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 28: Locks

You *might* want to go back and remind yourself about what a *thread* is, and some of the issues related to threading (specifically critical sections).

28.1: Locks: The Basic Idea

- The authors write this statement "… and thus exactly one thread holds the lock and *presumably is in a critical section*."

  Why would they write "presumably" here? Isn't a lock being locked a guarantee that a thread is in a critical section? Presumably means that there might **not** be a thread in the critical section. Did the thread lose the key?

- If you want a different analogy for locks here, think of a mutex lock as a "talking stick"; only the person who has the talking stick (has acquired the lock) is permitted to talk (enter the critical section). When the person who has the talking stick is done, they give it to the next person.
    - Take this analogy of a talking stick and think about the previous question, e.g., "*presumably* someone has the talking stick and is talking".

- "Locks provide some minimal amount of control over scheduling to programmers."

  The authors provide an explanation of this immediately below the statement, but make sure that you can convince yourself that this is true.

  - Expand on this idea: is this preemptive or non-preemptive?

28.2: Pthread Locks

- Think about this idea of coarse-grained vs fine-grained locking specifically in terms of a linked list: coarse-grained locking would mean that you're putting a lock on the entire list itself, something like:

```
struct LIST {
  pthread_mutex_t lock;
  struct NODE *head;
  // other stuff
};
```

  Any time *any* thread wants to do *anything* to the list, you'd have to lock the **entire** list, so only one thread could ever change the list at a time.

  Fine-grained locking would be more like putting a lock on every *node* in the list:

```
struct LIST {
  struct NODE *head;
  // other stuff
};
struct NODE {
  pthread_mutex_t lock;
  struct NODE *next;
  // other stuff
};
```

  Any time a thread wants to do something with a *specific* node in the list, it would lock **just** that node. Other threads would be free to modify other parts of the list concurrently.

  - OK, now expand on this idea in terms of a tree structure, with coarse-grained locking, where would you put the mutex? With fine-grained locking, where would you put the mutexes?

- Remember that code in chapter 2 that had a global counter? Yeah, go ahead and add a lock to that. It's figure 2.5, and you can find the source code for the code on GitHub.

28.3: Building a lock

- "We need hardware support." THE END.

28.4: Evaluating locks

- Hey look, it's this term "fairness" again! Where have you seen that before? What was the context?
  - Hey look again, it's this term "starve"! Where have you seen that before? What was the context?
  - It's almost like these two contexts are related or something.

28.5: Controlling Interrupts

- "Compared to normal instruction execution, code that masks or unmasks interrupts tends to be executed slowly by modern CPUs."
  - A point of clarification: masking and unmasking here means turning on and off.
  - With that in mind, the implication is that modern CPUs allow programs to do that, just that those programs run slowly. *Is* this real?
- The authors claim an OS might turn off interrupts as a primitive locking mechanism. This means a couple of things:

  1. Operating systems are multi-threaded.
  2. They don't necessarily have access to libraries like Pthreads.

  Why wouldn't an OS use a library like Pthreads?

28.6: A Failed Attempt: Just Using Loads/Stores

- Think about this concept of spin-locking in terms of what happened with pure round-robin when I/O was added, is the idea here any different?
  - What was the solution for "fixing" this issue with round-robin? Can you use something similar to "fix" the performance problem here? (Mutual exclusion is a problem that's not directly related).
- **Really** convince yourself about what's happening in figure 28.2 (on pg 6) in terms how this flag-based approach does not provide mutual exclusion.

28.7: Building Working Spin Locks with Test-and-set

- Oh gosh, there's an "aside" here that's a full page.
  - **Really** convince yourself that this "locking" and "unlocking" actually works compared to figure 28.2.
  - "Of course, this line of work became quite useless when people realized it is much easier to assume a little hardware support…" Uh, yeah, it's always easier when you let someone else do the hard work.
- What specific part of what's happening with test-and-set is fundamentally different from the load and store idea in figure 28.2?
  - Think about this specifically in the sense of figure 28.3 which **infuriatingly** spans 2 pages and has an "aside" in the middle of it. Figure 28.3 starts on pg 6, then resumes on pg 8.

