Implementation of Custom Malloc
And
Its Performance Against System Malloc
Report

CSE 3320: Operating Systems

Professor Trevor Bakker

By Eduardo Ramos

UTA ID:1001834020

Executive Summary:

As long as computing has existed, the dilemma of how we use its limited resources has been proven as a constant challenge for many brilliant minds around the world. From the usage of punching cards to the implementation of virtual memory, the process of how a computer reads and stores data is a fundamental question that requires careful consideration when developing the next breakthrough in computing. Luckily, we have developed many ingenious solutions that solve this problem. However, few come close to the brilliant idea that is the system function malloc, the double-edge sword of programming. It is important to note that although malloc provides a solution to storing large amounts of data, the possibility of incorrectly handling that storage is still a threat to be aware of. Yet, this begs the question, could we improve the system malloc even further? To answer this question, we devised our own version of malloc that implements four memory allocation algorithms to observe how they perform against native malloc. In so, we can conclude if further improvements are needed to the system malloc to meet the demands of future technological endeavors.

Description of Memory Allocation Algorithms

As previously mentioned, our version of malloc relies on four memory allocation algorithms: First Fit, Next Fit, Best Fit, Worst Fit. These algorithms decide the most optimal block that fits the needs of the user at the lowest cost to the operating system. To expand this further, these four algorithms work in the following manner:

First Fit:

The First Fit algorithm looks through the available data blocks to find the first free block with sufficient memory to satisfy the user's needs. The benefit of this algorithm is that it takes less time to allocate a block. However, this algorithm is prone to memory fragmentation since allocating the first block does not always mean that it is the most optimal block to allocate. When it comes to implementing the algorithm, it is one of the easier ones to implement.

Next Fit:

The Next Fit Algorithm behaves similar to the First Fit algorithm with a small exception. Unlike the First Fit algorithm, Next Fit does not start from the beginning of the data blocks, rather it starts at the last allocated block. The benefits of this algorithm are similar to First Fit, which is faster allocation speeds. However, just like First Fit, it is prone to memory fragmentation as well. In short, this algorithm reflects the First Fit Algorithm with some modifications. Concerning implementation, Next Fit does take a bit of ingenuity to implement but figuring out how to continue from the last allocation will be the biggest challenge you will face.

Best Fit:

As the name suggests, the Best Fit algorithm looks for the memory block that is the best fit for the user's needs. To determine if a block is the best fit for the user, it takes the difference in size between the memory needed and the available memory within that block. Thus, a smaller difference indicates a high probability that the current block is the best fit. The benefits of this algorithm is that it is highly resistant against memory fragmentation since little to no memory is left over after allocating a block. However, the downside of this algorithm is that it might take longer to allocate a block to the user. Thus, implementing this algorithm is simple, but the programmer must decide if they wish to prioritize speed or optimization before implementing this algorithm.

Worst Fit:

The last algorithm is the Worst Fit algorithm, where it is the complete opposite of the Best Fit algorithm. Instead of looking for the smallest difference, the Worst Fit algorithm looks for the block with the biggest difference. Unlike the previous algorithms, it is difficult to decide if this algorithm has any benefits. However, this algorithm has plenty disadvantages. As an example, the allocation of memory blocks with a high difference makes it extremely likely to suffer from memory fragmentation. Moreover, this algorithm takes a certain amount of time to allocate a block, which makes it unfavorable to programmers looking to prioritize speed. When it comes to implementation, it is on par with Best Fit, but its lack of advantages and serious disadvantages requires careful planning before deciding on implementing this algorithm.

