Andrew McLaughlin
1001988355
CSE 3320 - 003

Malloc Implementation Report

Summary/Overview

For this assignment I have been tasked with implementing my own version of malloc which is a function native to the C programming language that is used to dynamically allocate memory on the process's heap during runtime. The purpose of the arena allocator is to create and test the four common strategies/algorithms for allocating memory on the heap and test them against each other and against the optimized system malloc. The metrics used to compare performance are system time and a set of metrics used to describe the heap's state. This includes number mallocs, frees, reuses, grows, splits, coalesces, blocks, requested memory, and the max size of the heap. Based on my findings, the next fit and first fit algorithm on average record the lowest time compared to the other algorithms which is consistent across all tests ran. The optimized system malloc implementation is truly optimized for performance and has the fastest execution time. However, first fit and next fit come close to the optimized malloc and in certain test programs record a lower execution time.

Algorithms

The algorithms used to manage and keep track of the memory in the heap all utilize a linked list data structure where each node contains memory block metadata such as the size of the block, whether it is currently in use or not, and a pointer to the next block in memory. The first fit algorithm works by searching the memory block list from the beginning and allocating the first available block it finds. The next fit algorithm works similarly to the first fit algorithm however, it utilizes a global pointer to store the location of the last block that was allocated. A call to malloc will begin searching at the last location allocated searching for the first available block

that the requested memory can fit in. If we run into the end of the list, next fit will return to the beginning of the list and search the remaining blocks that have yet to be checked. The best fit algorithm starts searching at the beginning and searches the entire heap looking for the block that will result in the least amount of leftover space. The worst fit algorithm works in a similar manner in that it will search the entire heap but will end up choosing the block that will result in the most leftover space. Realloc() and calloc() were also implemented in order to complete the group of memory allocation functions that a user will use alongside malloc. Realloc() utilizes the same base cases that the system function uses to handle any obscure parameters the user will pass in. The implementation allocates a new block of the requested size, copies the users existing data over and marks the old block as free for use. Calloc() utilizes our malloc implementation and uses memset() to go through all the bytes that have been granted to the user and initializes them to zero.

Test Implementation

A series of four test programs were specifically fabricated to produce different results across all the algorithms implemented. A randomization approach was used to attempt to simulate user behavior. This allows for analyzation and comparison of metrics across the different algorithm approaches. A test program was also developed to stress test a high number of frees and allocations with the goal of finding edge cases in my personal approach to implementing malloc.

Andrew McLaughlin
1001988355
CSE 3320 - 003

Test Results

Test 1

	Mallocs	Frees	Reuses	Grows	Splits	Coalesces	Blocks	Requested	Max heap	Time
First fit	29	28	10	19	8	25	2	11,712	10,676	.002s
Next fit	29	28	10	19	8	25	2	11,712	10,676	.001s
Best fit	29	28	10	19	8	25	2	11,712	10,676	.002s
Worst fit	29	28	10	19	10	27	2	11,712	10,676	.003s
Sys Malloc										.002s

Test 2

	Mallocs	Frees	Reuses	Grows	Splits	Coalesces	Blocks	Requested	Max heap	Time
First fit	3999	2998	2000	1999	1799	1799	1999	25988024	19992000	.004s
Next fit	3999	2998	2000	1999	1800	1799	2000	25988024	19992000	.008s
Best fit	3999	2998	2000	1999	1799	1799	1999	25988024	19992000	.016s
Worst fit	3999	2998	2000	1999	2000	1999	2000	25988024	19992000	.008s
Sys Malloc										0s

Andrew McLaughlin
1001988355
CSE 3320 - 003

Test 3

	Mallocs	Frees	Reuses	Grows	Splits	Coalesces	Blocks	Requested	Max heap	Time
First fit	1008	250	254	754	244	0	998	1008048	884512	.004s
Next fit	1008	250	255	753	241	0	994	1008048	883520	.000s
Best fit	1008	250	254	754	244	0	998	1008048	884512	.005s
Worst fit	1008	250	252	756	249	0	1005	1008048	886484	.001s
Sys Malloc										.001s

Test 4

	Mallocs	Frees	Reuses	Grows	Splits	Coalesces	Blocks	Requested	Max heap	Time
First fit	4000	1333	2000	2000	1990	666	3324	1337333024	1333340000	.004s
Next fit	4000	1333	2000	2000	2000	666	3334	1337333024	1333340000	.011s
Best fit	4000	1333	2000	2000	1989	666	3323	1337333024	1333340000	.006
Worst fit	4000	1333	2000	2000	2000	666	3334	1337333024	1333340000	.004s
Sys Malloc										.010s

Andrew McLaughlin
1001988355
CSE 3320 - 003

Interpretation

The most straightforward data to interpret would be that of the system malloc execution time compared to the 4 algorithms for each test case. The system malloc's execution time on average produces the lowest time compared to any of my implementations. However, in test programs 2 next fit's time is incredibly close to the system malloc's time. In test program 3 the structure of the test produces a quicker system execution time with next fit than the system's malloc. These results coincide with my hypothesis that out of the 4 algorithms, next fit will have the greatest performance due to its nature of searching at the last allocation made. The results across all tests show that depending on the structure of the test, first fit and next fit on average have the highest performance with respect to their system execution time.

In test program 1, the worst fit algorithm results in having 2 more splits and coalesces compared to the other algorithm. These results do not come as a surprise due to the fact that the approach has the goal of finding a block with the greatest amount of leftover space. However, a comparison of number of splits to heap growth did not show a change. This could be a result of an existing bug still present or the specific structure of the test case. However, test program 2 does show a correlation between the number of splits and max heap size. The worst fit results have 5 more splits than any of the other algorithm resulting in a heap size that is at the minimum 1,972 bytes greater than any of the other algorithms.

Regarding heap fragmentation, test number 3's structure shows that heap fragmentation is best minimized between best fit and next fit. These results match the goal of the algorithms because

best fit works to find the block of memory that will result in the least amount of space left over which will minimize the number of splits. By selecting the smallest block of memory, it reduces the amount of space left over and therefore reduces the fragmentation of the heap.

Conclusion

From the results of the benchmarking test programs, the system's malloc out performed each of my four algorithms across the different test programs. This is strictly based on execution time due to not being able to record the other metrics of the system's performance. However, on average, first fit and next fit came very close to the system's performance. In the case of test programs 1 and 3, next fit actually outperformed the system's implementation. Regarding the management of heap fragmentation, next fit and best fit performed the best at keeping the size of the heap small by reducing the number of splits and wasted space. This observation lines up with the goal that best fit works to achieve. In conclusion, my implementation of malloc, the arena allocator, could be a viable substitute for the standard implementation of malloc. Based on the structure of the test programs and what a user wishes to do, the performance of the arena allocator can outperform the system's implementation depending on how a user utilizes malloc and structures their program.