CSCE 4600.003

Spring 2016

Project #2

By Andrew Thomas and Michael Thomas

**Introduction**

The purpose of this experiment was to compare the performance difference in using the standard C/C++ library functions malloc() and free() with our own custom implementations of the library functions, my\_malloc() and my\_free(). While working on this project, we had to make some assumptions, namely:

1. We can safely measure everything in kilobytes instead of bytes - Our memory allocator assigns memory in terms of whole kilobytes, so using kilobytes throughout the process enabled us to simplify certain aspects of the code, such as my\_malloc() which would have had to comb through 1000x more array elements if we had used bytes instead of kilobytes.
2. No malloc overhead - This was important, as malloc assigns pointers within 16-byte boundaries. Since we're treating each byte as a kilobyte throughout the experiment, if we took malloc()'s overhead into account, we could have ended up wasting a lot of memory if we took the overhead into account (every 1KB process would have taken up 16KB of memory, drastically reducing the performance of the standard functions).
3. Processes can be described as an array of bools (1's and 0's) - bools can be stored in 1 bit of memory (either a 1 or a 0). In c++, bools end up taking 1 byte anyways, but that's still much better than the 4 bytes that an integer would take. We could have alternatively multiplied the maximum memory amounts by 4 and used ints instead of bools, but we decided that it was ultimately irrelevant since our processes keep track of where they start in our custom memory array regardless (If we used ints, we could have written the process number in each memory array block instead of just a 1 or a 0, but then we have to constantly write, read, and compare ints instead of bools. We decided it wasn't worth it).
4. All processes can run simultaneously, assuming we have the memory available and that the processes have all entered - We decided that limiting our code to running one process at a time was a bad idea.Our goal was to compare the time spent calling malloc(), my\_malloc(), free(), and my\_free(), and running each process individually would have slowed the execution time down so much that those calls become a much smaller portion of the overall runtime, making them harder to compare. Also, if we were to only run one process at a time, the enter-time of each process would become irrelevant, as each processes is guaranteed to have a cpu time greater than 50, so the next process will always be ready before the current process finishes.
5. The memory locations where malloc() places the memory we request is irrelevant – We assumed that malloc() will use a first-fit algorithm to allocate memory for our processes, so that a set of x processes with an average of y memory used will perfectly fit inside a space of x\*y memory. In reality, the memory could become fragmented during the execution of our code to where our x\*y memory wouldn't fit inside our imaginary x\*y size block, but we don't really have a way to check or enforce that using the standard functions, so we decided not to worry about it.

**Description of Methods**

We used a first-fit memory allocation algorithm in our custom malloc algorithm. When we call my\_malloc(), we pass it the array of bool pointers that represents our total memory block (memArray), the number (identifier) of the relevant process, and the array of processes. We then start looping through memArray, starting at index 0. Any time there is free index location, we increment our counter contiguousSpace. Any time we find an index location that is not free, we reset contiguousSpace. If contiguousSpace ever matches the memory required by the process in question, we assign the startMemBlock element of the process, and then loop through memArray from startMemBlock to startMemBlock + the required memory, setting all the memArray values to 'true', indicating that the space is now in use. This was a very easy method of implementing the first-fit memory allocation algorithm.

In my\_free(), we receive the same parameters as my\_malloc(). We loop through memArray starting at startMemBlock through startMemBlock + the memory of the process, setting each index location back to false, indicating that it can be used again for another process.

**Procedure**

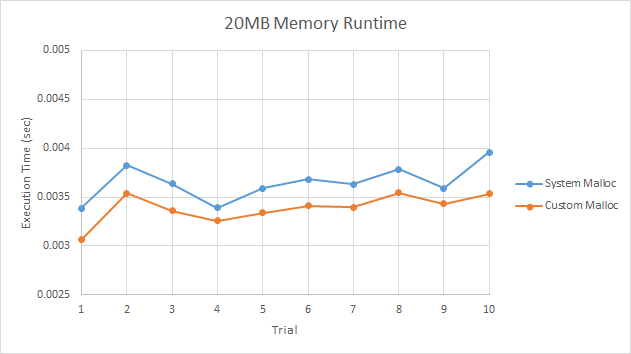
Our procedure for both methods are very similar, basically just swapping out my\_malloc() and my\_free() for their standard counterparts.

