BENCHMARK COMPARISON AND ANALYSIS OF C'S SYSTEM MALLOC V.S. STUDENT MALLOC

Prepared for

Trevor Bakker, Ph.D.

Operating Systems

University of Texas at Arlington

Prepared by

Preston Loera

March 14, 2024

Contents

PAGE

Executive Summary......iii

Description of the heap arena algorithms implemented......1
- First fit algorithm......1
- Next fit algorithm......3
- Best fit algorithm......4
- Worst fit algorithm......5

Description of test program......6

Explanation and interpretation of Test Results......8
- Explanation and interpretation of time table and graph......9
- Explanation and interpretation of stats from the four algorithms......10

Conclusion......11

FIGURES

Implementation of First Fit......1
Implementation of Next Fit......2
Implementation of Best Fit......4
Implementation of Worst Fit......5
Implementation of benchmark......6,7
Benchmark time table......8
Benchmark graph......9
Benchmark Splits, Blocks, and Max Heap table......10

EXECUTIVE SUMMARY

This report will compare and analyze the C libraries system malloc against the malloc developed by student Preston Loera. Using *GitHub Codespaces*, system malloc and the students malloc alongside four different heap arena algorithms will be benchmarked to compare the performance, relative comparison of splits and heap growth, heap fragmentation, and max heap size of each malloc.

Summary of the four different heap arena algorithms

Four different heap arena algorithms have been implemented by Preston Loera with each having different conditions on how a block in the heap is chosen and returned. The four algorithms that have been implemented are first fit, next fit, best fit, and worst fit. Based on the algorithm running during benchmarking, the stats should change based on the number of heap growths and splits that are needed during the runtime of the benchmark programs. Providing stats that can compare and contrast the algorithms.

Benchmark programs implemented

A test program has been implemented with the goal of the benchmark to be able to stress the students' heap algorithms and malloc alongwith C’s system malloc. The stress on the algorithms should help separate the performance of the algorithms based on the amount of time the program takes to run and the relative number of splits, grows, and heap size that is returned at the end of the program.

Results from benchmark

The four algorithms were separated by performance from the benchmark and were significantly outperformed by C’s system malloc. Even with this significance, it is expected due to system malloc being around and optimized for many years now whereas the students malloc was only implemented a couple weeks ago from this report being written. Even so, the results from this benchmark show the performance of the algorithms learned in the Operating Systems class in a real world application and gives a chance for us to be able to learn and understand them better.

Description of the heap arena algorithms implemented

Four different heap algorithms were implemented to be able to iterate through the heap called “heapList” and are used to find blocks that are currently labeled as “free” and return them to be able to split, coalesce, or properly allocate a new block based on “NULL” return. Each of these algorithms finds a free block in the current list a different way and can change the number of splits, coalesces, and possibly time needed to perform the proper instructions. *(program blocks included in the report are from Preston Loeras malloc.c program. Comments have been removed for the sake of convenience for the reader).*

```
#if defined FIT && FIT == 0
   while (curr && !(curr->free && curr->size >= size))
   {
      *last = curr;
      curr  = curr->next;
   }
#endif
```

