Nisarg Dabhi
Assignment I: Malloc Library Part I
ECE 650- Spring 2018

Introduction

The files that can be found in my zip file consist of *my_malloc.c*, *my_malloc.h*, *test.c* and *test.h*. Both *test.c* and *test.h* were used to individually test subfunctions to build the two implementations of malloc and free. Each implementation of malloc and free was subsequently tested using the allocation test file provided to us. This report will provide an overview of how allocation policies were executed, display results from performance experiments for each policy and offer an analysis of results.

Implementation Overview: First Fit

To implement malloc using a first fit allocation policy, I first elected to represent each memory block with a struct called "mem_block". This struct served as a header for each memory block, storing information on what block came before it, what block was after it, if it was free or not, and finally the size of the block. I then built a Doubly Linked List to represent my free list: a list of memory blocks that are free. I used a global mem_block * variable to serve as the head of the list.

I built a function "sortedInsert" to insert memory blocks into the Doubly Linked List and maintain the list's chronological order. The list was organized in ascending order from lower addresses to higher ones. This order was maintained to allow for easy merging of blocks later in the free list. I also built a function to remove items from the list, "RemoveNode", when memory blocks could be used to complete allocation requests. I created a function, "FirstFitFind", that would traverse the linked list starting from its head node to find the first node that was large enough to complete a call to malloc. This function returned NULL if no block was large enough to fulfill a request.

Next, I built a function, "addblock" which would push back the program break by calling sbrk() in order to create space for a new memory block. This was called when no free nodes in the free list were large enough to complete a malloc request. I created a function "partition" that would split up memory blocks. If a malloc request only needed half of a memory block, partition would create a new block out of the remaining space in the memory block, mark it as free, and add it to the freelist represented by the Linked List. To merge blocks that are next to one another in the freelist and are next to one another in memory, I created "single_merge". This function added the size of a Header and the size of the memory block to the size of the starting address of a memory block and compared this value to the starting address of the next block. By doing so you can determine if these free blocks were actually next to one another and should be combined.

The First Fit malloc function works by first looking for an available block (through FirstFitFind) in the free list. If one is found, it attempts to split it (through partition) and after doing so removes it from the free list. If no block is found on the free list, a new one is created and used to fulfill a malloc call. The First Fit free function works by using the pointer inputted to the function to get to the exact address of the start of the memory block that will be freed. The function then will add that block to the free list, and finally attempt to merge the block that was just added to the free list to any blocks it may be adjacent to.

Implementation Overview: Best Fit

Best Fit's implementation is almost entirely synonymous to First Fit's implementation. The distinction in the implementation is the way in which a memory block is found on the free list when allocating memory through malloc. The function I built, "BestFitFind" iterates through each node in the free list starting from the head. It calculates the difference between the size of the current node it is on and the size of the memory request and compares it to a minimum difference value. If it is smaller than the minimum difference value and positive, the minimum value is replaced by that calculated difference. This process is repeated for every node in the free list, until the smallest positive difference in the size of node and the size of a memory request is found along with the index of that node. The index of the node is then used to get that specific memory block from the free list.

The free implementation for a Best Fit allocation policy is a carbon copy of the First Fit allocation policy function. This makes sense as the only difference is the way in which nodes are searched for in the free list. Besides the "BestFitFind" function, the code for the Best Fit implementation is organized in the exact way the First Fit implementation code is.

Performance Experiments: First Fit Expectations and Results

Both malloc and free implementations were analyzed by using them to run the allocation policy executables provided in the starter kit. For the small range rand allocations simulation, we would expect this simulation to finish quickly. This is due to the fact that this implementation utilizes a LinkedList for the free list that just holds nodes free nodes. The program as we know mallocs a multitude of 128-512 byte regions, then alternates between freeing these regions and mallocing 50 more regions of 128-512 KB.

