Different Implementations of Malloc in C

Prepared for

Trevor Bakker, M.S.

Operating Systems

University of Texas at Arlington

Prepared by

Will Maberry

March 5, 2024

CONTENTS

LIST OF ILLUSTRATIONS

Description of Algorithm Implemented

This section of the report will discuss the techniques used by the algorithm tested. The algorithm's purpose was to stress test different implementations of malloc against one another. This section will talk about the specific techniques for the testing including multiple malloc and free calls.

Test Implementation

This section of the report will discuss how the test was implemented including the number of times it was run for each implementation of malloc. Furthermore, the implementation will provide a brief overview of how the different libraries tested were coded. The differences between each library will help us understand the test results.

Test Results

This section of the report will discuss the results of the algorithm tests. This section will contain tables, charts, and graphs showcasing the outcomes of my testing. The tables will showcase all of my data I obtained during each run of testing. The charts and graphs will show the results of each implementation that highlights how the implementations vary from one another. There will be a graph to measure the time complexity of each implementation and a chart to measure the number of coalesces and blocks at the end of the program.

Test Explanation and Interpretation

This section of the report will attempt to explain why some of the results from the previous section are the way they are. To do so, this section will investigate the inconsistencies and provide possible reasonings behind why the data is skewed how it is. Without definitive explanations and/or evidence, I will try to find potential reasons behind the results.

Conclusion

This section of the report will briefly explain the algorithm used to test the differences between the five implementations of malloc. On top of that, it will give an overview of the findings from the tests before describing potential issues with my testing.

DIFFERENT IMPLEMENTATIONS

OF MALLOC IN C

DESCRIPTION OF ALGORITHM IMPLEMENTED

The algorithm I implemented is very similar to test1.c combined with test2.c. The program starts with calling malloc() a few times, and the first call is meant to fit the rest of the malloc() calls in the program; the first malloc() is freed to allow the rest of the program take up that space.

Then, I initialize a couple of pointer arrays to be used in a loop sequence. I loop through the first array and malloc() 1024 bytes for every array position. To add to the complexity of the algorithm, I free 1024 bytes every 10th iteration through the loop and call malloc() to allocate 100 bytes into my second array. My hope with this technique is to generate different data between Best Fit and Worst fit. Every freed pointer will have at least 24 free bytes creating internal fragmentation. Once memory is allocated into the second array, I increase the counter for the second array to ensure no malloc() calls are overwritten.

Now that my arrays are filled with pointers, I call malloc() to allocate another 10 bytes. Once that is allocated, I free the pointer and call malloc() for 1 byte. My purpose for this section is to compare the time complexity of First Fit and Next Fit. Next Fit will work significantly quicker to allocate the memory because the malloc() call of size 1 will point to a free block. Next Fit, however, will have to iterate through the entire heap to find an open spot.

To generate more data between Best Fit and Worst Fit, I call malloc() to allocate 150 bytes every 10 iterations through my loop of freeing the data in the first way. Lastly, I free all the data in the second pointer array. In a real-world application, I would want to free the 150 bytes I continuously called malloc() for, but I opted not to do that. I made this decision to see how the final number of coalesces and blocks differed from one algorithm to another. [Appendix 1]

TEST IMPLEMENTATION

My tests were implemented by running my algorithm 15 times for each version of malloc(). The versions included the system's version, First Fit, Next Fit, Best Fit, and Worst Fit. The system's version of malloc was run without the "env LD_PRELOAD" command because it's automatically utilized during program execution. First fit searches the heap to find the first spot for the requested memory to fit into. Next fit is very similar to first fit but starts its search where it last left off when allocating memory. Best fit scans the entire heap and finds the smallest block of memory for the requested memory to fit into. Worst fit is the opposite of best fit and scans the entire heap to find the largest block that the requested memory will fit into.

All 75 total tests (15 * 5) were run on the same Codespace environment. My hope with this was to ensure that the same computer and/or server was used to run the tests. I wanted to mitigate the chances that some of the tests were run on one physical computer and other tests were run on another computer, so I ran every test on the same Codespace. Even if the software environment is the same across multiple Codespaces, there's a chance the physical environment could impact the algorithm's performance. For example, the room temperature and humidity the physical machine is in could alter my test data if the tests were run on different physical machines.

