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 30: Condition Variables

- We skipped chapter 29 "Locked Data Structures". If you want to see locks in practice, feel free to take a look at that chapter (there's as much code as text in that chapter).
- We've *seen* this idea of joining before, for both threads *and* processes. Now you can put a name to this idea.
- Figure 30.2, what is this, a spin lock? Didn't we figure out that those were "bad"?

30.1: Definition and routines

- "A **condition variable** is an explicit queue…" There's that queue data structure again.
    - Think back to chapter 28: we're using a new term (condition variable), but isn't this basically just a lock?
- Think back to *several* weeks ago now: can you do this with signals? How?
- **Convince yourself** that the two changes to figure 30.3 that are defined in 30.4 (replacing the `thr_exit` and `thr_join` functions) can result in a condition where the parent gets stuck waiting forever.

30.2: The producer/consumer (bounded buffer) problem

- Can you think of any real-life analogy for what the authors describe as the producer/consumer problem?
- The authors use an unnamed shell pipe | as an example of the bounded buffer problem. Think back to when we looked at the kernel source code for pipes: what was that bounded buffer data structure?
- "The `put` routine assumes the buffer is empty", how does it do that? *Can* it assume that, in general, without a lock? Can you think about how this is related to condition variables and signals before we get to the actual answer below?

A Broken Solution

- "However, putting a lock around the code doesn't work; we need something more." Why? Why doesn't a basic lock work here?
- **Convince yourself** about how this solution is broken using figure 30.9. This is getting difficult to think about 3 things running concurrently!
- This is a *very* subtle reference, but on pg 9 the authors write "An assertion triggers" — they're very subtly referring to the use of assertions (i.e., `assert.h`) and design by contract here.

Better, but still broken: While, not if

- Again, **convince yourself** that figure 30.10 is broken using the trace in figure 30.11.
- Really importantly: these problems are coming from the fact that locks and threading are permitting us to have some control over scheduling, but the scheduler is kinda fighting back and we can't predict when the scheduler is going to take over.

The Single buffer producer/consumer solution

- Once again, **convince yourself** that figure 30.13 is indeed correct compared to the previous solutions. What's different about this specific solution compared to the others?

The correct producer/consumer solution

- Finally, and again, **convince yourself** that what the authors have done in in figure 30.14 is indeed a correct and **general** solution to this problem (e.g., buffer size > 1, multiple producers, multiple consumers).

30.3: Covering conditions

- OK wat? A multi-threaded memory allocation library? We'll *eventually* take a look at memory allocation (when we take a deep dive into memory management), but the general idea here is good to think about: you've got pthreads, they're almost certainly going to call `malloc` at some point, how would the OS (or, in reality, the standard C library) actually handle multiple threads asking it to allocate memory for them on the heap?
- The broadcast variant of `cond_signal` is kind of cool, it wakes up all threads that are blocked on a condition variable instead of just one. *Can* you do this with the signals that we looked at with the `kill` and `signal` system calls?
  - As an exercise, try creating multiple pthreads that register the same signal handler, then send the signal to your process.