ECE 650 Malloc Library Report

In this assignment, malloc () and free () functions are implemented in C with the sbrk() Linux system call. The key idea in this project is using LinkedList data structure to keep a record of the allocated memory in the heap. The node of the LinkedList is actually meta-data that will be attached with the allocated memory. The node will tell us if the allocated memory is freed, where to find the adjacent node and where to find other free nodes.

Implementation

(a) Node in Linked List

The node of the Linked list will hold 6 fields:

1. The pointer to the previous node
2. The pointer to the next node
3. The pointer to the previous free node
4. The pointer to the next free node
5. A size_t number to tell the size of the allocated memo
6. An integer to tell if the allocated memory is free or not.

There are many reasons why I choose these fields.

Firstly, the first two pointers will be used when we free a node. When free a node, we have to check whether the previous node and the next node is freed and if they are we have to merge them. By having a prev and next pointer, we can check in O(1) time.

Secondly, the last two pointers free_next and free_prev are used to keep a list of freed nodes. Because when implementing malloc, we have to find whether there is a node in the heap that can satisfy the user request. The search time can be very long if there are many used nodes. By having a Linked List which only contains freed nodes, we can reduce the time to match nodes.

The last two numbers is necessary for computing and serve as a flag.

(b) Malloc (ff_malloc/bf_malloc)

The high-level idea of malloc is:

1. Search in the freed Linked List to find fit using first fit of best fit algorithm
   a) if there is a fit
      Try to split the node
      i. If the node is too small to split (we can not record the remaining space)
         We do not split, just remove free the freed Linked List
      ii. Else
         Split the node. Replace the searched node with the splited node in the freed Linked List.

   b) If there is not a fit
      Using sbrk() to create space for node and user required space.
      Add the nodes to the freed list and list.

2. Return the address to the user.

(c) Free

1. Get the node using the address and pointer arithmetic
2. Find the previous node next node in the Linked List
3. Check whether there is a freed node in prev and next
   a) If both are not free
      Add the node to the tail of free Linked List
      Return.
   b) If the next is free
      Merge the current node with the next and replace the next with current node in the freed Linked List. Remove the next node from Linked List
   c) If the prev is free
      Merge the current node with the prev node in the freed Linked List
      Remove the current node from Linked List.

Performance Result

The screen shot of the running result is attached in the Appendix.
The performance result is shown below:

First Fit Malloc and free

	Execution Time	Fragmentation	Heap Size	Free Space
Equal size alloc	3.05904	0.600000		
Small range alloc	5.544372	0.171534	3930560	674224
Large range alloc	15.792278	0.115142		

Best Fit Malloc and free

	Execution Time	Fragmentation	Heap Size	Free Space
Equal size alloc	3.279091	0.600000		
Small range alloc	3.649656	0.148036	3795408	561856
Large range alloc	44.071483	0.042116		

Firstly, we can compare the execution time of first fit algorithm and best fit algorithm.
(1) First fit vs best fit under equal size alloc
For equal size alloc, we allocate the memo sequentially and free the memo in the same order. Therefore, the node allocation and memo allocation should behave the same between first fit algorithm and best fit algorithm.
Also, because we allocate first and deallocate, this means when malloc, the free Linked List is always NULL. Therefore, in first fit and best fit, we do not go through the free Linked List.

Theoretically, the pelyrformance of first fit and best fit algorithm should behave the same under the equal size allocation pattern. And the experiment actually proves my speculation. The execution time just differ by 0.2 and the fragmentation is actually the same.

(2) First fit vs best fit under small size alloc

Intuitively, first fit malloc should be faster than the best fit malloc. However, from the experiment, we can see that best fit takes 3.6 seconds to finish and first fit takes 5.5 seconds to fit.

But after comparing the heap size, we can see part of the reasons. First fit malloc will produce 3930560 bytes heap, while best fit algorithm will produce 3795408 bytes heap. We can see that the heap size of best fit is much smaller than first fit. Because first fit and best are tested under the same data, we know that best fit malloc produces less nodes compare to first fit malloc. Also, because best fit algorithm find the most fit free nodes and these free nodes maybe too small to split or fit exactly. This will reduce the number of nodes inside of the Free Linked List. This means best fit malloc will search through a shorted linked list compare with first fit malloc. This may be part of reason why best fit runs shorter under this case.

