

++Malloc: Asst.1 - Systems Programming

By: Kevin Shah (kas665) & Manav Patel (mpp124)

Introduction:

The purpose of this assignment was to implement `malloc()` and `free()` in our own way using our knowledge about pointers, arrays, loops, casting, and other concepts of C programming. One of the main obstacles was to figure out how to work with header files, Makefiles, and redirecting all calls from `malloc()` and `free()` to `mymalloc()` and `myfree()` respectively using macros. After learning the basic concepts, our next objective was to understand how we can compile our functions in `mymalloc.c` and output it as an object `mymalloc.o` file. Putting all of these files together, we managed to create a Makefile that not only compiles, but also executes and cleans the files all in one command - “make”.

Design & Implementation:

One of the most important tasks of this project was to design an implementation that would allow us to traverse through our limited static array of bytes and determine if a particular block of bytes were allocated or free. A possible way to do this would be to store “metadata” for every block that is allocated or freed and then use a linked list to traverse from one block to another. This would require us that we have an integer or a short that tells us if a particular block is free or allocated and store a pointer to the next block. Thus, putting both of these elements into a struct of a linked list type it is possible. However, this is a very **inefficient** way to go about creating a function. Instead, what we did is we used one short as the metadata for every block and traversed through the array by finding the metadata and figuring out how many bytes away the next block is. The problem with this implementation is we only know the size of the block but we do not know if it is free or not. The way we fixed this issue was by assigning:

- **Negative Value = Free Space**
- **Positive Value = Malloced Space**

Example:

-13 = 13 Bytes of free space

13 = 13 Bytes of allocated space

0 = Error (Would never happen)

Now that we had a working and efficient implementation of the metadata, all we had left to do was how we determine during our first cold run of malloc or free if has a correct design structure or not. We did this by creating a “magic-number”: A random hard-coded number that would allow us to determine if our implementation exists or not.

Our final implementation looked something like this:

| | | | | | | | |
|--------------|----|------------------|----|-------------------|---|-----------------|-------|
| Magic Number | 10 | Data of 10 Bytes | -1 | 1 Byte Free Space | 2 | Data of 2 Bytes | |
|--------------|----|------------------|----|-------------------|---|-----------------|-------|

Space & Time Efficiency:

We wanted to make our implementation for mymalloc() and myfree() as space and time efficient as possible with a first-fit algorithm. So in order to do so, we found the best way to do this was by using shorts instead of structs or pointers because pointers would typically be 4 - 8 bytes long depending on the system and structs would be at least the size of all the data types in it and additional padding. Therefore, by making 2 shorts of (4 bytes) for the start of the array and metadata of just one short for every block, we minimize the need to store much data and by using arithmetic calculations to traverse through the array we would be time efficient as well because we would not be searching through every single byte for something.

Workload Data:

We tested all test cases A through F and intensively tested for any breaks or faults in our code 100 times and recorded the time for each iteration of the workload and calculated the average time of all workloads. The workload on memgrind.c was as follows:

A: malloc() 1 byte and immediately free it - do this 150 times.

B: malloc() 1 byte, store the pointer in an array - do this 150 times. Once you've malloc()ed 50 byte chunks, then free() the 50 1 byte pointers one by one.

C: Randomly choose between a 1 byte malloc() or free()ing a 1 byte pointer > do this until you have allocated 50 times - Keep track of each operation so that you eventually malloc() 50 bytes, in total > if you have already allocated 50 times, disregard the random and just free() on each iteration - Keep track of each operation so that you eventually free() all pointers > don't allow a free() if you have no pointers to free().

D: Randomly choose between a randomly-sized malloc() or free()ing a pointer – do this many times (see below) - Keep track of each malloc so that all mallocs do not exceed your total memory capacity - Keep track of each operation so that you eventually malloc() 50 times - Keep track of each operation so that you eventually free() all pointers - Choose a random allocation size between 1 and 64 bytes.

E: Allocate an array to maximum capacity in mallocs of 1 byte and then free each byte randomly.

F: Completely allocates the array to maximum capacity in chunks of 100 and then free random blocks, then reallocate memory in size of 33 and do this 150 times. Finally, free the rest of the space.

Findings:

After thoroughly testing each of the test cases in our memgrind.c, we found our results to be consistent to this data:

Average Time Taken by Test Case A: 0.000007 seconds

Average Time Taken by Test Case B: 0.000071 seconds

Average Time Taken by Test Case C: 0.000011 seconds

Average Time Taken by Test Case D: 0.000025 seconds

Average Time Taken by Test Case E: 0.019219 seconds

Average Time Taken by Test Case F: 0.000541 seconds

The interesting thing in these findings is significant difference between our Test Case E and others. The reason why workload E took so long compared to the other workloads is because the array of 4096 was allocated 1 byte at a time to its maximum capacity and then freed random 1 byte allocations. Freeing at random costs time as it must not only worry about checking if the pointer can be freed or not but also merge two or more free spaces every time it sees an opportunity from the random free()s made.