CS214 Systems Programming Assignment 1: Memory Allocation++ by Chris Gong and Eric Giovannini

Chris Gong RUID: 164008349
Eric Giovannini RUID: 162008912

I. Purpose

The purpose of this assignment is to create an improved version of *malloc* and *free* using marcos in our mymalloc.h file,

```
#define malloc(x) mymalloc(x, __FILE__, __LINE__)
#define free(x) myfree(x, __FILE__, __LINE__)
```

The extra parameters are used to indicate the file name and line number where an error occurred. However, unlike the conventional *malloc* and *free*, this version will treat errors nicely by not returning a vague Segmentation Fault and printing the reason as well as the location of the cause of a failed malloc/free.

Malloc(size_t size) will return a pointer to a block of memory whose size is at least the requested size. These blocks of memory will come from a 5000-byte char array defined in mymalloc.c,

```
static char myblock[HEAP_SIZE];
```

In this case, *HEAP_SIZE* is 5000.

*Free(void *)* will allow the user to tell the system that he/she is done with a dynamically allocated block of memory.

II. Common errors with malloc and free

When freeing an address to memory that was not dynamically allocated, the usual result is exiting the program. With our *free*, the user will instead be told that the address inputted was not on the heap. In this case, the heap is the 5000-byte char array that simulates virtual memory.

```
// inputted address was never returned by malloc
if (!found_block) { // **** WARNING **** Do NOT say if ptr==last. What if
                    // ptr DOES EQUAL last BUT we
                    // DID find a block (namely, the last one)!!!!
  //printf("Block not found. Invalid address!\n");
  //return NULL;
  errorInFree = -1;
  printf("ERROR: The address contained in pointer (%p) is not the address of a block on the heap.\n"
  "Filename: %s, Line: %d\n\n", p, filename, line);
  return;
}
```

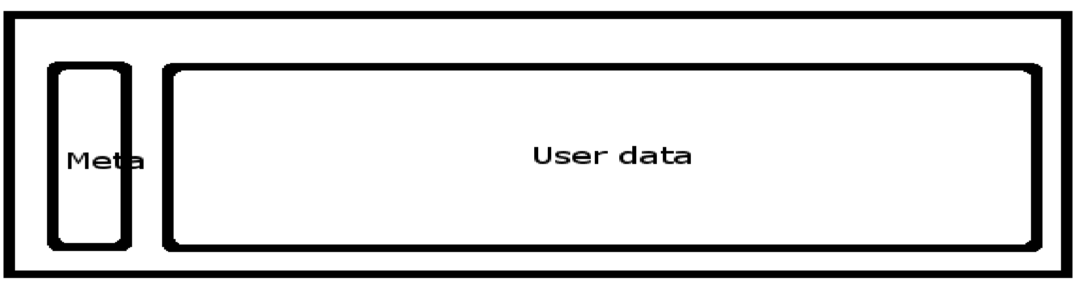
The variable *found_block* determines whether or not, after looking through every meta block within virtual memory, the address entered pointed to the first byte of a user data block. If the block was not found, then the address is not a valid address in the heap. Therefore, the user will be told this instead of exiting the program.

When asking for more data than can be requested, the usual result is once again exiting the program with no explanation. With our *malloc*, it'll tell you the reason why the operation cannot be executed. For example, if you tried *malloc(5001)*, you will be told that there isn't enough space in virtual memory to allocate 5001 bytes. This is reasonable since our emulated memory is only 5000 bytes. Taking into account the size of the meta block (2 bytes), the max amount of memory that can be dynamically allocated is 4998 bytes.

For other common errors in *malloc* and *free*, check the comments by the print statements in mymalloc.c

