ECE 650 Project #1 Report

Malloc Library Part1

Name: Yingxu Wang

Net ID: yw473

Date: Jan. 19th, 2022

I. Introduction

In this project, I am going to implement functions to malloc memory and free memory. There are two ways to implement malloc function. The first approach is to find the first place that is free and is large enough. Then we can assign this memory space. The second approach is to find the best place, where the space is the smallest, but is larger than or equal to required memory. After that, we assign that memory space. The free method is to free the memory space.

Therefore, I need to implement 4 functions:
1) void *ff_malloc(size_t size);
2) void ff_free(void *ptr);
3) void *bf_malloc(size_t size);
4) void bf_free(void *ptr);

Function 1) and 2) are for the first fit method. 3) and 4) are for the best fit method. I will introduce how I implement them in the next section.

II. Implementation and Methods

I define 1 struct, 3 variables and 15 functions in total, in order to implement and test this project. This is the overview of my functions defined in my_malloc.h.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

typedef struct memory {
    size_t size;
    struct memory * prev;
    struct memory * next;
} memory_t;

size_t memory_size = sizeof(memory_t);
unsigned long total_memory = 0;
memory_t *first = NULL;
void *ff_malloc(size_t size);
void ff_free(void *ptr);
void *bf_malloc(size_t size);
void bf_free(void *ptr);
void *find_first_from_List(size_t req_size);
void *find_best_from_List(size_t req_size);
void *split_into_two(memory_t *block, size_t req_size);
void *remove_from_list(memory_t *block, size_t req_size);
void insert_into_freed_list(memory_t *prevOne, memory_t *cur, memory_t *ptr_begin);
void merge_all_list(memory_t *block);
void *create_mem(size_t size);
void free_memory(void *ptr);
unsigned long get_data_segment_size();
unsigned long get_data_segment_free_space_size();
void print_list();
```

Main Idea of malloc and free:

Main Idea to implement First Fit Malloc:

I keep a doubly linked list to store all the memory space that has been freed. When I'm going to malloc a certain size of memory. I will first search in this doubly linked list from the beginning to the end. If I can find a space that meets the requirement (the memory size of it is larger or equal to required memory size), I will assign that free space to malloc. There are two situations here: 1. If that space is exactly the same as required memory or that space is between required space + metadata memory size, I will just remove that space from the list. 2. If not, and if that memory space can be split into 2 parts, I will allocate required space and keep the remaining in the list.

If there's no memory space in the freed list that meet the requirement, I will use sbrk() to malloc new actual memory space.

Main Idea to implement Best Fit Malloc:

I keep a doubly linked list to store all the memory space that has been freed. When I'm going to malloc a certain size of memory. I will first search in this doubly linked list from the beginning to the end. If I can find a space that meets the requirement (the memory size of it is larger or equal to required memory size), different from previous one, I will keep looking for memory space that is larger than required memory space but is the smallest. If I find that place, there are two situations here: If that space is exactly the same as required memory. In this situation, I will just remove that space from the list. If not, and if that memory space can be split into 2 parts, I will continue to find in the freed list to see if there exists a smaller one.

If there's no memory space in the freed list that meet the requirement, I will use sbrk() to malloc new actual memory space.

Main Idea to implement Free (Both First Fit and Best Fit):

Since the free operation is to free a memory space by its starting pointer, there's no difference between first fit and best fit. I can just free a certain memory space by the sum of the metadata and its memory size. To actually do "free" operation. I'm going to use a doubly linked list to keep track of the freed space because I'm going to reuse these memory spaces if next time I want to malloc memory space. Each node of this doubly linked list contains the information of size, previous node, and next node. It is ordered by the address of its memory space. So, each time I free a memory, I should find an appropriate place to insert. (starting address of previous node < starting address of insert node < starting address next node). If there exist adjacent nodes, I will merge them together. Keeping the doubly linked list in order can help me know if there are two adjacent addresses or not. If so, we can merge address together.

Function Implementation Explanation:

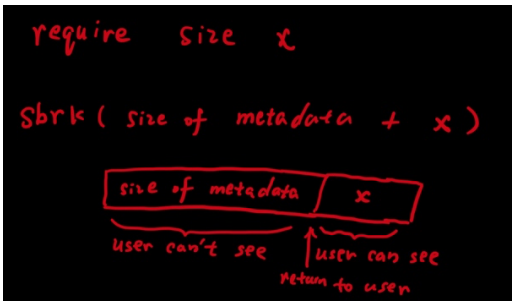
void *ff_malloc(size_t size)

This function will firstly look for appropriate memory size from the freed list by calling:

