Lab 2:
Original Heapster

Due March 9 at the **start of lecture.**

Overview

In this lab, you will implement your own dynamic memory allocator (heap manager) in C. We will use the simple (but somewhat ineffecient) *free list* system of heap allocation. You will demonstrate how to use your allocator to allocate dynamic memory for a variety of C types.

Implementation

You must follow these implementation guidelines:

1. Define a struct to represent an allocation block.
```
struct Block {
   int block_size; // # of bytes in the data section
   struct Block *next_block; // in C, you have to use “struct Block” as the type
};
```
2. Determine the size of a `Block` value using `sizeof(struct Block)` and assign it to a global const variable. We'll refer to this value as the “overhead size.”
3. Determine the size of a `void*` and save it in another const global.
4. Create a global pointer `struct Block *free_head`, which will always point to the first free block in the free list.
5. Create a function `void my_initialize_heap(int size)`, which uses `malloc` to initialize a buffer of the given `size` to use in your custom allocator. (This is the only time you can use `malloc` in the entire program.) Your global `free_head` should point to this buffer, and you should initialize the header with appropriate values for `block_size` and `next_block`.
6. Create a function `void* my_alloc(int size)`, which fills an allocation request of `size` bytes and returns a pointer to the data portion of the block used to satisfy the request.
   (a) `size` can be any positive integer value, but any block you use must have a data size that is a multiple of your `void*` size. So if a `void*` is 4 bytes, and the function is told to allocate a 2 byte block, you would actually find a block with 4 bytes of data and use that, with 2 bytes being fragmentation.
   (b) Walk the free list starting at `free_head`, looking for a block with a large enough size to fit the request. If no blocks can be found, returns 0. (null) Use the **first fit** heuristic.
   (c) Once you have found a block to fit the data size, decide whether you need to split that block.
      i. A block needs to be split if its data portion is large enough to fit the `size` being allocated, AND the excess space in the data portion is sufficient to fit another block with overhead and a minimum block size of `sizeof(void*)`.
      ii. If you cannot split the block, then you need to redirect pointers **to** the block to point to the block that follows it, as if you are removing a node from a singly linked list.
         A. WARNING: the logic for removing a node in a linked list is **different** depending on whether or not the node is the **head** of the list. Draw it out and convince yourself why you need to account for this.
      iii. If you can split the block, then find the byte location of where the new block will start, based on the location of the block you are splitting and the size of the allocation request. Initialize a

new `struct Block*` pointer to that location and assign its new `block_size`. The new block's `next_block` pointer needs to point to the same block as the `next_pointer` of the block you are splittling. Reduce the size of the original block to match the allocation request.

   (d) Return a pointer to the **data** region of the block, which is "overhead size" bytes past the start of the block. Use pointer arithmetic.

7. Create a function `void my_free(void *data)`, which deallocates a value that was allocated on the heap. The pointer will be to the **data** portion of a block; move backwards in memory to find the block's overhead information, and then link it into the free list.

Testing Your Code

Test your code thoroughly by allocating values of various types. You should write (and turn in) your own testing main, which **at least** includes the following tests, each a separate branch of main so that only one runs per execution of the program:

1. Allocate an `int`; print the address of the returned pointer. Free the `int`, then allocate another `int` and print its address. The addresses should be the same.
2. Allocate two `int`s and print their addresses; they should be exactly the size of your overhead plus the size of an integer apart.
3. Allocate three `int`s and print their addresses, then free the second of the three. Allocate a `double` and print its address; verify that the address is correct. Allocate another `int` and print its address; verify that the address is the same as the int that you freed.
4. Allocate one `char`, then allocate one `int`, and print their addresses. They should be exactly the same distance apart as in test #2.
5. Allocate space for a 100-element `int` array, then for one more `int` value. Verify that the address of the `int` value is 100 * `sizeof(int)` + the size of your header after the array's address. Free the array. Verify that the `int`'s address and value has not changed.

424H Requirements

If you are enrolled in CECS 424H, then you must complete the following requirements in addition to the requirements above.

1. Implement a **coalescing heap** by modifying your `my_allocate` and `my_free` methods as such:
   (a) You will no longer take the easy way out when freeing a block (which is to make the freed block the new head of the free list). Instead you will maintain the links in the free list so that the blocks are linked in order of increasing memory address. Visually, the free head will link to the "first" unallocated block from left to right; each other free block will have a `next_block` pointer leading to the "next" free block to the right.
   (b) This means that freeing a node is a little more difficult, but it gives the opportunity to **coalesce** adjacent free blocks. When freeing a block, you will need to walk the free list, comparing `next_block` pointers until you find two blocks between which the newly-freed block should be linked. When you have found these two blocks, you will need to coalesce them if either (or both) are adjacent to the block you just freed.
   (c) I will leave it up to you to determine how to do this. Note that if you coalesce a new block into its left neighbor, then you only need to change the left neighbor's size... but if you coalesce into your right neighbor, then you need to change the "previous" block's `next_block` pointer...
   (d) Your answers to the Testing your Code section may change when you implement this behavior.

Deliverables

Turn in the following when the lab is due:

1. A printed copy of your allocator code, **printed from your IDE when possible.** If you cannot print from your editor, copy your code into Notepad or another program with a fixed-width (monospace) font and print from there.
2. A printed copy of your testing main.
3. A printed copy of the output of your testing main.