Figure 1: Implementation of First Fit

First Fit algorithm

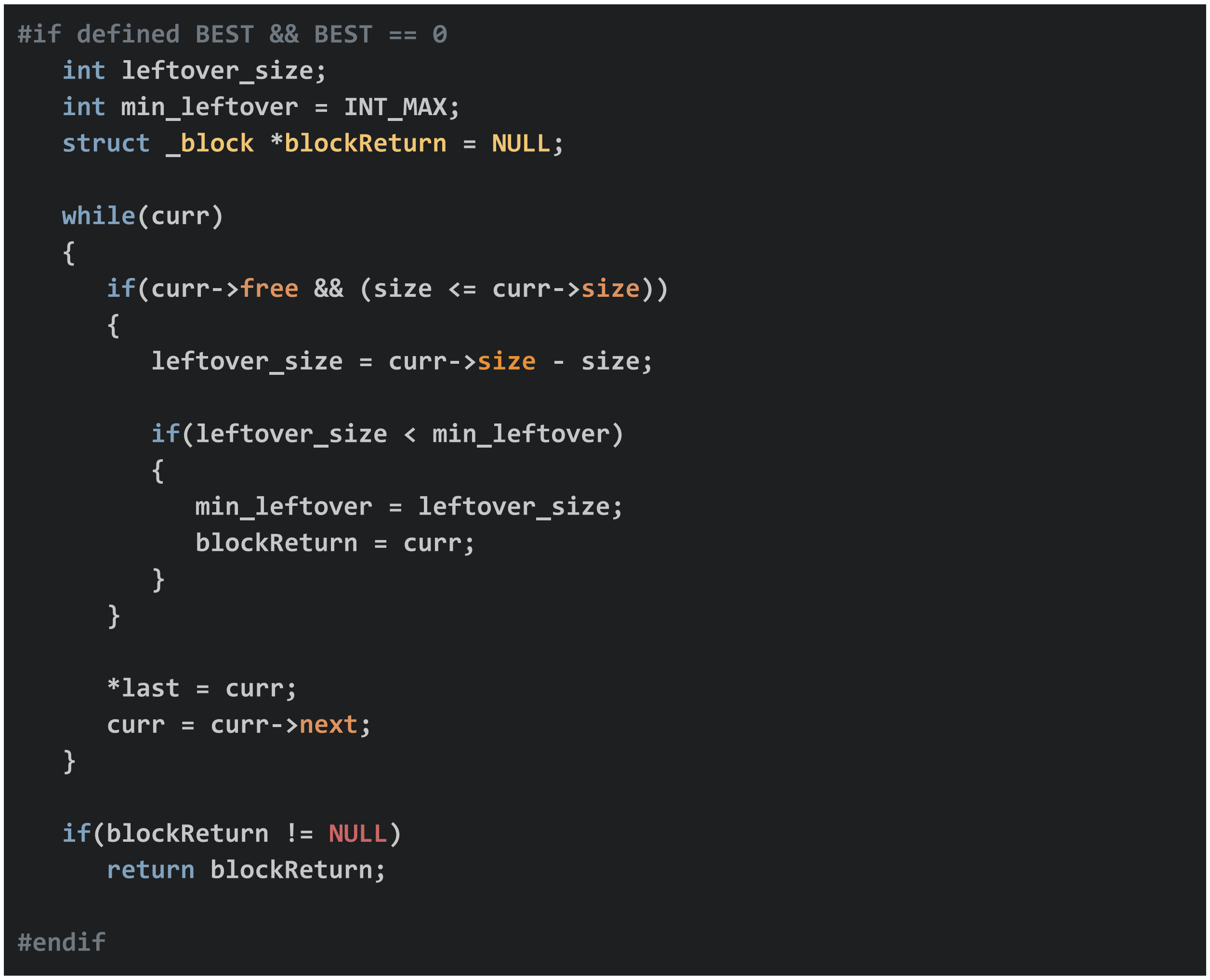
First fit was provided by professor Bakker as an example heap algorithm and works as follows. First fit will iterate through the current heapList using a while loop and will conclude its search until a block that is free and can fit the users requested size or till the end of the heapList is reached. In the case the end of the heapList is reached the function will return null and the heap will grow and create a new block of the users requested size. Otherwise if a block that fits the users memory request size and is free, the function will immediately return that free block and use that existing block's memory.

```
#if defined NEXT && NEXT == 0
   if(lastFreeBlockFound == NULL)
   {
      while (curr && !(curr->free && curr->size >= size))
      {
         *last = curr;
         curr  = curr->next;
      }

      if(curr && curr->free)
         lastFreeBlockFound = curr;
   }
   else
   {
      curr = lastFreeBlockFound;
      while (curr && !(curr->free && curr->size >= size))
      {
         if(curr->next == NULL)
         {
            *last = curr;
            curr = heapList;
         }
         else if(curr->next == lastFreeBlockFound)
         {
            lastFreeBlockFound = NULL;
            while (curr)
            {
               *last = curr;
               curr  = curr->next;
            }
         }
         else //Normal continue
         {
            *last = curr;
            curr  = curr->next;
         }
      }

      if(curr && curr->free)
         lastFreeBlockFound = curr;
   }

#endif
```

