CS214
Assignment 1
Professor Tjang
mymalloc() and myfree()
Eric Zhuang, Shah Rahim

We allocated 5000 bytes of memory for this implementation. We use a doubly-linked list in this implementation, which allows memory to be allocated in linear time O(n). Doubly-linked lists are also more helpful for our myFree() function when attempting to defragment separate sequential blocks of memory, as the array can communicate with previous and next pointers from each node.

When the first call to myMalloc() is made, myMalloc() will initialize a base struct that will initially hold all free memory. A pointer to the address of each memory entry to be allocated will also be saved into an array memEntries (as they are being allocated) to ensure that when a memory entry needs to be free'd, we can check that that specific memory entry was created by myMalloc().

Our implementation detects errors that occur when:

- User tries to free unallocated variables and free'd variables.
- User tries to allocate less than 2 bytes in size.
- User tries to allocate a block of memory larger than what was initially reserved (5000 bytes).
- User tries to free pointers that were not allocated by myMalloc().

myMalloc() will then proceed with a series of checks that compare the size of each memory block in our doubly-linked list and also the amount of memory needed for the malloc request and process with the size of the memory entry from user input.

> **Case 1**: Not enough memory.
>
> **Case 2**: There is enough memory for the malloc request but not enough to make a newmemory entry.
>
> **Case 3**: There is enough memory for both the malloc request and new memory entry.

myFree() communicates with the global memEntries array to check if the requested memory entry freeing operation is indeed valid, and that the specified memory entry was indeed allocated by myMalloc().

myFree() accounts for (de)fragmentation through a series of 4 cases:

Case 1: Prev and Next blocks are free, so both Prev and Next are merged into the now larger current block.

Case 2: Prev is free but Next is not free, Prev is merged into the current block.

Case 3: Next is free but Prev is not free, Next is merged into the current block.

Case 4: Both Prev and Next are not free or there is only 1 memory entry

Results:

Workload A: Total Execution time of workload A in microseconds is: 7312.0000
Average execution time of workload A in microseconds is: 73.120003

This workload involved a 6000 separate calls and over the course of several tests, the average execution of each separate instance of this workload proved to take a similar length of time and stress of overhead. Since each memory allocation call asked for the smallest amount of memory possible, I thought that this workload would take a shorter amount of time. From our findings, we can assume that the amount of memory requested and allocated through each call only incur negligible amounts of overhead.

Workload B: Total Execution time of workload B in microseconds is: 6636.0000
Average execution time of workload B in microseconds is: 66.360001

This workload proved to be shorter than workload A, which was very surprising. The number of calls in this workload and workload A were the same, and the numbers of each operations performed were the same as well but this workload proved to take less time. We surmise that this might be because the process of allocating memory becomes more expensive when there is more memory and addresses to track, but when you are only allocating 1 byte at a time and then freeing it, there is less memory being held in the machine.

Workload C: Total Execution time of workload C in microseconds is: 16928.000000
Average execution time of workload C in microseconds is: 169.279999

This workload proved to be very expensive because of the quantity of variables that we had to keep track of. Using the random function proved to be very expensive and so did determining how many times we had used malloc. We also had to ensure that each malloc call had to be negated by free calls by the end of the function.

Workload D: Total Execution time of workload D in microseconds is: 15687.000000

Average execution time of workload D in microseconds is: 156.869995

This workload proved to be more expensive than the previous ones because we had to randomly decide whether to use the malloc or free functions. We also limited the function to be pseudo-random because it would make our output cleaner (our code does account for attempting to free null pointers, but these error messages would have spammed the output.). This pseudo-random design meant that our workload would not attempt to use free more frequently than malloc had been called.

Workload E: Total Execution time of workload E in microseconds is: 3307949.5
Average execution time of workload E in microseconds is: 33079.49

This workload is used for the sole purpose of allocating an expensive call. Doing this will inform us in the future on how to optimize our function in order to maximize efficiency. The price that is to be paid is run-time, but as people say, lose the battle, win the war. The purpose of this test case was to exploit the function's inability to malloc in an efficient runtime. This test case was to show how a beginner programmer might inefficiently allocate memory, and how it affects the time.

Workload F: Total Execution time of workload F in microseconds is: 7030.000000
Average execution time of workload F in microseconds is: 70.300003

This workload worked quite efficiently, the purpose of this was to start of allocating large amounts of memory and decrease the number of bytes allocated by 1 to see how malloc and free work with memory that decreases overtime. This workload shows that if a large amount of memory is allocated and it decreases by 1 byte per allocation, the runtime to do so is quite low. Meaning this type of allocation is quite efficient although the first allocation is a large amount of memory in respect to the heap.