Assignment 4 - Malloc Replacement : Conclusion Document

Corinne Jones

To analyze the performance of 'MyMalloc' vs. the system's built-in 'malloc' and 'free,' I performed timed-tests for small/large object allocation, deallocation, and total time. I wrote functions that overrode the system's 'malloc' and 'free'. When I timed the built-in methods, I commented the override functions. I ran each test 10 times and found the average. The comparative analysis revealed that for small object operations, my allocator took significantly more time (MyMalloc average allocation time: 52.023 ms, MyMalloc deallocation time: 93.185 ms, MyMalloc total time: 145.211 ms) (SystemMalloc average allocation time: 1.682 ms, SystemMalloc deallocation time: 0.197 ms, SystemMalloc total time: 1.881 ms) compared to the system allocator (average allocation time: 1.6816 ms, deallocation time: 0.1966 ms, total time: 1.8812 ms). For large object operations, while the allocation times were comparable, the deallocation times and total operation times exhibited substantial differences (MyMalloc average allocation time: 10,619.12 ms, MyMalloc deallocation time: 37.728 ms, MyMalloc total time: 10,656.84 ms) (SystemMalloc average allocation time: 10,537.44 ms, SystemMalloc deallocation time: 135.283 ms, SystemMalloc total time: 10,672.74 ms).

There are some potential reasons why MyMalloc is slower than the system's build in 'malloc' and 'free.' This could include differences in functions such as hashing, growing, and probing. My custom hash table, while effective for tracking allocations, might have some increased computational overhead. Optimizations could more effectively hash data, as well as have alternative collision resolution strategies. Previous assignments have asked me to time different hash functions, and I found variability in hash function performance to be significant. Second, growing a hash table would take time. I had my hash table start at a capacity of 10, which would require many calls to "grow" for a large number of memory allocations. The systems 'malloc' might have a more optimized algorithm for growing once a certain capacity is reached. Last, my 'free' function seemed to perform comparatively to the system's 'free.' This might indicate that mine is as efficient, but most likely, mine might not be as thorough as the system's free. This could be due to mine implementing lazy deletion, which wouldn't go through and zero out large chunks of memory, or could be that mine doesn't consider fragmentation. Since I made many calls to make space for the same sized large chunk of memory, mine might be easier to deallocate since there might be limited fragmentation in that scenario, although not indicative of real-life. I am not sure how linear probing compares to different types of probing, but there might be a more efficient probing strategy as well.

This brings me to another reason that 'MyMalloc' may suffer since I did not consider fragmentation as the system allocator might. In my testing, I asked for allocations of the same page sizes, which would rarely happen in a real-life scenario. Due to this, mine is not optimized for speed based on size, or memory utilization. This could make mine more efficient if I were to consider small, medium, and large pages sizes, similar to the system's 'malloc'. The system's malloc likely incorporates advanced techniques to minimize fragmentation, ensuring efficient memory usage over time. Implementations that consider different page sizes techniques could offer significant performance improvements, especially for allocators dealing with a mix of small and large allocations.

In conclusion, the performance evaluation of ‘MyMalloc’ to the system’s ‘malloc’ shows that there is room for improvement in my implementation optimization. Mine especially lags behind in small object allocation/deallocation. Mine could improve in managing memory fragmentation and optimizing the custom hash table implementations.