Homework 4:
Original Heapster

Due April 19 by the start of lecture.

Overview

In this lab, you will implement your own dynamic memory allocator (heap manager) in C, using a free list with first-fit block selection.

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. We'll refer to this value as the "pointer size".
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 pointer size. So if a pointer 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, return 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 (rounded-up) 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 the pointer size. Example: if overhead size is 8 and pointer size is 4, we should split if the remaining space (after the allocation size is accounted for) is at least 12 bytes.
      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 splitting. 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.

Pointer Arithmetic

In C, one can add integer values to pointers to "move" that pointer forward or backward in memory from its current position. You will need to do this when splitting blocks: to determine the start of the new block, you will start at the current block, then move forward by enough bytes to skip the old block's data (the size of the allocation request) plus the old block's overhead. However, we must note that adding an integer x to a pointer does not move the pointer by x bytes; it moves it by *x multiples of the data type the pointer points to.*

Example: suppose `int *p` currently points to memory address 1000. The expression `p+1` would point to address 1004, not 1001, assuming `int` is 4 bytes. Example 2: suppose `struct Block *temp` points to address 1000. `p+2` would point to address 1032, assuming the size of `struct Block` is 16 bytes.

This is most likely not what you want; you know an exact number of bytes that you want to move forward in memory by, not the "number of multiples of X". To take a pointer to an arbitrary type and move it forward by x bytes, we temporarily convert that pointer to a `char*`, then move the pointer, then cast it back to the expected type. Since a `char` is always 1 byte in C, adding x to a `char` pointer will move the pointer forward by $x \cdot 1 = x$ bytes.

Example: to move `int *p` forward by 11 bytes, we would do `p = (int*)((char*)p + 11)`. In other words: treat `p` like a `char` pointer, increment it by 11 multiples of a character (11 bytes), then reinterpret it as an `int*` as intended. We could then dereference `p` to retrieve a 4 byte value at the destination address. [1]

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 larger of (the size of an integer; the minimum block size) apart.
3. Allocate three `int`s and print their addresses, then free the second of the three. Allocate an array of 2 `double` values and print its address (to allocate an array in C, allocate (`2 * sizeof(double)`); 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 80-element `int` array, then for one more `int` value. Verify that the address of

[1] Don't actually do this. Moving an `int` pointer by 11 bytes will cause it to no longer be aligned to a multiple of 4 bytes, and derefencing it will give a super-fun *segmentation fault* error.

the `int` value is 80 * `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.

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.