Sector 1025 Sector 1026 Sect 10

If the inode here is a directory, then the payload blocks will contain dirents

How many dirent's fit in one block? Assuming a block is 512 bytes.

$$512\ bytes \frac{1\ dirent}{16\ bytes} = 32\ dirents$$



What questions can I answer?

Bitmaps and Free Blocks

Block:	0	1	2	3	4	5	6	7	8	9
Bitmap:	0	1	1	0	1	0	0	1	1	1

└ allocated

└ free

Each bit represents a block and whether it's free or allocated

PRO: Allows you to get locality (aka store files in blocks closer to each other)

CON: As the disk fills up, files begin to scatter around and finding an open spot becomes slower

I (Linux) lie to my user about how much storage I have to mitigate this

What questions can I answer?

Block Cache

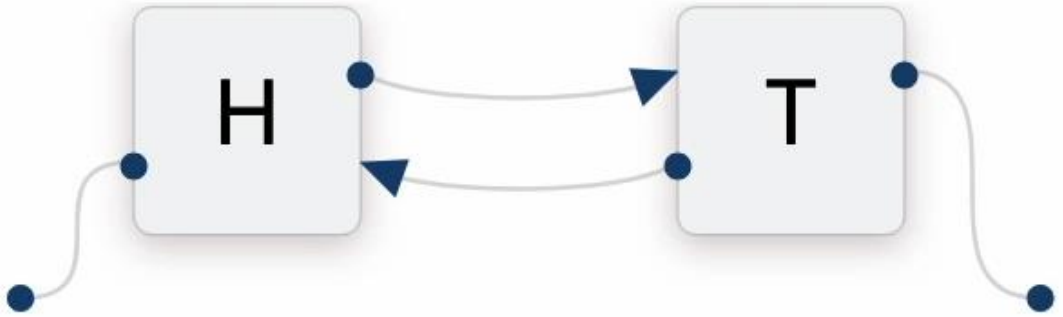
Why the block cache?

- We want to avoid *expensive* disk accesses
- **The key insight:** we usually operate on recently accessed blocks
- Think of this as a scratch pad – it's not committed to memory yet
- *They may reorder operations when commiting to disk*

