ECE 650 – Spring 2020
Project #1: Malloc Library

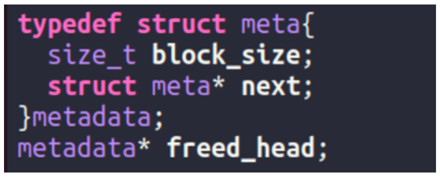
Name: Weihan Zhang NetID: wz125

1 An Overview

1.1 Malloc Function Implementation

- Meta-Information Storage & Free List:

Since we also need to implement the free function which is only passed in a pointer, we have to know what size the block is associated with. To implement this, we store the meta-information in the memory region which was hidden from the user. Additionally, when malloc was called, it is a best way to reuse the freed blocks without returning to the OS. So, I choose to use a singly linked list and use struct which contains block size and the address of the next block to implement it.
The following is the struct looks like and the freed_head is the head of the linked list.



```
typedef struct meta{
  size_t block_size;
  struct meta* next;
}metadata;
metadata* freed_head;
```

1.1.1 First-Fit Policy

The first fit policy will find the first block in the freed linked list that has enough space to allocate. So, I need to iterate through the whole linked list and compare the size of each block in the list with the requested size. There are two possible situations that we could handle the malloc without using the sbrk system call. One is that the current block size in the linked list is as same as the requested size. We just need to remove this block from the linked list. The other one is the block size is larger than the requested size plus the size of meta-information, so we need to split the block. Once we iterate the entire linked list without finding any blocks that satisfy these two situations, we need to use system call to allocate space in the heap.

1.1.2 Best-Fit Policy

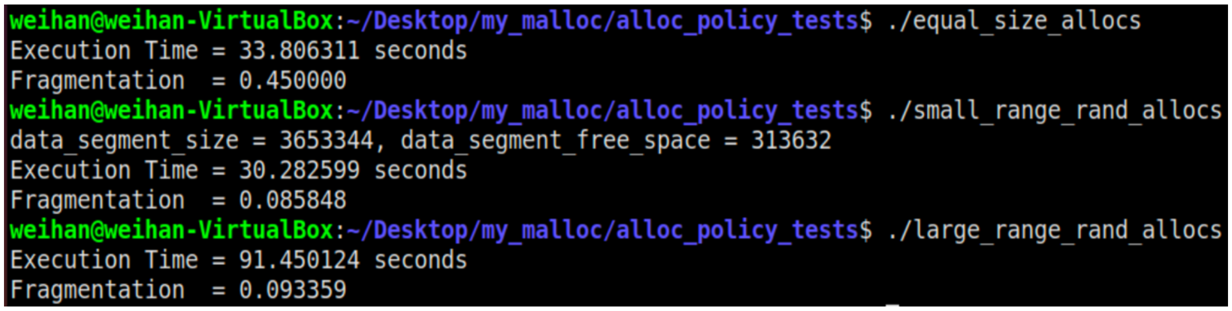
The best fit policy will iterate through the entire linked list and allocate the address from the free region which has the smallest number of bytes greater than or equal to requested allocation size. The while loop in bf_malloc function is to examine all of the free space information and record which block can be the best. If we find the best block, then we allocate that block. Otherwise, we use the sbrk system call to allocate a new space in the heap. In order to optimize the runtime, in the while loop, we firstly check whether the current block in the linked list is already the smallest blocks which can exactly fit the requested allocation size. If it is, we will return directly.

1.2 Free Function Implementation

Both first fit policy and best fit policy have the same free function. We insert the block which was pointed by the pointer into its correct place in the linked list (The node in the linked list was in sequence of their address). Then, we need to examine whether there are some blocks need to be merged. So, firstly, we check if the block which is needed to be freed and its next block are continuous. Then we iterate through the linked list to check whether the front block also needs to be merged. In this case, a doubly linked list will work better than the current singly one. Additionally, we do not need to examine the block whose address is already larger than the freed pointer.

