Elvin Palushi
CSE 3320-001
Malloc Assignment
3/16/24

Malloc Report

Executive summary:

In this assignment, I implemented my own version of malloc and free, as well as calloc and realloc, along with the four fit heap management strategies, which are first fit, next fit, best fit, and worst fit, in C. I made sure malloc, free, calloc, realloc, and the four fit strategies worked by running them against the tests provided. I also made sure the heap statistics in the program printed accurately regardless of the test it was given. After everything was working, I created a well enough benchmark to test malloc and free with the four fit algorithms in order to gain certain statistics when benchmarking. When I benchmarked my malloc with the four fit algorithms, I captured the execution times along with the heap statistics for each fit algorithm. I also benchmarked system malloc with my benchmark, but in this case, I was only able to capture the execution time without the heap statistics. Finally, I capture the heap statistics for each fit.

Description of the algorithms implemented:

When implementing the four fit algorithms, I started with best fit and worst fit since first fit was already implemented. The first fit code is not complicated, you are just iterating over the linked list from the beginning every time until you find a free node that has a big enough size for the request. Back to best fit, I initialized the size to the smallest integer possible since I will be comparing it to the space left over when getting the difference in sizes. I made sure to keep track of a winning node that is free, has a big enough size, and enough space in order for the node to best fit in the heap. After that, just keep iterating over the linked list while you don't fall off of it. Worst fit is the same, however, you just make sure the space available is the worst fit when assigning the node in the heap by initializing a size variable to the largest integer possible. Finally, I worked on next fit which required the most thinking in order to get it working. I started by keeping track of a previous node and a winning node. After that, I iterated through the linked list while there was a current node. If that current node was free and had enough space, I set the previous node to the current node. If the new node existed, was free, and had enough space, then I assigned the node to the winning node. I then set the current node to the previous node in order to start at the node where it stopped, which fulfilled the requirements for next fit. Basically, the four fit algorithms are just iterating through a linked list trying to find an available node to manipulate in order to meet each fit.

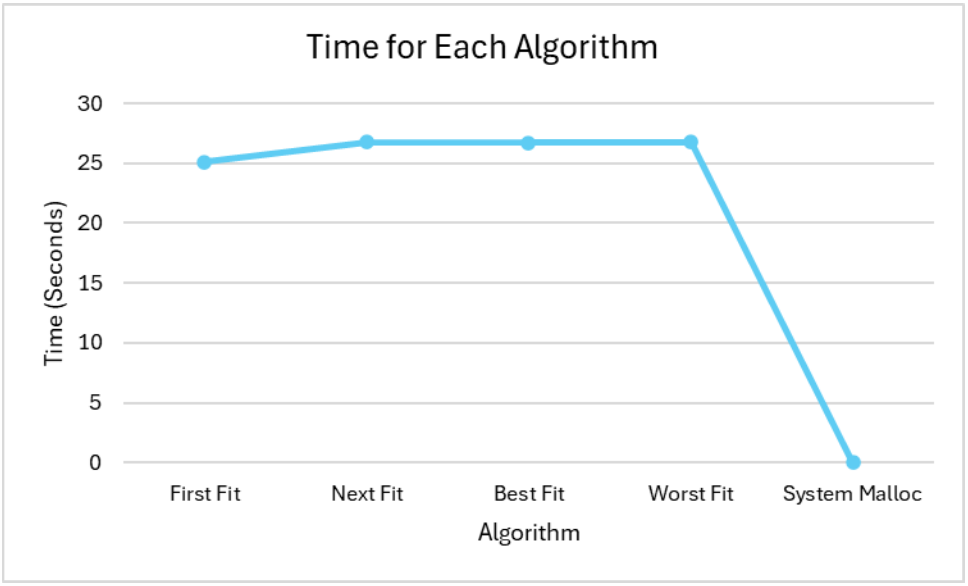
Test implementation:

