Project Report #1: Malloc Library

Name: Xushan Qing NetID: xq47

1. Overview of Implementation

1.1 Goal

For this assignment, the goal is to implement two malloc-related library functions, *malloc()* and *free()* in C code using a system call, *sbrk()*. We need to use two different allocation policies including First Fit and Best Fit to determine the memory region to allocate. Besides, we also need to conduct a performance study in time and fragmentation, which needs two functions, *get_data_segment_size()* and *get_data_segment_free_space_size()*.

1.2 Data Structure

To implement malloc and free, I need a data structure to record each allocated memory block including address, size and free status. The easiest way is to use a linked list, which collect all the allocated blocks, whose order is according to their virtual address. However, since we only need to merge or split the free block, we can simple the design by only connect free blocks. In this way, we save the space and save the time to look up in the list. This is how the data structure looks like:

```
typedef struct block_header{
   size_t size;
   struct block_header * prev;
   struct block_header * next;
} block_h;
```

Every block header located at the head of each block and contains information of size and its neighbor free blocks.

1.3 Function Implementation

For First Fit strategy, when we call malloc() to initialize memory space for an variable, we first call *find_first_free_block(size)* to find the first free block which size larger than *size + sizeof(block_h)*. If we cannot find it, we need to allocate new memory space for the new variable by calling function *void * allocate_new_block(size_t size)*. This function can help us call sbrk() to allocate memory and initialize a block header object to record information of this block. If we can find an adequate free block, if its size is larger than size + *sizeof(block_h)*, this block will be split to reuse free space. We will call function *void *split(block_h * cur,size_t size)*, to split the block into two nonfree and free blocks. The nonfree block will be deleted from the free linked list(*void delete_free_list(block_h * block)*), while the free block will be added in the free list, which at the same time initialize a new header for this free block(*void add_free_list(block_h * block)*). If the

adequate free block size is smaller, we do not need to split. The malloc function returns a pointer that point to the specific address where the variable is.
When we call free(), we first find the block header address by using the header size offset. Then, we add the new block to the free list (*void add_free_list(block_h * block)*) and try to merge this new free block with its neighbors(*void merge_free_block(void *ptr)*).

For Best Fit strategy, the only difference is when we call malloc(), instead of using *find_first_free_block(size)* to find the first free block, we use *block_h *find_best_free_block(size_t size)* to iterate the free list to find the smallest free block that can fit in the malloc size. Other part is all the same with First Fit strategy.

1.4 Problems and Improvement of Implementation

There are several problems I encountered during the implementation.
First, the linked list problem I told before. I tried to connect all the free and none-free block in the list, however, it did not pass the time requirement in gradescope. To make my algorithm more efficiently, I change the linked list to only maintain the free block.
Second, the uninitialized value error. I must initialize all the previous and next block to be NULL at the beginning, when I call malloc new block and initialize a new header.
Third, the merge problem. The logic of merge and reconnect neighbor blocks is a little complicated. So when problems happened, I use *printfree()* function to print the free linked list each time I malloc or free something to see what the exact address and size of block. I found problems usually happens with merge connection.

2. Results

First-fit strategy:

```
vcm@vcm-24033:~/ece650proj1/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3705640, data_segment_free_space = 273784
Execution Time = 14.129284 seconds
Fragmentation  = 0.073883
vcm@vcm-24033:~/ece650proj1/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 53.999595 seconds
Fragmentation  = 0.093421
vcm@vcm-24033:~/ece650proj1/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 30.380917 seconds
Fragmentation  = 0.450000
```

Best-fit strategy:

```
vcm@vcm-24033:~/ece650proj1/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3529960, data_segment_free_space = 95160
Execution Time = 5.134914 seconds
Fragmentation  = 0.026958
vcm@vcm-24033:~/ece650proj1/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 62.329109 seconds
Fragmentation  = 0.041318
vcm@vcm-24033:~/ece650proj1/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 30.587620 seconds
Fragmentation  = 0.450000
```

	First-fit execution time	First-fit fragmentation	Best-fit execution time	Best-fit fragmentation
Small	14.13s	0.07	5.13s	0.02
Large	54.00s	0.09	62.33s	0.04
Equal	30.38s	0.45	30.59s	0.45

3. Analysis

For *small_range_rand_allocs* test, the malloc size and order is random. The results show that best-fit cost less time than first-fit strategy. Although theoretically it may cost more time for best-fit to find new block, but it uses memory more efficiently. So, there is a situation that best-fit use existing free memory block but first-fit use up those free large block before, cannot find adequate existing block and create new block which cause more time. For example, if we try to allocate variable with memory size 3 4 5 to free block list with memory size 5 4 3.

For *large_range_rand_allocs* test, the malloc size and order is still random like the small range test, but the range is much larger, have larger amount of data to allocate. The results show that First-fit is faster than Best-fit though best-fit have smaller fragmentation. First-fit strategy just find the first block and allocate it. Instead, to save space, Best-fit strategy cost more time.

For *equal_size_allocs* test, all the malloc size is the same. So, the result of first-fit and best-fit is the same, both for the execution time and fragmentation. Indeed, the implementation of these two methods are still the same, cause if we can find a block which is large and free to allocate, it is also the best-fit block we can allocate.

Like in most cases, space complexity is opposite to time complexity. In real world, I would recommend First-fit strategy, cause in most cases, time matters more.