WILLIAM CHEN, GEON YEONG KIM

MyMalloc

The purpose of this project was to implement our own versions of the standard library functions malloc() and free(). Our library must include malloc and free that work like the actual malloc.
For example, when the user calls malloc with a specific amount of bytes, it must reserve a memory space in heap in order for the user to use that memory space until it is freed. In our little malloc project, an array of characters act as the heap in memory.

In our header file mymalloc.h, we have the following definitions:

#define MEMSIZE
    A preprocessor definition that allows us to test various memory array sizes.

```
void *mymalloc(size_t size, char *file, int line);
void myfree(void *ptr, char *file, int line);

#define malloc(s) mymalloc(s, ___FILE___, ___LINE___)
#define free(p) myfree(p, ___FILE___, ___LINE___)
```

    The features of C pre-processor allow malloc() and free() to report the source file and line number of the call that caused the error. By stealing the function names of the standard library, we are able to now begin creating our own little malloc.

Data Structure and Functionality:

The fundamental data structure we implement is a linked list. Each node contains 3 variables.

MYMALLOC

ISTAKEN
    Whether or not the block the node points to is in use. True if 1, 0 if false
BLOCKSIZE
    The size of the block the node contains, which starts right after the node in memory.
NEXT
    A pointer to the next node in the list. NULL if last pointer.

Our mymalloc() function handles the following cases:

SIZE

Malloc is called for a given size. Our function resizes using a brief modulus of 4 and addition of remainder in order to create consistent alignment of 4 byte intervals.

IF THE LIST IS NOT INITIALIZED

Initialize the list by creating the first node at the starting point in memory handled by testing if the first node is not taken and if it does not have a size, allocate requested size.

IF THE LIST IS INITIALIZED AND NO MEMORY HAS BEEN FREED

Iterate through the list until the end, create a new node and allocate the correct size.

IF THE LIST IS INITIALIZED AND MEMORY HAS BEEN FREED

IF THERE EXISTS AT LEAST ONE NODE WITH AT LEAST THE SIZE

Assign that node to the malloc'd size, the first available node is used.

IF THE AVAILABLE SIZE CAN ALLOW ANOTHER NODE TO BE CREATED

Create a new free node in between the existing node and the next node.

ELSE

Create a new node at the end of the list.

ERROR CASES

A) IF REQUESTED SIZE IS LARGER THAN TOTAL MEMORY

Prints Case and Returns NULL see Test 6

B) IF MEMORY IS OCCUPIED AND REQUESTED SIZE IS LARGER THAN AVAILABLE SPACE

Prints Case and Returns NULL see Test 12

C) IF REQUESTED SIZE IS ZERO

Prints Case and Returns NULL see Test 7

FREE

Given a pointer to a block location, free looks at its block header and frees it if it is a header of a block. This is determined by iterating through the linked list. Then, it calls the function coalesce.

ERROR CASES

A) CALLING FREE ON AN ADDRESS NOT OBTAINED BY MALLOC

Prints Case see Test 9

B) CALLING FREE ON AN ADDRESS NOT AT THE START OF A BLOCK (IN MALLOC)

Prints Case see Test 11

C) CALLING FREE ON A POINTER THAT HAS BEEN FREED

Prints Case see Test 10

D) CALLING FREE ON A NULL POINTER
Prints Case see Test 8
E) CALLING FREE ON AN UNALLOCATED BLOCK
Prints Case see Test 13

COALESCE

Checks if the block before and after the freed block can be merged. If possible, the first block in the link list that is free is assigned the total space of the following nodes and headers that are consecutively freed.

These functions are provided in the mymalloc.c library.

TEST PLAN

The goal of this project is to create a basic malloc and free that mimic the functionality of the existing functions.
The properties that are necessary and sufficient to determine success are:

1) Allocation
2) Coalescence
3) Deallocation
4) Proper error handling
5) Reasonable time efficiency

ALLOCATION

Test 1 displays the ability of the program to initialize a linked list and re-assign the location after immediately freeing.

Test 2 displays the ability of the program to sufficiently create and add to the linked list structure with proper allocations.

Test 3 showcases that order of operations of free and malloc does not affect the functionality of the library, and is therefore resistant to random calls.

Test 4 showcases the ability of the program to accept varying sizes without issue of alignment. The size range can be changed by changing its modulus.

Test 5 showcases the ability of error detection of allocation, by preventing continued allocation once memory is limited. This test allows additional small allocations as new freenodes can be created given varying sizes. IE Chunks of 50, freeing even chunks, then allocating chunks of 3, allows for additional allocations of certain sizes in the free region space and efficient space usage.

COALESCENCE

Test 3 showcases the ability to coalesce free chunks given random ordering of function calls.

A print statement has been commented out of the Coalesce function in mymalloc.c that can display the size of new coalesced nodes for visual interpretation if necessary.

DEALLOCATION

Test 1 displays the ability to free a given pointer successively.

Test 2 and 4 displays the ability to free given pointers in succession.

Test 3 displays the ability to free given pointers at random without triggering error cases

Test 5 displays selective freeing of allocations without causing issues to the data structure of memory

ERROR HANDLING

Potential error cases have been outlined under the functionality component of the library.

Test 6-13 test each error case individually.

TIME EFFICIENCY

Free at worst case searches through the entirety of the linked list structure (N allocations) and determines that a given pointer is not able to be freed for whatever reason. This is an N time efficiency worst case.

Malloc at worst case searches through the entirety of the linked list structure until the end of the linked list and either adds a node or triggers an error case. This is also at N time efficiency where N is the existing number of allocations.

See Average Test Times under Test Cases for specific time averages of 50 runs with sufficient N iterations.

Under these combined conditions. This library is completely functional and reports additional error cases that are unreported in standard variations.

TEST CASES

Our heap memory will have a space of 4096 with malloc that will return a pointer to a block of memory while free will tell the system that the user is done with the reserved space in the heap.

<u>TESTS PROVIDED BY PROFESSOR</u>

Each of Tests 1-5 are ran 50 times to find average time of completion.

TEST 1

MALLOC Then FREE a 1-byte chunk, 120 times.

TEST 2

MALLOC a 1-byte chunk 120 times, THEN FREE all 120 allocations.

Operated by storing the pointers in an array at the head of memgrind.c called ptr

TEST 3

RANDOM CHOICE OF

Allocating a 1-byte chunk and storing the pointer

Deallocating one of the chunks of array

Occurs until malloc has been called 120 times.

DESIGNED TESTS

TEST 4

Allocate random chunks of sizes 1-8 bytes, until the pointer storage array is full. Then free all allocations.

TEST 5

Allocate 50-byte chunks until all possible spaces are in use. Store pointers in the pointer storage array.

*Note: This causes error case of MALLOC CASE B to print 50 times as it occurs each run of the test as the test is designed to test the ability of stopping at the error case

Deallocate every other pointer in the array. For instance, the even indexes.

Allocate 35-byte chunks to the freed pointer spaces in the array.

Deallocate all pointers.

AVERAGE TEST TIMES

The average time to execute Test 1 2.800000 microseconds
The average time to execute Test 2 49.480000 microseconds
The average time to execute Test 3 39.919998 microseconds
The average time to execute Test 4 53.860001 microseconds
The average time to execute Test 5 48.000000 microseconds

ERROR TESTS

Tests 6-13 are designed to showcase the handling of potential error cases of the code and print once.

These test cases currently print and return, but are capable of calling for the exit of the program as well if needed to mimic real error case handling.

Through these tests on memgrind.c, we have fully implemented my little malloc.