A Visualization

- This block cache assumes that the block cache can store 3 blocks

The block cache is super fast, and we love operating systems that are fast

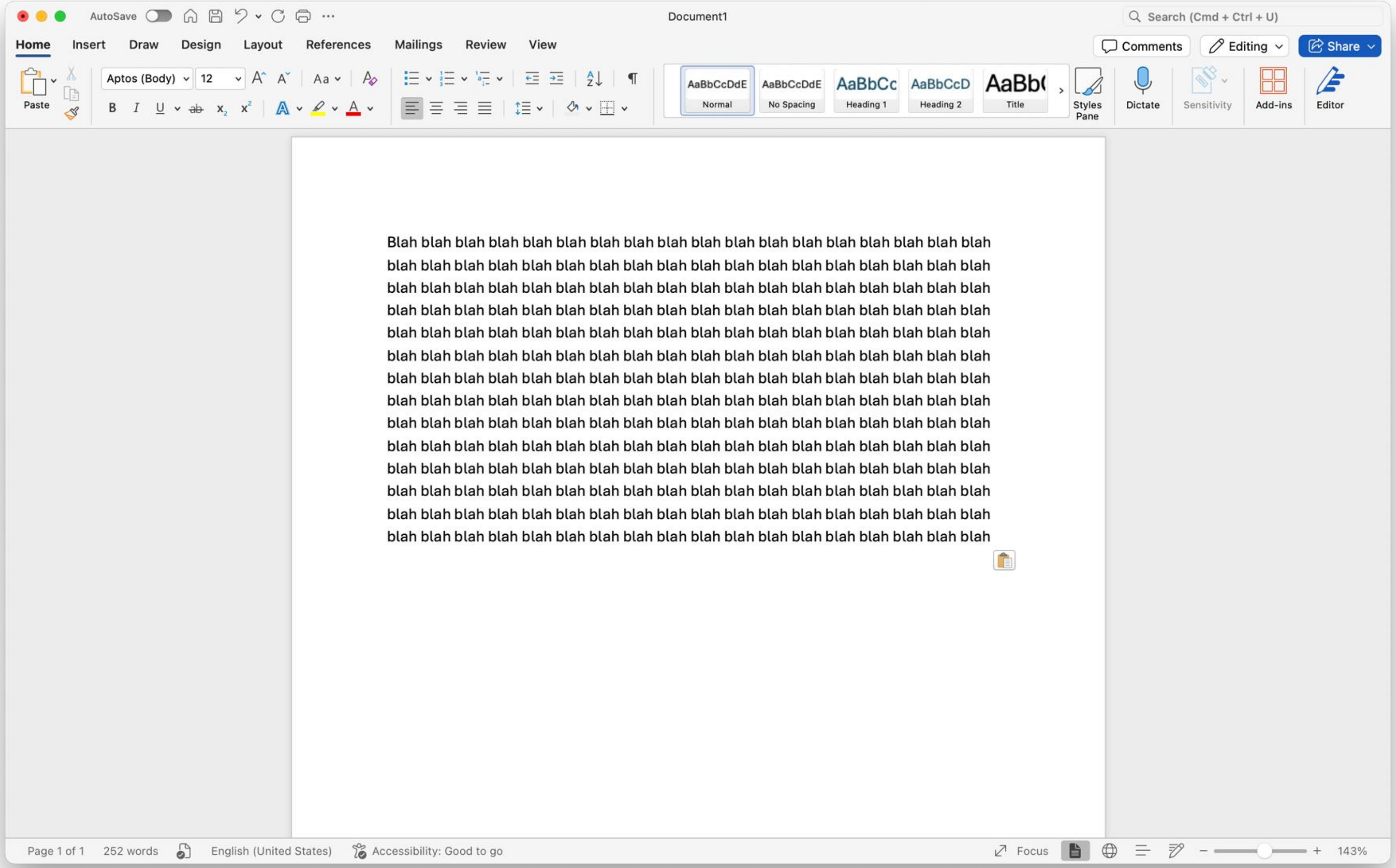


https://lrucache.netlify.app/

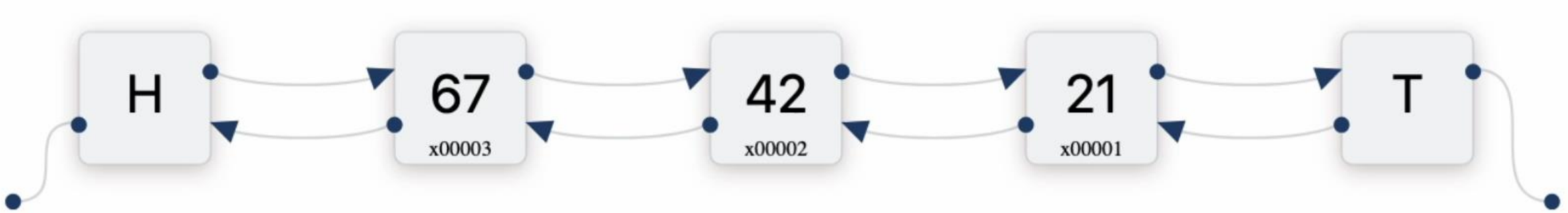
Locality

- **Temporal**
  - When we access a file and do something to it at a particular time, we are likely to continue to make modifications to that file
- **Spatial**
  - When we access some block containing data, and modify it, we are likely to make modifications to other data near it

Super Important Document



Block Caches are Stored in Memory



- So, what is a potential issue with using block caches?
  - We are susceptible to data loss with block caches
  - *What is our program crashes, what happens to what is inside the block cache?*

Write-through vs. Write-back

Write-through

- Immediately write to disk when you write to the block cache
- Slow but good because you reduce the time between a block cache write and a disk write

