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 27: Thread API

This is an **introduction** to the `pthreads` threading API.

Note that the authors switch back and forth between using `pthread_*` and `Pthread_*` (lower- vs upper-case p). The lower-case p `pthread_*` functions are those that come in `pthread.h`, the upper-case p `Pthread_*` are wrappers that the authors have written in their support code for the book that checks the return code from `pthread_create`.

Thread Creation

Creating a thread with `pthread_create`.

The discussion on the function pointer stuff is kind of confusing, specifically about how changing the types changes the signature of `pthread_create`. To be clear: it *doesn't* change the signature of `pthread_create`, it changes how you can call it. In fact, you **must** pass a function with the signature `void *func(void*)` (a function that takes a void pointer and returns a void pointer). Look at figure 27.1 for a more concrete explanation of how to create a thread, note the signature of the function `mythread` and how `mythread` unpacks its arguments manually.

Thread Completion

`pthread_join` and `wait` are pretty similar in terms of what they do for threads and processes, but they do have one pretty significant difference. What is it? (it's something that `pthread_join` can do and `wait` *can't*) Why *can't* `wait` do that thing?

In the code listing in figure 27.2, the `main` function doesn't call `malloc` to allocate memory for `rvals`, but this code doesn't crash. Why *does* this work? Where is `malloc` called? Why can this work with threads specifically?

The authors ask a good question on pg 4, summed up as "Why shouldn't you try to return a stack-allocated variable from a thread?" To add to their question: is this any different from returning a stack-allocated `struct` from a function? Why or why not?

Locks

Remember that mutual exclusion stuff from last chapter? Yeah, here's how we're going to do it.

In terms of design, why do you think the `pthread` library has two ways to initialize a mutex (i.e., `PTHREAD_MUTEX_INITIALIZER` and `pthread_mutex_init`)?

The code listed on pg 6 giving an example of how you might use a `pthread_mutex_t` is kind of misleading in the way it's written. Specifically, it's implying that `lock` is a stack-allocated variable (so it's in thread-local storage). Do you think that locking like this would work if each thread has its own 'lock' object? Where *should* such a lock go?

What's the main differences between the three variants of acquiring a lock: `lock`, `trylock`, and `timedlock`? Check out the man pages for each (you may need to look on aviary itself, or, you know, refer to our friend Google and ask "`man page pthread_mutex_lock`").

Condition variables

This looks *awfully* similar to an idea that we *briefly* saw before in terms of processes: the idea of "signaling". How is this different? *Is* this different beyond different functions and threads vs processes?

On pg 8 in the discussion about `pthread_cond_wait`, the authors are describing **a lot** of stuff going on behind the scenes. Convince yourself about what's happening here in terms of locks being released and acquired, and when that might be happening in terms of what happens when `pthread_cond_wait` is called and is returned from.

Compiling and Running

Weirdly, this is a really important part of using `pthreads`: being able to *compile* code that contains `pthreads`. Write a simple `pthread` program (even if it's just the first example the authors give in this chapter) and compile it on aviary.

Summary

Take special note of the `man` page reference here: they're passing the `-k` option, giving you the power to search through `man` pages by topic. Try running `man man` to get an idea of what kind of options `man` has.