To ensure consistent data across the testing period, I ran all the tests in one sitting. By running the tests back-to-back, I was able to mitigate the chances of someone else using the physical machine's resources for something taxing on the system. My test data would be skewed if I ran some tests when I'm the only user utilizing the hardware and other tests when someone else is using the hardware to train an AI model. By running the tests close together timewise, I was able to mitigate the chances of that issue arising.

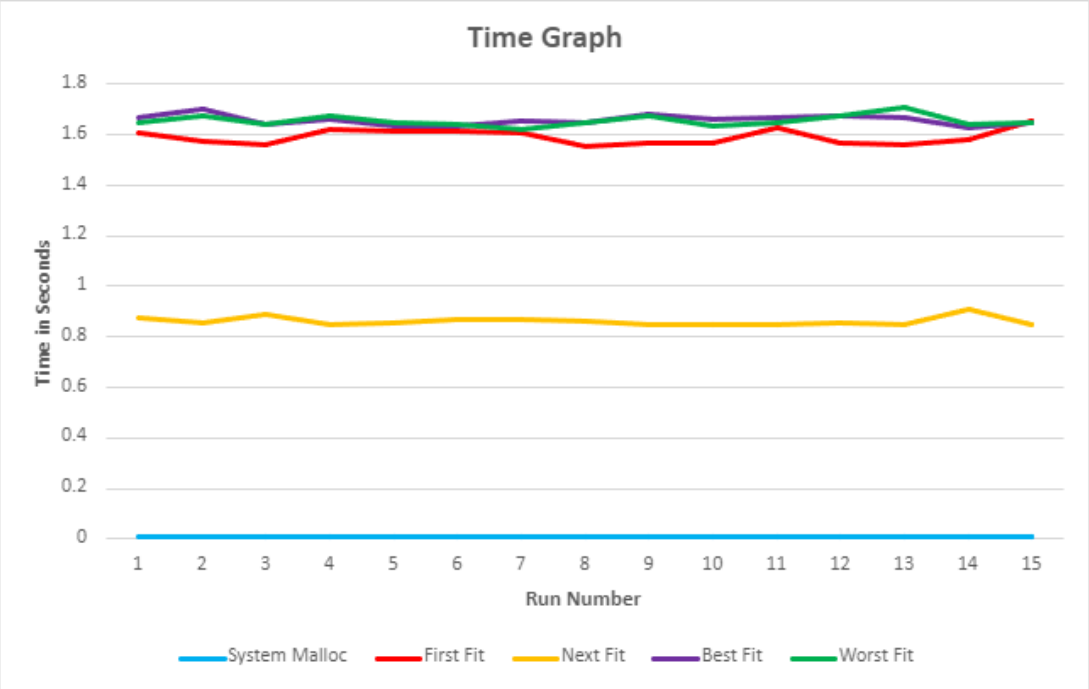
TEST RESULTS

My test results are split into two different parts. The first part of this section will focus on the time complexity of each implementation. This will show a table including all the individual data points from each test iteration and a graph showing all the data.

Table 1: Table showing time complexity between algorithms

Run #	System Malloc	First Fit	Next Fit	Best Fit	Worst Fit
1	0.01	1.607	0.873	1.67	1.648
2	0.01	1.574	0.853	1.7	1.674
3	0.011	1.559	0.887	1.644	1.645
4	0.01	1.619	0.851	1.665	1.672
5	0.011	1.615	0.853	1.632	1.648
6	0.011	1.613	0.87	1.636	1.642
7	0.011	1.605	0.868	1.653	1.624
8	0.01	1.554	0.864	1.647	1.648
9	0.01	1.567	0.852	1.682	1.675
10	0.01	1.569	0.848	1.659	1.635
11	0.01	1.628	0.849	1.667	1.649
12	0.01	1.565	0.853	1.674	1.673
13	0.011	1.559	0.851	1.669	1.71
14	0.01	1.584	0.909	1.631	1.643
15	0.01	1.652	0.85	1.649	1.651
AVERAGE	0.010	1.591	0.862	1.659	1.656