Test implementation

To evaluate the performance of malloc, we developed a total of 8 tests to examine how the different algorithms compare to native malloc. To use as a benchmark, we developed the tests to challenge the two versions of malloc in terms of performance, number of splits and heap growth, heap fragmentation, and max heap size. In a perfect scenario, malloc should have a relatively fast performance since it will be important to reduce the runtime of many programs. Furthermore, the number of splits and the growth of the heap will show how the different algorithms manage the demands made by the user, which demonstrates their efficiency. Additionally, we will be testing for heap fragmentation since a malloc with a high chance of heap fragmentation means we will have little memory to use due to the location of the allocated chunks of memory. Lastly, to observe how the different algorithms operate, we will output the max number of heap size to demonstrate the differences between our algorithms. To add as another benchmark, we will time our different algorithms against system malloc to deduce any potential differences.

Test Results

As we evaluated the different algorithms, it was observed that they displayed the same values for our different statistics. As observed below, each of the different algorithms displayed the same output after being used on test 1.

```
@EduardoRamos585 ➜/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-ff.so tests/test1
Running test 1 to test a simple malloc and free

heap management statistics
mallocs:        2
frees:          1
reuses:         0
grows:          2
splits:         0
coalesces:      0
blocks:         2
requested:      66559
max heap:       66560
```

```
@EduardoRamos585 ➜/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-nf.so tests/test1
Running test 1 to test a simple malloc and free

heap management statistics
mallocs:        2
frees:          1
reuses:         0
grows:          2
splits:         0
coalesces:      0
blocks:         2
requested:      66559
max heap:       66560
```

```
@EduardoRamos585 ➜/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-bf.so tests/test1
Running test 1 to test a simple malloc and free

heap management statistics
mallocs:        2
frees:          1
reuses:         0
grows:          2
splits:         0
coalesces:      0
blocks:         2
requested:      66559
max heap:       66560
```

```
@EduardoRamos585 ➜/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-wf.so tests/test1
Running test 1 to test a simple malloc and free

heap management statistics
mallocs:        2
frees:          1
reuses:         0
grows:          2
splits:         0
coalesces:      0
blocks:         2
requested:      66559
max heap:       66560
```

This phenomenon can be observed across other tests in our set of examinations, but the values vary within different tests. For example, we can compare the previous results to the results of test 3.

```
@EduardoRamos585 →/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-ff.so tests/test3
Running test 3  to test coalesce

heap management statistics
mallocs:        4
frees:          3
reuses:         1
grows:          3
splits:         1
coalesces:      2
blocks:         2
requested:      5472
max heap:       3424
```

```
@EduardoRamos585 →/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-nf.so tests/test3
Running test 3  to test coalesce

heap management statistics
mallocs:        4
frees:          3
reuses:         1
grows:          3
splits:         1
coalesces:      2
blocks:         2
requested:      5472
max heap:       3424
```

```
@EduardoRamos585 →/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-bf.so tests/test3
Running test 3  to test coalesce

heap management statistics
mallocs:        4
frees:          3
reuses:         1
grows:          3
splits:         1
coalesces:      2
blocks:         2
requested:      5472
max heap:       3424
```

```
@EduardoRamos585 →/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-wf.so tests/test3
Running test 3  to test coalesce

heap management statistics
mallocs:        4
frees:          3
reuses:         1
grows:          3
splits:         1
coalesces:      2
blocks:         2
requested:      5472
max heap:       3424
```

