Assignment 1: ++Malloc

Dennis Kager
Ali Awari

Objective

In this assignment, the goal is to re-create malloc() and free() so that both functions are smarter and more robust to error.

How it works

mymalloc.h:

- In this header file, function calls to malloc() and free() are respectively replaced with mymalloc() and myfree().
- There is a static character array of size 4096 bytes which simulates a heap of memory to allocate.
- This file also includes the function definitions used in mymalloc.c

mymalloc.c:

- mymalloc()
    - When the user makes a call to mymalloc() the number of bytes that the user requested is passed in as well as the file name and line number from the preprocessor.
    - The first thing checked is to see if the user asked for a number of bytes less than or equal to zero – if so, then throw an error.
    - Then the function checks for a magic number. Upon declaration of the static array, a 4-byte number is assigned to the front of it. The function declares an integer pointer to the front of the static array, which takes the first 4 bytes, and then compares the dereferenced pointer to the magic number. If the comparison is true, then there is no metadata yet and the linked list of metadata is initialized. If the comparisons do not match, then that means the magic number in the array has been overwritten by metadata and is already initialized.
    - The function then loops through the linked list to find one of the following:
        - A node that is large enough for the space requested along with the size of the metadata.
            - Adds a new node to the linked list using the addNode() function.
        - A node large enough for the space requested, but not the metadata.
            - Just returns the node's space with a few extra bytes for the user.
        - A node exactly equal to the space requested and the size of the metadata
            - Just returns the node's space to the user.
    - If no nodes with a large enough space are found, then NULL is returned along with an error message.

- myfree()
    - When the user makes a call to myfree() the pointer to be freed is passed in along with the file name and line number from the preprocessor.

    - This function starts off by checking to see if the pointer given is NULL – if so then output an error.
    - Then the metadata is looped through and with every iteration, the function checks to see if the address of the memory to-be-free minus the size of the metadata is equal to the address of the current node. If there is a match, then the loop breaks and a Boolean flag is set to true. If no match is found, then the flag stays false.
    - After the loop is finished, a conditional statement checks the status of the flag. If the flag is true, then continue. If the flag is false, then an error message is used to tell the user that the item given is not associated with mymalloc().
    - If the pointer is associated with mymalloc(), then the function checks to see if the metadata for the pointer is marked as in-use or not. If it is marked as not in-use, then an error is given saying that the freeing of that pointer is redundant – It was done before.
    - If the pointer given passes all of the previous tests, then the in-use flag is flipped to 0 (not in-use) and defragment() is called.
- defragment()
    - This function looks through the linked list of metadata and checks if there are multiple nodes, side-by-side, that are not in-use.
    - If two or more consecutive nodes are found to be not in-use, then the spaces are combined to create one larger node. This helps to maximize space available to the user.
    - Every time nodes are condensed, the function calls itself (recursively) to keep checking throughout the function for consecutive free nodes until no more nodes can be condensed.

memgrind.c:

- The purpose of this file is to run a series of heavy tests on mymalloc() and myfree() to check for correctness.
- Tests A-F are run at least 15,000 times total and the average time for each test is calculated and output at the end.

Workload Data and Findings:

- Out of the given Tests A-D, Test B stands out the most. Test B seems relatively simple and much less involved compared to Tests C and D, however, the results say otherwise.
- On iLab machines, Test B generally runs anywhere from 11.5-17x slower than Tests A, C and D.
- Test F generally runs slow compared to the rest of the tests except for Test B still. It makes sense that Test F would run slower than other tests since it is constantly using up all the memory and freeing it over and over along with testing for other edge cases. However, it is still faster than Test B, which is only allocating 1 byte 50 times and then freeing it all (with a total amount of 150 uses of mymalloc).

[Results are on the next page]

```
[dlk144@kill Asst1]$ ls
Makefile  memgrind  memgrind.c  mymalloc.c  mymalloc.h
[dlk144@kill Asst1]$ ./memgrind
AFTER 100 TRIALS:
TEST A AVERAGE: 4.41 microseconds
TEST B AVERAGE: 52.70 microseconds
TEST C AVERAGE: 3.32 microseconds
TEST D AVERAGE: 3.10 microseconds
TEST E AVERAGE: 6.66 microseconds
TEST F AVERAGE: 31.00 microseconds
[dlk144@kill Asst1]$ ./memgrind
AFTER 100 TRIALS:
TEST A AVERAGE: 10.21 microseconds
TEST B AVERAGE: 117.31 microseconds
TEST C AVERAGE: 8.76 microseconds
TEST D AVERAGE: 8.99 microseconds
TEST E AVERAGE: 15.27 microseconds
TEST F AVERAGE: 67.93 microseconds
[dlk144@kill Asst1]$
```