```
oid *find_first_from_List(size_t req_size);
```

. If I can find an appropriate memory space, I will return the pointer pointing to that memory space. If not, I will grow the size of processes data segment and return the pointer pointing to that newly created data segment. Detailed implementation of how I use sbrk() or malloc data

from freed list will be in each corresponding function description below.



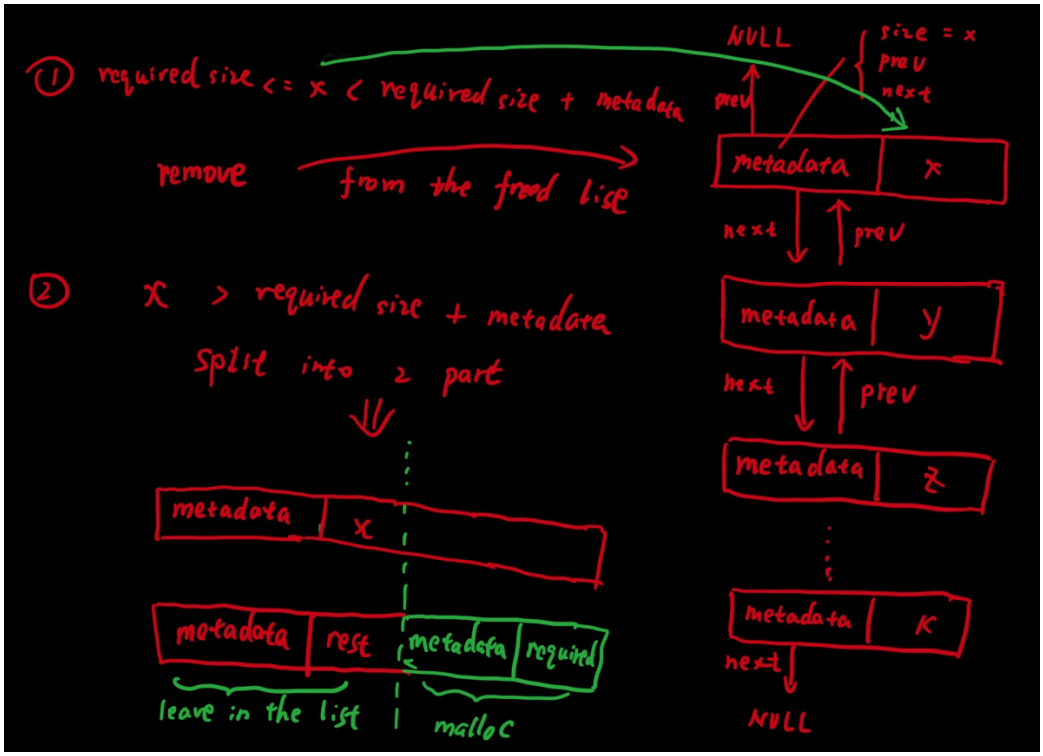
↑memory that actually sbrk()

void *bf_malloc(size_t size)

This function will firstly find best memory space in the freed list by calling:

