

## Contents

<b>About these reading notes</b>	<b>2</b>
<b>Chapter 5: Interlude: Process API</b>	<b>2</b>
The <code>fork()</code> system call . . . . .	2
The <code>wait()</code> system call . . . . .	3
The <code>exec()</code> system call . . . . .	3
Why? Motivating the API . . . . .	3
Process Control and Users . . . . .	4
Useful tools . . . . .	4
Summary . . . . .	4

## 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 5: Interlude: Process API

We're discussing specifically some system calls that are used to create new processes. Specifically, we're looking at `fork`, `exec`, and `wait` (at least).

### The `fork()` system call

Notice that this is **explicitly** referred to as a “system call” now.

The book suggests typing in and running the program. Do it! Seriously!

Surprise! `fork` *forks* into two different processes. This is **supremely weird**.

Based on what you read about processes in chapter 4, what do you think the OS has to *do* when `fork()` is called?

The order of operations of the messages that are printed by the code isn't deterministic, you can run it multiple times and possibly get different results each time.

Thinking back to the description of states of processes, what do you think that means about how the state of the calling process changes? What does that mean about the state of the *newly created* process?

There's some foreshadowing to concurrency and scheduling.

### The `wait()` system call

The `wait` system call lets a process wait for it's children to complete.

What's the difference between `wait` and `waitpid`? Check out the manual pages for each! Note that you might have to run `man 2 wait` to get the manual page for the `wait` system call and not a shell equivalent.

What is a process *doing* when it's waiting for children? What state might it be in?

### The `exec()` system call

Oh boy. This is an entire *family* of functions. They said that `fork` was weird, but I think this one is weirder, mainly because by calling it you're asking to **replace** your code with something else.

Based on this idea of `fork` and `exec`, what exactly do you think a [shell](#) is doing when you're entering commands? A shell is the name for the program that's printing out the prompt when you log in to a remote system.

So `exec` is basically loading up a different program (by name) and replacing the code that *was* running with the code for the specified program. Weird. `fork` is "normal" to me in that it's a clone, `exec` is like switching out bodies.

### Why? Motivating the API

The shell stuff is actually then discussed in this part of the chapter.

This is cool, because it's giving an example of the redirect operator, and it's kind of showing the idea that there's code running behind the scenes to set up the file for standard output et al.

Figure 5.4 *really* drives home this idea!

## Process Control and Users

Creating and waiting are not the *only* system calls on the block related to processes, there's also `kill`.

The main point here, though, I think is that the OS is restricting who can interact with processes based on *user* accounts.

## Useful tools

Here we're listing tools like `top`, `ps`, `kill`, etc

## Summary

This refers to [a system call called `spawn`](#). The `spawn` system call is kind of like `fork/exec`, but for other operating systems like Windows.