MEM3-MALLOC: 181-210

181 [T / F] I now know why in the example on slide 2, there's a 32-byte distance between the allocations even though I only malloc(4 bytes).

182 [T / F] The logical address boundaries (the BASE addresses) of the code, heap, and stacks are set together by the OS, CPU, and compiler architects. The malloc library has a macro (a constant variable such as "HEAP_BASE") that is predefined based on the OS/CPU architecture.

183 [T / F] A pointer value returned by malloc() is just a virtual value crafted by the malloc library. Also, a pointer value typically looks "random" (*e.g.*, 0x0800C840). This is because users call malloc() of different sizes and there are header/footer bytes in every malloc allocation.

184 [T / F] While the OS attempts to reduce memory fragmentation (wasted memory) within its own memory management, it cannot prevent wasted memory within your program (*e.g.*, you malloc() a lot of bytes but never use them).

185 [T / F] In our 32-bit lecture, a word is 32 bits (4 bytes).

186 [T / F] The major fragmentation problem that malloc() library faces is external fragmentation.

187 [T / F] After you malloc() a structure or object and get the memory address, the malloc() library is allowed to shuffle their positions.

188 [T / F] The "size" value in the block's header is the *total* size of the header+footer (8 bytes; in 32-bit addressing) *and* the payload (and perhaps some paddings). In other words, the "size" represents *the size of the block*, not just the payload.

189 [T / F] The best-fit policy will cause big leftover blocks.

190 [T / F] The worst-fit policy is slow as it needs to scan all the blocks to find the worst fit.

191 [T / F] The next-fit policy is an optimized version of the first-fit policy. The next-fit policy does not need to scan from the beginning.

192 [T / F] In the malloc library, merging is all about knowing where your and left/right neighbor's heads and feet are with only one information "p". From "p", you can reach all other information such as block sizes n, m1, m2, and the allocation bits.

193 [T / F] (Assume a 32-bit addressing), given the current pointer "p" in "free(p)", your foot is at $p + n - 4$. (n is the size of your block).

194 [T / F] Your left neighbor's head is at $p - m1 + 4$. ($m1$ is your left neighbor's block size).

195 [T / F] Your right neighbor's foot is at $p + n + m2 - 4$. ($m2$ is your right neighbor's block size).

196 [T / F] An "implicit" list contains pointers to allocated and free blocks.

197 [T / F] Explicit list is a list of direct pointers to all the blocks (both allocated and free blocks).

198 [T / F] When calling malloc(), we don't need to traverse through allocated blocks.

199 [T / F] When malloc library uses an explicit list, it doesn't need the "implicit" list anymore.

200 [T / F] free() is guarateed to have a O(1) complexity, but malloc() can be unluckily O(N) when no free blocks fit the new large allocation.

201 [T / F] In a 32-bit explicit list, the smallest block size possible is 12 bytes: header (4 bytes), payload (4), and footer (4).

202 [T / F] In a 64-bit addressing with explicit list, the smallest block size possible is 16 bytes: header (4 bytes), next (4), prev (4) and footer (4).

203 [T / F] Mallocing super tiny allocations too many times, *e.g.* malloc(1-byte), leads to heavy internal fragmentation.

204 [T / F] In explicit list, the "next" pointer contains the address of the first byte of the header of the next free block.

205 [T / F] In explicit list, the "prev" pointer contains the address of the first byte of the header of the previous free block.

206 [T / F] I have mastered the "C Pointer Primer" example (the "p" and "pp" example with all the *, **, & prefixes. This knowledge is important for updating the content of the "root", "next" and "prev" memory lines.

207 [T / F] Suppose a 32-bit addressing, and I call free(p) where the p's block will be the new root. This is a correct C code to update the root's content: `root = p;`

208 [T / F] (Referring back to mem2-seg lecture, slide 17). Say, you have a 16-bit segment offset, while in the class we simplify by saying that you can malloc 2^{16} bytes, in reality you can't because there are malloc header bytes.

209 [T / F] I want to have an early Thanksgiving break! No class on Friday! But I promise to watch sy1 and sy2 videos before Wednesday.

210 [T / F] The OS manages the entire physical memory (DRAM) and the malloc() library just manages the heap area of a process. One good analogy to think about these different levens of memory management is like this: The ovearhead bin is the physical memory (DRAM), and the flight attendant is the OS. A passenger is a process. A passenger bring several carry-on bags (code, data, stack, and heap segments). The flight attendant manages the overhead bin, storing and removing carry-on bags as the passengers board/deboard the plane (as processes start and exit). Now, you are a passenger (you are a process). One bag that you have (imagine a suitcase) is the heap segment. Inside this bag (suitcase), there will be many chocolate bars. When you buy a chocolate bar (*i.e.*, calls malloc()), you put it in your suitcase in a fixed position that *cannot* be moved around. When you want to eat a specific chocolate bar (*i.e.*, calls free()), you remove it from your suitcase. How you add/remove a chocolate bar in your suitcase is basically how malloc() library manages the heap. Over time in your suitcase you will have external fragmentation (gaps/empty spaces between the chocolate bars that are still there). Next, you buy a new big chocolate bar that does not fit any of the gaps. What happens now? Well you will call the flight attendant (*i.e.*, the malloc() library make an sbrk() system call to the OS). The flight attendant has a magical skill that will make your suitcase grow larger as long as there is room in the overhead bin next to your suitcase (*i.e.*, the OS will increase the "bound" of your heap segment). This is what happened inside the malloc library with *segmentation*. Today, with *paging* (which you should not worry about, but just FYI), the heap segment is broken into several fixed-size pages (several fixed-sized suitcases). If your suitcase is full, the flight attending will give you another small fixed-size suitcase where you can put your new chocolate bars. Wow .. what a long analogy! Hope helps :) — 210