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 4: The Abstraction: The Process

Process A running program.

"waiting to spring into action" — sproing.

It's the operating system's responsibility to take code that's just sitting on the hard drive and then actually *do* something with it.

Given what you know about writing programs, what kinds of things do you think an operating system has to do when it's starting a program? I'm thinking about things like reading files and populating data structures.

Most modern systems have *many* CPUs available, but, take a look at your task manager: how many processes are currently running? While I'm writing this, I have 238 running processes. As far as each of those processes is concerned, it has full, unfettered access to one (or more) CPUs. I **definitely** don't have hundreds of CPUs on my system.

Open up your task manager (on Windows 10/11, hit Ctrl+Alt+Delete and choose "Task Manager", then switch to the "Performance" tab). How many processes you have running right now? Do you have that many CPUs on your system?

A basic technique for virtualizing the CPU (giving every process the illusion that it's got a processor to itself) is time sharing, give A a bit of time, then switch to B, then switch to C, then switch back to A, *really* quickly.

They use the term "context switch", but don't go into detail in this chapter.

They *do*, however, say that time sharing is a mechanism that's used by all modern operating systems.

The Abstraction: A Process

The entire idea of a process *is* an abstraction.

The operating system needs to keep track of the state of a process while it's running (the **machine state**).

There's a tip here that sums up to: separate the implementation from the interface. The tip is about separating policy and mechanism.

Process API

This is listing some of the kinds of operations that an OS needs to expose in terms of how it deals with processes.

- Create (makes sense, we need to be able to create a new process)
- Destroy (also makes sense, we need to be able to get rid of a process and everything that's associated with it).
- Wait (sometimes it's useful to wait for a process to finish? Why???)
- Miscellaneous control ("most operating systems provide some kind of method to suspend a process (stop it from running for a while) and then resume it (continue running it)")
- Status (basically: "tell me about yourself")

In terms of destroying a process: what kinds of things do you think an OS has to do *to* destroy a process?

When do you think an OS might need to *suspend* a process? What kinds of things could a process be doing where the OS might want to tell it to stop? When might *you* want to tell a process to "stop", then continue again later?

You can spend some time spelunking in the `/proc` file system on a Linux machine, if you're interested.

Process Creation: A Little More Detail

An OS has to load a program into memory. The program itself is in some kind of executable format (e.g., ELF).

It's more complicated than just reading the binary directly into memory (beyond "lazy loading", the OS has to set up its own internal data structures), but we're just hand-waving it here. The ultimate description of what they're doing here is that they need to load code and "static" data, the data that's encoded in the program itself.

The OS *does* allocate memory for the stack (but note that this is the **entire** stack, not individual stack frames). The one stack frame that the OS *is* responsible for setting up is the frame for `main`, with `argv` and `argc`.

The OS may also allocate memory for the heap.

In terms of "allocating" memory, what do you think the OS is actually doing here? Note that the OS **does not** call functions like `malloc`, those functions live in standard libraries.

The last thing the OS does is set up *some* I/O stuff, specifically giving the process access to things like I/O (standard input, standard output, and standard error).

Process states

Processes can be in a state, and in the simplified book version, processes can be in 3 different states:

- Running (the instructions for this process are currently running on a processor).
- Ready (the process is ready to be run, but it's not *actually* running on the processor).
- Blocked (the process **should not** be run, it's waiting for something to happen).

Looking back at the Process API section, do you think that this description of "blocked" as a status here is related to what was described in miscellaneous control?

They've got some nice tables here showing how processes can move between ready, running, and blocked.

There's a reasonable question here under figure 4.4: "it is not clear if this is a good decision or not. What do you think?"

Data structures

I really like this section because it shows "real" data structures that an OS might use to represent a running process.

Take a look at Figure 4.5, there are more states than just running, ready, and blocked. What do you think each of these states represent? This is covered later, but still might be worth looking at.

How much do you think the OS has to know about the CPU it's running on? This diagram shows a bunch of registers in this thing called a "register context", is this going to be the same for *every* CPU?

They refer to "context switch", but say that it's for later, that's OK. You should actually have the *basic* idea of what's happening (we're switching contexts).

Let's imagine, for a second, that a machine has exactly 1 CPU in it. Once an OS starts running instructions for a process, it's effectively given up control of the processor to that process (the OS can't *also* be executing instructions, there's only one CPU!). How do you think the OS might get control of the CPU again?

Task list as an idea (a **list**!), and the term "PCB process control block" is used here to describe the data structures that represent a process.