Angel Cabrera

Professor Trevor Bakker

Operating Systems

14 March 2023

Executive Summary

The objective was to make a personalized version of malloc to test against the history system malloc. Through the many test cases and runtimes, my implementation of malloc is not as efficient as the system malloc. Overall functionality is essentially the same thing, but performance, efficiency, and optimization are not there yet. I worked with the already implemented first-fit to implement best, worst, and next-fit. Those algorithms all work as intended and the main purpose for each is to find a free space to allocate a block of memory in. Each just does it in a different way. First-fit looks for the first free and capable space that can hold the block while next-fit starts the same but it then goes back to search from where it left off with a new block to allocate. Best-fit searches for the smallest available free space that can hold the space while worst-fit does the opposite, it searches for the largest available free space that can hold the block. Each test varies in terms of efficiency causing different runtimes. For my benchmark test, I decided to go with a test that allocates One Million 100-byte size blocks of memory and then frees all of them. From my results, it was determined that worst-fit was, on average, the fastest algorithm to complete it and I concluded that it was able to do that because worst-fit benefits from large allocations. Also, it was determined that next-fit was, on average, the slowest algorithm and I determined that because it was 39s slower than the other fit algorithms. In terms of speed, it goes; worst-fit, first-fit, best-fit, and next-fit. The system malloc

was able to produce the benchmark case like it was nothing. The many years of optimization definitely played a part in that. Overall, the four algorithms gave the same statistics, which I recognized was possible due to the large number of allocations. However, only the worst-fit resulted in a different coalesce stat, which I linked to lower memory fragmentation. Another ambiguity I saw was the system malloc exceeding a second of runtime after averaging 0.05 seconds. I also decided that this was due to a lot of tasks running in my browser, which most likely slowed it down slightly. This program lacks performance and optimization, but it is expected from a two-week-old program.

Algorithm Descriptions

The first of the four algorithms implemented was first-fit, the one that was already implemented for us. First-fit is working as intended and choosing the right address when testing. Its purpose is to choose whichever address it the block will fit into first from the beginning to the end of the list. Whichever address is first, is the address chosen. Compared to next-fit, first-fit restarts at the beginning of the list after a block is allocated instead of where it left off like on next-fit. If a block does not fit it is ignored and not allocated. An advantage to first-fit is its simplicity, however it can produce poor performance if there is a lot of fragmentation.

Next-fit's purpose is to allocate an address into whichever slot it fits in as the order goes, however instead of restarting from the start the list every time, it continues from the last allocated slot. If it has iterated through the entire list it will restart from the top and look again, if it then doesn't find a slot, it will restart again with a new block ready for allocation. At least to my understanding this is how it works. An advantage to next-fit is it's more likely to have less external fragmentation, however it can be slower and its internal fragmentation may be large. I kind of went a different way when implementing this, I used my best fit code and tweaked it ever so slightly and it got me the right address when testing so I figured that it was correct. It starts with an if statement that sets curr equal to next if next is not NULL Then while current is true, it checks if the current node is free and the size of the current node is more than the size, if it is true, it goes into another if statement that checks if the leftover size is less than the next size, setting next size to current size and next to current. After it exits the if statements, it sets last node to current and current to next current. If current is NULL, the while loop terminates and it sets current to next. I do feel like I did too much on it, but when testing it gave me the right address so I didn't bother changing it. However, one thing that does happen is since I did use my best fit code, it also works for best fit and vice versa. I found that interesting since it works for the next fit, but it may be that the tests were easy to pass and a small error could also slip through causing a passing address.

