Operating Systems – Assignment 3

Aryan Mediratta
1001910692

November 2022

1 Introduction

The overall goal of this project is to implement an arena allocator that manages a large chunk of memory and allocates it to variables as needed. We use four different algorithms – First Fit, Next Fit, Worst Fit, and Best Fit – to allocate memory and benchmark them to compare their performance from an execution time perspective. For comparison, we also run the same tests on the standard `glibc malloc()` and `free()` functions. Given a ledger of free and allocated blocks of memory, the algorithms handle requests to allocate memory and decide which section of the memory to assign to fulfill that request.
The benchmarks suggest that strictly from an execution-time perspective, the optimal algorithm for memory allocation is Next Fit.

2 Overview of the Algorithms

2.1 First Fit

First Fit starts at the beginning and allocates memory in the first available hole.

2.2 Next Fit

Next Fit keeps track of where it last allocated memory. It starts looking from that point and allocates memory in the first available hole after that point.

2.3 Worst Fit

Worst Fit allocates memory in the smallest available hole.

2.4 Best Fit

Best Fit allocates memory in the largest available hole.

3 Data Collection and Benchmarking Process

The four algorithms were first written out, tested and implemented in C. Then, five nearly identical benchmark programs were written. The only changes made were the corresponding algorithms.

3.1 Benchmark Design

Each benchmark program was divided into T test cases, and each test case contained N iterations. In each iteration, an array of M character pointers (`char *`) was initialized and a request to allocate 10 bytes was made for each element in the array. Then for each of these, the string `"Hello"` was written at every pointed to by the array. Starting in the middle, every odd-indexed block allocated was freed. Then, the allocation function was called on these indexes again with one byte of space. Finally, all the memory was freed. All of these operations together were timed for each test case. The benchmarks were run using $T = 5000$, $N = 40$, and $M = 5000$. The time taken for each test case was printed into a file.

```
 ./benchmark1 > runs/malloc.txt
./benchmark2 > runs/first_fit.txt
 ./benchmark3 > runs/next_fit.txt
./benchmark4 > runs/worst_fit.txt
```

The data was then exported into Excel and analyzed.

3.1.1 Considerations Made in Design

1. It was ensured that the time spent in initializing and destroying the ledger was not counted towards the runtime of the algorithms.

2. For every call made to `malloc` or `mavalloc_alloc`, some data was filled in the spaces allocated, in an attempt to prevent an optimizer from discarding the allocation for being unused.

3. The `clock()` function was used from the `time.h` header, which helps count the number of CPU clock cycles that the process spent actually on the CPU, and ignores any time spent context-switched. This is important so that the result of our test does not depend on other processes running on the CPU.

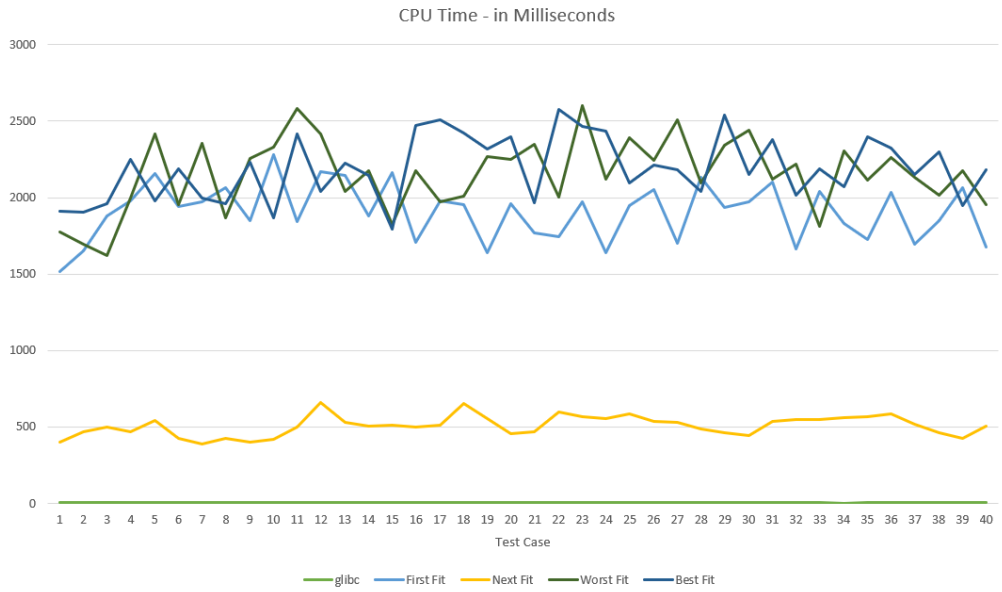
4. The values of the constants T, N and M were stored in a separate header file, to allow modularity.

