Muhammad Muawiz Farooqi
1001825601

Arena Allocator Assignment

Executive Summary

This project analyzes different allocation algorithms and compares their execution speeds using a clock-based program timing system. It will compare the execution speeds of a set of benchmarks that use allocation algorithms such as best fit, worst fit, next fit, and first fit. An additional benchmark will compare the user-declared allocation methods with the malloc() allocation algorithm used in C to dynamically allocate data.

The benchmarks use a variety of methods to test the allocation algorithms' full capabilities. Such methods include creating an array of pointers that can hold up to the maximum number of possible allocations at a time for the program, allocating memory to individual pointers and releasing them at different points to test the different allocating techniques, and iterating over the allocations to perform different combinations of allocations and frees.

The benchmarks conclude that the first fit algorithm used in benchmark 4 is the best allocation algorithm. This is because it had the lowest average execution time over a period of five executions. It was comparable to benchmark 3, which used the next fit algorithm, but was significantly better than both best fit and worst fit algorithms. Additionally, benchmarks 1-4 were not comparable to benchmark 5, as the malloc() library call function was exponentially quicker than the user-written allocation algorithms.

Purpose of benchmarks

The benchmarks were built to test the efficiency and execution speed of the linked list-based allocation algorithm. The benchmarks test a number of possible test cases for the program. An array of pointers of size 10,000 [max] is first declared and the same number is initialized in memory using the allocation algorithm of choice. The array is then allocated iteratively to completely fill it up and test if the allocator works for its maximum number of allocations.

The array is then iteratively freed and then reallocated for every 15^{th} term. This tests if the allocator can keep track of the appropriate index and not only fill up consecutive values. The array is then freed again, and the allocator finally allocates the array again for each term to ensure consistency before and after the skipped allocations. The array is finally iteratively freed.

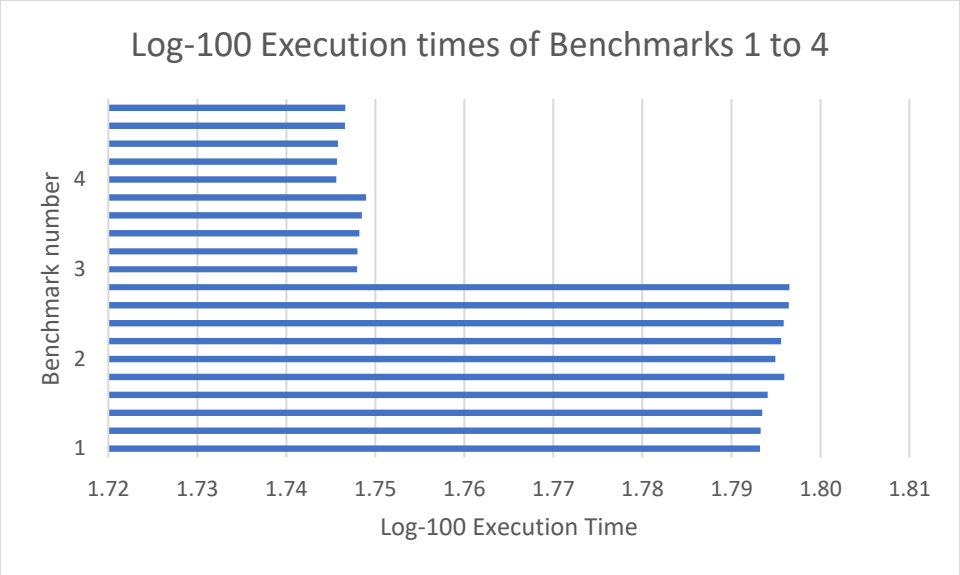
Next, three additional, non-array pointers are initialized. The first pointer is allocated 9000 units of memory and the second pointer is allocated 100 units of memory. The first pointer is then freed, and the third pointer is then allocated with 90 units of memory. This is to check for proper usage of the allocation algorithm, with the final pointer being allocated to a different block depending on the chosen allocation algorithm. The two remaining pointers are finally freed. The allocation for these three pointers is set in a loop that allocates and frees the memory 10,000 times to ensure that the allocation algorithm does not run into errors from multiple allocations.

This benchmark system is repeated for all allocation algorithm types and also an additional one for the malloc() function, to compare the program's efficiency to the built-in library call function.

