Jerry Olds  
1001533643  
CSE-3310-002  
11/25/2018

Assignment 4

1. Problem Statement

The program should be able to manage an initial block of allocated memory using malloc, calloc, realloc, and free functions that are created by the developer. The functions that are created should have the same functionality as the built in malloc, calloc, realloc, and free functions.

The malloc function should search the block of memory allocated at the beginning of the program for contiguous memory that is the size requested in the malloc call. If there is no contiguous memory large enough to satisfy the size requested within the malloc call, the program shall return an error statement and terminate the program. If contiguous memory is available, the malloc function will return the address of the first position of the contiguous memory to be used.

The calloc function will take a number (n) and a size as parameters and will call the malloc function n times. Each time the malloc function is called it will send the size to the malloc function as an argument.

The realloc function will take a pointer and a size as parameters. The realloc function will first call the free function and free up any memory that the pointer is using. The realloc function will then call the malloc function and allocate memory to the pointer with a new size.

The free function will take a pointer as a parameter and will set any memory that it is using to null.

2. Schedule

|  |  |  |
| --- | --- | --- |
| **Tasks** | **Expected Time** | **Actual time** |
| Design/Planning | 3 hours | 5 hours |
| Malloc function implementation | 5 hours | 2 hours |
| Calloc function implementation | 1 hour | 5 minutes |
| Realloc function implementation | 1 hour | 5 minutes |
| Free function implementation | 1 hour | 5 minutes |
| Testing | 2 hours | 4 hours |

3. Requirements

|  |  |
| --- | --- |
| **Requirement** | **Type** |
| The program shall accept a call to \_malloc function and it shall preform the same functionality as the built in malloc function | Functional |
| The program shall accept a call to \_calloc function and it shall preform the same functionality as the built in calloc function | Functional |
| The program shall accept a call to \_realloc function and it shall preform the same functionality as the built in realloc function | Functional |
| The program shall accept a call to \_free function and it shall preform the same functionality as the built in free function | Functional |
| The program shall declare a block of 1GB static memory and manage that block of memory throughout the duration of the program | Functional |
| The program shall return an error message if there is no contiguous block of free memory that is equal to the size requested | Functional |
| The program shall return an error message if the pointer requests more than 1 GB of memory | Functional |
| The malloc function shall be nonrestrictive, allowing the pointer to use more memory than requested | Functional |
| The program shall be able to distinguish between a null terminator of a string and a free byte of memory that has the value null | Functional |
| The malloc function shall be able to be used indefinite amount of times throughout the program and operate as intended. | Non-Functional |

4. Design

The way my malloc function works is it searches for contiguous NULL bytes in a character array of size 1 GB and returns the starting position of the contiguous NULL bytes. For example, in the diagram below, bytes 0 and 1 are storing the characters ‘A’ and ‘B’. Bytes 2-8 are contiguous null bytes. Bytes 9 and 10 are storing the characters ‘C’ and ‘D’ and byte 11 is null. If the program were to call the malloc function and send an argument of size 5, the malloc function would return the address of byte 3, leaving byte 2 as null terminator for the ‘A’ and ‘B’ values. The pointer would then be able to use the rest of the 1048572 bytes in the character array with the ability to overwrite already stored values. I wanted my malloc function to be very unrestrictive, much like the built in malloc function in C.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | \0 | \0 | \0 | \0 | \0 | \0 | \0 | C | D | \0 |

0 1 2 3 4 5 6 7 8 9 10 11

Y = (char\*) malloc(5)  
Y[0] = ‘X’  
Y[1] = ‘Y’  
Y[2] = ‘Z’  
Y[3] = ‘\0’

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | \0 | X | Y | Z | \0 | \0 | \0 | C | D | \0 |

0 1 2 3 4 5 6 7 8 9 10 11

5. Alternatives

One alternative design I considered implementing was returning a chunk of the character array equal to the size requested in the malloc. This would better ensure that the pointer not use more memory than requested. Another alternative design I considered implementing was resizing the 1 GB of memory every time memory was allocated or freed. That way memory that had been allocated could not be used again until freed. In the end I decided to go with a very nonrestrictive malloc method that gives the pointer a lot of freedom. The pointer will be allowed to use more memory than allocated if needed with the risk of overwriting already stored data.

6. Constraints

The first constraint is that the pointer cannot request more than 1 GB of memory. The program accommodates for this by checking to see if there are contiguous unused bytes equal to the size requested. If the size requested is bigger than the size of the initial 1 GB allocated, there will never be 1 GB + n contiguous amount of free bytes, therefore leading to an error message and termination of program.

The second constraint was that the size requested within the malloc call had to be contiguous free memory. The program accommodates for this by going through the initial character array and when there is a null byte, a counter increments by one and goes to the next byte. If the counter reaches the size of the malloc requested, the malloc function will return the position of the first null byte.

The third constraint is that the malloc function could not overwrite a null terminator of a string. The program accommodates for this by not counting the null byte after the last byte that has a value in it that is not null.