(3) First fit vs best fit under large size alloc

In this experiment, best fit malloc takes much longer time to run compared with the first fit malloc. This is normally because best fit malloc has to go through all the free nodes to decide which one is the most fit.

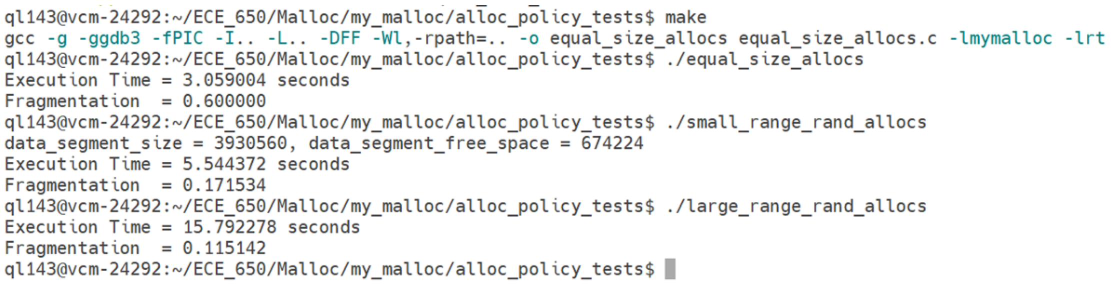
As for the fragmentation, best fit is about 0.042 and first fit is about 0.115. Best fit malloc performs better than first fit malloc. This is because best fit selects the most fit free node and reduce the times of splitting free nodes.

My Recommendation

I would recommend first fit malloc for the following reasons.

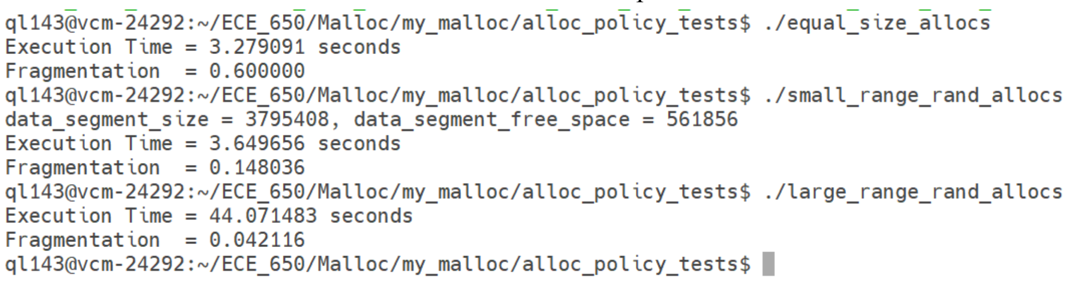
(1) Computer in nowadays has a quite large RAM for us to use. It is acceptable use space to improve performance.
(2) Best fit malloc runs very slowly under large chunk circumstance. This disadvantage shadows its storage performance.

Appendix



```
ql143@vcm-24292:~/ECE_650/Malloc/my_malloc/alloc_policy_tests$ make
gcc -g -ggdb3 -fPIC -I.. -L.. -DFF -Wl,-rpath=.. -o equal_size_allocs equal_size_allocs.c -lmymalloc -lrt
ql143@vcm-24292:~/ECE_650/Malloc/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 3.059004 seconds
Fragmentation  = 0.600000
ql143@vcm-24292:~/ECE_650/Malloc/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3930560, data_segment_free_space = 674224
Execution Time = 5.544372 seconds
Fragmentation  = 0.171534
ql143@vcm-24292:~/ECE_650/Malloc/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 15.792278 seconds
Fragmentation  = 0.115142
ql143@vcm-24292:~/ECE_650/Malloc/my_malloc/alloc_policy_tests$
```

Picture 1. First Fit Malloc performance



```
ql143@vcm-24292:~/ECE_650/Malloc/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 3.279091 seconds
Fragmentation  = 0.600000
ql143@vcm-24292:~/ECE_650/Malloc/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3795408, data_segment_free_space = 561856
Execution Time = 3.649656 seconds
Fragmentation  = 0.148036
ql143@vcm-24292:~/ECE_650/Malloc/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 44.071483 seconds
Fragmentation  = 0.042116
ql143@vcm-24292:~/ECE_650/Malloc/my_malloc/alloc_policy_tests$
```

Picture 2. Best Fit Malloc performance