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 16: Segmentation

- "Segmentation". That's a word you've seen before. Uh, a lot probably. Where have you seen that word before?
- How do we support virtual address spaces that are larger than physical memory? We had questions like this from the last couple of chapters, now we're going to figure out (a less than ideal and currently unused) solution to this problem!

16.1 Segmentation: Generalized Base/Bounds

- Basic idea: *don't* allocate the full address space, allocate the space you need for the different known "segments" of memory that your process needs (like code, heap, and stack), have each of those segments be relocatable by adding hardware support for n different pairs of base/bounds registers for n different segments.
    - Note that these segments are **explicitly** not the same size.
    - What kind of problems can you predict happening with segments that differ in size?
- The hardware support here seems to know a lot about what's going on with a program in that it assumes that the program uses a heap. What if your program *doesn't* use dynamic memory

allocation? What do you think the OS will do in terms of allocating a segment for heap if you never try to use dynamic memory allocation?

  - Extending on this, figure 16.3 shows 3 sets of registers. How many sets of registers is reasonable? Is it reasonable to assume that 3 will always be enough? Is it reasonable to assume that all operating systems will forever use this idea of 3 different code regions?

- When we allocated the full virtual address space for a process, we had this issue of internal fragmentation. One idea we saw from our classmates to solve the problem was to put code/heap/stack regions into the middle of the virtual address space and grow outwards. This was a great idea, but internal fragmentation still remained because we were allocating the full address space.

  - Does the idea of a sparse address space resemble the proposed solution above? Before you read on past pg 3, *does* sparse address spaces solve the issue of internal fragmentation?

- On the top of pg 4 is an **extraordinarily important aside**.

  - Read it first, then watch this video: https://www.youtube.com/watch?v=03QuygM0YB8

- Before you get to section 16.2 “Which segment are we referring to”, starting at the top of page 4, try to predict: given an address, how *might* you figure out which base and bounds registers to use? Would the hardware test them all? Something else?

16.2 Which segment are we referring to?

- If you have taken COMP 3370 (or some equivalent course in a different faculty), this process of splitting an address up into parts is directly related to cache memory implementations (e.g., the tag, offset, index, etc).
- If we use the top 2 bits to indicate which segment we’re referring to, how many segments could we have? What if we used 3 bits? n bits?
- In short: the mapping of segment to base/bounds register pairs is something the OS and hardware would have to agree on. The hardware doesn’t know the difference (necessarily) between the “heap” and the “stack”, it just knows that there are different segments. The OS would have to keep track of the semantic meaning of each of the segments, and configure the hardware such that “01” (or whatever it chooses) should use one pair of base/bounds registers, “00” (or whatever it chooses) should use another pair of base/bounds registers, etc
- On the bottom of pg 5: “Some systems put code in the same segment as the heap and thus use only one bit to select which segment to use”.

  - This is actually what we saw when we looked at the manual page for `brk` and `sbrk`, the "data segment" being a combined code segment and heap segment.
  - Can you think of any issues with this solution? (hint: should the heap be executable? code?)
  - Why would they choose the heap and not the stack? (hint: look at the direction that each of those segments "grows")

- On the bottom of pg 5

  > Specifically, each segment is limited to a *maximum size*, which in our example is 4KB

  Can you convince yourself based on how these addresses are separated into "segment" and "offset" that this is true?

  - In other words: how many addresses can 12 bits (the number of bits in the segment) refer to?

16.3 What about the stack?

- OK, now things are going to get *weird*.
  - The arithmetic is *straightforward* (not easy, straightforward), but your mind might be throwing exceptions while you're reading it. Take the time to make sure you get this idea of computing negative offsets.
  - I found this arithmetic really confusing the first time I read it, I didn't quite see why we had to do this offset - size thing to get a negative offset.
  - The base for the stack segment in physical memory (in figure 16.2) is at 28KB, but if you imagine the stack as an array:
    - the 0th stack frame (`main`) is at 28KB $- 0 \times$ sizeof(stack frame),
    - the 1st stack frame is at 28KB $- 1 \times$ sizeof(stack frame),
    - the 2nd stack frame is at 28KB $- 2 \times$ sizeof(stack frame),
    - the 3rd stack frame is at 28KB $- 3 \times$ sizeof(stack frame), …
  - The $offset − sizeof(stack segment)" to get a negative offset is effectively doing the same arithmetic.

- Serious question: Why don't we all just agree to let the stack grow forwards like the heap?

16.4 Support for sharing

- What kind of code memory do you think would be shared *between* processes?
- Why do we have to *add* bits (protection bits) to *share* memory? Isn't that counterintuitive?

- The authors are again using the terms protection and isolation here. Do they mean the same thing as we saw last time?

16.5 Fine-grained vs coarse-grained segmentation

- OK, so this is addressing a question that we saw earlier.
- Remember this term "segment table", the idea is going to come back again *real soon*.
- Why on earth would we even *want* fine-grained segmentation?

  > … expected a *compiler* to chop code and data into separate segments which the OS and hardware would then support.

  What kind of maniac would come up with a system like that? Compilers supporting OS-specific ideas?

16.6 OS Support

- Yay, we solved internal fragmentation.
- Surprise: when we have multiple segments instead of a fully-allocated address space, the OS still has to do many of the same things!
- Libraries performing memory allocation:

  > the memory-allocation library will perform a system call to grow the heap

  When you call `malloc()`, you're almost always using the glibc implementation of `malloc()`, but musl has its own allocator, and there are also dynamic memory allocators that are entirely separate from standard libraries, like jemalloc and mimalloc.
- Check out `man sbrk`, this is a real thing. It also kind of means that **you** could write your own memory allocator.
- We talked about external fragmentation last week, now we're going to see it rear it's ugly head.
  - Is external fragmentation the same thing that happens with file systems on disk?
- Compacting memory:

  > One solution to this would be to **compact** physical memory by rearranging the existing segments.

Depending on when you took COMP 2160, this is literally the same word we used to describe the feature of our compacting garbage collector that would shuffle allocated memory back to the beginning of the array. In fact, this *entire idea* of virtual addresses aligns *very closely* with that assignment, specifically about using an ID or tag instead of an actual address to interact with the memory allocator.

- "Compaction also (ironically) makes requests to grow existing segments hard to serve"
  - *Is* this ironic? https://www.youtube.com/watch?v=Jne9t8sHpUc
  - Why is this true? Why does compacting memory segments make growing segments harder?
- Tracking free memory sounds like a "hard problem" ™; "There are literally hundreds of approaches that people have taken". Maybe that means it's easy? How would *you* track free memory and decide which free space you should allocate a request in? Does this remind you of any algorithms?