Project 1: Malloc Library Part 1

Junyan Tang (jt418)

1 Overview

(a) Data structure of metadata
The space managed is known as **heap** and the generic data structure used to manage free space in the heap is free list. As this structure contains references to all of the free chunks and needs to track free space, it is reasonable to use linked list as free list. I designed this as below.

```
//free list data structure
typedef struct __node_t{
    size_t size;
    struct __node_t *next;
} node_t;
```

In the code above, **size** means the size of actual usable memory in one block of free list, and **next** is a pointer point to the address of next block. I only use this data structure to construct free list.
(b) Memory allocation policies
I implemented two basic strategies for managing free space: **First Fit** and **Best Fit**.
The difference between these two strategies is merely which free block to choose. Thus, I designed three functions to be called by both strategies.

```
void *sbrk_memory(size_t size);
void *split_block(node_t *curr, node_t *prev, size_t size);
void *remove_block(node_t *curr, node_t *prev, size_t size);
```

When the free memory is not enough, it will **sbrk()** to increase heap. When the appropriate block size if larger than sizeof(metadata) + sizeof(required), the block will be split and the allocated one should be deleted from free list. When the block size is larger than sizeof(required), but not enough for an additional metadata, it will be simply removed from free list.
(c) Memory free polices
My free list is ordered based on the address where the block starts. So every time a user calls **free()**, I will find that memory block and add it to the right place of the free list. As I need to coalesce free space when a chunk of memory is freed, I check previous block and next block of the inserted block and see whether they can be merged. **Best Fit** and **First Fit** can use the same free policy.

2 Results of performance

In the starter kit, it provides three different policies to test performance. The result is as below. FF refers to First Fit, and BF refers to Best Fit. I've run the program several times to get the average result.

Pattern	**FF Exec Time**	**FF Fragmentation**	**BF Exec Time**	**BF Fragmentation**
Small	11.27s	0.0745	4.32s	0.0269
Equal	15.05s	0.4500	15.18s	0.4500
Large	38.13s	0.0934	49.90s	0.0408

3 Analysis of results

For large-range allocation, the first-fit policy has less execution time, but almost two times of fragmentation to best-fit policy's fragmentation. For small-range allocation, the efficiency of first-fit policy is worse than best-fit one in both aspects. For equal-size allocation, it seems that they have similar performance.
(a) Large-range allocation
For large-range allocation, even when sometimes we find the most suitable block to allocate memory, we still need to split them. Thus, it will cause more time than first-fit strategy because we just simply choose the first one.
(b) Small-range allocation
When using best-fit strategy, it will always find a suitable block to allocate. Without splitting and merging, it executes much faster than first-fit strategy. Also, since we can remove the whole free block from free list(I think the situation when it isn't larger enough to split always happens), the fragmentation is lower than first-fit strategy.
(c) Equal-size allocation
Since this program uses the same number of bytes in all of its malloc calls, the best and first fit will find the same free block to allocate(I forced the while loop to stop traversing the whole free list when the block size equals required size).