Mymalloc implementation and explanation:

The first thing we did for implementing mymalloc was to create a global variable that handles whether or not malloc has been called before. On first call it gets flipped and creates a block containing all the free space (not including the meta data) and then sends it to be split into the first block of the list and the last block of the list.

Once initialized, malloc was supposed to loop through the blocks and check to see if they were in use or not, this is where we encountered an inexplicable problem where the third malloc request would some how not notice that the first block was in use(so it reset its in use flag). Why we use the term inexplicable is because it will work as intended across the board on initialization, and the initialized section works the same for malloc request 2 up until N(where N is the amount of malloc requests made) which means if it works for the second malloc request it should work for the rest.

This is not the case however and for some reason any request after the first two just overwrite each other. We attempted to test this out as much as possible, but nothing seemed to change it for any request made after the second malloc.

On the bright side, the split method created to cut blocks apart works as expected (almost no matter the situation, hard to tell since it does not want to work for more than 2 requests) and functions by taking a single block and basically making it into 2 blocks and putting them in their relevant positions. As far as the slew of other helper methods like read_used and read_size, they are just binary operations used on the meta data to obtain relevant information. We also thought about including our merge helper method, however, were unable to test to the point where it may have been relevant (more specific details about merge below).

Myfree implementation and explanation:

Implementing free was a bit more difficult in some ways but essentially worked by iterating through the list until the address of the pointer's data matched the address of the requested address to free. We also decided free was the better place to use merge and where it would be used most often. We identified 3 major cases where merge would be needed; one being the case of 3 contiguous blocks become free as well as the similar cases of either the current block+next block need to be merged or the prev block+ current block.

From our testing it seems that merge also works as expected, although in full disclosure we believe some of the conditionals are unnecessary or just plainly will never happen.

Extra findings and Information about Testcases:

While working with smaller versions of our testcases we noticed a few interesting findings. The first most interesting one was when it came to the size of the struct. We found if a struct was of size 1, or 2 it would show correctly, and anything more would be rounded up to the nearest power of 2(usually). In more detail, if the struct contained 3 bytes of information it would be rounded to 4, if the struct had 5 it would round to 8, and from there it seems to add 8

additional bytes if more is necessary. We found this interesting because it means getting the size of the meta data may not lead to the most accurate count of the bytes within someone's meta data and should either manually input the size of the meta data when necessary or just condense the meta down to fit into either 1, 2, 4, or 8 bytes of data exactly.

Additionally, since mymalloc has that weird issue, we were unable to test and see which errors are caught by malloc and which are not.

As far as our test cases go in memgrind.c, we are not sure if the times given are accurate or not because most of the workloads end up displaying as 0.0 which may be dependent on the way the ilab machines deal with floating point rounding and divide by 0 error. We opted not to include the results from our memgrind test because we are certain the results are inaccurate due to both the shortcomings in mymalloc.c as well as in memgrind.c.