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 17: Free-Space management

- Don't let "Page 1 of 18" scare you! There are many pages with nearly full-page diagrams.
- "Let's take a look at some algorithms for free-space management", then quickly decide that it's easier to use fixed-size things (pages) and mostly forget about the idea of free-space management.

17.1 Assumptions

- Always with the assumptions.
- Note that this chapter assumes you're familiar with the memory management API (`malloc` and `free`). I don't know how you *couldn't* be at this point, but if you need a refresher, you can read chapter 14.
- "Note the implementation of this interface: the user, when freeing the space, does not inform the library of its size" — …? Think about that for a second. When using `malloc` and `free`, have you ever *cared* about telling free how much space you're freeing?
- Dynamic memory allocation is kind of a split responsibility between the standard library and the OS. In fact, *all* of the things this chapter talks about apply both to the OS's management of free memory (physical memory) and your dynamic allocator of choice's free memory (virtual memory within the heap).

  - Why not just make this an OS responsibility? Why *aren't* `malloc` and `free` system calls?

- "Allocators could of course also have the problem of **internal fragmentation**" — do you think your allocator cares about internal fragmentation? What even is internal fragmentation to a memory allocator library? *Should* a memory allocator library care about internal fragmentation?
- "We'll also assume that once memory is handed out to a client, it cannot be relocated to another location in memory" ← note that this is **specifically** talking about "relocated to another *virtual* address", the OS is free to move the segment in physical memory wherever it wants, as long as it correctly updates the base/bounds registers for that segment.
- Again, check out `man sbrk` and think about how you could write your own memory allocator using this system call.

17.2 Low-level mechanisms

Splitting and coalescing

- A free list is literally a list, and you can see a visual depiction of that on the bottom of pg 3.
  - This diagram alone looks **extraordinarily similar** to an assignment in COMP 2160. Ask me (or anyone else in the class) about it if you don't recognize this image.
- This idea of splitting and coalescing is pretty neat, particularly in terms of how the list changes. The authors *don't* include what the physical memory allocation looks like corresponding to the list as regions are split and coalesced, take the time to draw out how this physical memory changes as the list changes.

Tracking the size of allocated regions

- OK, this idea is actually super cool. Not lame cool, but **super** cool. This same idea is actually used in some C string management libraries ("Cello", a C library liberally uses this pattern: http://libcello.org/).

  How would *you* use this idea to build a "string library" in C?
- Why would a library that uses this approach check and confirm a "magic number"? (hint: what happens when you `free` a pointer that you didn't get back from `malloc`? What would `free` even *do* if you give it a pointer you didn't get back from `malloc`?)

Embedding a free list

- "In a more typical list, when allocating a new node, you would just call `malloc()` .... Unfortunately, within the memory-allocation library, you can't do this!" — Think about this for a second,

and make sure that you understand *why* this is true. Why *can't* a dynamic allocation library call `malloc()`?

- `mmap()`? What's that? Check out `man mmap`.
  - What does this have to do with the heap?
- What the heck is going on here? This list doesn't look anything like what I imagined a list to look like! Where are the circles?!
  - Seriously, though: why is this list embedded within the free space? Why wouldn't they allocate a chunk at the beginning and use *that* for the list?
  - What they're doing here is directly required by this idea that the dynamic memory allocator doesn't have its own dynamic memory allocator to fall back on (insert Xzibit here).
- The authors describe figure 17.7 as "a big mess". I would agree with them. *When* should a memory allocator like this decide to coalesce memory?

Growing the heap

- Again, check out this system call `sbrk`. What does it do? What does `sbrk` *stand* for as an acronym or abbreviation?
- "To service the `sbrk` request, the OS finds free physical pages"… what does this have to do with *segments*?

17.3 Basic strategies

- As you're reading about each of these different policies, think about this: are these in any way related to the *scheduling* policies that we looked at? Thinking about time and space this way is a little bit weird, but definitely think about it.
- For each of the policies, try to imagine, or, you know, draw a picture of, the worst-case scenario for the policy.
- Is there a "best" policy of the ones that are described here? What does "best" even mean in this context? With scheduling policies we had some pretty good metrics, what are the metrics here?
- These policies are all fairly straightforward, so outside of "Worst fit", there aren't any questions about them.

Best fit

Worst fit

- Wait, what? *Worst* fit? Doesn't "worst" imply "bad"?
- Worst fit is related to best fit in that it's sort of the opposite. Are these policies similar in any other ways?

First fit

Next fit

Examples

17.4 Other approaches

Segregated lists

- How would a dynamic memory allocator know what a "popular-sized" request is? Think about the memory API (the interface; how it would get that information), and think about what the memory allocator could use that information in its implementation.
- Is this idea effectively the same as having fixed size blocks in terms of what we saw with virtual memory implementations?
- "when the kernel boots up, it allocates a number of **object caches** for kernel objects that are likely to be requested frequently" — that sounds *awfully familiar*. What is this idea similar to?
- Do you think this idea is at all related to the MLFQ scheduling policy? How is it related?

Buddy allocation

- https://www.youtube.com/watch?v=9jyCfRHumHU
- This whole idea of dividing spaces by two is an idea that comes up **constantly** in CS. Where's another place you've seen this?
  - One place you may never have seen this is in two dimensions: Quadtrees, and *three* dimensions: Octrees
- "note that this scheme can suffer from **internal fragmentation**, as you are only allowed to give out power-of-two-sized blocks." Do the other allocation policies that you've seen suffer from this problem of internal fragmentation? Why or why not?
- What kind of allocation patterns do you think would cause this policy to behave poorly? Specifically, given a 64kb chunk, what general kind of size requests would cause this policy to behave poorly?

Other ideas

- "Failing that, read about how the glibc allocator works"; you're more than welcome to do that, but know in advance that glibc source code is notoriously impenetrable to read.
  - You could also read about jemalloc
  - Or, uh, many others

17.5 Summary

- A lot of this sounds like it's responsibility of a dynamic memory allocator library that's running in user space. What does this have to do with operating systems?