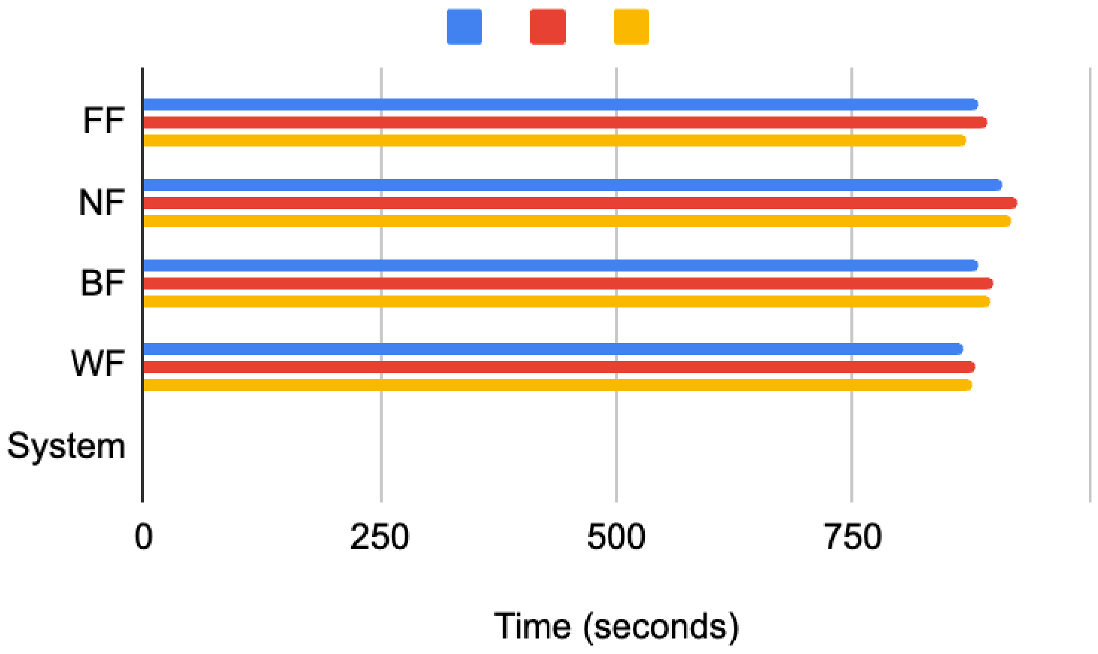
Best-fit's algorithm is to search the whole heapList and find the best suitable address space for the block. It always restarts from the top whenever a new block is going to be allocated. An advantage is its good at allocating and saving space, it's efficient, but it is slow and can waste time when it's looking for the best block of memory. I implemented best fit and worst fit similarly as there are only minor differences for both. For best fit I started with initializing a block named best and a long long value named best size that equals INT_MAX. Then it goes into a while loop that checks if the current is not NULL, in the loop it checks if the size is less than the size given to us, if it is it then checks if the leftover size is less than the best size, in this case INT_MAX, it sets best size to the current size. When it exits it sets the last node to the current and current to the next. After the while loop is done and the best size has been found, it returns the best fit size for that block that is allocated. I found this one kind of easy to work with, just making sure that the address is right when testing was a little difficult, other than that best fit is pretty good, time wise we will see soon.

Worst-fit's algorithm is to search the entire heap list for the worst spot to go in. It's like me being 5 '7 and wanting to drive a monster truck, while I can, but that doesn't mean that I should. It searches for the largest free memory block that can hold the size of the requested. It can cause a lot of fragmentation and it is slow, but in the case of my testing, it was one of the faster ones. I still don't know exactly why, but maybe it was all the frees that I did or I allocated just the right amount for it to go up in ranking. Moreover, my implementation is basically best fit with two changes, in the second if statement I changed it to, if leftover size is more than the worst size, in this case worst size is initialized to 0 at the start, then it continues to be the same as best fit until the end part. I was testing and when I removed the line for the current node to change to the next node, it was still able to find the right address. When testing it became apparent that my program needed optimization when it came to runtime.

Testing Implementation & Results

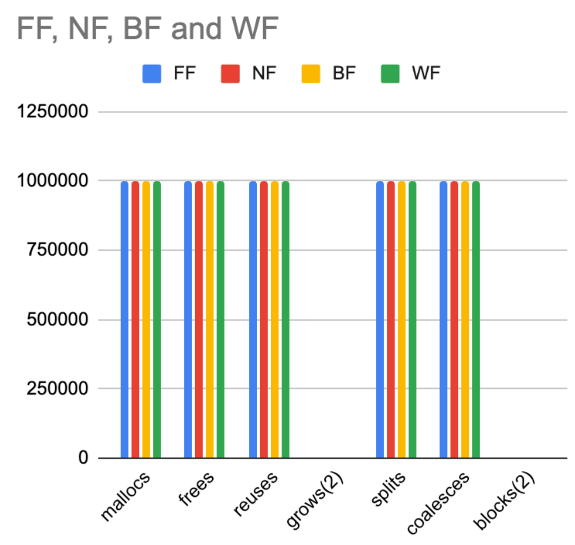
For my testing I just wanted to stress it out as much as possible within a timely manner, it still took a long time to show results. So, I made it do some regular malloc tests, one being a big size with division and multiplication, then it allocates one-million one-hundred byte size blocks. It definitely put my program to work. At first I had it allocate ten-thousand blocks but that still had a small runtime so I went to one-hundred thousand and while it showed a difference, it was kind of boring, so I did five-hundred thousand then I followed the Professor's advice and went for a million, and wow did it show a difference. Following all of those malloc's, it then has to free all of the blocks up to the second to last one in one for loop. It then frees the last one outside of the loops. I also did one test with ten-thousand allocations of one-hundred thousand byte size blocks, in that test I believe the amount requested overflowed because I got a negative number. At the end of it I chose one million mainly because it had a big runtime and it was enough for me to peak my interest. Overall my statistics from my four algorithms are constant, there was no change between the four when tested with my benchmark. The only difference that can be noted is in the run time, for each algorithm it was different enough to warrant this attention. Here we can see the times from my benchmark tests. Remember that these are in seconds.

