Khalid Akash
157-00-3614

Program function: allocates space requested by the user to a memory array that holds 5000 bytes. Allows the user to also request freeing of previously allocated space. The number of bytes available is variable due to the addition of metadata associated with each allocation request.

Files included:
mymalloc.h:
Defines malloc() and free() functions.
Includes the node struct used for the metadata, and enumeration of the booleans true and false
Includes the following headers:
sys/time.h - used for timing each workload
stdlib.h
stdio.h

mymalloc.c
Holds the main algorithms for allocating the spaces necessary in the memory array, and the algorithms to free the allocated spaces, concatenate unused sections of the array.

memgrind.c
The interface to the malloc and free functions. Used to request and free allocated spaces. Also the source of the workloads used to test the program.

Input:
(void*)malloc(int x); x = number of bytes requested
x must be greater than 0 to be accepted by the malloc function; will return an error otherwise. Malloc request will be return an error if allocation space is not available for the number of bytes requested.
void free(void* p); p = pointer to the start of the allocated location
If the pointer is NULL or is not the start of an allocated location, the function will return an error.

Program discussion:
The following is a description of how the malloc and free calls work.

Malloc: when using malloc for the first time, the memory array is initialized to the value 0. Using this as a condition for initiating the memory array, the program calls a function called ‘initblock()’, to add a node at the beginning of the array and one at the end. Each node is of size 8 bytes, and has the following attributes: an integer representing how much space is left in front of the node (FI algorithm), and if that space is being used (TRUE representing used, and FALSE representing not used). The last array has a value of space = 0, and used value of FALSE.

These values were chosen because it is impossible for a user to invoke a node being created with 0 space. The end node provides a stop condition for any array traversals. Assuming the memory size requested is valid (error is checked), a traverse function is called that traverses all of the nodes of memory block and stops at the ‘end node’ looking for a node with a used condition of FALSE, and space available greater than the size requested and the size of a node. If a node is found, it is returned to the main malloc function, other wise a null pointer is returned. When a valid pointer is found, a new node is allocated after the space requested to partition the array into a used space, and unused space. That is the basic process of the memory allocation function.

Free: Free is the function called when the user/interface requests for an already allocated pointer to be deallocated. It accepts a void pointer. First, the pointer is passed to a function that traverses through all the nodes of the array, and checks the first location after each node to see if the pointer passed is in the array. If it isn’t, an error a boolean value of FALSE is returned, and an error is returned. If the pointer is found to be valid, it is converted into a node pointer, and the pointer is shifted back by 1 (sizeof(Node)). The reason this is done is to go to the pointers metadata to see if the space is used or not. If the space is found to not be allocated already by the user, an error will be returned, otherwise the program continues. The node(metadata) usage flag is set to FALSE. After this, two functions are called. checkLeft(), which checks the node to the left of the node being freed, and checkRight(), which checks the right of the node being freed. If a node is returned (meaning an unused Node is found), the space of the two will be concatenated with the space of the current node.

printmem: a function used for debugging, traverses entire memory array and prints all the metadata using the printf() function.

Memgrind.c: Workload testing source file. Along with functions that represent each test cases (workloads), there is also a function that accepts other functions as parameters. This was useful during the timing portion of the project, in which the respective cases were simply passed to the timer() function to generate the average timing data. There is also one more function that calculate the averages of 100 numbers (time).

Workload timing data:
The following data was received using the sys/time.h header file using the gettimeofday() function. There are some cases where that function induces an error and gives a negative time interval but repeated runs of the file will yield approximate the following results.

663 ms timing average for case A - This case was by far the largest due to the fact that individual bytes had to be allocated, and the malloc function utilizes node traversals. The more nodes allocated, the more time it took to execute this function. 1000 malloc calls weren't reached due to the metadata taking a significant amount of space out of the memory array during large requests of very small data packets.

24 ms timing average for case B - This case was the smallest even though it is similar to the first. The reason for this is due to the fact that traversals weren't necessary for this as the allocated memory was immediately freed right after.

380 ms timing average for case C
This case took a very long time because the program had to keep going until 1000 mallocs were complete, this wasn't nessesary for the previous tests. It also had many cases of failure if an attempt to free an unallocated pointer was made. The program that facilitated the case likely took quite a bit of time due to the excess amount of loops involved. Like the very first test, this test also had a large number of requested small packets of space, so that was also a factor.

345 ms timing average for case D
This case was less than its counterpart in case C, and was very similar to case C. The difference was that larger packets were used so not nearly as much traversals through the memory array to find the pointer location was necessary. The packet sizes were still relatively small so it still took a bit of time.

Usercases:
These two cases were meant to complement each other.

119 ms timing average for case E
This first test took relatively little time to complete. The case randomized between a malloc call requesting 10 bytes, and 1000 bytes. Freeing them in random was also a part of the case until 1000 frees (1000 mallocs by proxy) were made. The purpose of this test was to see how very large data packets (relative to the memory array) and very little packets interacted with each other, whether they caused any segmentation faults and any other errors.

376 ms timing average for case F
This case was very similar to the first, but the difference between the two byte packets were 1 bytes and 100 bytes. Clearly, a trend seen in all other test cases were seen with the timing. Due to the smaller packet sizes, significant longer time was taken. Curiously, even with the inclusion of the 100 byte packet, the average timing for case F was near that of Case C. This might indicate that there is a relationship between the size of packets entered to the malloc function, perhaps that the time exponentially increases the lower the packet size.