David Acevedo
Kyle Thakker

Asst1: Memory Allocation++

Running the program:

To execute the program, run the make file and execute the output file. The makefile includes the mymalloc.h where we define the malloc and free functions.

Description:

For this program, we implement malloc and free library calls with the capability to detect common programming and usage errors. Both functions use the “first free” algorithm to select free blocks to allocate. A static char array of size 5,000 bytes is used to simulate main memory in a computer. A struct called “Node” is used to represent a metadata node that gives information on the following block of memory including its state and the size of the block. Both of these functions are located in the mymalloc.c file.

In the malloc function, there is a check to test whether the size of the pointer trying to be allocated is greater than the total memory minus the size of the metadata node. If the check is true, an error statement is printed and the program returns without allocating any memory. However, if the check returns false, the block of memory is initialized and the memory is then looped through. If a block of memory is being used the loop then points to the next block in memory. If the block is free, and the size of the block can fit a metadata node, that node is then initialized and its state and size are defined. If the size of the block is just big enough to store the data trying to be allocated, no node is created and the state is updated to show that the block is being used. Afterwards a pointer to the block of memory is returned to the programmer.

In the free function, there are three checks to test to see if the free function can free the pointer. The checks are if the pointer is null, if the pointer is within the heap, or if the pointer is already freed. If any of these checks are true, the program prints an error statement and returns without freeing the pointer. After these conditions are checked, the pointer is freed, and if the blocks before or after are also free, they are merged with the recently freed block.

The memgrind.c file contains the code to run our test cases. Each workload was run 100 times. The data outputted by these runs is shown below.

Workload A allocates1 byte 1000 times. Then it frees the pointers 1 by 1. This workload took the second longest to run. This is likely the case because as the amount of allocated memory increases, the malloc function has to loop through a larger segment of the memory block to find an unused segment.

Workload B mallocs 1 byte and then immediately frees it 1000 times. This workload took the least amount of time to run, indicating that allocating and freeing memory at the beginning of the block is very quick.

Workload C chooses randomly between allocating or freeing a pointer to 1 byte. 1000 mallocs are performed and eventually all pointers are freed. This workload was the second fasted to execute. Due to the function randomly choosing to malloc or free, the malloc function never has to search very far to find an unused segment of memory.

Workload D chooses between mallocing and freeing a pointer to 1 to 64 bytes. The allocation size is chosen randomly. 1000 mallocs are performed. The total amount of memory allocated is kept track of so the total memory capacity is not exceeded. Eventually all pointers are freed. The extra execution time of workload D compare to that of workload C is likely to due to the extra overhead contained in D. The program having to choose an allocation size randomly likely slowed the workload down.

Workload E consists of calling malloc 1000 times for 1 byte and storing these pointers. Then, pointers are randomly selected from the array to be freed. Workload E took to the longest to run. This slow run time is likely due to two reasons. First the memory is allocated 1 byte at a time consecutively, requiring the malloc function to loop through large amounts of the memory block. Then, random locations is chosen to try to free pointers from. However, if the pointer in that location was already freed, a new location has to be selected. Since, all pointers need to be freed eventually, the program will need to randomly select a location a very large number of times.

Workload F chooses randomly between trying to malloc and free 1 byte in a random spot in an array of size 1000. When choosing to malloc, the program stores a char in the memory segment that the pointer returned points to. When the program eventually frees that pointer, it first checks to see if the char is still there. If it is not, an error is outputted. Our program never printed this error, demonstrating that our malloc implementation did not overwrite the data in memory previously allocated. This workload took the third most time to run. The program decides between allocating and freeing, which speeds up the runtime, but also chooses the location to malloc or free to randomly, thus slowing down the run time.

Testcase data:

	Workload A	Workload B	Workload C	Workload D	Workload E	Workload F
Total Time	~ 1.2	~ .0022	~ .07	~ .36	~ 1.3	~ .6 seconds
	seconds	seconds	seconds	seconds	seconds	
Average	~ .012	~ .000022	~ .0007	~ .0036	~ .013	~ .006
Time	seconds	seconds	seconds	seconds	seconds	seconds