Figure 1: Graph showing time complexity between algorithms

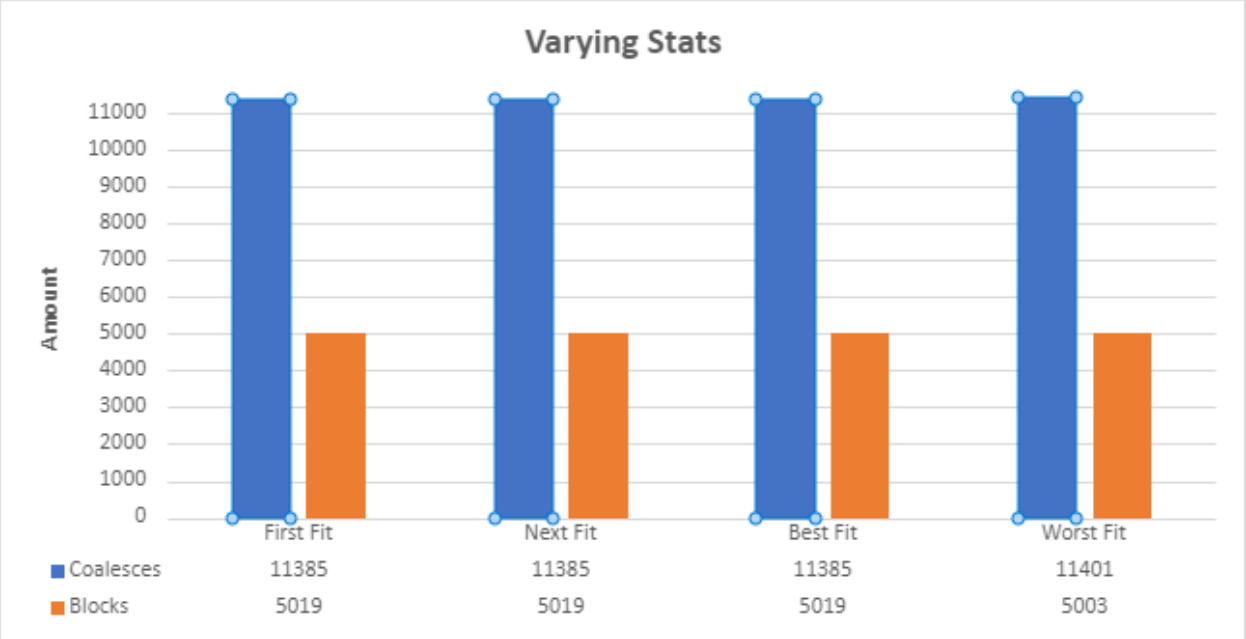


Above, we can see the time complexities of the different versions of malloc. Moving onto the varying statistics of the different iterations is quite interesting. I can't run the system's malloc with the print statistics method, so we'll leave it out of this section. Additionally, the chart only compares the different statistics between each implementation. I have also put a table underneath the chart showcasing the different statistics to make viewing easier.

Table 2: Table showing space complexity between algorithms

	System Malloc	First Fit	Next Fit	Best Fit	Worst Fit
Mallocs	xxxxxxxxxxxxxx	16405	16405	16405	16405
Frees	xxxxxxxxxxxxxx	11802	11802	11802	11802
Reuses	xxxxxxxxxxxxxx	16402	16402	16402	16402
Grows	xxxxxxxxxxxxxx	3	3	3	3
Splits	xxxxxxxxxxxxxx	16402	16402	16402	16402
Coalesces	xxxxxxxxxxxxxx	11385	11385	11385	11401
Blocks	xxxxxxxxxxxxxx	5019	5019	5019	5003
Requested	xxxxxxxxxxxxxx	719,660,698	719,660,698	vvvvvvvv	719,660,698
Max Heap	xxxxxxxxxxxxxx	705,033,764	705,033,764	706,033,764	705,033,764

Figure 2: Table showing space complexity between algorithms



The statistics for the different implementations are quite interesting. For starters, the total amount of memory requested to be allocated by the program is 719,660,698 bytes (719.66 Mb). Quite a bit of the data between the charts is identical, but the number of coalesces and blocks at the end of the program vary between Best Fit and Worst Fit.

TEST EXPLANATION AND INTERPRETATION

Looking at the graph shows us that there's a clear difference between the efficiency of the system's version of malloc and my implementations of malloc. This discrepancy is likely because the system's version has had years of optimization. Additionally, millions of people use this version daily, so it must be optimized to reduce the compute time it takes for programs to execute. On the other hand, my implementations have had 3 weeks of coding put into them and are only utilized by me. I was expecting my implementations to be slower than the system, but I'm very surprised with how much slower my implementations truly are. My assumption is that the system has an accelerated version to malloc() data iteratively like the loops my program has.

