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 26: Concurrency: An Introduction

The first little paragraph here summarizes last week's readings on processes (cool!).

Now we've got a *new* abstraction for running a single process: a "thread". I *don't* like the way that they're describing this concept ("for running a single process").

Remember: PC is referring to "Program Counter" the register, not "Personal Computer" or "Progressive Conservative".

Threads actually require a context switch, just like a process.

Processes required context switches beyond just registers: at least as far as we know so far, a context switch for a process implies that the address space is also switched out (they're not shared, right?). With that in mind, what kind of differences do you think there are in terms of context switching for a process and context switching for a thread? This is *hinted* at in the text ("there is no need to switch which **page table** we are using"). Page tables are something that we'll look at later in the course.

Process state goes into a process control block, thread state goes into a thread control block (TCB).

Where do you think a thread control block is *stored*? How is it related to a PCB?

Why use threads?

Remember how we could `fork` a new process and do work in it, then `wait` for that child to end? Why are we bothering with looking at this new structure?

The authors are describing two kinds of workloads here to justify using threads: one is to improve the speed of programs that are embarrassingly parallel (they call this parallelization), and the other is to let some parts of a program continue operating while another is blocked waiting on, for example, I/O. Can you think of any example problems or existing software that you currently use that uses threading in each of these ways?

An Example: Thread Creation

Oh boy, pthreads!

Considering the code listed in figure 26.2, how would you write this same code using `fork` and `wait`? How different is that code from what's listed here with pthreads?

Make sure that you can convince yourself of what the authors are saying immediately below figure 26.2: "Overall, **three** threads were employed during this run: the main thread, T1, and T2."

Take the time to compare figures 26.3 and 4.4! How are they different? How are they the same? How are the author's *descriptions* of these figures similar and different, specifically thinking about non-determinism.

Why it gets worse: Shared data

You were actually introduced to this in chapter 2! (pages 7–8).

Try entering and running the code in figure 26.6 on your own (or, uh, just download it from here: https://github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-intro).

Thinking back to chapter 2, can you briefly explain why 1) we don't get the correct result with multiple threads, and 2) *try* to explain why this might actually happen (keeping in mind that two threads are running "at the same time" and that one thread might be interrupted between *instructions*). This is described in gory detail in section 26.4 and figure 26.6, but try to be able to explain it to yourselves verbally, and at a high level.

The heart of the problem: uncontrolled scheduling

Can data races or race conditions happen with processes? Why or why not?

Let's try and relate what's happening in OS to other courses (CS or not). Where else have you seen this concept of atomicity? Where else have you seen this concept of a "transaction"? Where else have you seen this concept of "all or nothing"?

The wish for atomicity

It sure would be nice if we could just add a bunch of atomic instructions to our instruction set.

That `memory-add` instruction seems like a fine idea, but the authors do a pretty good job convincing us that adding instructions for *all* operations that need to be atomic isn't reasonable. But why not add instructions for things that are *reasonably* universal, *like* `memory-add`? This is outside the scope of the course, but: are there any instruction set architectures that have atomic primitive instructions like `memory-add`?

One more problem: Waiting for another

The authors here describe a problem for threads about "waiting around" for another thread to complete some action. Can you think of any reasons why they didn't describe this problem when talking about the `wait` system call? It seems *awfully* similar.

Summary: Why in OS class?

This is a fair question, and one that I think we often ignore because the exercises that we give to students are threads from the user side rather than threads from the OS side.

Don't worry *too* much yet about the critical sections described by the book in terms of what the OS has to worry about. Instead, think about it this way: if your process (as a user process) or thread could be interrupted during execution, the OS *probably* has similar internal issues. So let's think about the OS side of things for a second: Thinking *specifically* of the system calls that you know about so far, which of the system calls might have a critical section, and what might it need to protect?