

## Contents

<b>About these reading notes</b>	<b>2</b>
<b>Chapter 39: Interlude: files and directories</b>	<b>2</b>
39.1 files and directories . . . . .	2
39.2 The file system interface . . . . .	3
39.3 Creating files . . . . .	3
39.4 Reading and writing files . . . . .	3
39.5 Reading and writing, but not sequentially . . . . .	4
39.6 Shared file table entries . . . . .	4
39.7 Writing immediately with <code>fsync()</code> . . . . .	4
39.8 Renaming files . . . . .	5
39.9 Getting information about files . . . . .	5
39.10 Removing files . . . . .	5
39.11 Making directories . . . . .	5
39.12 Reading directories . . . . .	6
39.13 Deleting directories. . . . .	6
39.14 Hard links . . . . .	6
39.15 Symbolic links . . . . .	6
39.16 Permission bits and access control lists . . . . .	6
39.17 Making and mounting a file system . . . . .	6

## 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 39: Interlude: files and directories

- Note: we *haven't really* looked at address spaces yet beyond how they fit into processes and threads. Virtual memory is coming up real soon.
- Note: This chapter is primarily about the *client* side of the file management interface, and some stuff about its implementation.
- This chapter is describing a UNIX file system. As you're working through this chapter, try to think about how the ideas are similar to a Windows-based operating system. Some ideas are literally the same (e.g., files and folders), others are not *exactly* the same (e.g., links and permissions).

### 39.1 files and directories

- Make sure you agree with the author's description of the abstract idea of a file: a linear array of bytes that you can read or write.
- As the authors say: just accept that "inode number" is a thing, that a file has it, and that it is unique, you don't need to know what it is (yet) beyond "it's also part of the file".
- Notice how the authors are describing the structure for a directory: the directory "thing" itself contains the names of the files that are underneath it, paired up with inode numbers. Note specifically that this means that file "things" **do not** have a name embedded within them, but

instead *only* have this unique inode number. The name is attached to its relationship with the directory that it's contained within.

- This both sounds and *feels* really weird: I thought files would have had their own names embedded within them! Nevertheless, make sure you're keeping the relationship between files, inode numbers, their names, and directories in the right place.
- How does this concept of a “root” entry differ from Windows? Both macOS and Linux (and all UNIXes) share this idea of a root entry.

### 39.2 The file system interface

- We've seen `unlink` before when we looked at FIFOs (`mkfifo` specifically). Keep this `unlink` function in mind. Keep it in mind in the context of above (how directories have inode references + names to refer to a file), and keep it in mind in the context of figure 39.1 (which is a tree).

### 39.3 Creating files

- Weirdly, you create a file by using the `open` system call. Is this consistent with higher-level languages like Python and Java?
- Note two things about the API for `open` here:
  1. What *kinds* of options you can pass (permissions and file modification options), and
  2. **How** you pass those options (with some binary arithmetic).

We're going to see this binary arithmetic again real soon, specifically in the context of permissions.

- Hey look! File descriptors! Neat!
  - The authors show the xv6 PCB; the Linux kernel does something similar, there's a reference to a `files_struct` in the `struct task_struct` PCB, which leads you to [fdtable.h](#)
  - Wait, so a file descriptor is a number, the PCB has an array of pointers. What *is* the file descriptor in this context?

### 39.4 Reading and writing files

- The authors aren't joking about `strace`, it's an **incredibly** useful tool to see what system calls a program makes when it's running.

- “each running process already has three files open, standard output ..., standard input ..., and standard error”
  - If file descriptors are an array of pointers, do you think that the “standard output” pointer for each process in a single shell is the same? Do all processes share the same standard output pointer?
- TRY running the code snippet at the bottom of pg 5 yourself!

