CS 24: Introduction to Computing Systems

`malloc` Trade-Offs

When implementing your own `malloc` library, there are a *ton* of **design decisions** to make. This is not a complete list, but it contains the most important ones.

Fundamental Decisions

The *most influential* decision in your allocator is the type of free list you choose. There are four main types:

- **Implicit Free List.** Uses the size in the header to link together all blocks in memory.
- **Explicit Free List.** Splits the blocks into two types: allocated and free. Allocated blocks are identical to in an implicit free list. Free blocks store explicit `prev` and `next` pointers. This allows us to only search through free blocks for a fit rather than all of memory.
- **Segregated Free List.** Uses an array of explicit free lists split up by size (usually uses powers of two). It gains all the benefits of an explicit list while limiting the search to blocks likely to succeed.
- **Balanced Free Tree.** Uses some kind of balanced binary search tree to store free blocks.

After you've chosen a type of free list, you will need to choose a strategy to *search* that list. There are two main strategies:

- **First Fit.** Search through the list until a valid fit is found. Use it.
- **Best Fit.** Search through the whole list keeping track of the smallest block that is big enough. Use that optimal block at the end.

You will also need to choose a strategy to *add* to the free list. There are several choices here as well:

- **FIFO.** Insert freed block at the beginning of the free list.
- **LIFO.** Insert freed block at the end of the free list.
- **Address Ordered.** Insert freed blocks so that free list blocks are always in address order.

Splitting and Coalescing

Some form of splitting and coalescing blocks is necessary to have a performant version of `malloc`. While there is really only one version of splitting, there are several choices to make when coalescing.

Splitting

Before returning a block from `malloc`, it is essential that you break your block up into two pieces: one to return, one to retain in the free list. This avoids giving the user much more memory than necessary. Splitting is essential to any implementation of `malloc`.

Coalescing

While strictly not necessary, a **prologue** and an **epilogue** will make implementing coalescing simpler. These are special blocks with no payload put at the very beginning and very end of the heap. They serve as markers in either direction.

There are two places that we could coalesce, and, as with everything else, there are trade-offs between them. One strategy is to always coalesce whenever we `free`. This way, there will be no unnecessary fragmentation. This strategy more or less requires you to have *footers* in addition to your *headers*, because we must coalese a block with **both** the one before it and the one after it. The other strategy is a bit more open-ended (i.e., you could do it with footers or not). Since we have to search the free list anyway, we can coalesce *as we* `malloc`.

This strategy will lead to a bit more fragmentation, in general, in the case where you terminate your search before the end of the list.

Some Optimizations

Here are two optimizations you might consider implementing. There are many more, but this is intended to be a start:

- As we discussed in class, there is no reason to use two words per header/footer. They can be compressed into a single one because of the alignment restriction.
- If we keep an upper (or lower) bound on the free list, we will be able to avoid a search entirely for blocks outside this range.