Report for Malloc Library

- Student name: Xueyang Liu
- Student Netid: xl350

1. Design & Implementation

1.1 Data Structure

At the very beginning, I was confused about this project and had no idea where to start. After I fully understood the `sbrk()` function, including its purpose, its return value and what it did, the project was rather clear to me. What I needed to do was to implement a data structure, with which the memory space I got by calling `sbrk()` could be effectively used by the user. I designed a doubly linked list to do this, as shown below.

```
struct block_tag{
    size_t size;
    struct block_tag *next;
    struct block_tag *prev;
};
typedef struct block_tag block_t;
```

This data structure served as tags bonded to memory regions acquired from calling `sbrk()` . Inspired by the TA and the task sheet, I only kept free memory regions in my list. Since once the users want to free the allocated memory, they will provide the pointer and it would be useless and inefficient if I kept them in the list.

1.2 My Malloc

The project task sheet asked me to implement two kinds of malloc: First-fit and Best-fit. Firstly, let me illustrate the common feature of those two mallocs. They both needed to go to the linked list first and search for a suitable free-mem region to allocate. If they could not find one, `sbrk()` would be called to generate a new mem region. The difference between them is that First-fit will use the first suitable free-mem region and the Best-fit will go throught the whole linked list and use the smallest among all suitable free-mem regions. By suitable, I mean either the free-mem region size just equals to the `required_size` or it is larger than `sizeof(block_t) + required_size` . When it equals, the free-mem region will simply be removed from the free-list and returned for use. And when it is large enough, it will be split into two free-mem region, one remains in the free-list and the other returned for use.

1.3 My Free

When the user calls `free()` , the free-mem region needed to be inserted into the free-list. Note that to allow `merge` , the free-mem regions were designed to be address-ascendingly-sorted in the free-list. My implementation first traveres through the free-list and finds the correct place to insert, then insert the free-mem region and check if it can be merged with its previous one or next one.

1.4 Merge

Every time a free-mem region is inserted into the free-list, it will be checked if it can be merged with its previous free-mem region or its next one. The merge operation is basically the opposite of split mentioned above in `My Malloc` . One thing worth mentioned is that the newly inserted free-mem region should be better check its next region first and then its previous one to avoid some kind of pointer mess.

2. Functionality Tests

Upon finishing writing the code, I used the provided `general_tests` to test my code. I also added some printf statements to print the expected free-mem region list (with `merge` and `address-ascending sort` applied) and implemented a `printList()` function to print the status of the free-mem region list.

After fixing some bugs, the code run as expected and was valgrind-clean. Part of all tests and outputs are shown below.

Tests:

```
printf("expected empty list\n");
printList();

FREE(array[0]);
printf("expected list with 1 node with size 16\n");
printList();

FREE(array[2]);
printf("expected list with 2 node with size 16 and 32\n");
printList();

FREE(array[5]);
printf("expected list with 3 node with size 16 and 32 and 1024\n");
printList();

FREE(array[1]);
```

```
  printf("expected list with 2 node with size 160 and 1024\n");
  printList();
```

Outputs:

```
expected empty list
expected list with 1 node with size 16
curr: 0x562b0977d000, allocated size: 16, allocated address: 0x562b0977d018
expected list with 2 node with size 16 and 32
curr: 0x562b0977d000, allocated size: 16, allocated address: 0x562b0977d018
curr: 0x562b0977d080, allocated size: 32, allocated address: 0x562b0977d098
expected list with 3 node with size 16 and 32 and 1024
curr: 0x562b0977d000, allocated size: 16, allocated address: 0x562b0977d018
curr: 0x562b0977d080, allocated size: 32, allocated address: 0x562b0977d098
curr: 0x562b0977d184, allocated size: 1024, allocated address: 0x562b0977d19c
expected list with 2 node with size 160 and 1024
curr: 0x562b0977d000, allocated size: 160, allocated address: 0x562b0977d018
curr: 0x562b0977d184, allocated size: 1024, allocated address: 0x562b0977d19c
```

3. Efficiency Analysis

3.1 Test Results

Pattern	First-Fit Execution Time	First-Fit Fragmentation	Best-Fit Execution Time	Best-Fit Fragmentation
Equal	17.93s	0.45	17.81s	0.45
Small	16.96s	0.09	11.80s	0.04
Large	44.05s	0.09	64.64s	0.04

The provided tests were run several times, each `combination of the malloc policy and the tested size` resulting in a slightly `different` execution time but `same` fragmentation.

3.2 Analysis

3.2.1 Equal_size_allocs

For `equal_size_allocs`, the allocated mem-region is all 128B and all free-mem region in the free-list will be the same size (before merging with each other). In this situation, even with merging policy applied, FF and BF algorithms will do exactly same thing since they both use the head of the free-list every time, except for when the free-list is empty. In that situation, they also do the same thing -- call `sbrk()` to get a new mem-region. So their execution time and fragmentation resemble.

3.2.2 Small_range_rand_allocs

For `small_range_rand_allocs` , the test basically does the following: Malloc a bunch of similar sized mem-region, free them and then malloc again. In this procedure, if using FF, it will `split` or `merge` or `sbrk()` a lot, while using BF, it is easier to find a suitable one. So that BF is faster than FF in this test.

3.2.3 Large_range_rand_allocs

For `large_range_rand_allocs` , the test does similar thing compared with `small_range_rand_allocs` , only that the malloced mem-region sizes vary much more greatly. And in this case, the free-list will be longer and traversing through it will consume a lot of time, which is the reason that I beleive to cause BF slower than FF.

3.2.4 Conclusion

In our daily usage of malloc, it is more likely that we don't always malloc mem-regions with similar sizes. So according to the test result, using FF under this circumstance will be faster than BF. Of course, if we are informed in advance that the malloced mem-region will have similar sizes, BF will be a better choice.