### 39.5 Reading and writing, but not sequentially

- This is pretty neat, if you need to write to a specific and known part of a file, you can just go directly to that part of the file with the `lseek` system call.
  - Check and see: the commands `head` and `tail` let you print out the beginning of or the end of a file respectively. Do these programs use `lseek`? What would it mean for these files to use `lseek` on something like standard output?
- The authors again show some code from xv6 that relates to the PCB for that OS, you can also take a look at [the corresponding code from the Linux kernel](#)
  - Can you find the “offset” or “position” that they’re referring to in xv6 in the Linux kernel?
- What’s the difference between the open file table and the array of file descriptors that was described earlier? (hint: think about the level *where* these two structures are located, there’s only one open file table, but there are *many* file descriptor arrays).

### 39.6 Shared file table entries

- Try running the code in figure 39.2 yourself, and confirm that you can see how the file descriptors and the open file table are related here in terms of the parent and child process.

### 39.7 Writing immediately with `fsync()`

- The file system, for performance reasons, will **buffer** such writes in memory for some time

This was actually described before in the I/O scheduling policies section. What is the OS trying to accomplish by buffering writes like this?

- What kinds of things do you think could go wrong if, for example, the system suddenly lost power when it’s only buffered (in memory) the write you issued?

- this sequence does not guarantee everything that you might expect; in some cases you also need to `fsync()` the directory that contains the file `foo`.

Why do you think you would need to do this? What's the relationship between a file and a directory? Does the directory itself have changes that would need to be persisted to the disk?

### 39.8 Renaming files

- Rename is an atomic operation: thinking about the relationship between files and directories (specifically about where and how a name of a file is kept), how do you think this works?
- Convince yourself that the steps the authors are describing here about how an editor might overwrite a file makes more sense than just overwriting the file (using a temp file, then renaming the temp file).
  - Also: This is literally what `vim` does! [Neat!](#)

### 39.9 Getting information about files

- Actually try running the code snippet beneath figure 39.5 yourself, what does your output look like compared to the author's? What's different, and what's the same?
- Note now that we're talking about an **inode**, which is distinct from an **inode number**. They are *related* (an inode has an inode number), but they are not the same thing.

### 39.10 Removing files

- OK cool, here's that `unlink` again. Why do you think `unlink` takes the name of a file rather than a file descriptor?
  - Hint: Think about this in the context of the relationship between files, directories, and file names.

### 39.11 Making directories

- Note you can never write to a directory directly.

Why not? Why doesn't it make sense to write to a directory?

### 39.12 Reading directories

- Note that there are different system calls for working with directories. Is this consistent with how you work with directories in other higher level languages like Java or Python?
- How does `struct dirent` differ from the corresponding `struct` that represents files?

### 39.13 Deleting directories.

- Why do you think we don't use `unlink` to delete a directory?

### 39.14 Hard links

- OK, the authors are about to reveal the magic of link and unlink! Hooray!
- Try running the code that the authors show here to confirm what what they're doing actually works on your own systems.

### 39.15 Symbolic links

- What's the difference between a symbolic link and a hard link? Why would you use one over the other?

### 39.16 Permission bits and access control lists

- Here's where we need to look *way way* back at the open system call, in terms of the way that permissions are passed to it for creating a new file.
- Make sure you see the relationship here between the octal values (they are base 8) and the read (r), write (w), and execute (x) bits for each class of users (owner, group, world).
  - We saw this before when looking at FIFOs, when you got the chance to write to my FIFO on aviary.

### 39.17 Making and mounting a file system

- Note that what follows is **not** something you can do on aviary. You *can* run `mkfs` on an empty file, but you **cannot** run it on a device like `/dev/sdX`.
  - You can, of course, do this in a virtual machine or on a machine that you control.

- If you want to try making your own file system, you can quickly make an empty file with the `truncate` command (see `man truncate`), then you can run `mkfs` on that file. So, for example, if you wanted to make a new `ext2` file system, you could do something like:

```
truncate --size=5M my_file_system # make an empty 5MB file
mkfs.ext2 my_file_system           # format the file with ext2
file my_file_system                # ask `file` to tell you about
                                   # what's in this file
```

You unfortunately can't *do* anything with that file system now, but you made it!