CS 214 Assignment 1: MyMalloc

Nikita Kolotov, Michala Rose

Project Rundown:

- MyMalloc.h – Here we set the definitions to be used in the header file. We have malloc and free macros set to be (x,__FILE__,__LINE__) for both malloc and free.
- MyMalloc.c – Here is where we defined out static char array of size 4096. We decided to use 4 bytes in the beginning of the array to be used as free metadata. This metadata would keep track of how much free space we had to allocate in the array.
    - Malloc - The way our malloc set metadata was it would add 4 bytes to the requested number of bytes. These 4 bytes will be metadata and a magic number. Then it would traverse the array and look for any space that had the requested number of bytes plus 4, consecutively. It would then set those 4 bytes of metadata. The first 2 bytes of metadata are for an unsigned short that will keep track of how much was allocated. The second 2 bytes are an unsigned short with a magic number. This magic number is just large enough to not be accessible outside of MyMalloc.c. Then the mymalloc returns with a pointer pointing just after the metadata, so when the user returns it to free, we can handle it in the free function.
    - Free – Free will take in a pointer. We first began error checking to make sure the address of the pointer is within bounds of the array. Then we would check for the magic number, 2 bytes before the pointer using pointer arithmetic. Once found we will move another 2 bytes back to see how many bytes we need to free. Then we take that number, add 4 to it to account for metadata, and from that point start clearing space. We set all the indexes to the null terminator '\n' so we can have extra security in our array in case someone wants to read as string. We return nothing because that is how free works.
- Memgrind.c – Memgrind is where we will be testing our mymalloc.c. The first workloads are self explained in the writeup however, we will talk about our time function. Our time function brought to us by time.h is calculated every workload cycle. So when workload A is complete, we would have calculated 100 separate times, we then divide by 100 to get the average time for the work load. Once printed we continue to the next.
    - Case E – Case E tests how well our error catching methods are. It is supposed to return one error. This error will explain the cause of the error as well as include a file and line number at the end. When tested, the error repeated 100 times which can seem cluttering but this ensures that the program works. We try to catch the error of trying to malloc / free the address of an item not in our array. We can quickly catch is because it is outside the scope of our array and we print a detailed error followed by a file name and line number.
    - Case F – Case F We test to see that we have a definite end to our array. We try to allocate just enough to fill the array. Then we free a bit, and allocate more. Through this we see that our array is keeping up with what data is freed and what is malloced so when the new allocation goes through, we know it is because our free function works.

- Makefile – We used this to quickly test and clean our code. We have built in lines to build memgrind off of mymalloc.o and mymalloc.h. If mymalloc.o does not exist then we have a line to create it. Then we included a clean line to quickly get rid of old files.

Assumptions:

- We assume people would try to write to the array using the pointers we gave them. Thus we freed up space for them to be written.
- We assume that the testers will not write beyond what they allocated, based off of a statement made during lecture. Such an issue would cause metadata to be overwritten and we have no way to stop this.
- We assumed that an unused data block was completely optional. We found it more time and computation efficient to keep track of free space in a struct at the front of the array so we would not need to worry about free blocks. We also used null terminators instead of makers to gauge what is free. Because we assumed that the project would grant us that liberty to customize our code how we see fit.

Difficulties:

- We noticed that some test cases in the old code would randomly fail. Our new code is tested repeatedly without error now, however in the old code it seemed random. We assumed this could be the program moving too fast being unable to keep up with the pointers. One run would show 3 fails, then the next would show 5 with no change in code. We tried to implement the most stable tests now and it seems to have worked.

Space Efficiency Bonus:

- By only using 4 bytes to hold metadata for all pointers and an additional 4 bytes to hold free memory, we successfully cut down on wasted memory. If pointers were to be said in terms of x, our program only uses ((4*x) +4) bytes of metadata, greatly enhancing space efficiency.