For the system malloc, we did not assess the specific heap statistics. However, we were able to measure our implementations against native malloc in terms of run time, which truly showed the difference in our algorithms. As an example, we used test 2 as an example since it is one of the most malloc intensive examinations we currently have.

```
@EduardoRamos585 →/workspaces/malloc-EduardoRamos585 (master) $ time tests/test2
Running test 2 to exercise malloc and free

real    0m0.008s
user    0m0.004s
sys     0m0.000s
```

```
@EduardoRamos585 ➜/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-ff.so tests/test2
Running test 2 to exercise malloc and free

heap management statistics
mallocs:        1027
frees:          514
reuses:         1
grows:          1026
splits:         1
coalesces:      2
blocks:         1025
requested:      1180670
max heap:       1115136

real    0m0.013s
user    0m0.012s
sys     0m0.000s
```

```
@EduardoRamos585 ➜/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-nf.so tests/test2
Running test 2 to exercise malloc and free

heap management statistics
mallocs:        1027
frees:          514
reuses:         1
grows:          1026
splits:         1
coalesces:      2
blocks:         1025
requested:      1180670
max heap:       1115136

real    0m0.012s
user    0m0.012s
sys     0m0.000s
```

```
@EduardoRamos585 ➜/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-bf.so tests/test2
Running test 2 to exercise malloc and free

heap management statistics
mallocs:        1027
frees:          514
reuses:         1
grows:          1026
splits:         1
coalesces:      2
blocks:         1025
requested:      1180670
max heap:       1115136

real    0m0.013s
user    0m0.011s
sys     0m0.001s
```

```
@EduardoRamos585 ➜/workspaces/malloc-EduardoRamos585 (master) $ time env LD_PRELOAD=lib/libmalloc-wf.so tests/test2
Running test 2 to exercise malloc and free

heap management statistics
mallocs:        1027
frees:          514
reuses:         1
grows:          1026
splits:         1
coalesces:      2
blocks:         1025
requested:      1180670
max heap:       1115136

real    0m0.012s
user    0m0.002s
sys     0m0.010s
```

From these test results, we can observe how the system malloc fared against our implementation of malloc.

Analysis of Results

As observed in our test results, we can see that our statistics were exactly the same for all four algorithms. Although it may seem like an anomaly, this certain behavior was to be expected. To explain this idea, we have to analyze how our malloc is performing the tasks given by the test. In every test, the program is tasked to malloc and free memory a certain amount of times. Therefore, no matter the algorithm, our malloc will provide and recover memory as many times as the program is called to do so. The behavior of the other statistics are affected by this as well since they function around the number of times malloc and free are called in the examinations. However, where we see the true difference between system malloc and the four algorithms is when we measured the runtime of each program. As an example, when we used test 2 to measure the different runtimes of the different versions of malloc, the results were surprising. System malloc only took 8 seconds to allocate and free approximately 1024 blocks of memory, each with 1024 bytes. In comparison, our version of malloc took approximately between 12 and 13 seconds to perform the same action. Thus, it is evident that the system malloc is quite efficient despite the change in time. It is important to note the difference in perfromance between the different algorithms. As observed from the results, both Next Fit and Worst Fit were tied for the shortest runtime of 12 seconds, which is quite surprising since Worst Fit is considered a bit slower than Next Fit. Even more surprising, both First Fit and Best Fit tied for the longest runtime of 13 seconds. Although it might seem odd, there is a possible explanation for why this phenomenon occurs. It's been established that First Fit is usually the most likely fastest algorithm to allocate memory. However, this quick allocation is likely to result in heap fragmentation, where the location of the allocated blocks makes it harder to allocate new chunks of memory. Thus, the program is forced to grow the heap to compensate for the demands of the user. This extra step increases the amount of runtime, which leads to First Fit having a longer runtime than expected. Of course, there are some anomalies within our examinations since different tests have different demands, which might result in one algorithm performing better when it shouldn't. Thus, we can't expect a certain algorithm to rule over the rest. Additionally, we are limited to the 2 cores we are provided in code space, which might affect our results on how the different versions of malloc fare in our tests. Thus, despite the anomalies, it can be concluded up to a certain extent that the best algorithm for speed is First Fit while the best optimization algorithm is Best Fit.

Conclusion

From this testing, we can confidently conclude that the system malloc is superior in performance when compared to our implementation of malloc. Furthermore, it can be concluded that although the different algorithms displayed identical statistics, there was a substantial difference between how each of the algorithms organized the system's resources to accommodate the needs of the user. However, our results are not without interference of

anomalous behavior or by other behavior caused by human error. Thus, it can be deduced that contradicting results against what was hypothesized should be noted but further testing is needed to fully contradict the expected behavior of the four algorithms. Overall, this experimentation provided substantial data that expanded our understanding of the architecture of our computer, which is valuable to further discussions about system calls.