Contents

About these reading notes

These are my own personal reading notes that I took (me, Franklin) as I read the textbook. I'm providing these to you as an additional resource for you to use while you're reading chapters from the textbook. These notes do not stand alone on their own — you might be able to get the *idea* of a chapter while reading these, but you're definitely not going to **get** the chapter by reading these alone.

These notes are inconsistently all of the following:

- Me summarizing parts of the text.
- Me commenting on parts of the text.
- Me asking questions to myself about the text.
    - … and *sometimes* answering those questions.

The way that I would expect you to read or use these notes is to effectively permit me to be your inner monologue while you're reading the textbook. As you're reading chapters and sections within chapters, you can take a look at what I've written here to get an idea of how I'm thinking about this content.

Chapter 40: File System Implementation

- Haha. AFS to ZFS. Get it?
- As you're working through this chapter, try to keep the organization that was described in the previous chapter in mind, specifically this idea of inodes, inode numbers, and the relationship between files, their names, and directories.

40.1 The way to think

- Though this file system (and FAT and exFAT) are more array/linked structure based than trees.

40.2 Overall organization

> - simple file systems use just one block size

A block size is how many bytes there are in a block.

- They're calling back to an inode, you may want to remind yourself about what an inode might look like by referring back to chapter 39.
- We're going to think about this later, but how do you think the authors are choosing the number of blocks to represent an inode? The number of blocks *seems* like an arbitrary choice, too.
- Note that a bitmap is not something that you make in Microsoft Paint (it is, but not in this context). In this context, the authors are describing a sequence of bits, where the i^{th} bit being set (it's a 1) means that the corresponding i^{th} inode (in that inode region) is used. If the j^{th} bit is *unset* (it's a 0), then that means that the j^{th} inode is free (we can put a new file there).

- Be able to explain the difference between the superblock and an inode by describing the information contained within each, and by describing the *level* that each applies to.

40.3 File organization: the inode

- The general approach to figuring out where an inode is in an entire block is *straightforward*; there are many parts to include, but it's arithmetic underneath it all.
- The simplified ext2 file node is really good here, it's giving you an idea of the kinds of things that should be in an inode, and approximately how many *bytes* each one of those takes.
  - They're about to get to this right away, but: if you have a total of 15 disk pointers in an ext2 inode, how big of a file could you ultimately represent?

The multi-level index

- So yeah, direct pointers are not very scalable (as you just figured out).
- Thinking about programming, how might you distinguish between direct pointers and indirect pointers? How would you need to change the `struct` that would be represented by the simplified ext2 inode in figure 40.1?
- Then we get to double and triple indirect pointers. Why not quadruple? Using the numbers that the authors have here, how big of a file could you represent if you logically follow what they're describing to having 4 levels of indirection?

  > - One such finding is that *most files are small*

  Do you think this is true on your own system?

40.4 Directory organization

> - Such a delete is one reason the record length is used: a new entry may reuse an old, bigger entry and thus have extra space within.

This seems like a good idea that's going to come back and bite us.

- The aside on pg 9 is describing how this relates to FAT/exFAT. Pay special attention to this idea, FAT/exFAT and what's being described in this chapter are *similar* to each other.
- Remember how files and directories were treated differently at the API level? How are files and directories different at the implementation level? Are they both represented by inodes? Do they have similar or the same metadata?

40.5 Free space management

- In short: you have to know what's currently not used so you can find a place to write stuff.
- Why would ext2 or ext3 pre-allocate blocks? Why would the file system care if blocks in one file are contiguous with one another? (hint: This is deeply related to how disks work)

40.6 Access paths: reading and writing

Reading a file from disk

- Why does reading a file require reading directories?
- Why do you think the root inode number is 2 on most UNIX file systems? Why not 0?
- Be able to convince yourself about what's happening as a file is being read by both looking at figure 40.3, but also about what's happening near the bottom of pg 11, where the authors are describing how the data pointers are being used from the inode itself.
- On closing a file "No disk I/Os take place." <– this is a lie, a terrible lie, but… it's OK to assume that this is true.
- The authors claim that a directory with lots of files in it will be slow to access. Based on what you know about files now, and their relationship with a parent directory, can you confirm to yourself that this is true?

Writing a file to disk

- … is awfully similar to reading a file from disk (at first).
- Why does writing require so much reading? Why do we need to read and write inodes when we're writing data to a file? Think back to what inodes are, think back to how inodes keep track of data, and think back to the structure *way* back at the beginning of this chapter showing how the disk is separated into blocks.

40.7 Caching and buffering

- We're actually going to come back to this idea when we talk about virtual memory; the authors hint at this when they talk about a "unified page cache".
- LRU is least recently used (this is explicitly not the same as the oldest). A file might be around for a long time in a cache, but referred to constantly, where a newer file might have only ever been referenced once.
- This idea of buffering writes again comes up. Try to think about this from a design perspective: how might an OS know that the same block has been written to twice?

- “Some applications (such as databases)”; relational database management systems, the special snowflakes that they are, *effectively* re-implement a huge amount of stuff that’s otherwise the responsibility of an operating system (like file systems).