CSE 3320 – Operating Systems

Arvin Saju

1001705173

04/14/2023

## Executive Summary/Introduction:

The dynamic allocation of memory is a commonly used tool in programming when the amount of space needed is unknown or will otherwise vary. While useful, there are some complications that result from dynamically allocating memory such as memory leaks and segmentation faults that one must carefully fix to avoid issues in the program. In this coding assignment, the main goal was to actually rewrite the functions that would be needed to dynamically allocate memory such as finding free blocks of memory, malloc, calloc, realloc, and free.

To describe these functions in some more detail, the best function with which to start would most likely be malloc. In malloc, the goal of the function is to allocate a block of a provided size for the program to use. In order to do this, it must first find a block of memory that is currently free and not in use already. The way this works is that there is a linked list(called heapList in the code for the assignment) that contains the list of all free blocks that are in the heap and can be used. This memory is then traversed through, and the correct block is chosen depending on the desired algorithm. The first of these implemented algorithms is called first fit. In first fit, the list of blocks is traversed and the very first block that is at least as large as the requested size is the one which is allocated. Then there is next fit which is a variation of first fit in that the first free block in the list which is larger than or equal to the requested size is used to allocate the memory. The difference between first and next fit is that the position in the list is maintained and the next time that memory needs to be allocated, the list of free blocks is traversed starting from the block at which the last allocation stopped. The next algorithm that was implemented in the program is best fit. Best fit is when the list is traversed from the beginning just like in first fit however the program traverses through the entire list and keeps track of the smallest block of memory available which is still large enough to allocate the needed memory. In contrast to this, the final algorithm implemented for finding the block of memory to allocate is worst fit, in which the entire list of blocks is traversed once again from the beginning and then the largest memory in which the program can fit is selected. The final function that was implemented in the malloc function is the split function. The purpose of the split function is that when memory is being allocated, if the memory selected by the fit algorithm used is larger than needed then that block is split up into two separate blocks with one being the correct size and that is the one that is allocated, while the other one is inserted into the list of free blocks.

Calloc and realloc were also implemented in this program.  Calloc calls the earlier explained malloc function within it and then uses the memset function to initialize all the memory to zero and remove any garbage values that are in the addresses. Realloc also calls the malloc function however it calls it for a new pointer depending on the value you want to increase the size to. After this memory has been allocated it then use memcpy to copy over the values in the original block to the newly allocated larger block. The free function is then called on the original block. The free function used here as well as after any other allocation of memory used in the assignment is implemented by setting the pointers to be null and then declaring the block to be free as one of the struct properties for the block.

## Test Implementation

In addition to the originally provided test cases, 4 additional test cases were implemented. The first added test case was a allocation of array of 1000 integers, assigning values to each index, and then keeping a running total of all the values in the array and printing it out before freeing the array. The second added test case was allocating space for an array of 10,000 strings and then freeing each element of the array before freeing the entire pointer to the array as well. Essentially this program was creating a two dimensional array of characters due to the way in which strings are stored. The third added test case was creating two separate arrays, one which contained an array of pointers to integer values with the second containing memory allocated for an array of strings. Both of these arrays had 15,000 elements and were all freed at the end of the program. Finally, the fourth added test case was the creation of a massive array of 1,000,000 float values and assigning values to each element of the array. A running total of the sum was kept and printed out. The array was then reset to zeros and freed.

## Test Results:

| | Standard Malloc | BestFit | | | | WorstFit | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Performance(secs) | Performance(secs) | Fragmentation | Splits/Growth | MaxHeap | Performance(secs) | Fragmentation | Splits/Growth | MaxHeap |
| Test1 | 0 | 0.003 | 2 | 0/0 | 66560 | 0.002 | 2 | 0/0 | 66560 |
| Test2 | 0.001 | 0.003 | 1027 | 0/0 | 1,115,136 | 0.006 | 1027 | 0/0 | 1115136 |
| Test3 | 0.001 | 0 | 4 | 0/0 | 5,472 | 0 | 4 | 0/0 | 5472 |
| Test4 | 0.001 | 0 | 4 | 1 0 | 3072 | 0.002 | 4 | 0/0 | 3072 |
| MyTest1 | 0.001 | 0.001 | 2 | 0/0 | 5024 | 0 | 2 | 0/0 | 5024 |
| MyTest2 | 0.015 | 0.035 | 10,001 | 0/0 | 150,081,024 | 0.024 | 10,001 | 0/0 | 150,081,024 |
| MyTest3 | 0.207 | 0.208 | 30,001 | 0/0 | 22,338,464 | 0.194 | 30,001 | 0/0 | 22,338,464 |
| MyTest4 | 0 | 0.012 | 2 | 0/0 | 4,001,024 | 0.005 | 2 | 0/0 | 4,001,024 |
| | NextFit | | | | FirstFit | | | | |
| | Performance(secs) | Fragmentation | Splits/Growth | MaxHeap | Performance | Fragmentation | Splits/Growth | MaxHeap | |
| Test1 | 0.002 | 2 | 0/0 | 1024 | 0.004 | 2 | 0/0 | 66,560 | |
| Test2 | | Segmentation Fault | | | 0.001 | 1,027 | 0/0 | 1,115,136 | |
| Test3 | | Segmentation Fault | | | 0.002 | 4 | 0/0 | 5,472 | |
| Test4 | 0.001 | 3 | 0/0 | 1024 | 0.003 | 4 | 1 0 | 3,072 | |
| MyTest1 | 0 | 3 | 0/0 | 1024 | 0.001 | 2 | 0/0 | 5,024 | |
| MyTest2 | 0 | 2 | 0/0 | 1024 | 0.033 | 10001 | 0/0 | 15,081,024 | |
| MyTest3 | | Segmentation Fault | | | 0.2 | 30,001 | 0/0 | 22,338,464 | |
| MyTest4 | | Segmentation Fault | | | 0 | 2 | 0/0 | 4,001,024 | |

## Discussion of Results:

Based on the results shown above it seems that the implementations of best fit and worst fit seem to take the longest. This makes sense and is reasonable because in both of those algorithms, you have to traverse the entire list of nodes each time which takes a considerable amount of time. The standard form of malloc takes overall the least amount of time since it is using the optimized built in function instead of a custom implementation. The first fit is the next fastest which makes sense since it instantly allocates memory in the first block it can fit in without considering memory efficiency. Ideally next fit should be somewhere in the middle of the range since it takes space into consideration while not going through the entire linked list repeatedly, however it seems that there were some issues with the implementation of the code as several of the trials had segmentation faults which also indicates that the other values obtained for next fit may not be correct.

## **Conclusion:**

The creation of this program was a very valuable experience since it shows in depth how malloc and other dynamic memory allocation functions are done by the operating system as outlined in the introduction. By writing our own implementations of these functions, a greater understanding of how they work is required to actually rewrite them and debug any resulting issues. This assignment gave a clear view of memory management and how the system handles that in such an efficient and effective manner.