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 7: Scheduling: Introduction

You *might* want to go back and remind yourself about what a process *is* and what the OS does when it's switching between processes, as the book recommends here.

In short, we have things that are lining up and wanting to work, how do we decide which of these things actually gets to go next?

7.1 Workload assumptions

The assumptions that are stated here are *seriously* unrealistic. Be able to convince yourself that each of these are unrealistic, and be able to explain why they are unrealistic. (e.g., Why is the workload assumption that all jobs only use the CPU an unrealistic assumption? Why *is* knowing how long a process is going to take unrealistic? We can `time` a process, isn't it going to take the same amount of time the next time we run it? Why or why not?)

7.2 Scheduling metrics

Workload assumptions let us constrain the scope of our problem, it's an abstraction. We don't have to think about all of the nitty gritty details of what's going on within a particular process or job.

Workload assumptions actually let us try stuff out, but we can't just try stuff out willy-nilly, we need to actually be able to **measure** stuff.

Turnaround time is *a* metric that we can use to compare policies.

Can a process measure its own turnaround time? That is, can a process know when it arrived? Our unrealistic assumption that all processes arrive at the same time (at time 0) lets us assert that the process arrives at time 0, but in a more *realistic* situation, do you think that a process *can* know when it "arrived"?

Further, if a process itself can't know when it arrived, can *you* (i.e., you, the user that's actually physically hitting the ENTER key on your keyboard) know when a process has arrived?

7.3 First In, First Out (FIFO)

Look, it's a queue again. Dang, they show up everywhere in an OS.

FIFO is literally "the first process that arrives gets to go first, the second process that arrives gets to go next". Remember that we're running to completion here, all jobs take the same amount of time, and that they all arrive "at the same time".

The authors just finished talking about turnaround time, but now we switch to *average* turnaround time. Why do you think they did that? *Can* you reason about the performance of a scheduling policy if you **only** look at turnaround time for individual processes?

This is more of a statistics question than an OS question, but is the average turnaround time a reasonable way to reduce the turnaround time for all jobs and then compare the performance of a scheduling policy? Can you think of any other ways to reduce turnaround time to compare scheduling policies?

Considering the problem here with the convoy effect, how can you reorder the jobs in figure 7.2 to minimize average turnaround time?

I really like this analogy of grocery store checkouts. The analogy is currently more like a small corner store, though: one CPU, one checkout line. How might this analogy change if you go to a big store (e.g., Superstore, Safeway, or Sobeys)? How many "CPUs" do these big box stores have? Are any of the "CPUs" specialized in any way?

7.4 Shortest job first

Hey look, here's a solution to the convoy effect. Do you think this solution still works when you relax the assumptions/constraints on jobs? What if they don't all arrive at the same time? What if you don't know in advance how long a job is going to take?

> "no proofs allowed".

7.5 Shortest Time-to-Completion First (STCF)

Shortest Time-to-Completion definitely solves this issue with the non-preemptive shortest job first. Can you think of any circumstances where process A in the example in figure 7.5 (pg 6) *never* gets to run?

7.6 A New Metric: Response Time

Are response time and turnaround time related to one another? Do you think that optimizing for one might affect the other? In other words: if I try to optimize response time, am I (in general) going to change the turnaround time?

7.7 Round Robin

Oh no! Minimizing response time made turnaround time really bad. Can you think of any ways to adjust how round robin works to try bring down both? Or do you think that response time and turnaround time affect each other like a see-saw (push one side down, the other goes up)?

Our current abstraction only considers jobs that only use the CPU. What other kinds of things do *real* processes do that might help better inform when a process should be "switched off" instead of just using time slices like with round robin?

7.8 Incorporating I/O

Imagine, for a second, that we incorporate the idea of I/O into plain Round Robin. The scheduling policy is going to schedule processes that are currently blocked on I/O — what would the CPU be doing when a process blocked on I/O gets scheduled by the Round Robin scheduling policy?

The authors sort of omit that adding I/O is *effectively* mixing preemptive and cooperative. What should a Round Robin scheduling policy do when a process invokes an I/O system call before its time slice is over?

The authors are describing this very abstractly, so try to think about this a little bit more concretely: how would you implement a Round Robin scheduling policy that incorporates I/O? Think about which data structure you would choose to manage your workload, and think about what information you need to keep about each job in the workload as it executes (e.g., is it waiting on an I/O operation?). Expand on

this idea: what other kinds of properties of a process or job could you consider when implementing your scheduling policy?

7.9 No More Oracle

Oh no, now I feel like we're taking all the stops out.

"A general purpose OS … knows very little about the length of each job". Fair enough. *Can* it? How could it know about how long a job might take? Go beyond this: what kinds of things *does* an OS know about that can help inform it about scheduling, other than I/O and time to complete?