Jonathan Westerfield

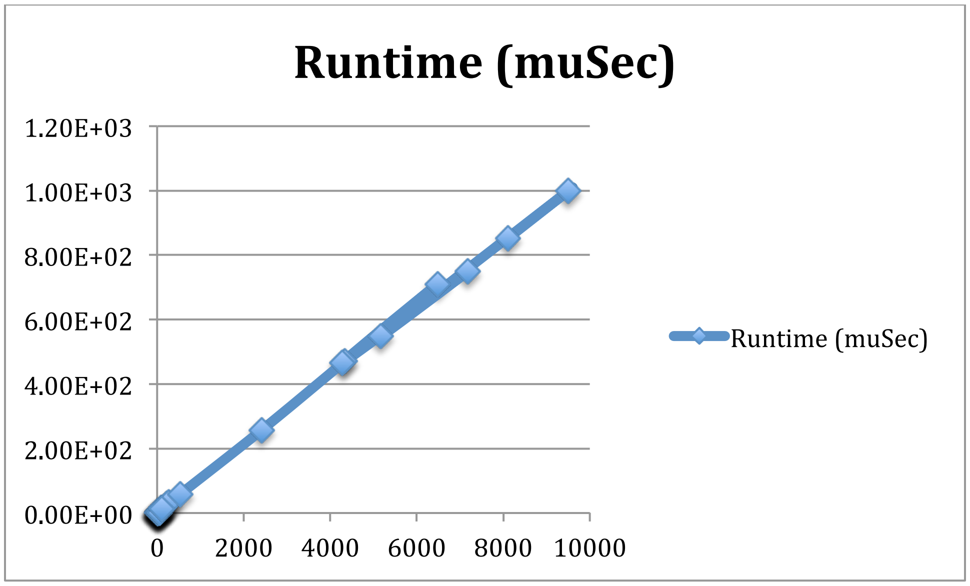
October 1, 2017

Machine Problem 1 Milestone 3

CSCE 313

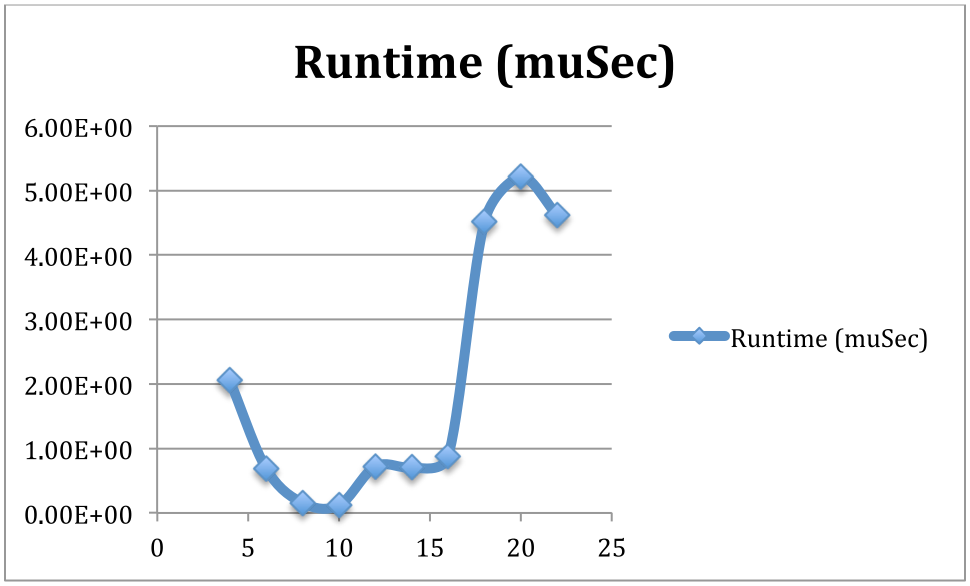
Bettati

When testing using memtest.c and my_malloc, the relationship of run time with respect to comparisons is mostly linear aside from the outliers (small comparisons). These outliers (which are much harder to see due to the scale of the graph) are due to less calculations allowing a greater chance of a different average value.

