Performance Study Report of Malloc Library

- Name: Wenting Yang
- NetID: wy77

1 Overview of Implementation

The basic idea of my implementation is to use a doubly-linked list to keep track of every free memory block.

1.1 Storage of meta-information

In order to keep track of each memory block that we allocated or freed in the heap, we need to store meta-information in the blocks, including the size of the block and two pointers to the previous and next node in the free list. A struct is used to store this information:

```
typedef struct block_t {
  size_t size;
  struct block_t * prev;
  struct block_t * next;
} block;
```

1.2 Basic Idea of Implementation

I use a doubly-linked list to manage the dynamically allocated memory in the heap. The list is organized in ascending order of address. For MALLOC operation, first, scan through the free list to check if there is an appropriate free block. For the first fit strategy, we choose the first block with a size bigger than or equal to the requested size; for the best fit strategy, we have to either scan through the entire list to find the most appropriate one or stop when we see a block with size equal to the requested size.

If a proper free block is found, we check if the block size is larger than the requested size + the size of meta info; if it is, we split the block into two blocks. Then, we remove the original block from the free list and add the newly separated block into the list.

If there is no applicable free block, we have to call sbrk() to allocate a new block in the heap. The size of this new block will be the requested size + the size of meta info. Each time we use sbrk() to allocate a new block, we increase the total heap size by the specified amount.

For FREE operation, we insert the block into the free list. For the implementation we use a pointer to pointer method to find the appropriate position to add to the linked list. Then

we check the adjacent nodes in the linked list; if their physical address is adjacent to the newly added block, we have to merge the adjacent blocks into one free block.

2 Results and Analysis

Results of performance experiments:

Test	First-Fit		Best-Fit	
	Execution time (s)	Fragmentation	Execution time (s)	Fragmentation
Small range	14.62	0.07	6.33	0.02
Equal size	22.50	0.45	22.87	0.45
Large range	40.65	0.09	54.48	0.04

1) For equal_size_allocs, the allocation size is fixed at 128 bytes; therefore, every block in the free list has the same size. So the first-fit and best-fit strategies will both pick the first block in the free list. So the runtimes of the two approaches are very close; the fragmentations are the same, around 0.5, because they are calculated halfway through the iterations.

2) For small_range_rand_allocs, the allocation size is random, ranging from 128 to 512 bytes. Before starting, the program mallocs NUM_ITEMS blocks and stores them in the array malloc_items[0]. Then in even iterations, the program frees 50 blocks in malloc_items[0] and mallocs 50 blocks in malloc_items[1]; in odd iterations, it switches the two operations.

   It turns out that the best-fit strategy has a better performance than the first-fit, the execution time is 42%, and the fragmentation is 28% of the first-fit result. Theoretically, first-fit should have a shorter execution time because if we look at one free operation, first-fit will always take the first block that is bigger than the requested size. In contrast, best-fit has to possibly loop through the entire list to find the most appropriate one. However, best-fit turns out to be faster because it is more efficient in managing spaces; in other words, it can avoid the problem when we use a huge block for small data, this could divide the memory into small fragments, some might be too small to be used again, and pile up in the free list, leading to extra sbrk() calls to allocate new memory.

3) For large_range_rand_allocs, the allocation size is random, ranging from 32 to 64K bytes. It does the same thing as small_range_rand_allocs except that the size of blocks varies greatly. In this case, the free list will be much longer. As I printed out the free list length, it’s around 4000, whereas the length is about 2000

in small_range_rand_allocs. Now scanning through the entire free list will take significantly longer than just using sbrk() to allocate a new block. In this case, it is more efficient to use the first-fit strategy and make use of the first usable block. So the execution time of first-fit is shorter than that of best-fit, but the fragmentation of best-fit is still smaller, indicating a high utilization of the memory.

In conclusion, the efficiency of the two strategies depends on the range of allocation size. The best-fit approach is preferred if the system requires a regular allocation of close sizes. However, it is more common than the allocation size can vary from small to large, and the first-fit approach is way more efficient in this case.