

## Contents

|   |          |
|---|----------|
| <b>About these reading notes</b>                  | <b>2</b> |
| <b>Chapter 18: Paging: Introduction</b>           | <b>2</b> |
| 18.1 A simple example and overview . . . . .      | 2        |
| 18.2 Where are page tables stored? . . . . .      | 4        |
| 18.3 What's actually in the page table? . . . . . | 4        |
| 18.4 Paging: also too slow . . . . .              | 4        |
| 18.5 A memory trace . . . . .                     | 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 18: Paging: Introduction

- On pg 1:

... it may be worth considering the second approach: to chop up space into *fixed-size* pieces.

that sounds almost the same as what we were doing originally with allocating the full address space. What was wrong with that? How is this going to fix things for us now?

- “The Atlas” that the authors are referring to is here:

[https://en.wikipedia.org/wiki/Atlas/\\_\(computer\)](https://en.wikipedia.org/wiki/Atlas/_(computer))

- Make sure you're keeping the terminology straight here: a **page** is a small piece of the virtual address space for a process, a **page frame** is where that page might go in physical memory. Just like in real life, you've got a **picture** and you put it into a **picture frame**.

### 18.1 A simple example and overview

- Looking at figure 18.1: before you get too far into this chapter, try to think about what a virtual address is (a number), and how you might get information from that virtual address to figure out which page an address belongs to.

- How many bits might you need to represent 4 pages?

- On pg 2:

We won't for example make assumptions about the direction the heap and stack grow and how they are used.

but, but, but, our programs still *have* a heap and a stack! How can the underlying virtual memory implementation not *care* about those things?

- On pg 3:

To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a *per-process* data structure known as a page table."

In terms of what you know about processes, where do you think this page table might be stored?

- The answer to our first question above (how many bits) is at the bottom of page 3.
- Something important to note about the discussion here is that the virtual address space is smaller than the physical address space again (we have 8 page frames and 4 pages of virtual memory per process). This is **explicitly not required** though, the virtual address space can be as big or bigger than the physical size of memory.
  - This idea is reinforced in the middle of page 4, where the number of bits in the virtual address is smaller than the number of bits in a physical address.
- Be able to convince yourself that the VPN (on the bottom of pg 3) can uniquely map each address to a page in the virtual address space.
- Given the number of bits in a VPN (say we have 5 bits for VPN), how many pages would we have in virtual memory?
- Before you get much farther than page 4, try to think about this: how much of this address translation work is the OS responsible for? Hardware?
- On page 5:

Note the offset stays the same (i.e., it is not translated), because the offset just tells us which byte *within* the page we want.

Why isn't the offset translated? What would it be translated to if you did translate it?

## 18.2 Where are page tables stored?

- Near the top of page 6:

Instead, we store the page table for each process in *memory* somewhere.

— wait a minute. We store the table that has the mappings between virtual and physical memory *in memory*?

## 18.3 What's actually in the page table?

- We're starting to talk about this idea of valid bits here. Try drawing a diagram that shows the "page table", and how it might correspond to physical memory.
  - What might that diagram look like?
- If you have already taken COMP 3370, some of this might look *awfully* familiar to you, both in the context of paging **and** in the context of caching.
- If you're actually interested in taking a look at the Intel Architecture Manuals, you can find them here:

<https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html>

Note that these are pointing at PDFs that are bigger than textbooks, weighing in at about 60+MB a piece.

- The aside "why no valid bit?" is a pretty nice example of hardware and the OS *working together* to do something better than the hardware might be able to provide alone.

## 18.4 Paging: also too slow

- Try to think about how many memory accesses are actually required to do this lookup, considering that we have to both translate the virtual address to a physical address, then actually go to that physical address to get the value. Add on top of that the idea that we're also needing to *fetch an instruction*, how many additional accesses would that take?

## 18.5 A memory trace

- At the top of page 10, the authors refer to `obj dump` and `otool`, try them out on a program you compile!
- This section (18.5) is slowly stepping through the answer to the question posed above for 18.4, use this to confirm what you thought about how many memory accesses this would take.