For my benchmark, I decided to work with big numbers in order to shake out any errors I could have had in my code. I started with a simple malloc of 65535 bytes, and then immediately freed it in order to make sure malloc and free worked properly. After that, I created a pointer array that held 100,000 elements. I looped through the first 50,000 elements and malloced 100 bytes each iteration. I then freed the first 25,000 elements of the pointer array which then creates a fragmentation effect of the heap. After freeing the first 25,000 elements, I allocated the last 50,000 elements of the pointer array which mallocs 100 bytes each iteration. I then iterate from 50,000 to 75,000 and malloc alternating sizes of 32 bytes and 128 bytes. I do the same thing from 75,000 to 100,000 and malloc alternating sizes of 64 bytes and 256 bytes. These loops overwrite the 50,000 to 100,000 loop in order to create a fragmentation effect since you are changing the 100-byte allocations to alternating allocations of sizes smaller and larger than the 100-bytes. I also did this because I wanted large amounts of data to be used in order to weed out any errors in my code. Finally, I get the execution time in order to compare the results among the four fit algorithms and system malloc.

Test results:

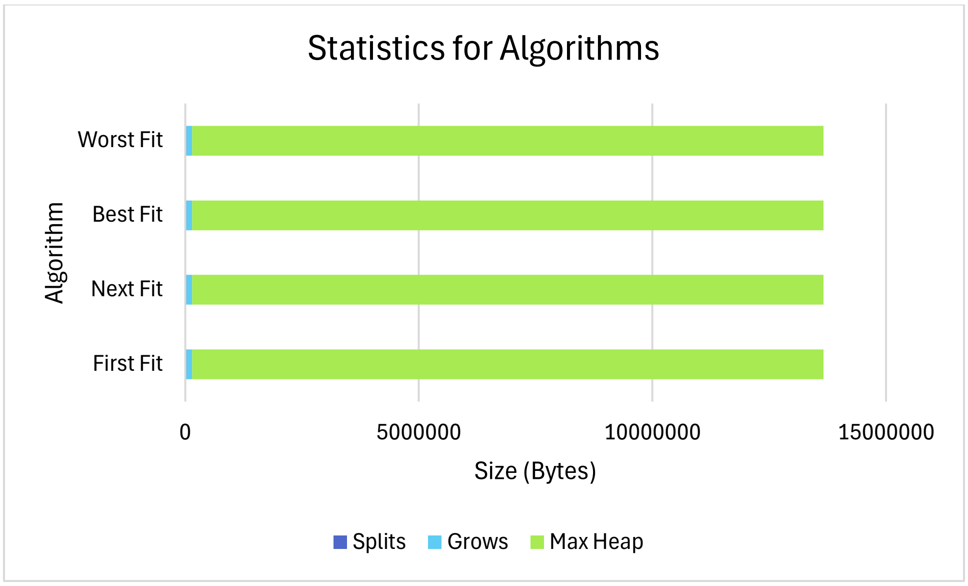
When running the benchmark, I tested in the order of first fit, next fit, best fit, worst fit, and then system malloc. For the first chart and table, I captured the execution time for each entry, which can be seen below…

	First Fit	**Next Fit**	**Best Fit**	**Worst Fit**	**System Malloc**
Time	25.083 Seconds	26.754 Seconds	26.704 Seconds	26.759 Seconds	0.00824 Seconds



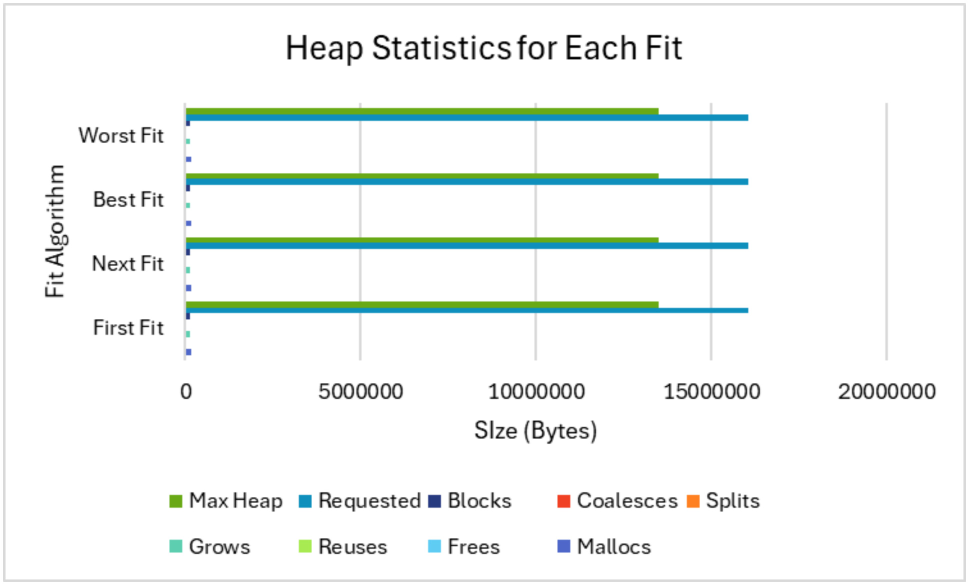
After that, I captured the amount of splits, grows, and the size of the max heap when running the benchmark. I could only obtain these statistics from the four fit algorithms, which leaves out system malloc, which can also be seen below…