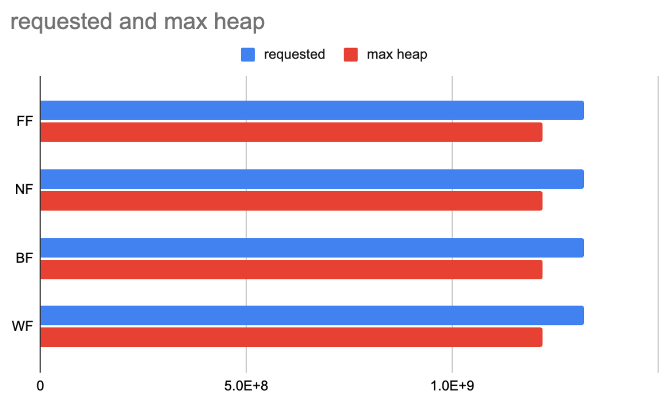
FF	883.22261	892.645569	870.19873
NF	907.654541	923.995544	918.191895
BF	882.300354	897.45343	896.553467
WF	867.125427	879.740479	876.709656
System	1.826751	0.088799	0.107088



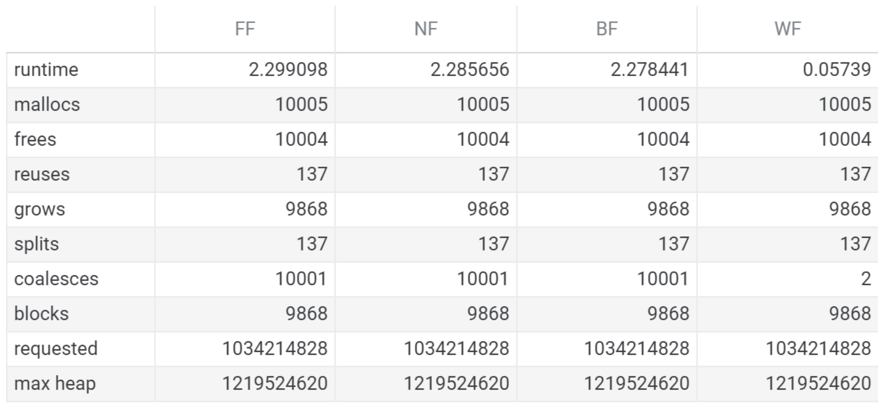
Looking at my results, there is obviously room for improvement, especially when it comes to optimization. Each fit had its own distinctive time and even the system had some distinctive results. First fit had an average return time of 882.02s with the fastest being 870.198s and the slowest being 892.645s. Next fit had an average return time of 916.613s with the fastest being 907.654s and the slowest being 923.995s. Best fit had an average return time of 892.102s with the fastest being 882.3s and the slowest being 897.453s. Worst fit had an average return time of 874.52 seconds with the fastest time being 867.12s and the slowest being 879.74s. The system had an average runtime of 0.674s with the fastest return time being 0.107s and the "slowest" return time being 1.82s. After these runtimes I got my statistics and it stayed consistent between all fit algorithms. I wish I was joking but they are all the same when I malloc one-million 100 byte size blocks, when I don't I get different results.

	FF	NF	BF	WF
mallocs	1000005	1000005	1000005	1000005
frees	1000004	1000004	1000004	1000004
reuses	1000003	1000003	1000003	1000003
grows(2)	2	2	2	2
splits	1000003	1000003	1000003	1000003
coalesces	1000001	1000001	1000001	1000001
blocks(2)	2	2	2	2
requested	1319528144	1319528144	1319528144	1319528144
max heap	1219524620	1219524620	1219524620	1219524620





