Malloc Report

Duy Tran

EXECUTIVE SUMMARY

The report comprises details used to implement malloc and free which is then later compared to the system library. To implement these functions, it incorporated 4 different independent algorithms to allocate memory; best fit, worst fit, first fit, and next fit. To assure its validity and compare it to the system library 8 tests are given and 3 are made. The 3 tests evaluate performance, number of splits, heap growth, heap fragmentation, and max heap size. Following these tests, their results are shown as heap statistics and then analyzed. All work provided is not done entirely by one person but also with the help of an AI assistant, its performance is reviewed along with the other details.

ALGORITHMS IMPLEMENTED

The customized allocator, malloc.c, mimics the system functions malloc(), free(), malloc (), and realloc() by incorporating different allocation strategies, such as Best Fit, First Fit, Next Fit, and Worst Fit.

Beginning with Best Fit, the function scans the entire list to find a block that is large enough but the smallest size to fit the allocation. It iterates through the list while holding onto the pointer to the smallest size, and then, if a better block is found, the best is updated. Finally, at the end of the iteration, if the best is found, the function returns, else null if none is found.

Next, First Fit, this algorithm scans the list from the beginning to find the first free block that is large enough. In code, it will start iterating the list using a while loop looping the current block that is not free or not big enough and ending when otherwise or returning with a null if non if found.

Next Fit scans the block starting from where memory was last allocated at. It will begin from the last allocated if present and continue if it is near the end of the list by wrapping around to the start search once more.

Then the 4^{th} algorithm is Worst Fit, an algorithm that scans the list to a space that is free and large enough, but the space is the largest size of the other available spots. It will iterate the list holding a pointer to the largest block and update the worst when a larger block is found.

TEST IMPLEMENTATION

There are a combined total of 11 test programs where 9 small-scale codes were interpreted which improved the custom allocator to be as close to mirroring the system allocator. The other two test5.c and test6.c were implemented to compare the custom allocator against the standard system call malloc().

In all tests custom allocator tracks memory management, including the number of successful malloc and free calls made by the user, frequency of block reuse, requests for new blocks, and block splitting. It also monitors the coalescing of adjacent free blocks to reduce fragmentation, the number of blocks currently in the free list, the total amount of memory requested by the user, and the maximum size of the heap during program execution.

Test5 evaluated the programs' splits, heap growth, and maximum heap size by aligning allocations and deallocations. Test5 consists of allocating memory using malloc with various sizes which will grow the heap 4 times, two frees of specific blocks to empty those blocks in the heap, two more mallocs that can fit into the recently freed space causing splits, then freeing the rest of the blocks. There are limitations in C on how the standard system call could display its statistics. So, the test for the system's splits, heap

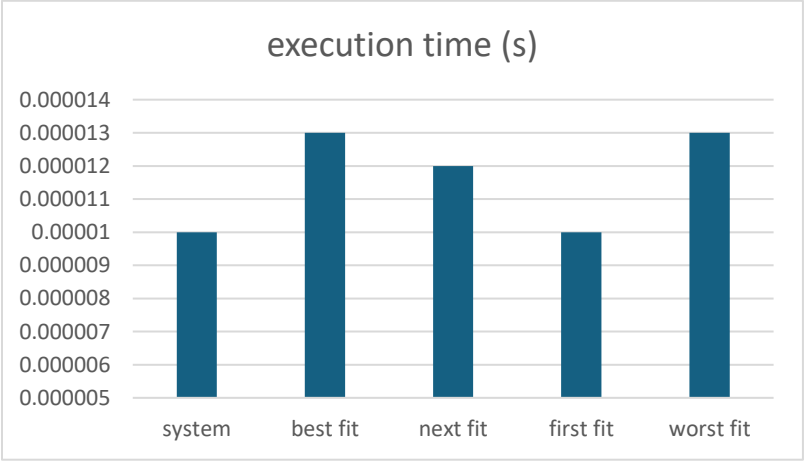
growth, and max heap size are commented in the code on what is to be expected for those results. But, the sequences are timed to view both programs' execution time and actual peak physical memory usage is tracked to be compared with memory usage tracked by the customer allocator.

Test6 evaluates heap fragmentation and performance by allocating various sizes and creating fragmentation. It is accomplished by the allocation of memory in blocks of different sizes and freeing every third block creating fragmentation. This is followed by allocating more blocks to analyze the program scanning process in a fragment scenario and then allocating a large block to view the impact of fragmentation. This sequence is timed to view both programs' execution times as part of the performance comparison. Then the fragmentation is calculated by the ratio of unusable fragmented space.