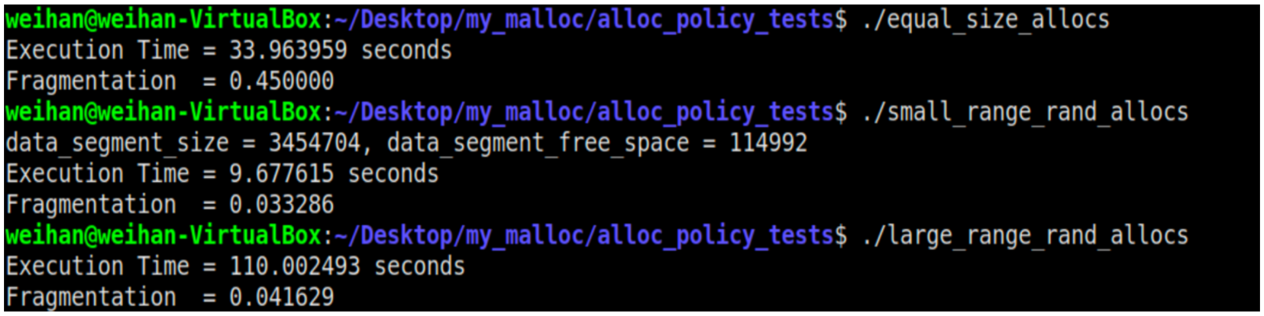
2 Results

The following screenshots are the results of different tests:



```
weihan@weihan-VirtualBox:~/Desktop/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 33.806311 seconds
Fragmentation  = 0.450000
weihan@weihan-VirtualBox:~/Desktop/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3653344, data_segment_free_space = 313632
Execution Time = 30.282599 seconds
Fragmentation  = 0.085848
weihan@weihan-VirtualBox:~/Desktop/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 91.450124 seconds
Fragmentation  = 0.093359
```

Figure 1. First Fit Performance



```
weihan@weihan-VirtualBox:~/Desktop/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 33.963959 seconds
Fragmentation  = 0.450000
weihan@weihan-VirtualBox:~/Desktop/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3454704, data_segment_free_space = 114992
Execution Time = 9.677615 seconds
Fragmentation  = 0.033286
weihan@weihan-VirtualBox:~/Desktop/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 110.002493 seconds
Fragmentation  = 0.041629
```

Figure 2. Best Fit Performance

3 Analysis of Results

The following table is for comparative analysis:

Policy	First Fit		Best Fit	
Performance	Execution Time(s)	Fragmentation	Execution Time(s)	Fragmentation
equal_size_allocs	33.806311	0.450000	33.963959	0.450000
small_range_rand_allocs	30.282599	0.085848	9.677615	0.033286
large_range_rand_allocs	91.450124	0.093359	110.002493	0.041629

3.1 Compare the performance of the two policies

3.1.1 In equal_size_allocs

This test will malloc the same size blocks which are all 128 bytes.
Since I optimize the Best-fit function, in this case, it will find that the first freed block is already the smallest one and return directly rather than iterating through the entire linked list. So, the execution time of the two policies will be roughly the same. In terms of the fragmentation, since the two functions measuring the data segment size is called in halfway and all the blocks are in the same size, the fragmentations will all close to 0.5 normally and the result presented above also shows that.

3.1.2 In small_range_rand_allocs

This program works with allocations of random size, ranging from 128 - 512 bytes.
The performance shows that the Best-fit has better execution runtime and fragmentation. Regarding runtime, the First-fit policy logically is the best because it will allocate region and return when it finds the first available block. However, the result shows different in small_range_rand_allocs case. By analyzing the testcase code, I purpose that the Best-fit policy sometimes uses more system calls that need to ask for the control of OS, so it will take a longer time to execute. For fragmentation, the best fit policy is better obviously. Because when we traverse the linked list and split the original freed memory, the Best-fit policy would make sure that the remaining part is in the minimum size. It will make use of as most free space as possible. So, when we calculate the fragmentation, the Best-fit always gives us a better result.

3.1.3 In large_range_rand_allocs

This program works with allocations of random size, ranging from 32 - 64K bytes.
The First-fit policy has better execution runtime because it will return when it finds an

available block to allocate the requested memory rather than traversing through the entire linked list. The reason for the Best-fit policy has the better fragmentation is the same as the reason in small_range_rand_allocs.

3.2 Compare the performance of different range size

Small_range_rand_allocs is the fastest one and large_range_rand_allocs is the slowest one. In the case of the same number of items (NUM_ITEMS= 10000), the difference between block size in large_range_rand_allocs will be larger, indicating that it is less likely to find the block which can fit for the requested memory size. So, the large range one will take longer.

4 Conclusion

For the question which policy seems effective, I would say it depends. If we need fast speed, then we will be better to choose the First-fit policy. But if we want to make use of the memory efficiently, we need to choose the Best-fit policy instead. Also, the requested memory size range needs to be considered as well. The First-fit policy performs better than Best-fit policy when the range size is relatively large.