CS 61 Problem Set 1

Grading notes

External metadata implementation (phases 1-3):

This implementation of malloc and free relies on three key data structures:

- `allocated_blocks`, a map containing all currently allocated blocks. Each element in the map is ordered by their starting position in memory (i.e., their distance from `&default_buffer.buffer[0]` stored as a `size_t` variable; this is the key), and also contains information about the size of the allocation (`sz`), the size of the padding at the end of the block to ensure the free block immediately after it will be correctly aligned if allocated (`alignment_sz`), and the line in the source code where the block was allocated (for error messages; `line`). The latter three values are packaged into the `struct` called `metadata` for ease of use.
- `free_blocks,` a map containing all currently free blocks. The map is ordered by the starting position of each free block (the key value) and each element stores the size of a the free block it corresponds to.
- `freed,` a set of `void*` pointers that tracks already-freed pointers. It is used to detect double frees.

`free_blocks` makes it easy for malloc to find a sufficiently large block for allocation. We simply traverse through the block, stopping only if we see a block with big enough size. This is $O(F)$ in the worst case, where F is the number of freed blocks. If a large enough block is found, then we break it up into the allocated portion and a remainder free portion, ensuring the latter starts in an aligned position. We also take care of some other implementation details like marking the region (at most 16 bytes) immediately following an allocated block with `'!'` for boundary write detection.

Once we find an appropriate block, we just need to insert it into `allocated_blocks` and update relevant statistics. We also erase its starting pointer from `freed` if applicable. This takes $O(\log N)$ time in the worst case, where N is the number of allocation blocks. Thus, the overall complexity is $O(F + \log N) = O(F)$ in the worst case.

The bulk of the work for `free` is memory error detection. We leverage `allocated_blocks` to check for pointer validity and `freed` to check for double frees. The metadata in `allocated_blocks` is crucial here for outputting the correct error message. This step takes $O(\log N)$ in the worst case.

If no error is detected, then we proceed to freeing the block. This is simple: we just need to remove the block with the given starting position from `allocated_blocks` and insert it into `freed_blocks`. This takes $O(\log N)$ time (worst case). We also need to update `freed` for error detection and coalesce adjacent blocks of memory if applicable. Note that we only need to consider the free blocks immediately to the left and right of the current freed block; since we are coalescing every time we free a block. This step again takes $O(\log F)$ time. Therefore, the time complexity for `free` is $O(\log N)$ in the worst case.

Extra credit attempted

None in this Github push, but will be in future updates.