Report of Malloc Library Project

Name: Yue Yang | NetID: yy258

In this project, I implemented memory allocation function malloc() and free() with C.

1 Overview of Implementation

1.1 Definition of Block and Metadata

In my implementation, an allocated block contains two components. The first part is metadata that contains the information about the block, including the data size, whether the block is free, and where are the previous and next free blocks. The second part is the actual data that we store. Specifically, I define a struct for the metadata. It is as below:

```
struct metadata {
    int size;
    int isfree;
    struct metadata * prev;
    struct metadata * next;
};
typedef struct metadata Metadata;
```

I used the above struct to realize a linked list which tracks all free blocks. Once a block is freed, it is added to the free list. Once it is occupied, it is removed from the free list.

1.2 Basic Idea of Implementation

For MALLOC operation, first we check whether there is a freed block with proper size that we can use. If there is not, we call sbrk() and allocate a new block. If there is, we reuse that block. If the size of the freed block is big enough, we split it into two blocks, use one and keep another block as free for later use.

For FREE operation, we just mark that block as free by adding it to the free list. Check if the newly freed block's physically adjacent block is free. If it is, merge the two adjacent free blocks as one free block.

1.3 Improvement of Implementation

During my implementation, I reduce the runtime of my program by tracking free blocks and adding newly freed block to the linked list in sorted address order.

At the very first of my implementation, I used the linked list to track all blocks. So, when we want to merge adjacent free blocks, we can just follow current block's pointers to find the physically adjacent blocks, then check whether they are free or not. This takes O(1) time. However, when we want to reuse a free block, we need to iterate through all blocks to find whether there is a proper one. This takes O(n) time. Taken all these into consideration, my first version of the program ran slowly, so I used the linked list to only track free blocks.

By only tracking free blocks, the time of finding a free block is reduced since we only need to iterate through free blocks. However, merging becomes a little tricky. We cannot use current block's pointer to find the physically adjacent blocks anymore.

To find the physically adjacent blocks, my first thought was to iterate all free blocks, compare their addresses with the current one and check whether they are adjacent or not. This made my program run very slow.

Then I changed the way of adding a newly freed block. Instead of just adding it to the back of the linked list, I added it in a sorted way. Every time we want to add a block, we iterate all free blocks from its head, compare the addresses and insert the newly freed block to the right place that keeps the addresses of all free blocks in ascending order. In this way, if we want to merge adjacent free blocks, we can follow newly freed block's pointers to find the free blocks that have the closest addresses, then check whether they are physically adjacent or not. If they are, we perform merge operation. These changes made my program run faster.

2 Results and Analysis

My results of performance experiments(using default value) are as below:

Pattern	First-Fit		Best-Fit	
	Execution time /s	Fragmentation	Execution time /s	Fragmentation
Equal	15.52	0.45	15.42	0.45
Small	5.75	0.06	1.31	0.02
Large	93.44	0.09	121.11	0.04

For equal_size_allocs, the program uses the same number of bytes in all of its malloc calls. The results show that the two strategies have close runtime and same fragmentation. I think this is because the size of every block is the same. If First-Fit and Best-Fit search the free list for a proper block, they will always find the first free block available and take that. The two strategies work the same way, so they have close runtime. Both of them have a fragmentation around 0.5, because it is calculated halfway.

For small_range_rand_allocs, the program works with allocations of random size, ranging from 128 - 512 bytes. The results show that Best-Fit run faster than First-

Fit. Best-Fit takes more time than First-Fit in searching the proper free block, but it uses the free blocks in a more efficient way. If we have three free blocks with size of 3, 2, 1 in order and we ask for three blocks of size 1,2,3 in order. Then in First-Fit, we can only use free block 3 to put new size 1 and use free block 2 to put new size 2. While in best-fit, we can use all three free blocks and call sbrk() less. The results show that under this situation, the runtime for calling sbrk() is more dominating than iterating and searching the free blocks. That's what makes Best-Fit run faster that First-Fit.

For large_range_rand_allocs, the program works with allocations of random size, ranging from 32 - 64K bytes. It works the same way as small_range_rand_allocs except that the size of blocks ranges from small to really large. The results show that First-Fit run faster than Best-Fit, I think this is because under this situation, iteration plays a more dominating role than sbrk(). I think it is because the sizes of blocks in large_range_rand_allocs are more random than those in small_range_rand_allocs, it has less possibility to find a freed block with proper size to use. Best-Fit spend more time on finding the proper free blocks, that makes it run slower under this situation.

In conclusion, in real life we have various sizes of data, the actual situation is more like large_range_rand_allocs , so I recommend First-Fit policy.