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 36: I/O Devices

Hard drives are cool and stuff, but it's probably a good idea to look at the generalization or abstraction of input and output devices before we look at *a specific example* of I/O devices.

Everything that's attached to our computer (pretty much) is some kind of I/O device. Keyboards, mice, screens, stuff that you attach with USB, WiFi.

Before you get into this chapter:

- How do you (as a programmer) currently interact with I/O devices?
- How do you (as a user) interact with I/O devices?
- Based on what you saw in COMP 2280 (or an appropriate related course in a different program, COMP 2280 is introduction to computer systems: assembly languages, architecture, etc), how do you think an OS actually interacts with hardware devices? It can't just make system calls like `open`, can it?

36.1 System architecture

- Whoa, hold on a second here. I thought COMP 3370 was computer architecture, what's the deal?

- We're really just starting to get comfortable with terms and acronyms (so many acronyms). We're also starting to get comfortable with the physical relationship between devices (e.g., where they are located physically relative to each other, who's connected to what).
  - "memory bus": beep beep.
- This architecture stuff is about "physics, and cost". What does an operating system care about this stuff?
- While the detailed description on pg 2 is *interesting*, it's not really that relevant beyond "here are the definitions of acronyms, and where these are physically related to each other".
  - Are there any acronyms here that you don't recognize?

36.2 A canonical device

- Alright, so this is actually more like what an operating system needs to care about (*does* it need to care about physical location of devices relative to others?)
- "From Figure 36.3 (page 4), we can see that a device has two important components. … interface … internal structure …"
  - Wait, what? Isn't that what we're looking at in COMP 3350? Are software and hardware design related to each other somehow?
- "more complex devices will include a simple CPU, some general purpose memory, and other device-specific chips to get their job done.", this feels very much like the kind of device that Xzibit would like.
  - Does that mean that there's a computer inside my computer? Does that mean that there are **multiple operating systems** running on my machine?

36.3 The canonical protocol

- Protocol: we've seen this word before in the context of pipes and client/server architectures. Protocols are really more of a distributed computing thing, but(!), given what we just learned above, *is* this a distributed computing situation?
- "By reading and writing these registers, the operating system can control device behaviour", OK, this is our first hint about how an OS actually communicates with hardware devices.
- Given *just* the pseudocode just below the start of this section: what might happen if multiple threads tried to interact with this device?

- Have you ever seen the term “polling” before? (outside of the context of politics and outside of the context of surveying people)
- The description of what this pseudocode is doing (poll, write, poll): what does this look like to you? Does it remind you of anything we’ve seen before?
  - What kinds of problems with *that thing* did we eventually try to overcome? HINT: we got to use new system calls that are only available on ancient operating systems (Solaris): `park` and `unpark`.

36.4 Lowering CPU overhead with interrupts

- OK, cool. We put the process to sleep and let the I/O device **tell** us when it’s finished rather than constantly asking it “Are you done yet?”.
- Yay! Our CPU is more efficiently utilized now!
- Why would really fast I/O requests make the system slower? Can an OS actually recognize this? Do operating systems need to know about how fast a device is?
- This term livelock was used previously when discussing solutions to deadlock. Does it mean the same thing here?
- Is coalescing the same thing that the OS is doing in the hybrid approach? Maybe. Coalescing is something we’ll see in the next chapter in terms of scheduling writes to disk.

36.5 More efficient data movement with DMA

- OK, so while maybe we improved resource utilization (we’re not just constantly asking the I/O device if it’s finished, we can do meaningful work!), performance still has room for improvement!
- The main problem that’s left is that the CPU (and thus the OS) still is responsible for *transferring* the data somewhere once the I/O device indicates that it’s finished. The OS actually has to *give* those bytes to the process that did the `read` system call.
- Wait: didn’t we say earlier that there’s basically a tiny computer inside that I/O device? Instead of us (the OS and the CPU) pushing data at the device from memory, why don’t we let the device *pull* the data directly from memory? Yay! This is DMA!
  - The OS tells the device *where* and *how much* data to copy from/to memory, then the device does the rest of the work itself.

36.6 Methods of device interaction

- Wait a minute. We didn't say how the OS communicates with the device? I thought we said that the OS communicates by reading and writing registers? How does it read and write those registers? Does x86 assembly have specific notation for writing to drives?
- Uh, I guess really old machines had I/O instructions.
  - Whoa, x86 *does* have special instructions for this, `in` and `out`. The instruction lets you specify which device should be communicated with using a *port*.
- "Such instructions are usually **privileged**." – why would this instruction be privileged? Why *wouldn't* you want a user program to be able to directly issue reads and writes to a disk, for example?
  - What does it even mean for an instruction to be privileged? Think *way way back* to direct execution in chapter 6!
- Memory-mapped I/O: steal/hide a *tiny* bit of memory from the system and use those addresses to communicate with a device. You *may* have done something like this in COMP 2280.

36.7 Fitting into the OS: the device driver

- Based on what's being described here, try to think about where this device driver fits; the file system is a pretty integral part of an operating system. If you were to imagine you sitting in a chair at your keyboard and the disk, where the file system, the operating system, and a device driver are "in between" you, how would you actually order these things?
- OK, awesome. Abstraction. That was something we learned about… well, everywhere in CS I guess.
  - Thinking about something like a hard drive, what kinds of abstract operations might an operating system or a file system need to do?
  - Thinking about something like a web cam, what kinds of abstract operations might an operating system need to do?
- You might not pick up on this, but an interesting thing about what's being described here:

> it simply issues block read and write requests to the generic block layer, which routes to them to the appropriate device driver, which handles the details of issuing the specific request.

is that this means that the "generic block layer" could service an I/O request that spans across multiple physical drives. Think about that: a file spread across multiple physical hard drives! Neat!

- Why would something like a file-system checker need "raw" support?
- "Note that the encapsulation seen above can have its downside as well.", what's described here is also a downside of the entire idea of OO: yeah, we get it, a dog barks to speak and cat meows and they're both animals, but, uh. Wait, what was this about? Right. OO. Can you think of any ways that you might approach changing this relationship (device drivers, OO, encapsulation) so that, for example, the "rich error reporting" from SCSI would be available?
- "Studies of the Linux kernel reveal that over 70% of OS code is found in device drivers;" — the Linux kernel is **millions** of lines of source code. Millions and millions. We've looked at an infinitesimally small fraction of the Linux source code in class.

36.8 Case study: A simple IDE disk driver

- *ACK*. Figure 36.5 is… not a sight for sore eyes. It's a sight that maybe makes sore eyes.
- Really, this is just showing all of the addresses where you would write to in with the `in` and `out` instructions to communicate with an IDE drive. We're not going to write x86 assembly, so we don't really have to care very much about these specific addresses.
- The protocol here is describing *how* you would interact with the device, using these addresses, and you can see some actual code in figure 36.6 on pg 12 that's *implementing* this protocol.

36.9 Historical notes

- "Interrupts are an ancient idea", yeah, a lot of things (a surprising number of things) in modern operating systems are ancient ideas.
- DMA was available **70 years ago** (as an idea, anyway).

36.10 Summary

- Make CPU get used for good stuff more, let the I/O device do more of the work, abstraction makes supporting many kinds of things easier.