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 6: Mechanism: Limited Direct Execution

We're going to talk about some of the nitty gritty details of time-sharing as a concept. Fun!

Two main concepts here: performance and control. Performance is specifically defined here as "without adding excessive overhead to the system". Control is defined as "how does an OS keep control of a CPU?"

Basic Technique: Limited Direct Execution

> "The OS will often use a judicious bit of hardware support in order to accomplish its work effectively." — why do you think the OS can't do this by itself?

Direct execution Run a program directly on a CPU.

Two concerns once a process is running:

1. How do we stop a process from doing something we (the OS) don't want it to do, and
2. How do we stop the process so that we *can* do time sharing?

Problem #1: Restricted operations

> "Restricted operations" here are described as "issuing an I/O request to a disk, or gaining access to more system resources such as CPU or memory". Thinking about the responsibilities of an OS, why do you think these operations are "restricted"?

Here they note "How can the OS and hardware work together to do so?" This is going to be an ongoing theme — how does the OS (software) work together with the hardware to accomplish what it needs to do?

There's an aside here describing how a system call is actually doing a trap instruction instead of just directly calling more code. It also approaches describing the distinction between a standard library and the OS.

They answer the question above right below here, I/O is a restricted operation because we want to build permissions into the FS.

> This is a bit outside the scope of the course, but do you think that there might be more kinds of modes that actual CPUs have beyond user and kernel mode? Why do you think a CPU would need those additional modes?

System calls are described as both a responsibility of the OS and the hardware. This makes it a bit weird, but you have to remember that the OS *is* software itself, the same as user programs. So if the OS has the entire responsibility to handle system calls (which it may have at some point), then user programs could (in theory) just do the same things as the OS.

> "To execute a system call, a program must execute a special **trap** instruction. The instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode;" — Who do you think is responsible for setting up these connections (e.g., where in the kernel the CPU jumps to for a specific system call), when those connections need to be set up, and what kind of a data structure might be used to store those connections?

The trap table is then described almost *immediately* below this. Figure 6.2 actually goes through and describes this entire setup, both timing, trap table, and trap handlers.

> The x86 platform has a per-process kernel stack. Why do you think this is only on a specific processor architecture?

> The tip on pg 6 says "Otherwise, it would be possible for a user to read all of kernel memory; **given that kernel (virtual) memory also usually includes all of the physical memory of the system, this small slip would enable a program to read the memory of any other process in the system.**" *Why* does the OS have access to all physical memory? Does an OS care about what

> a process is doing within its own virtual memory space?

> What kinds of things might a user program be able to do if it were able to install its own trap table?

> Thinking about `fork`, `exec`, and `wait`, where in figure 6.2 would these calls fit?

Problem #2: Switching Between Processes

The main problem with running user code directly on the processor is that you (the OS) needs to be able to regain control of the CPU so that you can let *other* programs run.

A Cooperative Approach: Wait for system calls

This seems like a reasonable approach: eventually a program is going to have to ask the OS for something, so just wait around for the program to ask the OS to do something before deciding to switch something out.

> Here's an example: if a program is computing digits of Pi, when might it issue a system call? Here's a C program that computes pi: https://crypto.stanford.edu/pbc/notes/pi/code.html

A Non-Cooperative approach: The OS Takes control

So, uh, turns out we (the OS) can't do much in the cooperative situation in terms of bailing out if someone's stuck in a loop. That requires manual, physical intervention to reboot the machine.

The next approach is … falling back to letting the hardware help us out.

> Notice that we *again* have to let the hardware help us out here — timer interrupts are *kind of* like traps, but instead of being invoked by a user program, the hardware is set up to run it on a schedule.

> Compare and contrast interrupts and interrupt handlers and traps and trap handlers. What are similar between the two ideas? What's different?

Saving and Restoring Context

Oh boy context switches!

The **scheduler** is the part of an OS that's responsible for deciding who's next, but for now we're going to assume that who's going next is decided for us. We just need to think about the mechanics of actually getting that program running.

> Looking back at the previous chapters describing a process, where do you think all the information that an OS needs to save about a process gets stored?

> Thinking about two parts of a context switch (the switching between kernel mode and user mode and switching to a new process), how much do you think is the same between different operating systems (e.g., Windows vs Linux vs macOS)? How much is different?

Worried about Concurrency?

What happens if multiple of the events happen at the same time? It's a problem, yes, but let's acknowledge it's a problem, then ignore it for now.