Lab 3.2

Out: Friday, February 18th

1 What you'll do

In Lab 3.1, you've allocated the same amount of memory to the user regardless of how much was requested. This means you'll almost always be wasting memory every time you assign a block. The solution to this problem is to dynamically break blocks apart to form blocks that are exactly the right size. When they are freed, contiguous blocks will then be merged back together. This is very similar to the way in which real operating systems implement malloc and free.

2 Task 1: Breaking Blocks

Instead of formatting the pool of memory into a series of uniformly sized blocks, you will start with just one block that is the same size as the whole pool of memory. As memory is requested from the pool, blocks will be broken off and allocated. Keep in mind that you will not always need to break a block. If a block happens to be only slightly bigger than the requested allocation, there may not be enough space left in the block to store the header for another block. If that happens, you should assign the whole block and not break it.

Since blocks will no longer all be the same size, you won't always be able to simply use the first block on the free list when the user requests an allocation. Instead, you'll have to search the free list for a block that is big enough to accommodate the request.[1] If no such block exists, you should return `NULL`.

Task: Fill in the method called `break_block`, and modify your code so that instead of having uniform size blocks, the blocks can be arbitrary sized to match the amount of memory that is actually requested. Instead of calling `format_uniform`, you should fill in and call `format_onebig`. This function should set up a single block that is the size of the entire available memory. Test your code by modifying your `main()` function to call `malloc` a few times on various size blocks, and then use `myalloc_print` to verify that the blocks were broken properly.

[1] This algorithm is called "First Fit Allocation" and may seem like a worse option than "Best Fit Allocation" (scanning for the block that fits the requested memory most tightly). In fact, First Fit usually performs as well or better than Best Fit, and runs faster too. If you want to know why, take CS169.

3 Task 2: Merging Blocks

Now that you are breaking blocks apart to fit the requested allocations, your pool of memory may start to become fragmented as more and more blocks are created. If this process is allowed to continue, your memory allocator may not be able to accommodate requests for large blocks of memory, even when there is plenty of free space available.

The solution to this problem is to merge adjacent blocks together when they are freed. That way, any time two blocks on the free list happen to be next to each other in memory, they will be combined into one larger block.

Task: Fill in the method called `merge_blocks` and modify your code so that when `myalloc_free` is called, adjacent free blocks in memory get merged together. Test your code with the `test_merge` function.

4 You're Done!