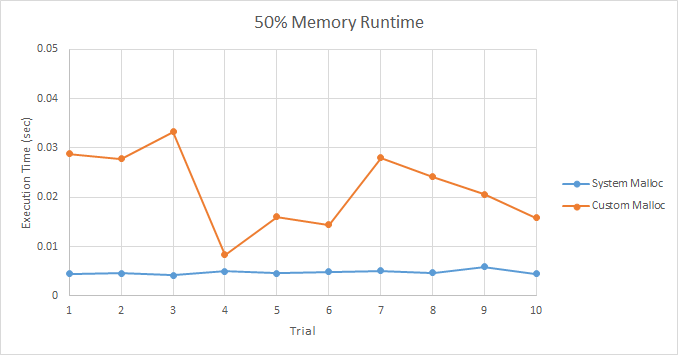
First, we loop through the set of all processes. If any process has entered and has a startMemBlock value of '-1' (what it's initialized to), we check that there is enough memory available to try to allocate some to the process. If there is not enough memory, we don't try to allocate memory. If there is enough memory, we attempt to allocate the required amount to the process. At this stage, we assume that malloc() will always succeed in allocating the memory (since enough was available), but my\_malloc() has a chance to fail if the available memory didn't have a large enough contiguous block to fit the process. If my\_malloc() fails, it leaves startMemBlock as '-1', meaning that the process is not running. If a process did have memory allocated, we add that amount to memUsed so that we can keep track of how much available space is remaining. Then, we check if there are any currently executing processes (there is remaining CPU time, and startMemBlock is not '-1'), setting oneRan to true and decrementing the CPU value of the running process. After decrementing the CPU time, we free()/my\_free() the memory of any processes that finished.

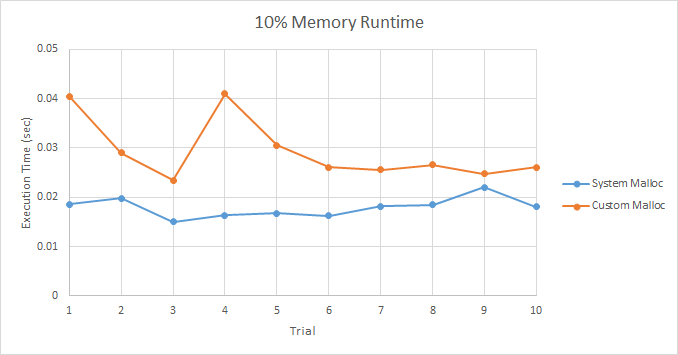
If we set oneRan to true, we can skip the check at the end that decides if we're finished. If we did not set oneRan to true, we check to see if there are any processes that have not finished. If there are any unfinished processes, we start the main loop over. If there are not any unfinished processes, we print out some analytical information and then return.

During each of the methods, we time the total run time by checking the start time and the end time and subtracting the two. We did not time the malloc() request to request the initial memory block for our custom allocator, as we felt that that wasn't the point of the project.

**Analysis**

Below are graphs representing the runtime of the system malloc() and our custom my\_malloc() for the three different memory sizes. I got this data by running my program 10 times. For each of the 10 runs, I ran the process for each of the three memory values, using the same randomly generated list of processes. 





With the excessive 20MB memory limit, our custom my\_malloc() actually ended up being slightly faster, with an average runtime of 0.0033876 seconds (can be found on our excel data file which we uploaded along with our other files), while system malloc() had an average runtime of 0.0036479 seconds, which is an increase of only 0.0002603 seconds. Given the sample size of 64 processes, this difference in runtimes is almost negligible.

With only 50% of the total required memory, system malloc() was faster with an average runtime of 0.0047882 seconds, while our custom my\_malloc() function had an average runtime of .0217356 seconds. The difference here is almost a factor of 5, showing that system malloc() is very clearly the winner.

Finally, with only 10% of the total required memory, system malloc() beat our custom my\_malloc() with an average of 0.0179577 seconds to 0.0293662 seconds. This difference here is still almost a factor of 2, showing that system malloc() is still clearly superior.

**Conclusion**

While our custom memory allocator won across the board (by a small margin) when there was enough memory available for all of the processes, the performance hit of looping through all memory indexes repeatedly caught up to it when we had only 50% and 10% of the required memory available. This ultimately led to our custom allocator being significantly slower when there was less than the total required memory available. This is likely caused by how malloc() and free() actually keep track of where memory is available, instead of having to loop through every memory index to find a large enough block.