Assignment 1 Report

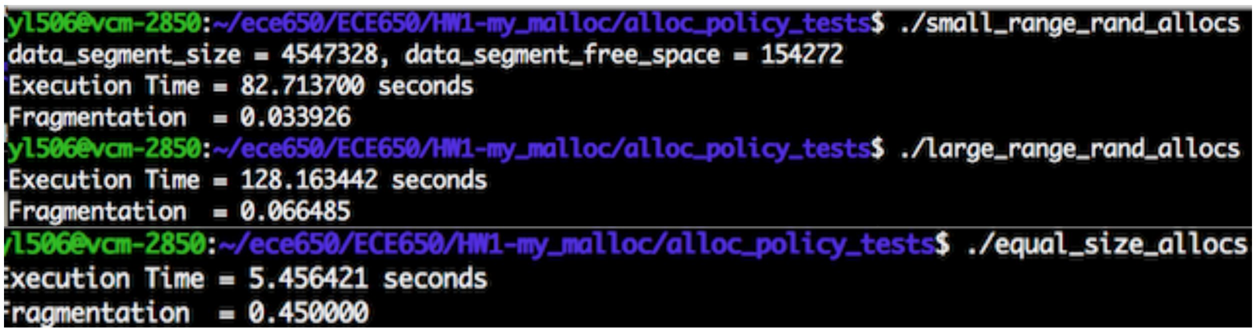
My implementation of malloc library is pretty straightforward and succinct. The implementation makes use of the sbrk system call to manage and manipulate heap and uses a doubly linked list to store and track the meta-information of the allocated memory blocks. Meanwhile, the policy to minimize the size of the process's data segment is used. Both first-fit and best-fit methods are implemented and the corresponding performance is measured.

The meta-information structure consists of 4 fields: the size of the block, the pointer to its next block and previous block, and the flag indicating if the block is free. Each time the user requests X bytes of memory, we actually use sbrk to request X + the size of meta-information structure. Then we update all the fields inside the structure. We also have a global pointer which is initialized to NULL to indicate if the user calls malloc for the very first time or not. We make use of these fields to form a linked list that tracks the memory blocks. Memory allocation/free operations then become insertion/deletion of a node in the linked list.

Specifically, the first-fit(FF) policy examines the linked list and compare the requested block size with each available free block currently in the list. A for-loop is used to iterate through the linked list, and as long as any available free block is found and its size is larger than the requested size, it is immediately selected as the proper space for the new allocated block. If such block is found and its size is way larger than the requested size plus the size of meta-information, we split the original block into two parts to minimize the size of the entire data segment. If no such block is found, we create a new block and append it to the tail of the list.

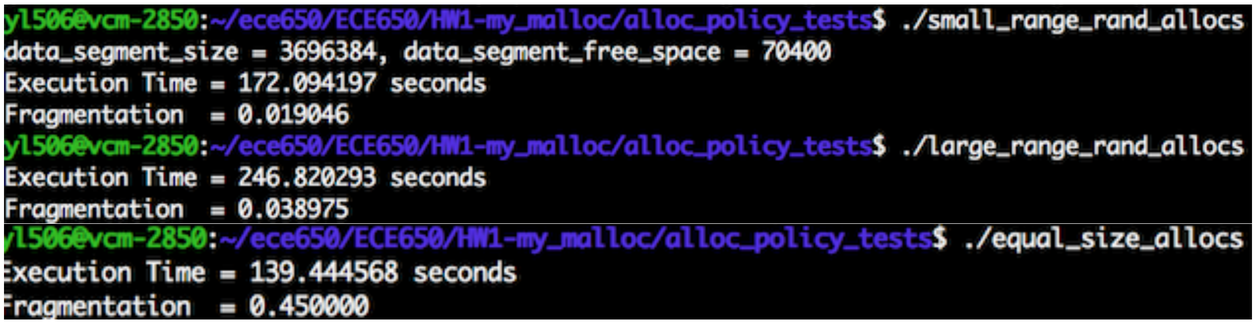
The best-fit(BF) policy has the similar effects as the FF policy but differs from the way to determine the replacement block. BF policy examines all of the free space information and allocate an address from the free region which has the smallest number of bytes greater than or equal to the requested allocation size. A for-loop is used to iterate through the list, and as long as any given block is free and is sufficient to fit the requested block, the size difference and the address of the block is recorded. These recorded information is updated as long as a better block is found (by better I mean the difference is smaller). After we iterate through the list entirely, we could now decide where to put the block.

The performance of the two policies with different allocation patterns was tested and measured. The results are shown below:



```
yl506@vcm-2850:~/ece650/ECE650/HW1-my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 4547328, data_segment_free_space = 154272
Execution Time = 82.713700 seconds
Fragmentation  = 0.033926
yl506@vcm-2850:~/ece650/ECE650/HW1-my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 128.163442 seconds
Fragmentation  = 0.066485
l506@vcm-2850:~/ece650/ECE650/HW1-my_malloc/alloc_policy_tests$ ./equal_size_allocs
xecution Time = 5.456421 seconds
ragmentation  = 0.450000
```

FF performance



```
yl506@vcm-2850:~/ece650/ECE650/HW1-my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3696384, data_segment_free_space = 70400
Execution Time = 172.094197 seconds
Fragmentation  = 0.019046
yl506@vcm-2850:~/ece650/ECE650/HW1-my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 246.820293 seconds
Fragmentation  = 0.038975
yl506@vcm-2850:~/ece650/ECE650/HW1-my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 139.444568 seconds
Fragmentation  = 0.450000
```

BF performance

Based on the measurement results, I drew the conclusion that, in general, FF policy has a faster run time than BF policy, but BF policy has a better memory usage efficiency (lower fragmentation ratio). The conclusion is in agreement with the algorithm and definition of FF policy and BF policy. The first two allocation patterns are quite generic and arbitrary. In these two cases, FF policy exits the for-loop as long as a proper block is found no matter if its size is actually way larger than the requested size. This logic guarantees fast runtime but lowers memory usage efficiency. By contrast, BF policy has the exactly opposite effect which sacrifices runtime but provides high memory layout. The equal_size_allocs pattern allocates and frees memory blocks that have a fixed size, which is exactly the nightmare for BF policy. So we can see that FF policy has much better performance than BF in this test case.

So, in terms of which policy is better, my answer is “it depends”. If one wants better memory usage efficiency and does not care too much about runtime, BF policy is generally better. But if he wants faster runtime, FF policy is better. The specific memory allocation request pattern determines which policy is better. If a program requests memory blocks that varies a lot in terms of size, BF policy excels under this condition. FF policy performs better if a program asks for memory blocks with fixed size.

The runtime of my program is not actually optimized. I am thinking that one way to get a faster runtime is to have another linked list that tracks only the free memory blocks. Since when the size of the linked list is large, the iteration takes more and more time. Currently, I have one list that links all the blocks no matter if it is free or not. Then, to calculate the free space of the data segment, the entire list must be iterated while not all the nodes are useful in this case.