Deep Patel
RUID 159000639
netid dp767

Program Design:

I utilized an implicit list to store the size of blocks in the memory. To do this I used a short type as the size and flagged the least significant bit to indicate the block being allocated or free. This way, I only allowed allocated blocks to be of even size.

I initialized the block to make the first 2 bytes the size of the rest of the data block.

I made helper functions to help with mymalloc, myfree, and mymerge.

1. getSize: takes in a char pointer to metaData and outputs short size value
2. setSize: takes a short size value and index (used to indicate where in myblock to set this size)
3. isAlloc: takes a short size value and outputs 0 or 1 depending on whether it is flagged or not (allocated or not allocated)

For malloc, I returned an error if the requested memory was more than the total block size. (Mainly because this ws discusses in Piazza as an error and because never would a malloc call of this size make any sense or be feasible and it would be a mistake to ignore this is just constantly return NULL pointers). I then traversed the list until a free block of suffienct size is found. If the block is just big enough by within 3, it would not make sense to split because the remaining 3 bytes would consist of 2 bytes metadata and 1 byte for data, but data can only be store in even amounts. Thus if this is the case I would just give a pointer to the whole chunk of data. If there is more that 3 extra bytes, I would split the blocks and create a new short and size for both. If sufficient memory is not available in the block, I return a NULL pointer.

For free, I start off by returning an error if a null pointer is passed as the parameter or if the pointer does not point to something in myblock. Then I traverse myblock, comparing the input pointer to the address of the start of every allocated block. If they are equal, I free the block by deflaging its size and then running merge to merge possible adjacent free blocks. If the pointer is not found, function prints an error saying the memory was not allocated by malloc.

For merge, it traverses the blocks to find 2 adjacent free blocks and if it finds them then it adds the size of the second block to the first plus the overhead size it used and deletes the size of the second from the array (set it equal to 0).

In memgrind, I made 6 functions, 1 for each workload and called the functions in main 100 times each, keeping track of the total time for each and found the average by dividing by 100 and printed this out.

Test Case Analysis:

Workload A

Average Time: ~60000 microseconds

This workload is expensive, because after 1250 mallocs, there is not enough memory in store more 1 byte mallocs, so for the rest of the 3000, it returns NULL pointers, so when freeing those 2750 null pointers, an error statement from free is printed each time so that is what is consuming the bulk of the time.
It is after 1250 because each block takes 2bytes (2*1250 = 2500) and each block is an even amount of bytes so (((1+1)*1250) = 2500) so 2500+2500 = 5000, which is the limit of the size of the memory.

Workload B

Average Time: ~200 microseconds

This workload is not very expensive because there are no errors associated with it. 1 byte is readily available every time because it is immediately freed in a standard manner. The only factor that makes it longer than E & F is that it has 3000 iterations (6000 operations).

Workload C

Average Time: ~4000 microseconds

This workload did not have many errors, becasue the mallocs were of size 1 and the frees always followed the mallocs.

Workload D

Average Time: ~8000 microseconds

This workload had many random errors including mallocing of size bigger than size of the array and also because malloc would not have enough memory to return it would oftern return a null pointer, which would cause an error when freed. This caused a lot of error printing and longer run time. The random behavior really tested malloc's ability to adapt and free to recognize errors.

Workload E

Average Time: ~500 microseconds
The random nature of this freeing leads to pointers being freed multiple times and other such errors, so it tests free in that sense Its speed is affected by the high number of random error stantments.

Workload F

Average Time: ~50 microseconds

This workload is quite inexpensive in runtime because the merge, malloc, and free functions are not called on too many times and there is not much room for error. The only error occurs when you free all pointers in the char array again and run into null pointers or already freed.