```
void *find_best_from_List(size_t req_size);
```

If I can find one, I will return the pointer pointing to that memory space. If not, I will grow the size of data segment by using sbrk(). Detailed implementation of how I use sbrk() or malloc data from freed list will be in each corresponding function description below.



↑Process of malloc from freed list

void ff_free(void *ptr)

&

void bf_free(void *ptr)

These two functions are the same. They are to "free" a certain memory space by its start address. Here, "free" means to add it to my freed list. I keep a doubly linked list (called freed list) to keep track of each memory space that is freed. Every time I free a memory space, I insert this memory space in this list according to its start memory address. Therefore, this

memory address will be sorted. Every time after I insert a new memory space to this list, I check if it is adjacent to its previous memory space and next memory space or not. If so, I will merge these memory space.

void *find_first_from_List(size_t req_size);

This function will find first place in the freed list that meet the requirement. Here meeting the requirement either means the size of allows it to be split into 2 parts (size > metadata size + required size), or the rest part is too small to manage (required memory size<= size < metadata size + required memory size). For the first one, I will split it into 2 parts and return the pointer pointing to that required memory space, leaving the rest in the freed list. For the second one, I will directly remove that memory space from the freed list, and return the pointer pointing to that memory space.

void *find_best_from_List(size_t req_size);

This function will find best place in the freed list that meet the requirement. Here meeting the requirement is same as previous function: either the size of allows it to be split into 2 parts (size > metadata size + required size), or the rest part is too small to manage (required memory size<= size < metadata size + required memory size).

Different from previous function, I will traverse all the elements in the freed list to find a element that meets the requirement but occupy less memory space. If I find that best one, I will either split it into 2 parts, returning a pointer pointing to that memory space, or directly remove it from the freed list, according to the size of it.

void *split_into_two(memory_t *block, size_t req_size);

Calling this function means the size of target memory space in the freed list is larger than required size +metadata size. Therefore, it can be split into 2 parts. I will split by adjusting the size of this block to be (block->size – req_size – metadata size). And after that, I will return the pointer pointing to the required memory space, meaning that I malloc this part from freed memory space. Check the graph unrder bf_malloc() to understand my idea clearly.

void *remove_from_list(memory_t *block, size_t req_size);

Calling this function means the size of target memory space doesn’t allow it to be split into two parts, but the size of it is still larger than required size. Therefore, we can malloc this memory space by directly deleting from the freed list, and return the pointer pointing to that memory space.

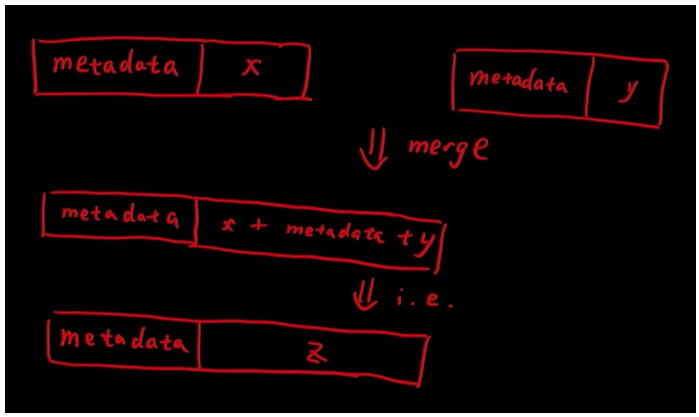
void insert_into_freed_list(memory_t *prevOne, memory_t *cur, memory_t *ptr_begin);

I call this function in the free function. When I either call ff_free() or bf_free(), it will find the correct place to insert and then call this function. Then it will insert into the corresponding place in the freed list.

void merge_all_list(memory_t *block);

I call this function after inserting into freed list. After inserting into freed list and keeping

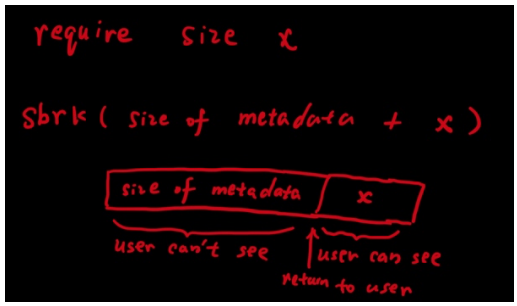
the address in order, I will firstly check if it is adjacent to its next block. If so, I will merge these two and adjust the size of the first block. After that, I will check if current block is adjacent to its previous block, if so, I will merge these two and adjust the size of the its previous block.



⬆Merge in the freed list

void *create_mem(size_t size)

This function will call sbrk() to grow the size of the processes data segment by (size of metadata + required size). Then, this function will return the new created memory address. Details are in the graph below.



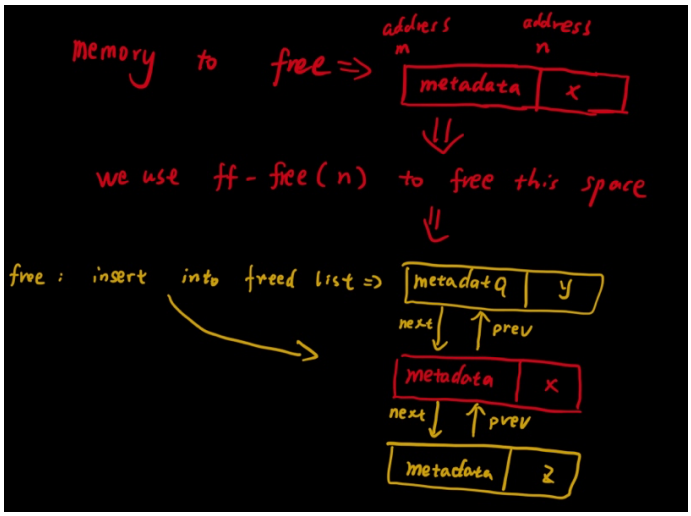
⬆memory that actually sbrk()

void *free_memory(void *ptr)

Both ff_free(void *ptr) and bf_free(void *ptr) will call this function because they are the same. In this function, I will firstly find a place in the freed list to insert to keep it in order by address. After that, I will call:

```
void insert_into_freed_list(memory_t *prevOne, memory_t *cur, memory_t
*ptr_begin);
```

to insert into the freed list. After that, I will check if it can merge with its next block. Then, I will check if it can merge with its previous block. Details are in the graph below.



↑process of free a certain memory space

unsigned long get_data_segment_size()

This function is used to calculate total memory space that has been increased by calling sbrk(). I use a variable called total_memory to record it. Whenever I use sbrk() to grow the size of processes data segment, I will add that size amount to it. This function will return this variable.

unsigned long get_data_segment_free_space_size()

This function is used to calculate total free memory space. To achieve this, I iterate from the freed list and sum up total size inside freed list. Then I can get total free space.

(The following part is performance test)

III. Performance Result and Analysis

First Fit:

```
yw473@vcm-23981:~/ece650/second/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 22.496978 seconds
Fragmentation  = 0.450000
yw473@vcm-23981:~/ece650/second/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 40.429888 seconds
Fragmentation  = 0.094559
yw473@vcm-23981:~/ece650/second/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3707312, data_segment_free_space = 275616
Execution Time = 14.360236 seconds
Fragmentation  = 0.074344
```

Best Fit:

```
yw473@vcm-23981:~/ece650/second/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 22.730878 seconds
Fragmentation  = 0.450000
yw473@vcm-23981:~/ece650/second/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 55.010493 seconds
Fragmentation  = 0.042176
yw473@vcm-23981:~/ece650/second/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3528608, data_segment_free_space = 93680
Execution Time = 5.953899 seconds
Fragmentation  = 0.026549
```

Analysis:

For large_range_rand_allocs, every time we generate random size. Therefore, we cannot observe their efficiency differences from this variable. However, we can find that the fragmentation for best fit is smaller than first fit. It means that best fit makes better use of memory. Every time when we malloc new memory from freed list, best fit approach search in the freed list for the smallest memory that meet the requirement. Therefore, there will be smaller memory garbage. It is because when we malloc a more appropriate memory, it will have less memory left. If it has less memory left, we can make the best use of it. Otherwise, we will leave a lot of space not used. That's why first fit approach has more data segment free space. As for execution time, the best fit approach execute more time because in average it has to search all the freed list to find the best fit memory space to malloc. However, first fit approach can just find the first fit place and return. That's why their execution time differs.

For equal_size_allocs, both first fit and best fit behave almost the same. It is because every time we malloc same amount of memory space and free same amount of memory space. After that, we malloc also the same amount of data. In this way, they will always find the first place in the freed list to malloc. In this way, they are almost the same and the execution time and fragmentation are almost the same.

For small_range_rand_allocs, best fit approach has better execution time than first fit approach. At the same time, best fit approach has smaller fragmentation than the first fit approach. I think the fragmentation of best fit approach is smaller than first fit is reasonable because it always finds a smaller memory space to malloc. Therefore, it has less freed memory space left. However, as for the execution time, I think in most cases best fit will have more execution time than first fit because in most cases it has to traverse all the freed list to find a best choice. However, for first fit, it can

just find a first fit place to malloc. Here, however, the result is different from what I think. I think it maybe because for first fit approach, every time it finds the first fit place and have much left. Therefore, next time when it also has to malloc same amount of data, it doesn't have that much. Therefore, it has to use sbrk() to generate new data segments. However, for best fit approach, it makes better use of freed memory space. In this way, it can have much more available space than first fit approach. Therefore, it has less execution time.