Based on the averages my fastest algorithm is worst fit, which is interesting because I would not have expected worst fit to be the fastest. I think it's the fastest because it was more efficient than the others and it has such a big allocation of blocks it favors the worst fit method. I fully expected worst fit to be the one of if not the slowest algorithm, but with my benchmark it was the fastest. I also tested it with allocation ten thousand nine-hundred thousand size blocks, in this test the worst fit still outperformed the rest of the fits by 190%. Also, based on the averages, next-fit is the worst possible fit algorithm that can be chosen. Based on the stats it is slower by an average of 39s. Even when it is basically the same code as best fit, for some reason it runs slower than best fit. I believe this is the case because of the extra line where the current node equals next node if next is not NULL. Aswell as that line I believe it's slower because next-fit goes through the entire heap list to look for a free space that can allocate the block. Not only that, but if it doesn't discover an empty space, it returns to the top to look at the places it missed, only then if it doesn't find one it tries for a new block. The order of efficiency from best to worst goes from worst, first, best and next fit, however when compared to system malloc it stood no chance. The system malloc outperformed my implementation by a mile, but it did have some interesting stats. On average it was insanely fast, mainly due to the decades of optimization, compared to my fits. On one interesting occasion it actually reached one (1) seconds. However, I don't think it was anything to do with the actual system program, I have come to a solution as to why it reached one second. I believe that my browser played a role in it, I think since I had too many active tasks that might have caused the runtime to run a little longer. Although the runtime for each of the fits and the system was distinctive, the statistics were quite the opposite. There was no way of measuring the statistics for the system malloc, so it will be voided for this part. So, all of my fit algorithms had the same statistics, there was no difference in anything. I came to the conclusion that it did not change because there were too many allocations for it to have any distinctive statistics. However, when I did something other than my benchmark, the only fit algorithm that had any difference would be the worst fit.



	FF	NF	BF	WF
runtime	2.299098	2.285656	2.278441	0.05739
mallocs	10005	10005	10005	10005
frees	10004	10004	10004	10004
reuses	137	137	137	137
grows	9868	9868	9868	9868
splits	137	137	137	137
coalesces	10001	10001	10001	2
blocks	9868	9868	9868	9868
requested	1034214828	1034214828	1034214828	1034214828
max heap	1219524620	1219524620	1219524620	1219524620

From this table it can be said that worst fit is the best implementation in terms of runtime and memory fragmentation. In this test worst fit had only two coalesces. Meaning that memory was not being broken down enough for it to warrant a coalesce. Overall the fastest algorithm is the system malloc, obviously but if we don't count that, my worst fit algorithm had the best performance in terms of runtime. The only ambiguity that I noticed was the system malloc performing over 1 second, I learned that it was because of too many tasks running.

Conclusion

In conclusion, my implementation of malloc and fit algorithms performs as expected from a two-week-old program. The fit algorithms do not even come close to competing with system malloc, it was meant to be a stress test and the only thing that stressed were my tests, system malloc chilled. The first fit did its job in finding the first allocation that can hold the block as did the next fit in finding the next available allocation to hold the block. The best fit underperformed my expectations but it did its job and the worst fit surprised me the most. Overall all of the fit algorithms did their job and if they were optimized to their full potential there would be no debate as to which one is the best. However, right now I can confidently say that worst fit is the best-fit algorithm to use for big and medium-size allocations. It outperforms every other algorithm and has less memory fragmentation when tested with medium-size allocations.The results showed that worst fit has capabilities to be the best and I believe it can. However, I think it would be better if it was used with another fit algorithm, both together would create a strong and efficient way to allocate blocks. If we can get the runtime and memory fragmentation efficiency together with another algorithm, it would be able to compete with the system malloc. I enjoyed working on this program and my results showed that no matter what, there will always be room for improvement. Even if it is done, that doesn't mean that it is efficient or optimized.

Appendix

Test Code

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define SIZE_BLOCK 100

int main()
{
   printf("Running benchmark test \n");
   clock_t time;
   time = clock();
   char *ptr1 = (char *)malloc(123155156924/124314*1231);
   free(ptr1);
   char * ptr2 = ( char * ) malloc ( 1200 );
   char * ptr3 = ( char * ) malloc ( 1200 );
   free( ptr2 );
   free( ptr3 );
   char *mill_ptr[1000000];
   for(int i = 0; i <= 1000000; i++) {
      mill_ptr[i] = (char *)malloc(SIZE_BLOCK);
   }
   for(int i = 0; i <= 999999; i++) {
      free(mill_ptr[i]);
   }
   free(mill_ptr[1000000]);
   time = clock() - time;
   printf("I took %f seconds. \n", ((float)time) / CLOCKS_PER_SEC);
   return 0;
}
```