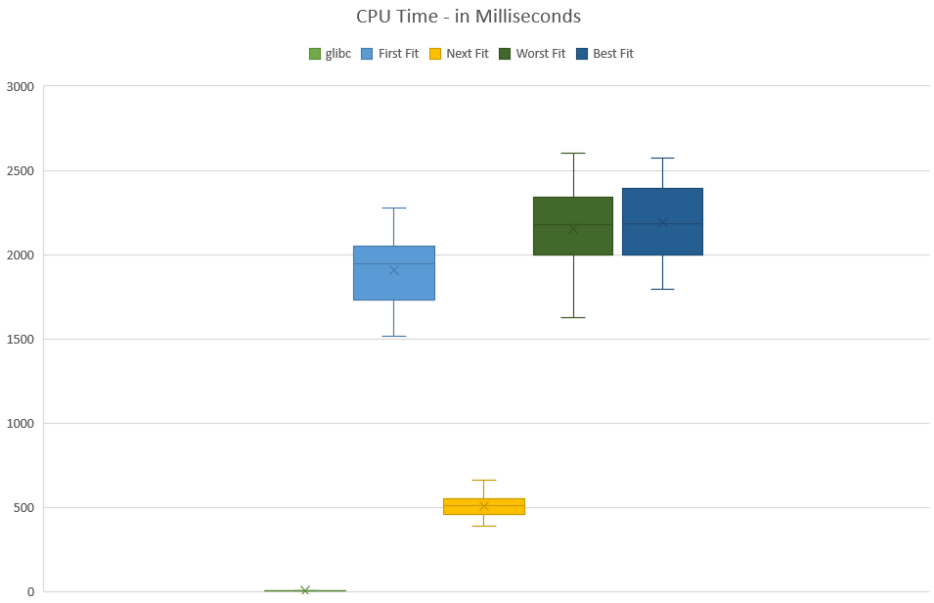
3.1.2 Possible Flaws in the Approach

This approach in benchmarking only compares the runtime of each of these algorithms. We must realize that these algorithms do not necessarily produce the same behavior for the same input. For example, First Fit may fail with a certain allocation sequence that Worst Fit may handle normally. Therefore, a desirable algorithm for memory allocation does not just need to be fast; it also needs to be able to manage space efficiently. We keep that in mind as we go through the results of our testing.

4 Observations and Results

The `glibc malloc` implementation was found to be several orders of magnitude faster than any of our algorithms. Among the algorithms tested, Next Fit ran the fastest and took about 508.44 ($\pm$65.56) milliseconds. The slowest one was best fit with an average running time of about 2190.53 ($\pm$208.48) milliseconds. The speed of Next Fit can be explained by the fact that it stores where memory was last allocated and it is likely that there will be space available next to where some memory was allocated. This way, Next Fit results in good performance if we care about runtime.





We see, as expected, that Best Fit and Worst Fit have similar running times. This is because both of these algorithms need to traverse through the entire ledger to find the smallest/largest available spot. This is not necessarily true for First Fit or Next Fit. The most surprising result here is that the `libc` version of the benchmark had an CPU time of 6.55 ($\pm$0.88) milliseconds. This is significantly lower than any of our implementations, and shows how the standard malloc and free are optimized.

5 Conclusion:

Next Fit worked the fastest based on our tests (after the standard malloc and free).

6 Appendix: Raw Data – Time in Milliseconds

Test Case	glibc	First Fit	Next Fit	Worst Fit	Best Fit
0	7.538	1514.547	404.358	1773.174	1911.863
1	6.956	1649.832	470.391	1695.09	1907.667
2	8.672	1877.995	500.383	1624.449	1959.727
3	7.338	1976.128	470.646	1996.058	2250.107
4	8.064	2155.69	541.688	2418.936	1977.849
5	6.872	1942.014	424.14	1951.66	2186.945
6	7.108	1970.543	390.605	2352.851	1994.703
7	6.972	2066.027	424.882	1866.419	1958.19
8	6.305	1847.195	400	2255.745	2234.623
9	7.281	2279.033	420.698	2329.825	1866.951
10	6.017	1846.33	497.08	2584.048	2418.689
11	5.648	2169.14	662.713	2413.651	2040.907
12	5.913	2145.664	533.963	2038.406	2225.282
13	6.127	1883.402	506.367	2176.73	2143.833
14	7.055	2163.989	514.153	1822.409	1796.378
15	5.934	1710.375	498.755	2176.784	2474.475
16	6.688	1980.593	513.71	1970.952	2510.127
17	6.914	1953.606	653.883	2012.495	2425.734
18	6.956	1640.964	558.515	2266.108	2316.933
19	6.413	1958.348	455.36	2253.299	2399.339
20	6.104	1768.016	471.531	2346.379	1967.415
21	6.166	1743.232	597.306	2001.452	2574.927
22	7.078	1975.096	564.991	2603.211	2463.313
23	7.086	1640.903	553.78	2117.818	2438.17
24	7.465	1947.894	583.356	2389.575	2097.167
25	7.104	2052.26	536.066	2242.024	2212.329
26	6.989	1701.246	530.518	2508.988	2181.536
27	8.001	2130.153	486.623	2098.227	2037.734
28	7.011	1935.152	461.163	2340.553	2541.554
29	7.308	1972.344	441.612	2441.606	2150.613
30	5.567	2105.38	536.594	2118.232	2381.849
31	5.125	1667.417	550.923	2221.322	2013.105
32	6.404	2038.42	550.741	1815.533	2188.669
33	5.065	1834.181	561.289	2305.614	2069.712
34	5.378	1728.427	564.896	2116.019	2397.752
35	5.466	2035.277	588.587	2263.066	2326.113
36	5.146	1696.099	517.945	2133.776	2149.323
37	5.423	1847.693	460.956	2013.059	2297.92
38	5.988	2065.161	428.27	2175.786	1950.923
39	5.361	1676.052	508.325	1953.424	2180.906