28.8: Evaluating spin locks

- Try to think about the assessment of performance here in terms of round-robin without I/O support. Are spin locks and plain round-robin suffering the same problems?
- Spinning on one CPU: wasted cycles. Spinning on n CPUs? Perfection.
  - What? Why? Why would n threads on n "CPUs" (where CPU here is any of CPU, core, threading unit) be *OK* for spin locks?
- The authors make an assertion that spin locks aren't fair, but I feel like their explanation isn't good enough; wouldn't any locking mechanism be unfair if the current thread that holds the lock never gives it up?
  - I'll expand on their behalf: the basic idea of fairness here is more that *if* the threads were guaranteed to all give up their lock, *eventually* all of them would get access to it. That guarantee isn't true with spinlocks. If we have n threads, we have no guarantee about when or how the threads acquire the locks (we're at the whim of the scheduler).

28.9: Compare-and-swap

- Convince yourself that this is actually different from test and set. (it *is*, I promise)

28.10: Load-linked and store-conditional

- The authors tell you to stop here (midway through pg 11) and think about how you might build your own spin lock with the load-linked and store-conditional primitives. Do it!
  - They give an answer in figure 28.6.

28.11: Fetch and add

- THIS IS AN ATOMIC ++ OPERATION! Aww yiss.
- Considering figure 28.7 and the description of the ticket lock, why doesn't the `unlock` function *also* have to use the fetch and add instruction? Is mutual exclusion still guaranteed?
- OK, so *here* we've got something fair, "one important difference with [the ticket lock] solution versus our previous attempts: it ensures progress for all threads."
  - Think *really* hard here: we're talking about tickets and counters and increments, but we're implicitly talking about a data structure. What is the data structure? Hint: it's like a list. But, uh, not *just* a list. It's a list with an order. Yeah, that's a good hint.

28.12: Too Much Spinning: what now?

- We're *again* talking about wasted time. It's like we keep having these same problems over and over again. First with round-robin, now this. It's ALMOST LIKE THEY ARE RELATED SOMEHOW.

28.13: A simple approach: Just yield, baby

- Yeah, OK, the thread that doesn't get the lock should just give up its timeslice. OH HEY. IT'S ALMOST LIKE THE PROBLEMS THAT WE SAW WITH ROUND-ROBIN ARE RELATED TO THIS PROBLEM SOMEHOW.
- The authors mentioned this directly before, but are implicitly mentioning it again here: "the yielding essentially **deschedules** itself"; locking mechanisms give the programmer (and the user space program) some amount of control over scheduling.
- Many threads yielding has issues with context switching: OH HEY, remember how we could set timeslices with scheduling? Remember how making that *really small* had problems with context switching?

28.14: Using Queues: Sleeping instead of spinning

- OOOOHHHH HEEEEEYYYYYY. It's almost like round-robin and thread locking have *similar problems*. Are they related to each other in some way? Remember what we did in round-robin when I/O was initiated?
- The authors describe two system calls here: `park` and `unpark`, but they're *Solaris* (SunOS) system calls. Does Linux have anything similar? What about macOS? Windows? On Linux you can try searching with `man -k`.
- "Second, we use a **queue**" (this is on the bottom of pg 17).
  - IT'S ALMOST LIKE SCHEDULING IS RELATED TO THIS SOMEHOW.
- What do you think this "something bad" is that's mentioned on pg 18 in terms of releasing the guard lock after `park`?
  - Remember how the authors talked about "presumably" there's a thread in the critical section?

28.15: Different OS, Different Support

- Thinking back to the primitives described before, what *is* a futex?
  - André Almeida wrote a *great* blog post that describes the idea of a futex in Landing a new syscall, part 1: What is a futex?

28.16: Two-phase locks

- Sometimes doing two things is better than doing one thing. What kind of circumstance does spinning work better than sleeping? (This is described earlier in the chapter).

28.17: Summary

- The summary here brings an important point: for **user-space** locking to work, there needs to be both **hardware** and **OS** support.