CS214 Systems Programming
Karen Lee – kwl47 – Section 07
Michael Loh – ml1328 – Section 03

++MALLOC

About

The goal of this project is to build a simple, custom implementation of *malloc(*) and *free()* that will not only provide the same basic functionality, but also detect common user errors and address them.

The following are the core files of our program:

- **mymalloc.h:** Contains include statements, struct and macro definitions, function prototypes, and the declaration of a static char array that will be used to simulate memory.
- **mymalloc.c:** Contains our implementations of *malloc()* and *free()*, named *mymalloc()* and *myfree()* respectively, and all the their helper functions.
- **memgrind.c:** Contains the test cases and workloads used to evaluate our implementation of *malloc()* and *free()*.

Design

To manage memory, we divided our array into nodes that could split and merge as needed. Each node consists of a fixed metadata node, and space reserved for the user to store data. The metadata node is represented by the struct *node_t*, and consists of the following attributes:

- *size (12 bits):* Represents the size of the user-reserved space associated with the node. Since memory is only 4096 (2^{12}) bytes, 12 bits is sufficient to store any possible size that can be associated with a node.
- *active (1 bit):* Represents whether or not the user is currently using the space associated with the node.

While the data types of *size* and *active* are listed as unsigned short and bool respectively, using a struct allowed us to utilize bit fields and tailor the size of our data to exactly our needs. In total, our metadata is 2 bytes. Although the individual components add up to only 13 bits, memory padding and the need for the struct itself to store metadata increase the size of our struct.

mymalloc

This function attempts to return a pointer to usable memory of the user's requested size, and returns NULL if unsuccessful.

Upon calling *mymalloc()* for the first time, one large, inactive node encompassing all of memory will be created. Then, the user's input will be validated to ensure that it falls within the acceptable range of sizes – a user cannot request less than 1 byte, and cannot request more bytes than there is room in the array. Afterwards, *mymalloc()* will iterate through each node of memory and attempt to locate an inactive one to accommodate the user's request. Once a viable node is found, the node will either be split or filled.

- Nodes are split when the available space of a node can accommodate both the user's request, and another node with at least 1 byte of available space associated to it. The original node is split into two – one node that encompasses exactly what the user requested, and a second node that encompasses the leftover space of the original node.
- Nodes are filled when the available space of a node can accommodate the user's request, but there is not enough leftover space to create a new node with at least 1 byte of free space associated to it. In this case, the node will simply be activated without changing its size, and the user may receive slightly more space than they requested.

If there are no viable nodes in the memory array whatsoever, *mymalloc()* will return NULL and produce an error stating that the user is out of memory.

myfree

This function attempts to deallocate memory. Freed memory can no longer be accessed, and becomes available to be overwritten by another allocation. If the user attempts to free a pointer not allocated on the array, free something twice, or free something that is not a pointer, *myfree()* will produce an error.

When the user calls *myfree()*, the function first validates their input by checking that it is neither NULL nor a value outside of the memory array. Next, *myfree()* will calculate where the metadata corresponding to a data pointer should be and iterate through the memory array, searching for a corresponding node. If one is not found, *myfree()* returns an error.

If a matching node is successfully identified, *myfree()* will first check to see if the node is active to ensure the user is not freeing the same pointer twice. If the node is active, it will be set to inactive. Then, *myfree()* will check the adjacent nodes, and if they too are inactive, will combine them into one to make more room for future calls to *mymalloc()*.

Workload Data

Workload	Mean Runtime
A	0.000007
B	0.000038
C	0.000007
D	0.000008
E	0.000040
F	0.001128

Findings

Both *mymalloc()* and *myfree()* had O(n) worst case runtimes, with n being the number of nodes. This occurs because both *mymalloc()* and *myfree()* navigate by starting at the beginning of memory and iterating through nodes with a series of calculations. Workloads where a lot of data is stored at a time will run much slower, as shown by the huge increase in runtime by workloads B and F. In contrast, workloads where data is constantly being freed, such as workloads A, C, and D, enjoy substantially faster runtimes because nodes early in the linked list are continuously being made available and combined, decreasing the total number of nodes and increasing the amount of cases where *mymalloc()* does not have to iterate to the end to find a suitable node.

While *myfree()* enjoys the ability to instantly access a memory address, our implementation chose not to risk trusting the user and interpreting their argument as a safe address to immediately access and operate upon. Because space efficiency and error checking was a bigger priority than speed, our implementation of *myfree()* opted to iterate through every node rather than store additional metadata that would allow safe instant access to a node. At the expense of speed, users will be able enjoy additional memory and the ability to free invalid addresses without worrying about breaking their existing data.

One important thing to note is that with this implementation, the amount of allocations done at a time is much more impactful than the sheer quantity of data allocated on the array. This is evidenced by the speed difference between workloads E and F – although both fill up the array entirely, workload E operates substantially faster because it frequently requests amounts of data much larger than workload F and runs out of memory after fewer allocations.