ECE 650 Project 1: Malloc Library Part1 Report

By Junqi Sun js895

1. Design of my malloc and free

The main function of this project could be divided into several parts:

1. First Fit search function
2. Best Fit search function
3. New space allocation function
4. Free space merge function
5. Split free space function
6. ff_malloc and bf_malloc
7. ff_free and bt_free

In order to fulfill these requirements, I design each function that correspond to each of the requirement. It seems that using linked list to represent the contiguous memory distribution of heap will be wise. Also, I create a struct type “myblock” as metadata of each space sector we operate on as well as the pointer type block_ptr in my_malloc.h. The struct contains four elements: the size of each allocation space (except for metadata’s space); the empty symbol of each allocated space indicated if it is freed (1 is for free, 0 is for allocated); the pointer points to next metadata block, as well as the pointer points to last metadata block.

My model of heap structure:

Figure 1.1 Heap model

The blue parts re the blocks of linked list, white space are data space I referred as sector.

I define the BLOCK_SIZE, head and tail pointer as global variables. Since I have four elements in struct, so their total size will be 32 bytes (8 bytes for each in 64-bits system). Head and tail pointer are used to keep track of the whole linked list.

Block_ptr first_find(block_ptr curr, size_t size); function correspond to requirement 1, When I input pointer curr as well as required size, I will search the entire linked list from location curr to the first memory sector that “is free” , “has enough size ” and also not an unavailable space (NULL), otherwise the search pointer keeps going on.

Block_ptr best_find(block_ptr curr, size_t size); function correspond to requirement 2. When I input pointer curr as well as needed size, it will use first_find at first to acquire a comparable address and its corresponding free size. Then it will search through the entire heap, compare every free sector’s size to get a smaller one as long as they are big enough and

free. When it finds a sector. Once it finds a smaller one, it will renew the space size for comparing and the track pointer for return. If it finds nothing, just return NULL.

Block_ptr coalesce (block_ptr merge); function correspond to requirement 3. It can merge the sector in front of the current sector or the next block. The mechanisms are almost the same. For merging the next block, first it will judge if the current block is empty, if it is the last block in linked list as well as if the next sector is freed. Then it will expand its size by adding an additional BLOCK_SIZE, the size of next free sector. Also it will judge if the next elements is tail, if it is then we will refresh the tail pointer. If it is not, just renew the next and last pointer of the two block

Void split(block_ptr use, size_t size); function is for requirement 4. Its inputs are current block address and the size that malloc apply for. At first it will generate a new address for the divided new part's metadata. Then it will renew the size of sector that new pointed to as well as symbol variable empty. After that it will judge if we are at the end of linked list. If it's not, just renew the next, last pointer of two divided parts. If it is, there is no "next" address for the new metablock and the tail pointer should be adjusted either, so it will be NULL.

For different types of malloc or free, they just differ in find function (first or best), all other setting are the same. If we call malloc, we have to check if it is an empty linked list. If it is, we need to allocate the first block, sector and initialize the head, tail pointer. Otherwise, it will find the first or best space for malloc. If we find it, we will see if we could split it, or we will allocate a new space at the end of linked list. Finaly we will return the address of this block.

For Free function, it will judge if it is not a NULL and use the address to the input of merge function. Since the input would be the address of data sector, It need to minus the BLOCK_SIZE to get the correct meta data's location.

2. Experiment result

First_find:

```
js895@vcm-12427:~/ece650prj1/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3725600, data_segment_free_space = 175936
Execution Time = 92.734690 seconds
Fragmentation  = 0.047224
```

Figure 2.1 ff small size test

```
js895@vcm-12427:~/ece650prj1/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 230.216924 seconds
Fragmentation  = 0.026806
```

Figure 2.2 ff large size test

```
js895@vcm-12427:~/ece650prj1/my_malloc/alloc_policy_tests$ ./equal_size_allocs

Execution Time = 439.027587 seconds
Fragmentation  = 0.450000
```

Figure 2.3 ff equal size test

Best_find:

```
js895@vcm-12427:~/ece650prj1/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3616832, data_segment_free_space = 80288
Execution Time = 142.805082 seconds
Fragmentation  = 0.022198
```

Figure 2.4 bf small size test

```
js895@vcm-12427:~/ece650prj1/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 293.797303 seconds
Fragmentation  = 0.019611
```

Figure 2.5 bf large size test

```
js895@vcm-12427:~/ece650prj1/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 456.306084 seconds
Fragmentation  = 0.450000
```

Figure 2.6 bf equal size test

3. Analysis of result

According to my design, since first find function will use the first big enough space sector to allocate, while best find will find the sector whose space is the closest to what we need (even equal size). So the running time of First find will be shorter than Best find methods. Based on data I collect in these picture, the result perfectly match my design.

Utilization of space: the total data segment space for two methods are not vary too much, almost the same, while the data segment free space of first find methods is much larger. This is because when first find method would like to malloc a space with size1, it will use the first find method that find the first free space (or allocate)with size2 that bigger than size1. It is possible that when we need to malloc a new place with exact size2, it will not fit the the block that size1 has already used, than it need to allocate new block or find the next available one. For best find method, there is no such problem.

Processing time: As I analyze at the beginning, Since best find method need to find the most fit sector, it will definitely cost more time on comparing and searching.