Figure 2: Implementation of Next Fit

Next fit algorithm

Next fit is the most complex algorithm that is implemented out of the four but functions similarly to first fit. The first time this algorithm runs, it will run almost exactly to how the first fit runs. Looping until the first block that can fit the user requested size and is free then returns that block immediately. Next fit differs from first fit by remembering where the loop last “left off” by storing the found free block into a variable called lastFreeBlockFound. This will be used as a bookmark to mark the start position the next time this algorithm is used to loop the heapList. The next time neft fit runs; the loop will start from the place of lastFreeBlockFound instead of the beginning. Looping the entire heapList until curr is equal to the node that lastFreeBlockFound was equal to. Next fit also has a unique quirk to see if the end of heapList is reached but still needs to check the beginning of heapList if lastFreeBlockFound was not null. The program will point curr to the top of the list and allow looping to continue. Having the same condition of immediately stopping if a block that can fit the user request size is found. If a new free block is found lastFreeBlockFound will be updated to the new block and will loop from that new position. In the case of no free block being found curr will be pointed to the end of the list and lastFreeBlockFound will be set to null to allow looping from the beginning of the list.

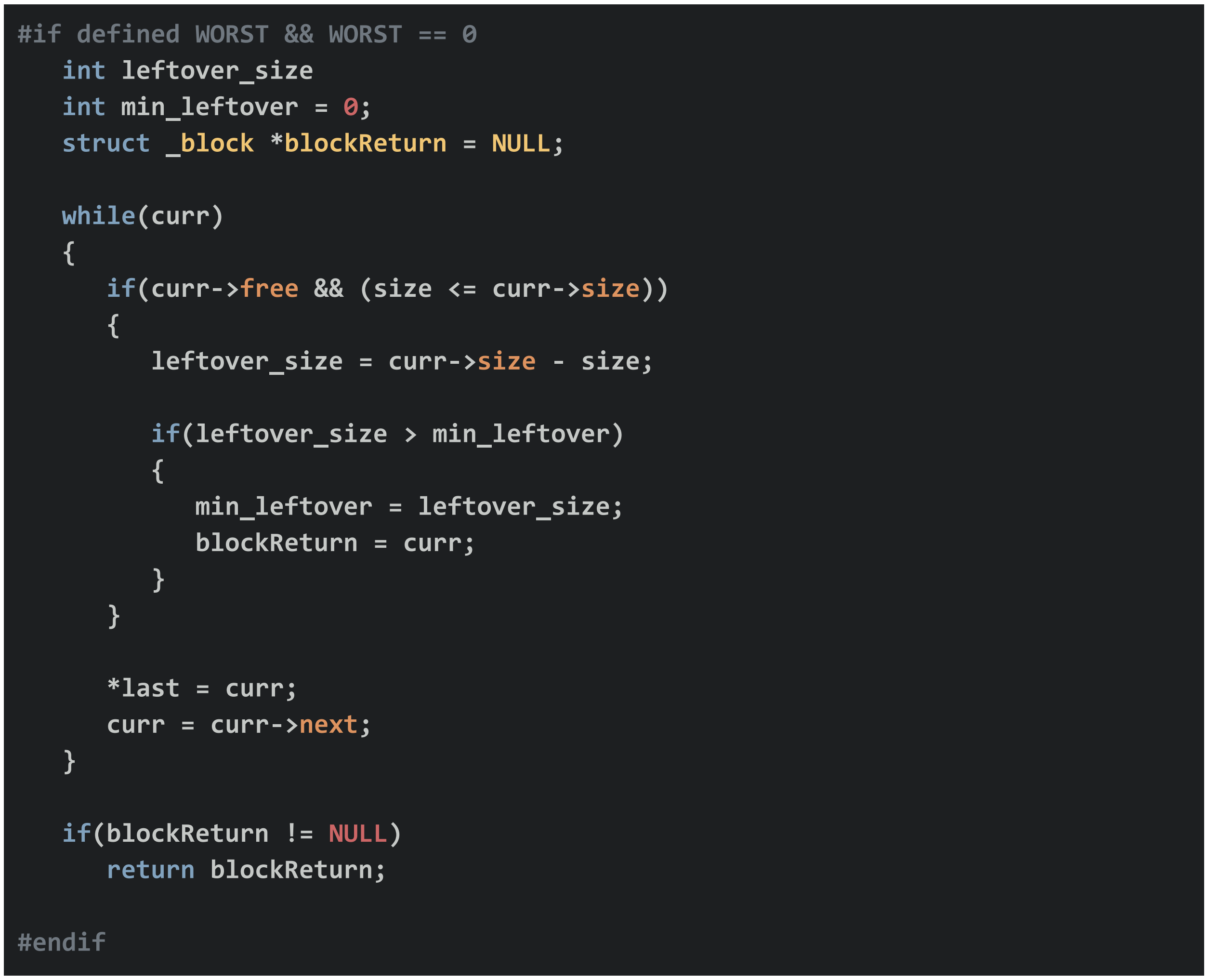


```
#if defined BEST && BEST == 0
   int leftover_size;
   int min_leftover = INT_MAX;
   struct _block *blockReturn = NULL;

   while(curr)
   {
      if(curr->free && (size <= curr->size))
      {
         leftover_size = curr->size - size;

         if(leftover_size < min_leftover)
         {
            min_leftover = leftover_size;
            blockReturn = curr;
         }
      }

      *last = curr;
      curr = curr->next;
   }

   if(blockReturn != NULL)
      return blockReturn;

#endif
```