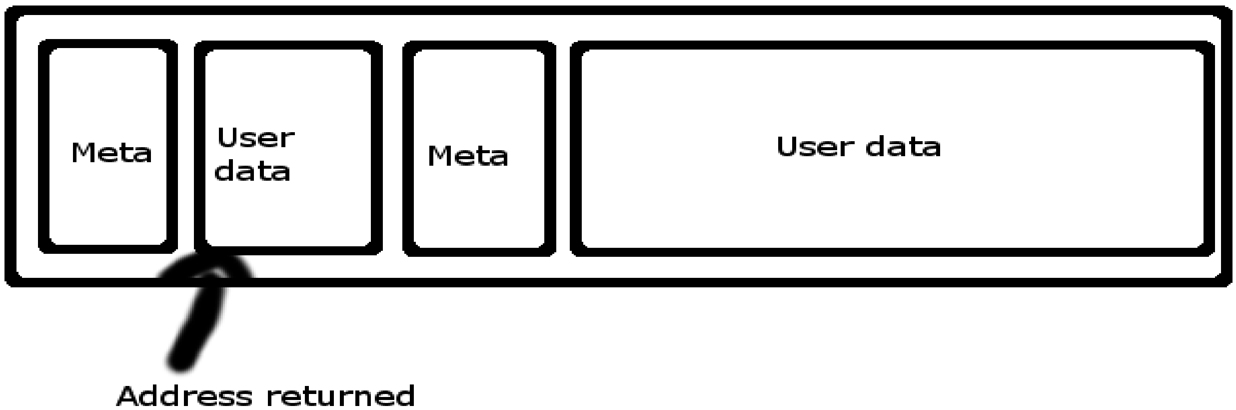
<u>III. How malloc works</u>

The first *malloc* call will call our *bootstrap* method which essentially "sets up" virtual memory for us. Setting up virtual memory is basically initializing virtual memory with one meta block that covers all the memory left in the char array. Since a meta block is 2 bytes, the user data it encompasses is 4998 bytes as a result.



When allocating memory, *malloc* will search through every meta block and find the smallest one that can hold the requested amount of memory. For the first malloc of course, the first and only meta block will be the source of the address returned.

Assuming the user called *malloc(100)*, 4998 bytes is too much memory to supply the user. In fact, if the address, *meta_block + sizeof(meta)*, was returned, then over 4000 bytes would go to waste. So what happens is that the user data block you see above would be split and the result would be two meta blocks that correspond to two different user data blocks,

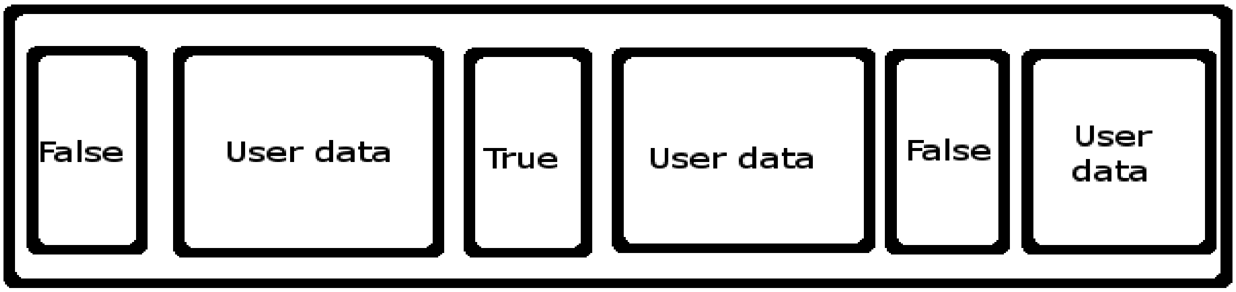


Assuming the first meta block corresponds to the user data to the immediate right of 100 bytes and the second meta block corresponds to the user data to the immediate right of the leftover 4998 – 100 – 2 = 4896 bytes, the address returned from a *malloc(100)* call would be the address of the first meta block plus *sizeof(meta)*.

IV. How free works

Calls to *free* will look at the inputted address and try to determine whether it is a valid address within main memory. Valid as in if it points to the first byte of a user data block that is in use, meaning it has not been freed already. Each meta block consists of a *size* variable and an *in_use* variable (both are stored in one variable called *info* and bit masking is used to read and write to the two variables). The benefit of bit masking is keeping the size of the meta block down in order to hold more data.

If the address inputted is valid, then the *in_use* variable of its corresponding meta block will be set to *false*. Imagine if the address of the first user data block was entered into free where virtual memory consisted of three meta blocks and three user data blocks with the first two meta blocks whose *in_use* is true while the last meta block's *in_use* is false before the call to free the first block. Here would be the resulting main memory,



Now imagine if the second data block was requested to be freed, the interesting result would be a coalescing algorithm used in our *free*, which would be to look for nearby meta blocks to determine whether we could combine blocks that were not in use. So the resulting main memory from freeing the middle two blocks would be one meta block and its user data block that encompassed all of main memory.