README:

Claudia Pan cp728

Yelin Shin ys521

Our struct meta, contains size_t size which is the number of bytes available to be used or is being stored. It is also includes int usage which will be 0 or 1. If the usage is 0 then the block of memory is free and if it is 1 then it is currently being used.

Our mymalloc.c code initializes a block of memory of 5000 bytes. When user malloc()'s, we look for a spot to malloc the size requested. If a spot is found, using our metadata, we change the usage to 1. Furthermore, we check if we are able to create another metadata plus 1 byte. If this is the case then the current metadata size is changed to the requested size, and the new metadata size is what's left over (usage is 0). Another case we have is that if the requested size is exactly the size of the metadata we found, all we have to do is change the usage to 1. When malloc is successful, we returned a void* pointer that we just malloced. Otherwise, when we are unable to find a spot (none of the usages are 0 or there is not enough memory left) we print an error statement "fail to malloc." In this case we return a null pointer.

In the myfree function, we get a pointer as the parameter. First, we check if the pointer address is within our myblock(our 5000 bytes we have available). If it is not within, we return an error statements saying to please provide a valid pointer allocated by malloc. In the case that the address is within my block, created a char pointer that points to the requested address. If the usage of this address is 1, then all we do is change the usage to 0 [free is successful] and call merge. If the size at this address is 0, this address doesn't have a metadata so we return an error statement saying please provide a valid pointer allocated by malloc. Otherwise the usage is 0 and we still return an error statement stating that this pointer was already freed.

In the merge() function, we made two meta pointers called tmp and next. Everytime merge is called, we always start at the head of myblock [the top]. We loop through myblock, to look for the first metadata's usage that is 0. By using pointer arithmetic, we move next to point to the next metadata. We must check that next is also within myblock; if it isn't then there is nothing to merge and we return. If next's usage is 0, we merge by increasing tmp's size with next's size and our metadata size. Using a while loop, we keep checking for the next to see if there is more metadata with usage is 0. At most, we will merge 3 metadata's together.

Findings:

- Test A: Average time for Test A: 61733 microseconds
    - We had successful mallocs until we hit our limit of 5000 bytes. After we hit our limit, our malloc prints out an error message, “Fail to malloc.”
    - When we free 1000 times, every time there is a successful free, our program is also able to check for merging. After we freed and merged all of our data, our free function prints an error stating, “Please provide a valid pointer allocated by malloc.”
    - At the end, myblock only contains one metadata with usage = 0, and size = 4984.

- Test B: Average time for Test B: 35858 microseconds
    - When we first malloc(1), a new metadata is created with usage= 0 and size 4967.
        - Size is 4967 because the first metadata’s size is 16 + the 1 byte it is storing, and the new metadata’s size is 16. [5000-16-1-16 = 4967]
    - Then we immediately free what we malloced and it will merge with to usage = 0 and size 4984.
    - We realized that there should be no errors because every time we malloc it will free right after. And this does happen.
- Test C: Average time for Test C: 69916 microseconds
    - We generated a random number, either 0 or 1 into an int array. Using a for loop we make sure to do this 1000 times, we check the index of the array and if it is 1 then we malloc(1) and increment our malloc count. If the index of the array contains 0, then we free a 1 byte pointer and increment our free count.
    - Then we loop through the pointers and make sure everything is free.
    - We also print our malloc counter and free counter.
    - So in the instructions we are asked to either malloc or free 1000 times in total, then it says to eventually malloc 1000 bytes. It is not possible to guarantee that we will malloc 1000 bytes because we are asked to randomly choose between mallocing or freeing 1000 times.
        - We found it more important that we were able to choose between mallocing or freeing 1000 times so that’s what we did.
    - We also realized that generating a random number takes make our average time longer than the other test cases.
- Test D: Average time for Test D: 84320 microseconds
    - In this test, we used similar steps as Test C. In this case we had to randomly pick between 0 and 1[instruction to free or malloc], and also 1 to 64 [size to malloc]. This test stops when our malloc counter hit 1000.
    - If our random number was 1 then we malloced the random size we were given. We also kept track of our total memory used so that we could print a statement saying that our memory is full. When our memory is full and we still ask to malloc because our malloc counter hasn’t hit 1000, an error statement will print, “Failed to malloc.”

    - If our random number was 0, then we freed a pointer at a spot that was just possibly malloced. Since it is random, our free can actually free or fail to free (if the address was already freed, our of memory bound, or not malloced yet).
    - Afterwards, we eventually freed all pointers by using a for loop.
    - Our findings were the same as test C such as randomly generating a number makes our average time longer.
- Average time for Test E: 70 microseconds
    - This test case is for making sure that when we find exact memory, we don't have to create a metadata, we just need to change the usage to 1 (used)
    - It works. more description in testcase.txt.
- Average time for Test F: 71 microseconds
    - This test case is for making sure when we free the middle one that is used, we merged twice for that. The surrounding two metadata's usage are both 0.
    - (like, usage=0 - usage=1 - usage =0 , and free middle one, so we need to merge twice and make one big memory that usage = 0)
    - It works, more description in testcase.txt.