Figure 3: Implementation of Best Fit

Best Fit algorithm

Best fits goal is to find the “best” free block in the current heapList and return it. The way best fit determines which free block is considered the “best” is based on the bytes of leftover space when the block size is deducted from the memory size request from the user. The best fit algorithm uses a while loop to iterate over the current heapList until the end. While looping an if statement will check if the current block is free and if it can fit the size of the requested user. If the current block can, we check to see if the leftover size of the block is smaller than the previous leftover size. On success, the current block will be remembered using blockReturn and be returned once the while loop concludes. Otherwise, blockReturn will continue to be assigned to what it was previously and continue. The leftover size will start out as the greatest possible value an integer can be and will continue to decrease as more free blocks are found.



```
#if defined WORST && WORST == 0
   int leftover_size
   int min_leftover = 0;
   struct _block *blockReturn = NULL;

   while(curr)
   {
      if(curr->free && (size <= curr->size))
      {
         leftover_size = curr->size - size;

         if(leftover_size > min_leftover)
         {
            min_leftover = leftover_size;
            blockReturn = curr;
         }
      }

      *last = curr;
      curr = curr->next;
   }

   if(blockReturn != NULL)
      return blockReturn;

#endif
```

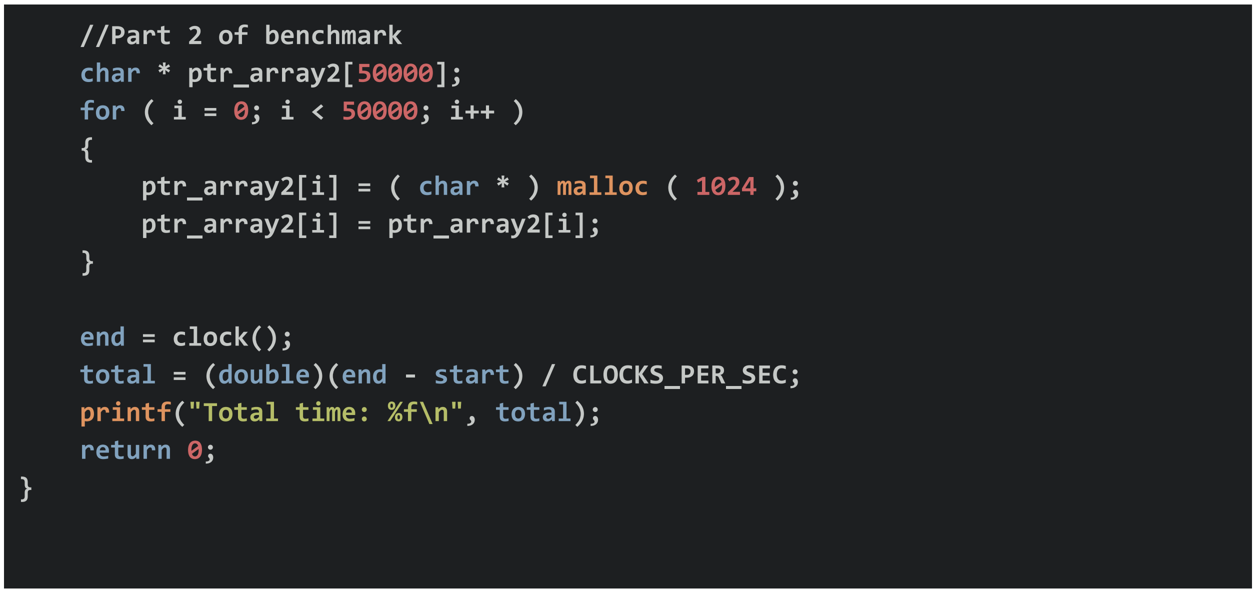
Figure 4: Implementation of Worst Fit

Worst fit algorithm

Worst fit takes the same idea as best fit but instead of minimizing leftover space, the algorithm maximizes this loss. Besides a few line changes, worst fit functions the same as best fit. looping through the current heapList until the end is reached. While looping, an if statement will check if the current block is free and can fit the size of the user's request. If the current block fits these conditions another if statement will check if the leftover size is greater than the previous leftover space. On success, this block will be remembered using blockReturn and will be returned later. Otherwise blockReturn will retain its previous assignment and continue to loop.

Description of test implementation

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

int main()
{
    char * ptr_array[50000];
    srand(0);
    clock_t start, end;
    double total;
    start = clock();

    //Part 1 of benchmark
    int i;
    for ( i = 0; i < 50000; i++ )
    {
        int block_odds = (rand() % 10) + 1;

        if(block_odds <= 4)
        {
            ptr_array[i] = ( char * ) malloc ( 1024 );
        }
        else
        {
            int block_size = (rand() % 5120) + 1024;

            ptr_array[i] = ( char * ) malloc ( block_size );
        }
        ptr_array[i] = ptr_array[i];
    }

    for ( i = 0; i < 50000; i++ )
    {
        int free_odds = (rand() % 10) + 1;
        if( free_odds > 6 )
        {
            free( ptr_array[i] );
        }
    }
```



```
    //Part 2 of benchmark
    char * ptr_array2[50000];
    for ( i = 0; i < 50000; i++ )
    {
        ptr_array2[i] = ( char * ) malloc ( 1024 );
        ptr_array2[i] = ptr_array2[i];
    }