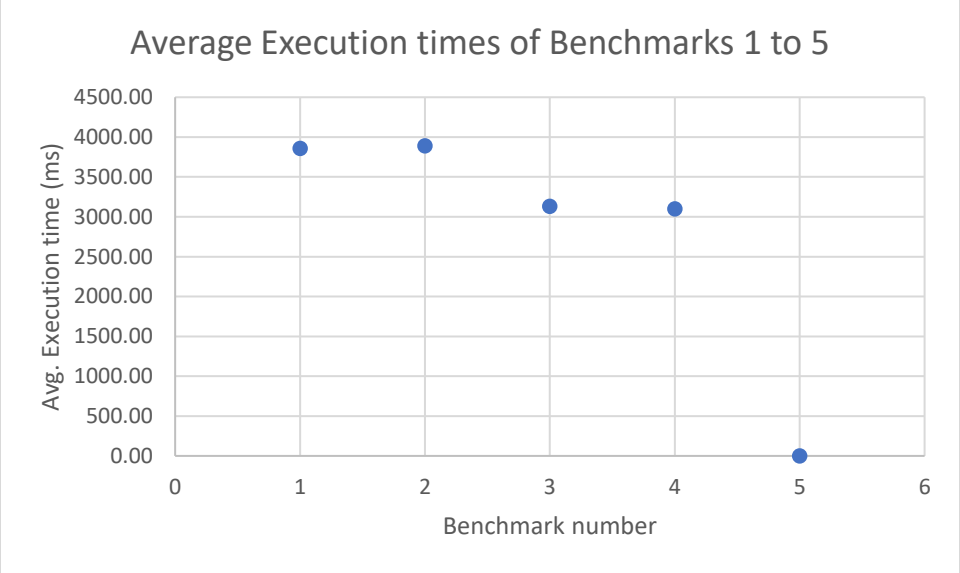
Results of benchmarks

Benchmark no.	Allocation algorithm	Execution Time (ms)	Average Execution Time (ms)	Log-100 Execution Time	Log-100 Average Execution Time
1	Best Fit	3858.93	3872.62	1.79	1.79
		3859.68		1.79	
		3862.84		1.79	
		3874.21		1.79	
		3907.45		1.80	
2	Worst Fit	3889.56	3906.24	1.79	1.80
		3900.84		1.80	
		3906.00		1.80	
		3916.67		1.80	
		3918.12		1.80	
3	Next Fit	3132.48	3137.84	1.75	1.75
		3133.09		1.75	
		3136.05		1.75	
		3140.37		1.75	
		3147.23		1.75	
4	First Fit	3099.15	3105.43	1.75	1.75
		3099.98		1.75	
		3101.69		1.75	
		3112.93		1.75	
		3113.42		1.75	
5	Malloc (library call)	1.59	1.67	0.10	0.11
		1.62		0.10	
		1.62		0.11	
		1.71		0.12	
		1.83		0.13	

Table 1: Data collected for execution times of benchmarks 1-5



Graph 1: Log-100 execution times of benchmarks 1 to 4



Graph 2: Average execution times of benchmarks 1 to 5

The data in the graphs and table above shows the differences in execution time between the five benchmarks. Table 1 contains the values for five execution times of each benchmark in milliseconds, an average execution time for each of the five benchmarks, a log-100 representation for each of the execution times as well as a log-100 representation for the average execution times. The log-100 data is included to show the differences between the larger measurements of time in a clear and concise graphical format.

Graph 1 represents the log-100 execution times for benchmarks 1-4 and graph 2 represents the average execution times for benchmarks 1-5. The first graph does not include benchmark five

due to the skewed nature of the results, which would make it harder to differentiate between the larger execution time values for benchmarks 1-4. For a full picture of the execution times for all benchmarks, the second includes the average execution time for each of the five benchmarks. This graph helps visualize the significant difference in execution time between benchmark 5 and benchmarks 1-4.

A significant difference can be observed between benchmark 5 and benchmarks 1-4. This is a result of the significantly quicker allocation time of the malloc() library function that was used to allocate memory in benchmark 5, as opposed to the defined mavalloc_alloc() function that was used in benchmarks 1-4. Benchmark 5 was the fastest, with an average execution time of 1.67 milliseconds. Next, benchmark 4, which used the first fit allocation algorithm, had an average execution time of 3105.43 milliseconds, followed by benchmark 3, which used the next fit allocation algorithm, with an average execution time of 3137.84 milliseconds. Benchmark 1 used the best fit allocation algorithm and had an average execution time of 3872.62 milliseconds, and finally, benchmark 2 used the worst fit allocation algorithm and had an average execution time of 3906.24 milliseconds.

The allocation algorithm for benchmark 4 was found to have the lowest average execution time (3105.43 milliseconds), which means that the best performing allocation algorithm was first fit. This is a result of the process in which first fit executes, simply looking for the first free block large enough to allocate memory and not comparing sizes of every free block, as in the cases of best and worst fit. Next fit was comparable to first fit in terms of execution time as this algorithm also uses a similar approach but instead maintains a record of the last used block to continue from. This takes a small amount of computing power to execute, and this difference is exemplified in the long run with larger allocations and longer program files.

There is one anomalous value that can be spotted on the first graph for an instance of the execution time for benchmark 1. This execution time value is further from the rest of the execution times for the same benchmark and is comparable with the worst performing benchmark, benchmark 2. This anomaly could have been a result of concurrent programs/executions being carried out at the same time which meant that the benchmark for this particular instance ended up with an unnaturally larger execution time. Therefore, it was essential to carry out five different executions to get a large enough sample size to estimate an average for the execution times for each benchmark and accurately predict the best allocator.

Conclusion

After analyzing the data for the execution times for each benchmark, benchmark 4 was found to have the lowest average execution time, with 3105.43 milliseconds, meaning that the best performing allocation algorithm was first fit. This is a result of the process in which first fit finds a block to allocate data in, where it simply looks for the first free block (hole) large enough to allocate memory and does not compare sizes, as in the cases of best fit and worst fit. In conclusion, first fit was the best algorithm in terms of execution speed due to its lightweight allocation system.