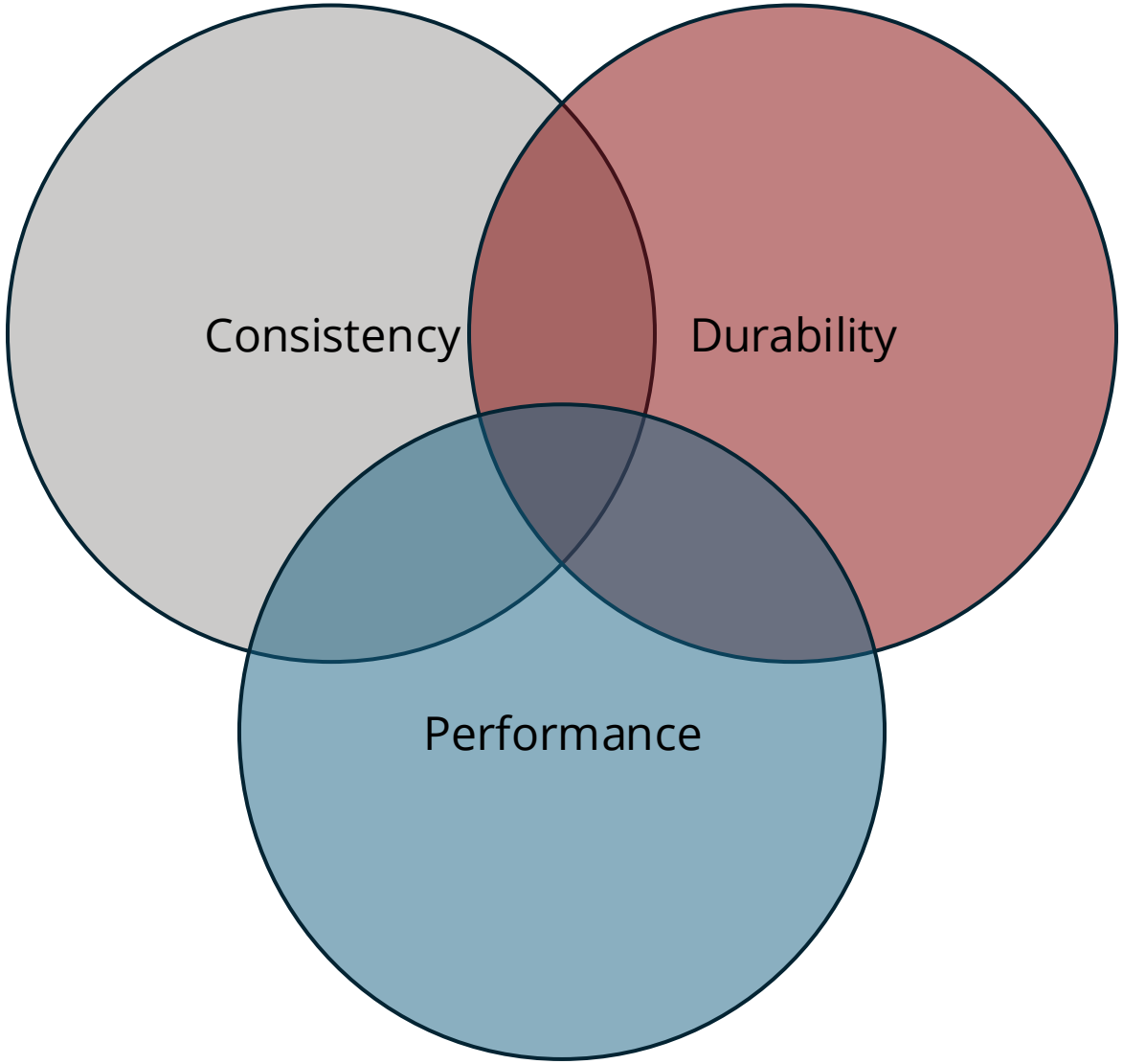
Write-back

- Write the the disk later
- This is quick – like we saw in the video but opens us up to data crashes

What questions can I answer?