7. Source code

#include "malloc.h"

// the big cheat

static char X[1000];

void \*\_malloc(size\_t size)

{

// Variables

short free = 0; // Boolean variable

int free\_contiguous = 0;

int position = 0;

int i;

// Search for free memory

for(i = 0; i < sizeof(X); i++)

{

if (X[i] == '\0')

{

if(!(i > 0 && X[i - 1] != '\0'))

free\_contiguous++;

}

else

free\_contiguous = 0;

if(free\_contiguous == size)

{

position = i - size;

free = 1;

break;

}

}

// If memory cannot be allocated

if(free == 0)

{

printf("Error: No contiguous memory available for malloc request.\n");

exit(EXIT\_FAILURE);

}

return &X + position;

}

void \_free(void \*ptr)

{

memset(ptr, '\0', sizeof(\*ptr));

}

void \_calloc(size\_t nmemb, size\_t size)

{

int i;

for(i = 0; i < nmemb; i++)

\_malloc(size);

}

void \_realloc(void \*ptr, size\_t size)

{

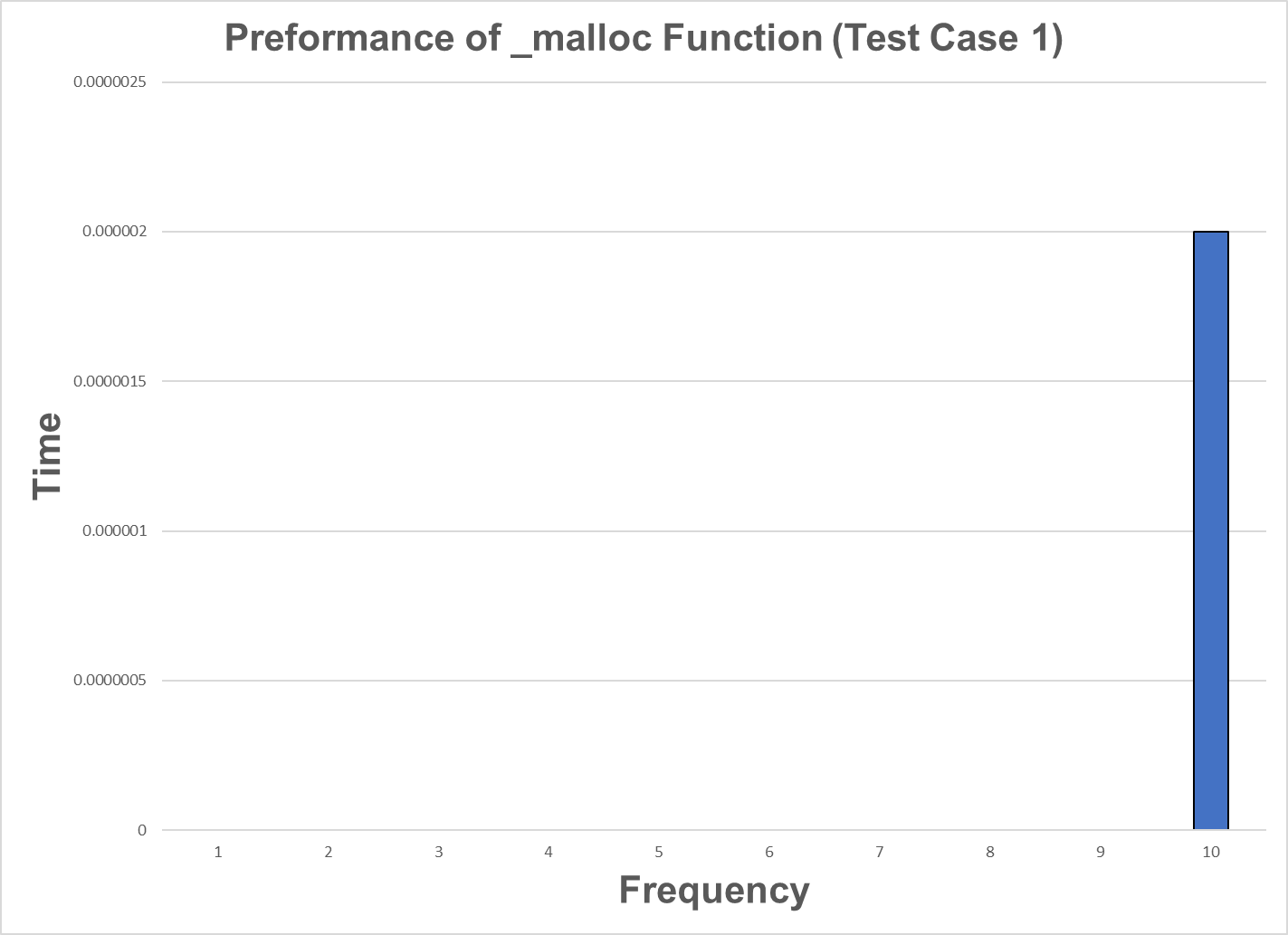
\_free(ptr);

\_malloc(size);

}

8. Analysis

I created two test cases to test the performance of my malloc function. The first test case tested the consistency of the performance of the malloc function. The test case measured the amount of time it took to execute the malloc function given an argument of 10 bytes. The test case was run 10 times and it took 0.000001 seconds to execute each of the 10 times. The malloc function was very consistent.



The second test case I created was to see if there was any relationship between the time it took to execute the malloc function and the size of memory requested. The test case generated a random number between 10 and 200 and sent that as an argument to the malloc function. After 20 tests I found that there was no relationship between size and time of execution. Each time the test case was run the malloc function executed in either 0.000001 seconds or 0.000002 seconds. Sometimes a number below 100 would take 0.000002 seconds to execute, and sometimes a number above 100 would take 0.000001 seconds to execute and vice versa. There was no correlation with the range of values that I used in the test case.