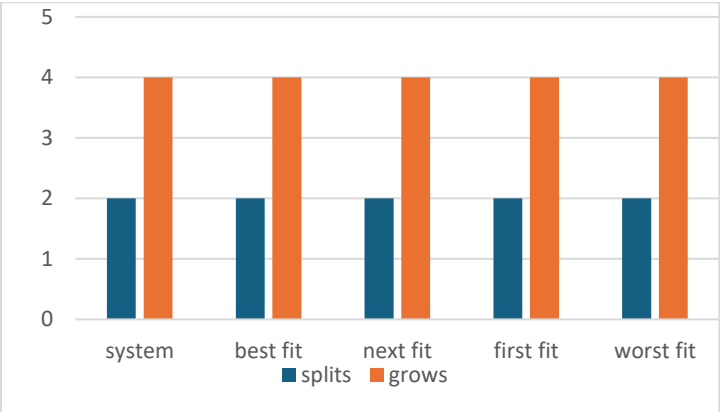
TEST RESULTS

In these graphs “system” is the standard system call and “best fit”, “next fit”, “first fit” and “worst fit” is the algorithms in the custom allocator.

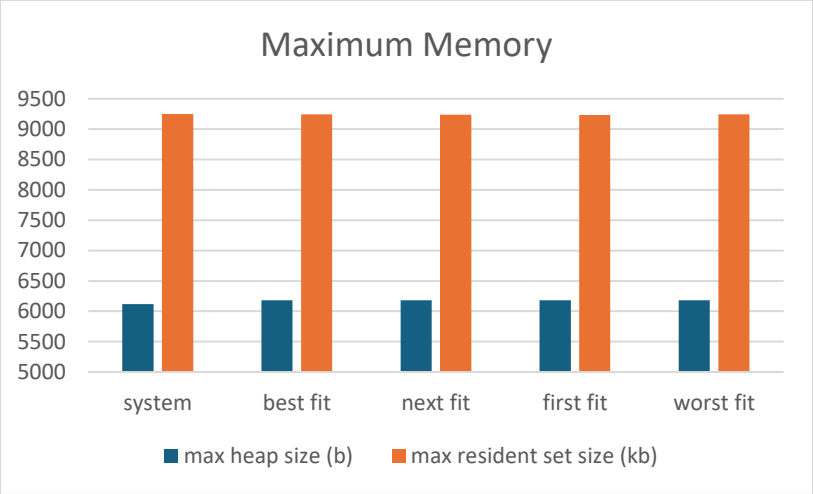
Test5



graph 1. Displays execution time in seconds running test5.c.

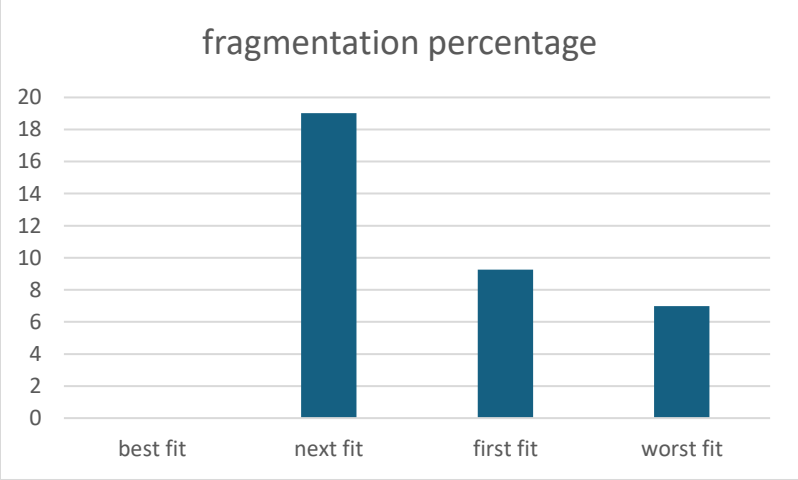


graph 2. Comparison of the number of splits and growth in a heap between standard system call and the custom allocator with 4 algorithms.