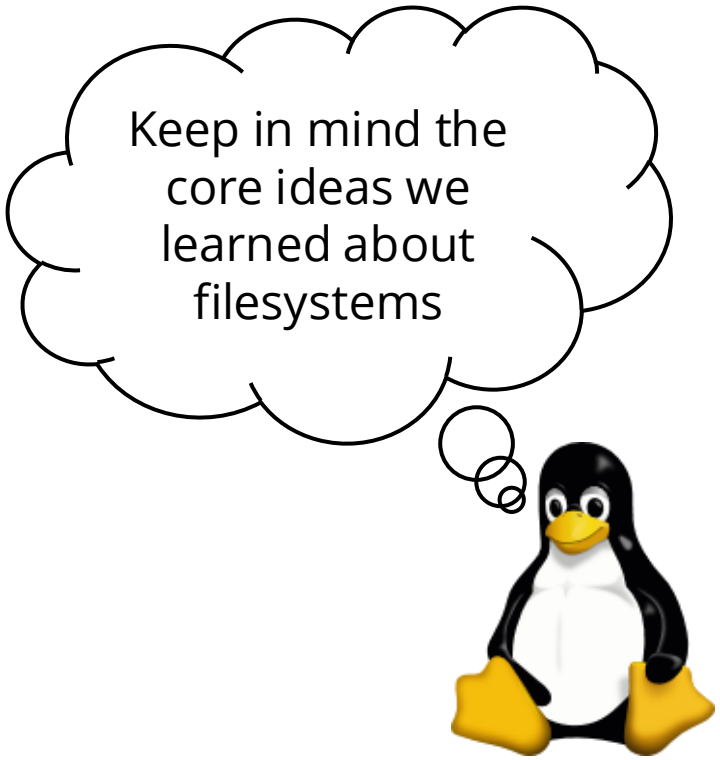
Crash Recovery

Goals of Crash Recovery



How?

- fsck
- Ordered writes
- Write-Ahead Logging (WAL)



fsck

What the fsck?

- fsck is a program that investigates what happened after a crash and how to potentially fix any issues.

- fsck tries to sus out the filesystem

Think-Pair-Share (3 min)

For each, think discuss with your table 1) how could it happen, and 2) how might fsck fix it?

1. A block is listed as being in a file **and** as “unused” in the free list

2. A block is listed as being a part of 2 files

3. An inode has a reference count > 1 (indicating that we can’t delete it because a directory references it), but we can’t find its directory entry anywhere…

What questions can I answer?

Ordered Writes

What are Ordered Writes?

- A *heuristic* that your operating system can use to prevent some of those sus scenarios that fsck dealt with

- The rules:
  1. When adding a block to a file we always write back to the free list before writing to an inode
  2. Initializes inode before pointing directory entries at it
  3. BUT remove i_addr references to blocks before marking a block as free
  4. Always keep 1 live reference to a block.

Con of Ordered Writes?

- We get a performance loss because we are forced to perform *synchronous writes***.
- Like discussed during lecture -- almost defeats the purpose of having a block cache.
- So, what can we do about this?

***writes that require us to wait until they are complete to move forward*

Write-Ahead Logging (WAL)

What is WAL?

- Keep a record of disk-operations that are to be performed
- In this log we keep *metadata operations*
- The log must be synchronously written to disk *before* any of the actual operations you are storing in the log are

Think-Pair-Share: Why do we have to "commit" the log to disk before we perform any of the operations we've outlined in the log itself?

Problem: if we don’t commit log first

- If we don’t commit the log first and we crash, then that defeats the entire point of the log itself.