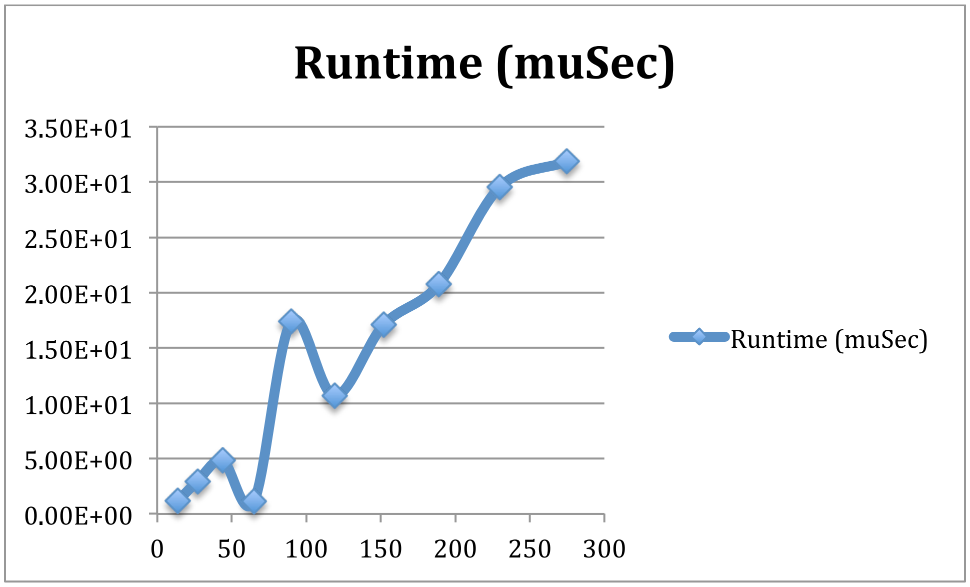


The memtest.c file uses the Ackerman function to test our implementation of malloc. This is because the Ackerman function is very, very, very recursive so it will really make heavy use of the my_malloc function. The results of the test are shown below.

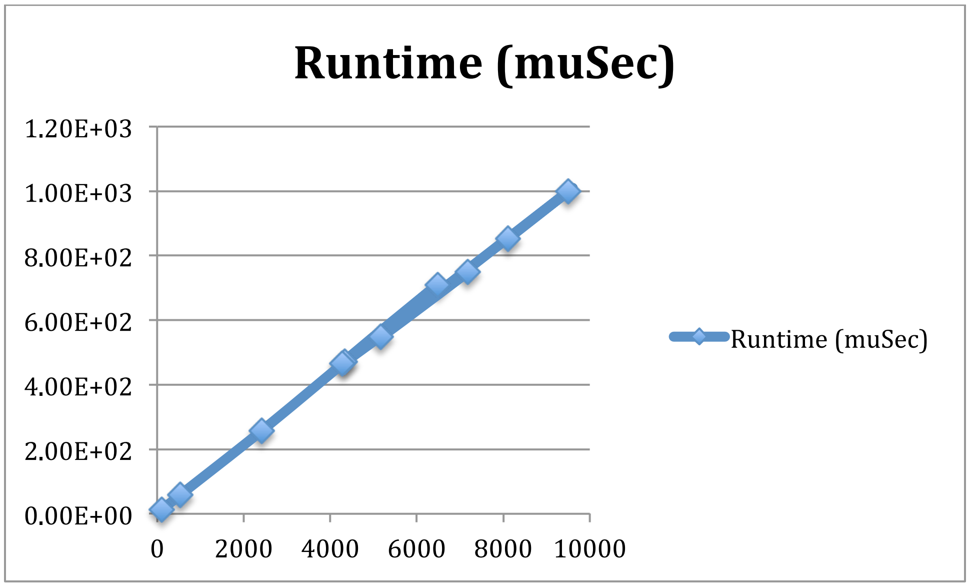
A = 1



A = 2



A = 3



Analysis
As you can see, as the number of comparisons goes up, the trend of the time versus comparison becomes much more linear and obvious.

Bottlenecks

- The bottlenecks in the system are plainly obvious when large amounts of memory are initialized with only a few small block sizes and relatively small amounts of memory are requested from the system. This is because the free blocks are large most of the time and must be broken down into smaller block sizes. Because of this, there is increased block fragmentation. The efficiency of memory management is better for bigger block sizes due to the fact that every block has a small constant size added for headers since header size is constant no matter the size of the block. However, the biggest bottleneck with my implementation is in the joining of 2 blocks. This is because my implementation checks for the possibility of a block having the capability of being joined even when it cannot. This happens for every block that is checked. This check happens every time a block is freed.

Improvements

- The freeing and joining of blocks is by far the slowest part of my implementation of malloc. One possible solution to speed up the program is to refactor the code so that it only joins blocks when there is no more memory of that block size available to be allocated. Essentially, it follows the philosophy of most operating systems in that it will only carry out the action once it absolutely has to. With this fix, the joining and breaking of blocks are both done in the allocate function. The benefit of this is that it would be able to skip joining blocks for every free. However, the disadvantage is that without specific blocks being free, the operation of finding a buddy block to join with would be much more difficult to implement due to the fragmentation and length of the linked list. This in turn would create a more expensive operation to do. To make the algorithm more efficient, it would need to make cases where blocks need to be joined very rare. For example a program that consistently allocates and free blocks from your smallest block size would theoretically only need to break blocks not join them. This would create a much faster implementation of so that the Ackerman function would theoretically take much less time to run.