The graph showcasing the different implementations' time complexity is primarily consistent with a few dips and spikes. While I can't confirm anything, I believe that the spikes in the data are due to other users utilizing the same hardware my program was using. This is fairly clear to see with the First Fit implementation when the last few runs showed the compute time steadily increasing.

I was pleased to see that the Next Fit implementation takes about half the time as the other three implementations. I expected the time complexity between Best Fit and Worst Fit to be nearly identical because they both scan the entire heap. Next Fit's nature of starting where it leaves off is much quicker for my testing. However, I was a bit surprised to see Next Fit take so long to run. I thought it would split the difference between the other three implementations. My assumption about why this is the case is because I'm primarily adding data to the end of the list.

Moving the chart showing the difference between the number of coalesces and blocks at the end of the program tells an interesting story between Best Fit and Worst Fit. While the algorithms are very comparable, Worst Fit had more internal fragmentation by the

end of the program. My assumption as to why this happened is because Best Fit was more efficient with where it placed my malloc() calls every 10 iterations through my loops. I was interested to see what data would be generated by implementing Best Fit alternating with Worst Fit. I found that the number of blocks and coalesces of that new implementation falls between the previous two. It's not included in the table or graph as it's out of the scope of this assignment.

CONCLUSION

This report contains five different implementations of malloc() in C and compares them to one another. These implementations contain the system malloc, First Fit, Next Fit, Best Fit, and Worst Fit. First Fit finds the first spot in the heap a block of memory fits into, Next Fit is like First Fit but starts where it left off, Best Fit looks for the smallest block of memory to fit the data into, and Worst Fit is the opposite of Best Fit.

Altogether, the data I found during my five different implementations of malloc() agree largely agree with what my assumptions for the algorithm would be. My premonition was that Next Fit would be the quickest of my implementations, and the system malloc() would be quicker than all of my implementations. Additionally, I was expecting Best Fit and Worst Fit to generate slightly different data but have nearly the same runtime. I was surprised to see First Fit take so long to run the algorithm because I thought it would be about 0.3 seconds quicker than Best Fit and Worst Fit; I assumed this because First Fit doesn't have to look through the entire heap like the other two. That said, First Fit was still quicker than the two previously mentioned algorithms.

There were a few potential issues with my testing. These issues include running my tests on a shared machine. My tests aren't entirely consistent with one another because they're being run on a machine that other Codespace users have access to. As a result, the time complexity of my algorithms might vary more than if the tests were run on a local machine. On top of that issue, I could have tried to create a more advanced algorithm to generate significantly different results. For instance, I could have created an algorithm that continuously allocates large blocks of memory, frees it, and allocates small blocks. This would generate a larger gap between the time complexity between Next Fit and the other three algorithms.

This section of the report will briefly explain the algorithm used to test the differences between the five implementations of malloc. On top of that, it will give an overview of the findings from the tests before describing potential issues with my testing.

Appendix

https://github.com/CSE3320-Spring-2024/malloc-dinosaur-oatmeal

[1] screenshot showing my tested algorithm

```
// Modified version to test for report
/*
char * ptr1 = (char *)malloc(5000000000);
char * ptr2 = (char *)malloc(1024);
char * ptr3 = (char *)malloc(35);
free(ptr1);

char * ptr_arrayOne[14000];
char * ptr_arrayTwo[10000];
int j = 0;
for (int i = 0; i < 14000; i++ )
{
  ptr_arrayOne[i] = (char *) malloc(1024);

  if(i % 10 == 0 && i > 0)
  {
    free(ptr_arrayOne[i]);
    ptr_arrayTwo[j] = (char *) malloc(100);
    j++;
  }
}

char * ptr4 = malloc(10);
free(ptr4);

char * ptr5 = malloc(1);
free(ptr5);

for(int k = 0; k < 10000; k++ )
{
  if(k % 10 != 0)
  {
    free(ptr_arrayOne[k]);
  }

  else
  {
    malloc(150);
  }
}

for(int m = 0; m < j; m++)
{
  free(ptr_arrayTwo[m]);
}

free(ptr2);
*/
```