	First Fit	**Next Fit**	**Best Fit**	**Worst Fit**
Splits	25528	25528	25528	25528
Grows	124473	124473	124473	124473
Max Heap	13513728	13513728	13513728	13513728



I also captured the heap statistics for the four fit algorithms…

	Mallocs	Frees	Reuses	Grows	Splits	Coalesces	Blocks	Requested	Max Heap
First Fit	150002	25001	25529	124473	25528	24999	125002	16066560	13513728
Next Fit	150002	25001	25529	124473	25528	24999	125002	16066560	13513728
Best Fit	150002	25001	25529	124473	25528	24999	125002	16066560	13513728
Worst Fit	150002	25001	25529	124473	25528	24999	125002	16066560	13513728



Analysis of Results:

When running the benchmark, I ran each fit and system malloc five times and got an average execution time, which is what you see in the table. I also ran the benchmark on codespaces exclusively, which could result in a bit slower times compared to if I had done it on my personal machine. Comparing first fit to next fit, first fit has a faster time than next fit because after I free the first 25,000 elements of the pointer array, I am allocating the second 50,000 elements of the pointer array. When I free the pointer array, it causes some fragmentation which first fit and next fit are affected by. First fit starts from the beginning, which means it can be quicker in finding suitable blocks to use while next fit starts from the last allocated block. This adds more time to the execution since the fragmentation allows next fit to be disadvantaged compared to first fit, which is why first fit is faster by 1.67 seconds on average. After that, the program doesn't really differ for first fit and next fit. Onto best fit and worst fit, you can see that best fit outperformed worst fit by about 0.05 seconds on average. When best fit and worst fit work through the program, the only thing that somewhat affects them is the fragmentation effect from freeing the 25,000 elements in the pointer array. You can then see how best fit and worst fit work through the program after that. For the second half of the array, it iterates from 50,000 to 75,000 and mallocs alternating sizes of 32 bytes and 128 bytes. It then does the same thing from 75,000 to 100,000 and mallocs alternating sizes of 64 bytes and 256 bytes. This allows the heap to allocate alternating sizes of smaller sizes and a little bit larger sizes, which allows best fit and worst fit to differ in a small sense. Worst fit would be advantaged if the request sizes were large, i.e. 2048 bytes. However, best fit and worst fit won't see much of a performance difference since they both have to iterate over the entire heap to find a slot available. I then tested system malloc and unsurprisingly enough, it was very fast. System malloc has over 50 years of optimization, so it is understandable that it is around 0.008 seconds fast compared to my 26 second malloc. Moving on to the second table, you can see that all of the statistics shown are the same values for each fit. This is because I am freeing a lot of blocks and they are being coalesced together, which doesn't allow too much fragmentation to occur in order to change the statistics. I designed the benchmark in order to work the same for each fit in terms of statistics. I wanted to get performance differences to show how the four fits differed in my benchmark. This is also the case with the last table. All of the values are the same due to the reason I gave previously. The most significant difference I have seen is in the performance.

Conclusion:

Implementing my version of malloc and free, along with calloc, realloc, and the four fit heap management strategies took quite some time. Splitting, coalescing, the four fit algorithms, calloc, and realloc were difficult at first, but now that I completed the assignment, it's just iterating over a linked list to determine which node to use for whatever use you need. The benchmark to test the implementations was also a little confusing, but it made sense when I was benchmarking for the results. I would say that the four fit algorithms all depend on the purpose you are trying to achieve. Something like first fit and next fit have different uses when implementing them for a program. I also have a better understanding of what's going on under the hood with memory management, especially now that I did it in C.