Idan Levi (il177)
Alex Varshavsky (av653)

CS214
System Programming Assignment1: ++Malloc

In this assignment we implemented our own malloc() and free() functions in order to observe what happens in memory and how dynamic memory works during run time. In this implementation we use static array (memblock) of size 4096 bytes. This array will be used to keep track of pointers to memory locations of available/unavailable memory. mymalloc() works as “first find” method which means it finds the first available location with the required amount of memory that is entered by its user.
We used struct that represents our Meta-Data. This structure allowed us to track down and look at each block. Metadata contains 1 bit to store 0 or 1 which represents FREE or IN_USE respectively. Another 15 bits are used to store data about the size of the next available block. When trying to allocate memory, we check if a block is available by checking his size and status, which are stored in the Meta(header) and if not, we move on to check the next block until we either find a block that will do the work, or run out of blocks to check.
If no free space was found, returns NULL. As part of the optimization process, we implemented mymalloc() to not only allocate a block only if it has the requested size + meta, our implementation also uses blocks that have enough space for the requested size even if Meta can’t fit. For example if user needs 10 bytes to be allocated, which means 12 bytes of space needed (10 data + 2 meta), but we found a block that has exactly 10 bytes, we will allocate that space, and simply use the previous (already existing) meta space.
Our implementation checks for errors:

- Requested size doesn’t exceed the available space.
- Requested size is not 0.

Myfree() function compares the requested address with all the valid addresses (only pointers to the first bit of data). If it finds a match between the requested pointer to a valid pointer, it moves one address back (now points to the Meta) and changes the IN_USE to FREE and by that releasing the whole chunk and making it available for future allocation.
Our implementation checks for errors:

- Requested pointer is in our block range.
- Requested pointer is a valid pointer.
- Checking if we are trying to free an already FREE block.

Test Results:

```
*********************************************************************************
***************************** Summary Test A-F ********************************
*********************************************************************************

Total runs for Test A: 100
Average run time for test A is: 0.000421000 miliseconds

Total runs for Test B: 100
Average run time for test B is: 0.005497000 miliseconds

Total runs for Test C: 100
Average run time for test C is: 0.001024000 miliseconds

Total runs for Test D: 100
Average run time for test D is: 0.008422000 miliseconds

Total runs for Test E: 100
Average run time for test E is: 0.007311000 miliseconds

Total runs for Test F: 100
Average run time for test F is: 0.000536000 miliseconds
```

Above attached the results of running 6 different tests 100 times each, and finding the average time for each one.
The findings show that the fastest execution time was done by test A and the slowest execution time was done by test D. Those findings make sense as test A is the simplest one and it uses malloc() and free() once each and then repeats it 150 times.
Test B was slower even though it also has a total of 150 allocations and 150 freeing. It was slower because this test involves a nested loop, as well as an array that is constantly used to store a pointer.
Tests C and F had also extra operations that made them a bit slower.
It made sense that test D and test E were the slowest ones even though the malloc() was used only 50 times in both of them. The reason is because in those two tests, we used a random number to pick the allocation time. We had much more errors and more operations because of the extra loops it had to go through.
The findings didn't show anything unusual to us and it seems that our predictions were correct.
Test E and test F are described more in the testplan.txt file.