Zac Blanco
Mike Botti
Assignment 1

Implementation of Malloc and Free

In order to keep track of the metadata for our memory block we split the block into two halves. The first half houses the memory pointers while the second half houses the metadata. This half-half approach works due to the fact that the ratio between metadata and memory space is 1:1. We use a basic approach to keep track of whether a block has been allocated or not. By simply using three characters to represent malloc'd space, we are able to accurately keep track of how many bytes are in use. The character 'o' is used for a single byte malloc'd, 'b' is used for the beginning of a malloc'd space, 'e' is used for the end of the malloc'd space and 'x' is used for everything in-between a 'b' and an 'e'.

Using this linear approach it creates an area where to check if any single byte is free or not can completed in *O(1)* time. In order to check if an entire block of n-bytes is free or not, this necessitates the need for n comparisons resulting in a time of *O(n)* to check if a block of memory if free or not. With this in mind, given a memory block of size *m* bytes, the entire time to check if we can malloc a pointer results in a maximum of *O(m/n)* time.

For the implementation of free, we simply must check in our metadata whether the memory block begins with 'o' and simply free the single byte. Otherwise check for 'b' and erase the metadata (and null the memory block) until we hit 'e'. In the worst case of memory block of size *n*, this implementation will run in a worst-case of *O(n)* time.

With regards to the behavior of malloc and free on edge cases:

- Malloc: If the size is less than or equal to zero OR cannot be accommodated, print an error and return NULL
- Free: print an error on a NULL pointer
- Free: print an error if the pointer doesn't point to the beginning of a memory block.
- Free: print an error if the pointer was not issued by malloc.

Structure & Makefile

This project houses the following files:

- Makefile - makefile with targets to compile and test project
- memgrind.c - Workload implementations
- mymalloc.c - malloc & free implementation
- mymalloc.h - header file for mymalloc
- test_malloc.c - A file containing some unit tests

- testcases.txt - A file which explains Workloads F and E

Makefile Targets:

- all - compiles the make grind (alias for memgrind target)
- clean - deletes all traces of memgrind and mymalloc executables and compiled objects
- test - runs a small set of unit tests from 'test_malloc.c'
- debug - compiles an debuggable version of memgrind and mymalloc
- memgrind - compiles a memgrind and mymalloc into 'memgrind.out'

Workload Data

Below is a sample of the mean time to run each workload after 100 iterations*.
Workload A Runtime: 18263.589844 Microseconds
Workload B Runtime: 109.930000 Microseconds
Workload C Runtime: 396.420013 Microseconds
Workload D Runtime: 72378.132812 Microseconds
Workload E Runtime: 53642.601562 Microseconds
Workload F Runtime: 197168.703125 Microseconds

* A short note on how these were obtained: We specifically commented out the lines in mymalloc and myfree which printed out the __LINE__ and __FILE__ on errors. This way when we ran the memgrind, the running time wasn't affected by the extra I/O time it takes to print to the console and are solely reflective of the running time of each workload.

The code submitted as a part of the assignment does print out errors and thus running time increases dramatically due to the I/O calls when printing to the user console. However it is possible to mitigate by piping the output to a file.

Another way to obtain the running times without having to print all of the information to the console is to compile and run memgrind by piping output to a file and then using "cat output.txt | grep seconds" to find the lines with the print statements. Example command below:

```
make memgrind; ./memgrind.out > memgrind_output.txt; cat memgrind_output.txt | grep seconds
```

Conclusion

We believe this to be one of the simpler, and more memory efficient situations in which it is necessary to malloc small amounts of data (< 20-30 bytes). When mallocs are kept to this size or smaller it results in a more efficient memory:metadata ratio. However as soon as the size of memory requests begin to increase the memory:metadata ratio becomes less efficient than that of a linked-list implementation. This could prove to be useful in devices where only small pieces of data need be transmitted. One possibility is in tiny embedded systems or sensors. One improvement that can be made to this implementation without changing the general scheme is partitioning out the memory size and allocating one partition to smaller pieces of data, and another partition for larger pieces of data. This would alleviate the condition on this algorithm where after many malloc and free attempts we end up with a very fragmented piece of memory resulting in an inability to allocate even small to medium sized pieces of data.

However we can can see that from this implementation (and a quick comparison with the *real* malloc implementation, that ours if wildly inefficient compared to the stdlib malloc. This, for one is due to the fact that in our implementation that malloc and free both require at best n search time (where n is the size of memory we're trying to allocate. Compared to a linked list or heap based implementation you would be able to search for free memory blocks much, much faster due to only indexing free chunks of memory rather than mapping each byte individually.