Mallocing regions in the first part of the program that consists of a series of malloc calls should not take m much time as each call will just push back the program break. There is no free list to search through so by default the program break will just be pushed back. The program then alternates between freeing regions and mallocing regions. This alternating sequence will keep the free list small meaning any free list traversals will not take too much time when looking for memory blocks to satisfy allocation requests. The last part of the program will take the most time as it is a series of mallocs. This requires searching the free list on every call. The free list will remain constant in its number of nodes through this series of mallocs, as no nodes are being freed. This part of the program may take up most of the execution time, but the free list shouldn't be very congested as there never was a series of free calls without a malloc call soon after.
The results of the simulations for the First Fit implementation are shown in the tables below:

Table I: First Fit Performance Results Running "small_range_rand_allocations"

Variable	Value
Data segment size	3689780
Data segment free space	2775
Execution Time	309.610143 seconds
Fragmentation	.000751

The above simulation was run for 1000 iterations. We see from Table I that our expectations were accurate in the execution time would not be very high and fragmentation would be low. We also can compare this with Table III and see that our execution time is a bit longer. This would make sense with greater fragmentation as there would be more remaining nodes at any given time in the free list for this simulation. As a result, we would have to traverse more nodes when finding blocks to satisfy malloc requests.

Table II: First Fit Performance Results Running "equal_size_allocs"

Variable	Value
Execution Time	1.924211 seconds
Fragmentation	.003409

The above simulation results were run with 1000 iterations. We should expect this simulation to actual run with the least amount of fragmentation as well as the smallest execution time. This is due to the fact that this program mallocs a series of memory blocks and frees those blocks in the order that they were malloced. As a result, we should see low execution times as the free list will never be very large as we should see blocks coalesce every free. In addition, fragmentation should be extremely low as every block is uniformly allocated at 128B. The results above are consistent with our expectations.

Table III: First Fit Performance Results Running "large_range_rand_allocs"

Variable	Value
Execution Time	249.716518 seconds
Fragmentation	.000019

The above simulation for Table III was run with 50 iterations. The execution time may be longer than the short_range simulation as more memory is allocated since chunks of up to 64 KB's could be in our LinkedList. This may take elongate the time it takes sbrk() to push back the break point. This program is set up exactly the same way as the small range except nodes may range from 32B to 64KB. Since blocks are much bigger, we should expect lower fragmentation as blocks will be big enough to satisfy most requests. Our result is consistent with our expectation as our fragmentation value is .000006. We expected to definitely have a fragmentation value much lower than the value for the simulation ran for the small_range program. Our results are consistent with that notion.

Performance Experiments: Best Fit

Table IV: Best Fit Performance Results Running "small_range_rand_allocations"

Variable	Value
Data segment size	43340276
Data segment free space	913
Execution Time	156.7634777 seconds
Fragmentation	.000021

The above simulation results were run with 1000 iterations. Comparing this to the First Fit simulation, we should expect this simulation to have slightly lower fragmentation but higher execution time. This is due to the fact that on every malloc, this implementation will search the entire free list to find the best memory block as opposed to the first block that can satisfy the request. Our results are somewhat consistent with our expectation as fragmentation is lower but the execution time has actually dropped. This may be explained by a much lower fragmentation value, corresponding to less blocks in the freelist. As a result, traversals through the free list may not take very long leading to a drop in the execution time.

Table V: Best Fit Performance Results Running "equal_size_allocs"

Variable	Value
Execution Time	476.330786 seconds
Fragmentation	.003409

The above simulation results were run with 1000 iterations. Once again, we should expect this simulation to run with the smallest execution time when compared to the other simulations. Again, fragmentation should be extremely low as every block is uniformly allocated at 128B. Our results seem to contradict our expectations. Although it would make sense for the execution time to rise due to the best fit policy, the free list is always small in this program so the execution time should not have magnified by these extreme margins.

Table VI: Best Fit Performance Results Running "large_range_rand_allocs"

Variable	Value
Execution Time	595.309332 seconds
Fragmentation	.000006

The above simulation for Table III was run with 50 iterations. Comparing this to the First Fit simulation, we should expect this simulation to have slightly lower fragmentation but higher execution time. This is due to the fact that on every malloc, this implementation will search the entire free list to find the best memory block as opposed to the first block that can satisfy the request. Our results support our prediction. Additionally, the larger blocks and wide disparity in range seems to have tripled the execution time.

As a whole, it seems that different policies are effective in different situations. It seems that if blocks are consistently the same size, a first fit scheme makes the most sense as every block will be the best fit memory block to allocate. It seems the Best Fit policy performs better than a First Fit policy for a small range of allocations however the First Fit policy performs better in a larger range memory program. The Best Fit policy seems to guarantee a much lower fragmentation amount, which makes sense as only the ideal memory blocks are used for each allocation.