Report for ECE 650 Homework1

Dynamic Memory Allocation

Advisor: Prof. Ivan Mura

1. Requirement & Summary of development

In this section, I will briefly introduce my coding environment, the objectives, main requirements of the development and the organization of this report.

1.1 Coding Environment

- Programming language: C code
- Ubuntu 18.04.6 LTS
- gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
- GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
- GNU Emacs 25.2.2

1.2 Objectives & Main Requirements

Implementation of two versions of dynamic memory allocation functions **malloc/free function** from C standard library based on two different strategy: First fit strategy & Best Fit Strategy.

1.3 Report Organization

My report will be divided into three parts, Requirement and summary of development, the design architecture of my program, the performance and data analysis of different implements and finally the conclusion

2. Design Architecture

In this section. I will introduce my design of data structures, functions and logic of implementation, for both (First Fit & Best Fit) function. I will use diagrams and words to describe my algorithms. After sharing my design architecture, I will also share my experience of debugging.

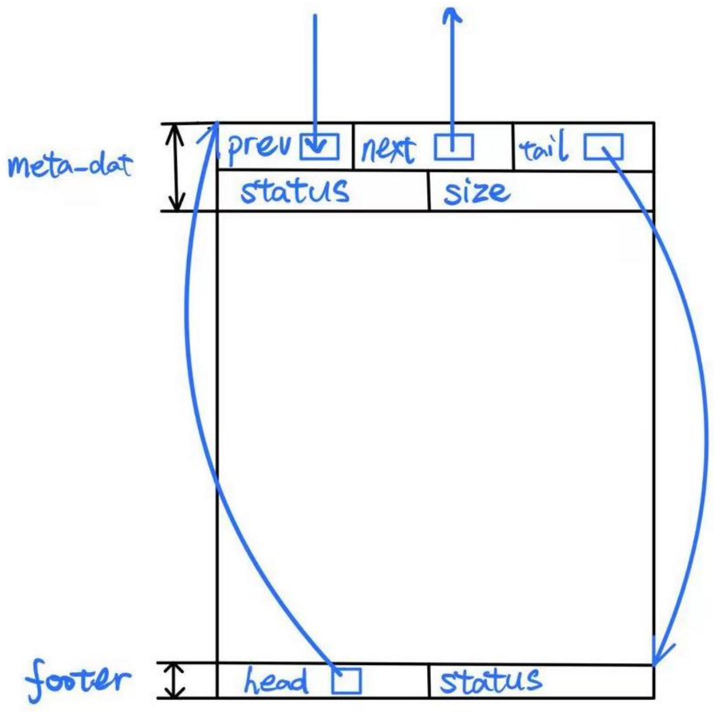
2.1 Data Structure

I use Double Linked List as my basic data structure. During the process of allocating and deallocating memory, each free data segment will be marked as a node in a linked list, and your system can get access to free information from this list. Each data segment contains three parts of data: *meta data, actual data, footer*.

- meta date: meta_data is a struct contains meta information of a data segment.

```
struct _meta_data{
   size_t status;  //the status of this node: FREE/MALLOCED/DIRTY
   size_t size;    //the total size of this block
   struct _meta_data * prev;          //a pointer to its previous node
   struct _meta_data * next;          //a pointer to its next node
   struct _footer * tail;        //a pointer to the tail of this node
};
```

- actual data: The actual user controllable data segment after memory allocation.
- Footer: a struct contains a pointer to the head of this node and a status variable indicates whether this data is clean.



2.2 Functions

2.2.1 First Fit Malloc

My first fit malloc function will first iterate through the free list to find the first free node with sufficient space to store the data user required. If fail to find a well-fitted node (reach the tail of list). My malloc function will then call sbrk() system call to extend heap and add into free list for allocation. When a node is found, ff_malloc function will allocate memory for user and update free list node information.

2.2.2 Best Fit Malloc

My best fit malloc is almost the same as first fit malloc. The only difference is the searching strategy. In my best fit malloc function, the searching process will only stop when the pointer reach the tail of free list or find a best-fitted free node (the size of free node is the same as user required). Whenever find a better node, my function will update best fitted node information.

2.2.2 Best Fit Free & First Fit Free

My best fit free and first fit free just do the same task. My free function will first shift to the meta date block of this node and check whether the status of this node is MALLOCED. If so, my free function will start deallocating memory.

My free function will first read the status of its previous block and the status of its next block to determine whether this new deallocated node need to be added to free list. If either of these two blocks is a free block, this deallocated block merge into the free block. If both previous and next are free blocks, then these three blocks will be merged together, and the next block will be deleted from the free list. Otherwise, the newly deallocated block will directly add to the free list.

2.3 Debugging Process

The debugging process was actually so twisted. In the first time of developing, I had not erased all the data inside block after free or merge. As a result, my program was easily destroyed by some dirty nodes with enormous block size and some pointers pointed to somewhere outside of heap. In order to fix these bugs, I kept checking my functionality by reading the data from memory address, printing out free & malloced list and designing test cases with different block size. Finally, I succeeded!

3. Performance & Result Analysis

In this section, I will make an analysis of the time complexity and space complexity of my ff_malloc, bf_malloc and free function. I will also use a chart to show the performance of these two implements in different situations (Large size/equal size/small size malloc) and finally make a summary.

3.1 Time Complexity

- First fit malloc: my first fit malloc function will check the whole free list node until find a suitable place. Thus the time complexity of my ff_malloc function will be O($N_{\text{number of free node}}$), but the average time is actually much more less than the number of free node.
- Last fit malloc: the time complexity of my best fit function will also be O($N_{\text{number of free node}}$), but the constant part is larger than first fit in most of time than first fit malloc.
- Free: my free function only need to check the status of its previous and next block and merge in different situation. Thus my free function has O(1) time complexity.

3.2 Space Complexity

In my program, the program need to maintain a free double linked list, thus the space complexity is O($N_{\text{free node num}}$). But it is actually less than N while the merging process happened when user deallocating memory.

3.3Performance Analysis

I run my first fit malloc and best fit malloc in different data range(small/equal/large) and finally get the following result.

- In small data range test case, bf_malloc performs much more better than ff_malloc. The execution time of bf_malloc is shorter, fragmentation is about 17 percent lower than ff_malloc.
- In equal range test case the performance of two implements are almost the same.
- Finally in large range test case. The performance of first fit malloc is better than best fit malloc. The execution time is about four times quicker than best fit malloc.

	Data Scale **(NUM_ITERS ,NUM_ITEMS)**	**FF**	**BF**
Small range Execution Time(s)	(100,10000)	2.525	1.057
Small range Fragmentation (%)	(100,10000)	99.15	82.19
Equal range Execution Time (s)	(10000,10000)	2.249	2.224
Equal range Fragmentation (%)	(10000,10000)	99.68	99.68
Large range Execution Time (s)	(50,10000)	11.28	38.87
Large range Fragmentation (%)	(50,10000)	99.41	96.81

4. Conclusion

In conclusion, choosing a correct malloc strategy according to different implements in different situations is important. When your test case data size is in large data range, it is hard to find a perfectly fitted free block, thus the cost of traversing through the whole free list is enormous. While sacrificing long time to do repeated task but get limited fragmentation improvement, it is better to choose first fit malloc. However, when the data range is small, bf_malloc can easily find a best fitted free block in a short amount of time (Looking for 10$ in a wallet with five tens and five nines, the process can be so quick, right?).