- On reboot, after a crash, the log will allow us to replay these operations

Checkpoints

- Periodically we can have a *checkpoint* which guarantee that up until this point, everything is written to disk

- Allows you to have shorter logs, since they can get big

Transactions

- The smallest unit of operation in our log
- The transaction was only successful if every operation within it was completed
- Another mechanism for avoiding inconsistency

Example Of Incomplete Transaction

LogBegin(ADD_FILE_A)

REMOVE: block 56 from free list

ALLOCATE: inode for file

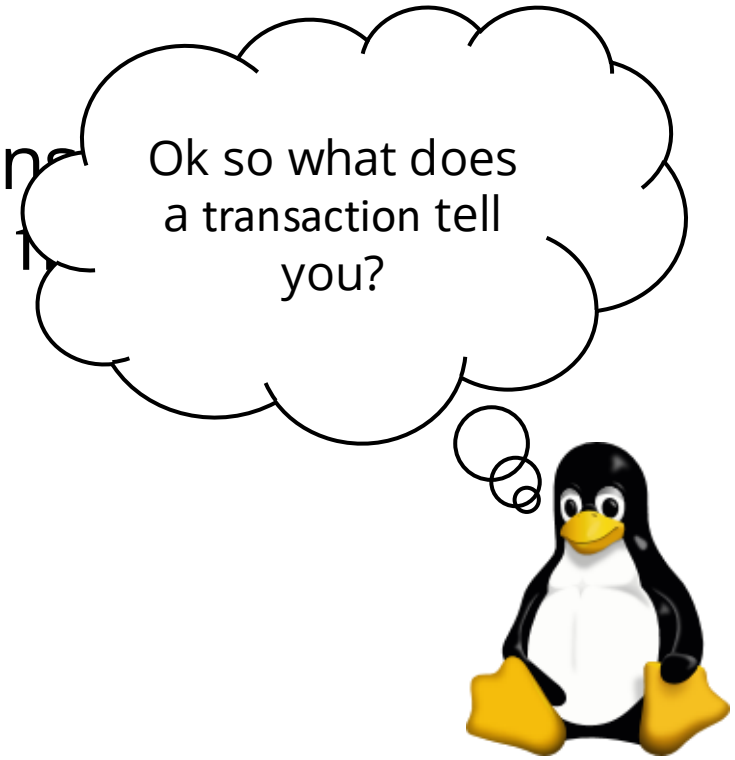
WRITE: inode with contents

CRASH

ALLOCATE: dirent and point to inode

WRITE: contents to file

- In this case you don't want to replay this because it's not a complete log.
- Avoids incon
within your f



Think-Pair-Share: Filesystem Steps

- Consider the following *high level* file system steps:
  - Create 3 new files, X, Y, and Z
  - Write data to files X, Y, and Z
  - Delete file Y
  - Create file A, using the freed blocks from file Y

- In your groups answer:
  1. What steps does the file system have to do for each of these things (accessing bitmap? creating inode? creating dirent? writing to blocks?)

  2. Do the steps you thought about have a particular ordering? What kinds of things could happen if we don't respect that ordering

Create 3 new files, X, Y, and Z, write data to X, Y, Z

1. Find free blocks for the contents of X, Y, and Z
2. Mark the bitmap entries for these blocks as “used”
3. Make 3 inodes that point to X, Y, and Z’s blocks, respectively
4. Add directory entries for X, Y, and Z to make sure they ‘exist’ in our file system somewhere
5. FINALLY write to them

Delete file Y

1. Remove internal inode entry for Y
2. Mark bitmap entry for file Y as ‘free’
3. Remove dirent for Y / references

Create file A, using the Free'd blocks from file Y

1. Repeat the first group of steps for file A

What questions can I answer?

Think-Pair-Share: Transactions

LogBegin1

1. Find free blocks **x1, x2, x3** to store file 1
2. Update the free map to mark **x1, x2, x3** blocks as used
3. Allocate a new inode **i1** pointing to **x1, x2, x3**
4. Add dirent to our new file's parent directory that refers to **i1**
5. Start writing into **x1, x2, x3**

LogCommit1

LogBegin2

1. Find free blocks **y1, y2** to store file 2
2. Update the free map to mark **y1, y2** as used
3. Allocate a new inode **i2** pointing to **y1, y2**

Suppose the system crashes now. Note that Begin/Commit pairs mark logging *transactions.* Describe the state of the filesystem after the system recovers and replays the log.

(ex. Are the files gone? What about the contents of the files?)