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 13: The abstraction: Address spaces

- Alright, file systems were fun, but let's go all the way back to processes now.

13.1 Early systems

- Thinking about what *your* processes see in terms of address spaces, how does it differ from what's presented in figure 13.1? *Does* it differ from what's presented in figure 13.1?
- The authors explicitly state that "there would be one running program" in this kind of early system. What do you think might happen if multiple processes were allowed to run in this environment? What kinds of things *could* go wrong?

13.2 Multiprogramming and time sharing

- Yay, scheduling!
- Multiprogramming, yielding the processor on I/O, hooray!
- "The notion of **interactivity** became important" — can you imagine using a computer where you had to write your programs entirely on paper, then encode them on some other kind of physical medium, then *give* the program to someone else to run?

- "In particular, allowing multiple programs to reside concurrently in memory makes **protection** an important issue" – it seems obvious *that* we wouldn't want a process interacting with the memory that belongs to another process, but *why not*? What kinds of things do you think could happen if the OS were to allow that to happen?
- *Before* reading the next part, can you think of any issues with getting a process to figure out what its own memory region is? Noting specifically that the address space in figure 13.2 for (for example) process A **does not** start at 0.

13.3 The address space

- NOTE: **address space** != actual memory. We're going to keep seeing this, but "it is the running program's **view** of memory in the system".
- "The **code** of the program (the instructions) have to live in memory somewhere"; this makes sense (a pro*gram* is inert on disk, a pro*cess* is running), but make sure that you can mentally convince yourself that this is true.
- Heap and stack are separated from each other. Why do you think that is?
- In figure 13.3 we're showing a 16KB **address space**. In terms of the first point here, that means that *actual memory might be much larger*.
- Important: "… somehow the OS, in tandem **with some hardware support**." This is a very common theme when talking about operating systems: Software itself can't do everything, and neither can hardware, so the two have to work **together** to get things done.

13.4 Goals

- Transparency: when was the last time you cared about *where*, physically, in memory your program was running?
- Efficiency: this is a topic that's getting into hardware (TLBs).
- Protection: the authors are using two different words here, "protect" and "isolate". What's the difference?

13.5 Summary

- Try actually running the code listing in the aside on pg 7. Run it on the same machine multiple times, run it on different machines (e.g., rodents and aviary and your own machine). Is the output always the same? Different? Based on the description here, what would you have expected?