    end = clock();
    total = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Total time: %f\n", total);
    return 0;
}
```

Figure 5: Implementation of Test Benchmark

The benchmark program starts off by calling malloc 50,000 times to set up the initial heap that the algorithms will have to loop through. Although we are utilizing "random" numbers to be able to change the size of the block that is being called. When a new block is about to be allocated a random number is generated, if that number is less than or equal to 4 the block will be a set size of 1024 bytes. Otherwise the block size will be between 5120 bytes to 1204 bytes. To prevent randomness completely from test results the function srand() is used to be able to "remember" these generated numbers. This allows all of the random numbers to be the same regardless of the malloc being used or the heap sorting algorithm used for testing. Once 50,000 blocks are allocated the program will free the blocks based on a random generated number and loop through the current heap list to possibly free the allocated block. The second part of this program allocates another 50,000 blocks of 1024 bytes and is meant to put stress on the algorithms to find the valid block that algorithms finds. The way time is recorded is by using the clock() function. This function is provided by the "<time.h>" library and returns the number of clock ticks since the function was called. The clock function is called before allocating begins and immediately after the second part of the program allocation concludes. The way total time is calculated is by "(end - start) / CLOCK_PER_SEC" this formula returns the time in seconds and then will be recorded onto a table for further testing.

Explanation and interpretation of Test Results

Each of the algorithms and system malloc were tested twenty times to be sure that the data being given was accurate and to be given a good average of the test results. Testing was done on the evening of March 14th and the morning of the 15th over the course of a couple hours.

Benchmark (time in seconds)

	NF	FF	BF	WF	MALLOC
1	126.288585	146.926659	170.136035	172.085502	0.978811
2	128.215849	150.251621	171.03522	172.408772	0.97618
3	128.399234	149.618529	172.28513	173.744957	0.972781
4	127.536721	176.899772	171.940884	172.987527	0.980732
5	127.269223	150.172677	203.973141	168.841069	0.973298
6	126.088082	147.830578	202.04165	169.482314	0.960092
7	129.537687	149.054168	199.914061	193.031454	0.971532
8	127.618167	148.079823	167.983722	180.991786	0.969842
9	131.517739	148.263829	165.861949	169.801395	0.972446
10	131.626959	149.425158	167.728377	172.14672	0.972148
11	129.952593	153.126948	170.184373	165.908756	0.968177
12	130.429732	185.401168	169.898655	168.93131	0.973149
13	130.30885	179.148286	168.226157	169.087884	0.971942
14	127.338867	192.266177	171.018599	169.181476	0.971612
15	127.227665	188.540725	168.155448	168.411906	0.969793
16	128.860923	152.29901	168.222867	170.279504	0.968051
17	132.383404	180.374065	169.649265	169.019803	0.965901
18	131.492561	186.923726	165.219676	170.400704	0.964158
19	130.030729	176.30347	165.609478	170.411665	0.980902
20	130.888871	188.030235	166.520481	173.523017	0.964067
Average	129.1506221	164.9468312	173.7802584	172.0338761	0.9712807

Figure 6: Benchmark time table

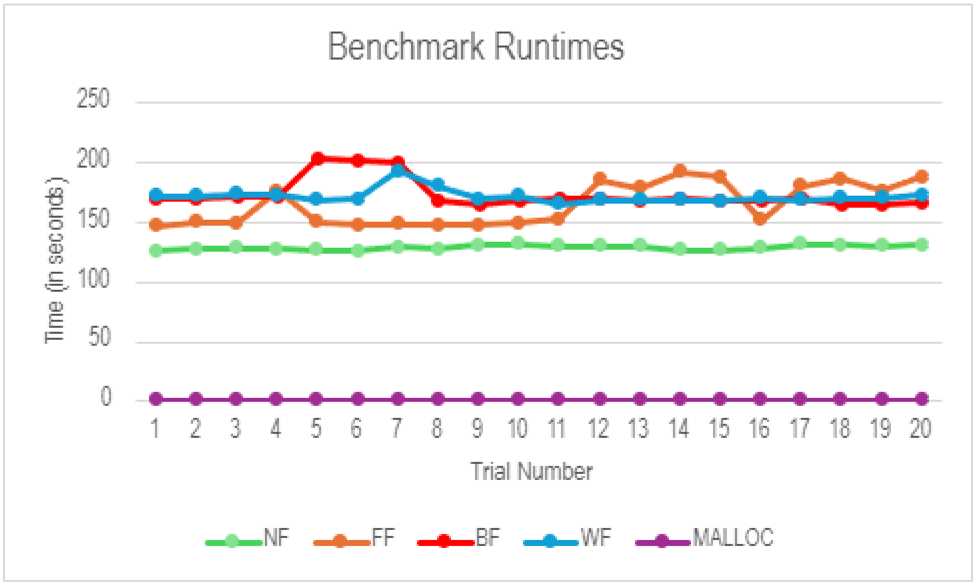


Figure 7: Benchmark graph

Explanation and interpretation of time table and graph

From the table and graph the times from benchmarking clearly show differences of performance based on the four algorithms that were running. With next fit having the fastest average time of 129.1506221 seconds and best fit having the highest time of 173.7802584 seconds. As for why next fit performed the fastest might be due to next fit algorithm being able to remember the location of the last block it selected. Due to the nature of the benchmark; blocks were freed in the current heapList from top to bottom order. The benchmark should already favor both first fit and next fit. Due to no more blocks being malloced after the end of part 2 of the program. Next fit most likely saves on time with the help of remembering where the algorithm last left off giving the edge to next fit to being searching from a head start when compared to first fit. Best fit and worst fit tank in