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 32: Common concurrency bugs

32.1 What type of bugs exist?

- FWIW: this book is kind of showing some age here: MySQL was acquired by Sun, which was acquired by Oracle, which is kind of a "bad thing" ™ because Oracle itself is a huge RDBMS company, so owning *two* really big relational database implementations makes people kind of nervous. MySQL was forked to MariaDB.

  Mozilla itself isn't really a thing anymore, you're *probably* more familiar with Firefox, which is ultimately derived from Mozilla's codebase.

  OpenOffice itself is also not really a thing anymore, but LibreOffice is (OpenOffice has its heritage in StarOffice, which was also at some point a Sun product, which was then bought by Oracle).

  The Apache web server is the only software here that's still what it was when this book was originally written. Good going, Apache.

  END OF HISTORY LESSON.

- Why do you think "Non-Deadlock" concurrency bugs are more common than "Deadlock" concurrency bugs? At least in the table the authors are reproducing in figure 32.1, deadlock bugs are less frequent than non-deadlock bugs.

32.2 Non-deadlock bugs

Atomicity-Violation bugs

- At the top of pg 3, the authors invite you to try figuring out the solution to the atomicity-violation bug listed in figure 32.2. DO IT. Stop and think about it, how would you ensure that the value of `thd->proc_info` doesn't change while it's being used?

Order-violation bugs

- Ok, this is tricky to do, but try covering up the text that's immediately below figure 32.4 and see if you can figure out what the bug is in the code.
- Why does the solution for this bug use condition variables *and* locks? Why wouldn't it just use a lock? Why does it have to use this additional variable `mtInit`? (Hint: *is* the code that follows the lock+condition variable in Thread 2 a critical section? Assuming that `mState` itself is a stack allocated local variable.)

Non-deadlock bugs: summary

- The authors reproduce a statistic here by Lu et al. that "(97%) of non-deadlock bugs…", what are the other 3%?
  - This paper is called Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics.

32.3 Deadlock bugs

- Deadlock bugs
- BEFORE READING AFTER figure 32.6: Looking *just* at the code in figure 32.6, can you come up with the sequence of execution and interruptions that would be required for a deadlock situation to occur between thread 1 and thread 2?

Why do deadlocks occur?

- Check out the *current*(-ish) Java implementation of `Vector#addAll`.

  The authors describe that this uses a lock, how does locking in Java differ from locking pthreads in C? Does the `Vector` class have anything *related* to threading in its imports? Try to translate the ideas in Java here (e.g., `synchronized(this)` with the corresponding pieces of locking in pthreads (e.g., the functions you call and the objects you create).

Conditions for deadlock

- Be able to convince yourself about these four conditions, specifically from the perspective of "If one of these four conditions *isn't* met, then deadlock cannot occur because…"
- We've used this word "preemption" before, but in the context of timeslices. Does the use of the word preemption here match what you understand preemption to be in terms of timeslices?

Prevention: Circular wait

- The Linux kernel source code that the authors refer to is `mm/filemap.c`.

  What the authors describe (lock ordering required by the kernel) is the **huge** block of comments at the top of this file.

- This is outside the scope of our course, but: could lock ordering by identified by automated tools? For example, could static analysis software (programs that read and understand source code) be designed to identify patterns of lock acquisition and report errors when locks aren't used in the same order? (Hint: the answer is "Yes", but only to a limited extent).
- Is total or partial ordering relevant when you only have 1 lock?

Prevention: Hold and wait

- The solution here is to basically lock lock acquisition. (Yo dawg).
- Can you convince yourself that this prevents the "hold-and-wait" problem? How does it do that?
- How would **you** design/implement/write code such that multiple locks are acquired atomically wherever you needed to acquire them? What kind of programming structures would you use for this (e.g., conditionals, loops, functions, etc)?
- If you had a set of locks that you wanted to acquire in *most* threads, but **one** thread only needed to acquire 1 lock, would it be OK for that thread to acquire the lock by itself? What if the thread needed to acquire 2 locks? (where 2 is less than the total number of locks)

Prevention: No preemption

- How exactly is the use of the `pthread_mutex_trylock` function different from locking lock acquisition in the "Hold-and-wait" section above? Don't they both accomplish the same thing?
- "ordering-robust" (this is just above the figure on pg 9). What does this mean, exactly? The authors used the terms "total ordering" and "partial ordering" before.
- "You might also notice that this approach doesn't really *add* preemption … but rather uses the trylock approach to allow a developer to back out of lock ownership (i.e., preempt their own ownership)" – voluntarily forcibly remove a lock from yourself?

- Convince yourself both that the trylock approach *does* solve the issue of deadlock, but also convince yourself that this concept of livelock can actually happen.

Prevention: Mutual exclusion

- How do you prevent deadlock from happening as a result of locking critical sections? Easy: just don't lock anything! Of course! That makes *so* much sense! :/
  - This does actually make a *little* bit of sense, specifically in this context of "lock-free" data structures: data structures that can be accessed and modified concurrently *without* the use of locks.
  - You can find some more information about lock-free programming on LWN.
- Notice that the "lock-free" concepts that the authors are describing here use the same hardware primitives that locks themselves use to guarantee atomicity (e.g., compare-and-swap). *Is* this really lock-free? Or is it just locking at an *instruction* level?
- Really be able to convince yourself that the author's solution to "locking" the `insert` function on pg 11 with the compare and swap function actually does what it's supposed to do.

Prevention: Avoidance via scheduling

- Given what we learned about locking last week by looking at musl code (this is a user-space problem), is what the authors are describing here *practical*?
  - "Unfortunately, they are only useful in very limited environments, for example, in an embedded system where one has full knowledge of the entire set of tasks that must be run and the locks that they need."

Prevention: Detect and recover

- The basic idea here is: Just don't worry about it until it happens. Don't waste your precious time figuring out what the precise order of lock acquisition is, don't waste your time with trylocks, don't waste your time with lock-free approaches, just write a bunch of locks. Uh, then also write some extra code that runs in the background and detects when other code has entered a deadlock, then interfere with those threads (e.g., preempt the lock) so that work can proceed.
  - Is this practical? Is it possible to do this within your own code? Can you think of any ideas about how to do this?

Summary

- So much of these problems come from the fact that threading is a user-space problem (remember how locks are related to scheduling?). How do you think each of the four conditions of deadlock

could be solved if locking was raised into the OS level, where the OS regains control of the CPU (preemptively) and can inspect the state of running programs and threads?