**Report for ECE650 homework1**

**Yilin Yang | yy379**

## Section 1: Requirements and summary of development.

The objective for this project is to develop Malloc and Free function to achieve dynamic memory allocation, and to manage that memory blocks efficiently and effectively. To be more specific, the malloc function allocates require bytes and returns a pointer to the allocated memory, while free function frees the allocated memory and makes the memory block available for allocation.

I used linux 64-bit virtual machine that previously used in ECE551 as my development environment to develop my code. For the debugging and testing process, I used valgrind and gdb to debug the program.

The organization of report will be as follows: Section 2 will introduce the data structure and algorithm design for malloc and free functions. Section 3 will cover the analysis for different malloc and free methods.

## Section 2: Design, implementation and test

## Section 2.1: Basic Design for Malloc and Free

A dynamic memory allocator can use Malloc() and Free() to maintain the heap in address space. Memory blocks in the heap are managed by allocator to handle arbitrary request sequence of allocate and free request.

For Malloc design, the first thing to check for allocation is ensure whether there is memory block with enough space. If there is no memory blocks with proper size (equal or bigger than the requested size), sbrk() function will be called to require additional heap memory and return the address at the top of the heap. If there is, the block will be selected and reused according to different allocation policies (First-Fit and Best-Fit). If the freed block is larger enough, the block will be split into two blocks to save memory space.

For Free design, I used explicit free list to manage all the freed blocks as a doubly linked list. Once a memory block is not allocated, I will inset it to the free list. If two memory blocks in free list are adjacent, they will be coalesced into one free block.

## Section 2.2: Implementation of free list, Best-Fit and First-Fit

To store the free blocks in the explicit free list, I created a doubly linked list as picture below:

typedef struct block{

struct block \* next;

struct block \* prev;

size\_t size;

int free\_flag;

} block\_fl;//free list block

By using this struct, all the free blocks can be organized and placed in a proper way as an explicit free list. In my Free function, if the address of adjacent blocks is continuous, these two blocks will be merged as a new block.

The Best-Fit algorithm is achieved by searching the free list to find the smallest block, which could handle the allocation. In this context, I designed my Best-Fit algorithm to traverse the doubly linked list to find the smallest allocatable block.

The Frist-Fit algorithm is to traverse the free list and find the first free region with enough space to allocate the request size.

## Section 2.3: Test and Improvement

I test my program by using the test functions in the start kit. In my test stage, I have encountered bugs like forgetting to check if the free list is empty when inserting new free blocks to the free list. I successfully use Valgrind and GDB to solve most of my logical errors but find my Best-Fit algorithm is extremely slow when I used equal\_size\_allocs function to test. The execution time per iteration is about 0.44s before my optimization.

To solve this issue, I first tested it with multiple iteration times and found the time consuming for iteration is almost linear, which means I might spend the same time per iteration. Hence, I rechecked the logical operation and realized that when the block has the exact required size was found, the traverse for linked list is still executing, which caused a lot of time. At the end, I added a break in the loop for a specific case and saved a lot of running time.

## Section 3: Performance Results & Analysis

From the provided benchmark, I got the performance result below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | First-Fit | | Best-Fit | |
| Execution time(s) | Fragmentation | Execution time(s) | Fragmentation |
| Small | 7.220406 | 0.602488 | 1.555310 | 0.834630 |
| Equal | 17.645855 | 0.999889 | 17.691081 | 0.999889 |
| Large | 70.951969 | 0.836830 | 71.166775 | 0.912994 |

For small range test function, the function allocates random size, ranging from 128 - 512 bytes in 32B increments. The result indicates that Best-Fit performs better in terms of both fragmentation and runtime. From the time perspective, I think that Best-Fit has a better selection strategy for free blocks, resulting in fewer calls to sbrk() compared to Frist-Fit, and therefore running faster. From the fragment perspective, the Best-Fit decreases the external fragmentation compared to First-Fit. That’s because Best-Fit always choose the best block to allocate memory.

For equal size allocation test function, the program uses the same number of bytes (128) in all of its malloc calls. Both First-Fit and Best-Fit have same fragmentation as all the memory blocks have the same size. And these two algorithms have very similar runtime due to the fact that they both use the first block which is available.

For large range allocation test, the program works with allocations of random size, ranging from 32 - 64K bytes (in 32B increments). The results show that Best-Fit again exhibits better fragmentation but has a slightly longer runtime than First-Fit. From the time perspective, I think the reason for longer time is that Best-Fit spends a lot of time on traversing the linked list. In my programming, bf\_malloc() traverses the free list to find the block that saves the most block memory before it finds the block that happens to be exactly the same size as the one it needs. From the fragmentation perspective, the Best-Fit works better as previously stated.

From this project I can conclude that Best-Fit algorithm is more efficient as it can minimize the external fragmentation caused by memory allocation. And for large scale allocation, First-Fit algorithm runs faster than Best-Fit as First-Fit will stop searching as soon as it finds a suitable block. In conclusion, choosing which algorithm in real life highly depend on the specific requirement. If fast allocation time is a priority, First-Fit may be the better choice, but if memory usage is the first concern, Best-Fit may be a better choice.