performance due to having to search the entire heapList anytime malloc is called to have a chance to find a free block. Although as expected from the results is the performance of both best fit and worst fit being similar. With the algorithms similarities and how they function relatively the same. Some results that were not expected were the sudden time jumps that first fit, best fit, and worst fit all experienced. Since testing was done over the course of a couple hours on *Github Codespaces* this could've tanked our performance when running our benchmark due to the free version of *Codespaces* being used. Potentially running the benchmark program on a lower priority rather than paying customers on a higher priority. First fit seemed to have been the most affected by this due to the sudden increase in time during the later portions of trialing. Although the true winner of this benchmark is C's system malloc with an average time of .9712807 seconds. Being undeniably the fastest performer out of the algorithms that were tested.

Heap fragmentation stats of four algorithms

	NF	FF	BF	WF
Malloc	100001	100001	100001	100001
Frees	19958	19958	19958	19958
Reuses	45256	45256	45256	41847
Grows	54745	54745	54745	58154
Splits	41585	41585	41585	41585
Blocks	88369	88369	88369	91778
Requested	179258508	179258508	179258508	179258508
Max Heap	132109388	132109388	132109388	135600204

Figure 8: Benchmark Splits, Blocks, and Max Heap table

Explanation and interpretation of stats from the four algorithms

Worst fit out of the four heap algorithms resulted in the largest max heap and number of blocks that were needed. Which makes sense based on how the algorithm chooses a block that is free in the heapList. The likelihood of the selected block in worst fit needing to be split is the highest out of the four. Although some potential anomalies might have come from the stats that are returned by the four algorithms with the splits, blocks, and max heap matching for the first three algorithms in *Figure 8*. As to how this could happen between next fit and first fit is due to the way the benchmark program allocates and free blocks. Freeing blocks from top to bottom both next fit and first fit are going to be selected the same first block that is free at the top of the heapList. Best fit however is not similar in the way the best block is selected. It can be possible that the utilization of random numbers to be able to set up a standard heap could've been made in such a way that best fit would select the same blocks as next fit and first fit but is unlikely. Another reason as to why best fit has the same number of blocks is that the blocks that were found are in fact different but produce the same numbers as first fit and next fit. Giving the illusion that the same blocks were chosen but in reality the heapList could be different by the end of compiling the program. Another possible reason is that the stats programmed by the students' malloc could be wrong and give the impression that the stats are the same when in reality they are different. Worst fit however does show differences in the amount of blocks and max heap size that was returned. With the results making relative sense due to worst fit resulting in more splits when compared to the other algorithms resulting in the max heap size to increase in byte size by the end of compilation.

Conclusion

From the results of benchmarking, C’s system malloc significantly outperformed the other four algorithms time wise. Although this result was to be expected due to the fact that system malloc has been around for at least fifty years now when compared to the students malloc lifetime of a couple weeks. The purpose of this benchmark was to be able to gain a better understanding and real world application of the algorithms learned from Operating Systems and to see how these algorithms perform today and the issues that they have when put under stress. Interpreting and analyzing why the results from the benchmark happened based on the previous understanding given and leaving room for how these heap sorting algorithms can be improved or innovated on. Giving a better understanding of the advantages and disadvantages of each sorting algorithm. On Top of realizing just how fast the system malloc is when compared to creating your own memory manager.