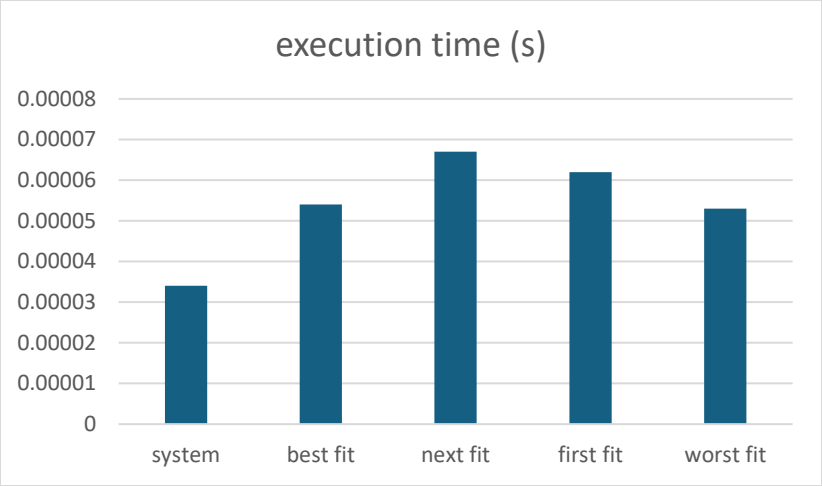


graph 3. Comparison of peak physical memory usage of the program with the memory usage tracked by the custom allocator.

Test6



graph 4. System percentages are unknown while best fit percentages are at 0%.



graph 5. Execution time in seconds running test6.c.

	best fit	next fit	first fit	worst fit
mallocs	27	27	27	27
frees	26	26	26	26
reuses	4	3	3	3
grows	22	23	23	23
splits	2	2	2	3
coalesces	23	22	23	24
blocks	2	4	3	3
requested	70100	70100	70100	70100
max heap	64336	42272	67368	67368

graph 6. Heap Management Statistics.

RESULTS INTERPRETATION

Test5's primary focus was on drawing statistics for block splitting, heap growth, and max heap size. Splitting blocks occur when a freed large block is occupied by a smaller block when the smaller allocation is called. Heap growth increments relate to each user's call for more memory when the previous heap size could not fit. As seen in graph 2 the number of splits and growth are consistent throughout all the algorithms. This is to be expected as this test is not to purposefully test algorithm limitations but to accurately gather results on paper which then can be translated to correctly constructing the algorithm.

As small as the test code is the difficulty of constructing the algorithm is to accommodate for underlying system calls that would add to the statistics. Heap growth is an example where the system would increment by printf function but has been resolved to only accept growth by the user. Another example of challenges presented by the system was accurately tracking memory usage for both the custom allocator and the standard system call. There were several options to consider such as Valagrind's massif, malloc.h, and sys/resource.h to compare to the manual custom allocation tracking. Ultimately sys/resource.h is chosen for its consistency, and simplicity and is the closest to obtaining the desired results. The results of this are displayed in graph 3. The massive difference between the max heap gathered by the custom allocator and sys/resource.h is that the custom allocator includes the total size

allocated and the overhead of 24 bytes per allocation. Sys/resource.h includes all memory used in the process like heap, stack, code and data segment, and much more.

Test6's larger test size demonstrates the difference between execution time (graph 5) and the results are not alarming as the results are expected in comparison to each other. The system allocator is the fastest as it has been tuned through many years and next fit is the slowest concluding that the algorithm of searching in a loop would be slower than the other alternatives. As for fragmentation, best fit 0% fragmentation demonstrates optimality in choosing small blocks to minimize free space. While next fit fit with the highest fragmentation at 19.02% is not optimally choosing free memory for allocated memory to be in.

To conclude this section, the standard system call is more optimal in overall performance though no modular in extracting wanted results compared to the custom allocator. But as flexible as the custom allocator can be it is also more susceptible to errors such as miscalculations. These tests provide a reliable insight into the custom allocator's accuracy in tracking fragmentation, inaccuracy in memory usage output, strength in flexibility, and weakness in inefficiency.

AI PERFORMANCE

In this case, an AI assistant did help more than it did hinder, but not by much. AI excelled in providing many options of code to choose from but when asked to be corrected it will continue to push the same code with minor changes with the same error. An assignment like this has never been done before now so topics and ideas are foreign to newcomers. Because of AI, beginners can learn from subjects introduced by AI, so any information presented prevalent to the subject is bound to learn more than less. On the other hand, it is based on the user's judgment to seek out the many errors and correct them because AI is never perfect in this type of situation. Users can only learn so much from AI before it just reiterates the same statement with or without error. With the topics it presents, I had to research more into it and fact-check it. Because of that, I learned about the logic of each function which allowed me to implement it in code.