Duke University

Assignment #1: Malloc Library Part 1 Performance Report

ECE 650 – Spring 2019

Ruihan Xu
rx29

Date of submission: 2019/1/22

1. Objective

The objective of this assignment is to implement the malloc() and free(). The performance will be tested to check whether the implementation is good or not. Moreover, the memory allocation will be based on two algorithms, first fit and best fit.

2. Design

To keep track of the data allocated, we need a special data structure. As we will be doing lots of delete/add, doubly linked list will be the best choice. Originally, my design was to link all the data blocks together. However, after testing, I found out that this approach was extremely slow. Therefore, I changed my implementation and only kept a linked list that contains free blocks. The data structure of a single block is as shown below.

```
typedef struct block{
  size_t size;
  struct block* prev;
  struct block* next;
  void* data;
  int fr; // 0 : not free; 1 : free
}block_t;
```

The size is the actual size of the data that can be stored after the block header. *prev points to the free block before the current one and *next points to the block after the current block. *data points to the actual data, which can be represented by (void*)((block*)curblock+1). Fr indicates whether the block is free or not. This field is useless in my final design. However, to compare the two designs, I decided to keep this part so that sizeof(block_t) is consistent.

3. Implementation

The implementation is very standard for this assignment. The functions are shown. The only difference between bf_malloc and ff_malloc is that they have different searching methods, which are provided in search_ff andn search_bf. New_block is to create a new block after the program break. This may happens in three cases: when we first malloc, when the free list is empty or when there is not a fit in the free list. Another case that may happen during malloc is that we found a fitting block in the linked list using either ff or bf. Note that this free block may be larger than the space we required, and we will need to perform the partition. The partition function will split the selected block and put the new block in the linked list and remove the allocated block from the linked list. Moving on to the free() method. Ff_free and Bf_free are the same in this case. We will need to add the freed block into the linked list using add_to_ll. Then perform a merge function to see if the newly added block is adjacent to the free blocks linked to it. If it is, we will merge them together. Also, we need to reset the program break is the last block in the heap is free. The merge_rstbrk() will be called every time we free(). So we only need to check the freed block and the blocks connected to it to see it they can be merged.

Lastly, the get_data_segment_size() and the get_data_segment_free_space_size(). The first one calculates the size of the heap while the second one is the free blocks' size, including the block information. I used a static variable start_addr and first to record the start address of the heap when we sbrk() the very first block. The total heap size can be thus calculated by program break – head start. For the free space, we simply traverse the linked list.

```
static block_t* head = NULL;
static block_t* tail = NULL;
static void* start_addr = 0;
static int first = 0;
void* new_block(size_t size, block_t* last);
void parition(block_t* fitted_block, size_t size);
block_t* search_ff(block_t* head, size_t size);
void *ff_malloc(size_t size);
block_t* search_bf(block_t* head, size_t size);
void *bf_malloc(size_t size);
void add_to_ll(block_t* cur);
void merge_rstbrk();
void ff_free(void *ptr);
void bf_free(void *ptr);
unsigned long get_data_segment_size();
unsigned long get_data_segment_free_space_size();
```

4. Optimization

The implement can be optimized in many ways. First of all, the first approach I took, using a doubly linked link to store all the block is very easy to implement. But as the linked list becomes longer, it will take a long time to find a free block. It took me 300 seconds to complete 100 iters for equal_size tests. So I decided to implement a linked list that only contains the free blocks. However, the speed was not well improved at all when I first implement this. Around 200 seconds to complete 100 iters for equal_size tests. After trying, I found that it is because it takes too much time to traverse the list only to find the last block. So I added a static block pointer, tail, to keep track. The tracking of tail however, is troublesome. At the end, I can complete the 10000 iters equal_size tests in ~20s.

5. Performance Matrices

Fitting algs.	small_range_rand_allocs	equal_size_allocs	large_range_rand_allocs
FF	7.700176 seconds	17.922737 seconds	111.536848 seconds
BF	2.498062 seconds	18.001393 seconds	141.069935 seconds
fragmentation			
FF	0.045407	0.450000	0.080707
BF	0.019785	0.450000	0.039482

The performance of my result was very good. The fragmentation is generallysmall and the speed is fast. Compare the FF and BF first. For equal_size_allocs, there are no significant speed and fragmentation difference due to the implementation of the test cases. But for the other two, the fragmentation is significantly reduced. This is reasonable because BF will partition less. Small range tests get faster and large range test becomes slower after switching to BF. In small range, the data size is 128 - 512 bytes while large range is 32 – 64k bytes. Apparently, FF will run faster than BF in large range because only when BF find a block that perfected fits the size will it stop. In large range allocation, this is not an easy task and it will always need to traverse the whole linked list. But for FF, we only need to find the first block that fits. But in short range, it becomes much easier to find the perfectly fitted block and exit traversal. So the FF will not have a significant lead at this point. Conversely, it will generate more fragmentation, meaning more blocks in the free list. This will lead to a worse performance during the free() process.