Assignment3 Report

Team member: Hanxi Chen, Rongsheng Qian

Overall:

In this assignment, we use AVL Tree to manage the memory blocks when we my_malloc or my_free memory. In order to reduce the fragmentation we use AVL tree to find the most suitable memory block (will be explained below). AVL Tree managment is used both in my_malloc and my_free. So the time complexity of these two functions is in O(log n) with the low fragmentation (as low as using best fit algorithm).

We have a defined constant in `my_malloc.h` file for the size of the initial memory block.

```
#define INITIAL_BLOCK_SIZE 4096
```

Structure:

I split the whole memory into several blocks which can support variable number of tasks and form them into `struct`.

There are two AVL tree. One is for storing free blocks in memory where its key is `size_t size`. Another is for storing used blocks in memory where its key is `void* start`. Each block can only be stored exactly in one tree (either tree for free blocks or tree for used blocks).

`bool free`: This variable recodes the status which is to determine whether the block should be in the used block tree or the free block tree.

`Block* depth`: Traditional AVL tree doesn't support duplicate keys. However it is possible to have free blocks with the same size. So we invent this varible to allow blocks with the same size store in the same position in the tree by creating a 3D tree, which is similar with `pop` in `stack`.

`Block* prev;`, `Block* next`: It is Used to check whether adjacent blocks are free, and merge them if they are when we invoke my_free function.

`Block* free_curr_root`, `Block* used_curr_root`: This two global varible in .c file is stored the roots of two tree (free block tree and used block tree).

```c
typedef struct Block_mem Block;
struct Block_mem
{
    void* start;
    size_t size;
    bool free;
    // for AVL tree
    Block* right;
    Block* left;
    int height;
    Block* depth; // store the same size block in the same position in tree
    Block* prev;//Used to check whether adjacent blocks are free, and merge them if they are
    Block* next;//Used to check whether adjacent blocks are free, and merge them if they are
};
```

Algorithm:

mem_init: `void mem_init();`

- Malloc the whole memory into one block according to the `define INITIAL_BLOCK_SIZE 4096`
- Set its `bool free = ture` and insert it into the free block tree.

my_allocate: `void* my_malloc(size_t size);`

- Using AVL Tree Search to find best block which is free and have the smallest size bigger thant the size that is needed, remove this block from free block tree.
- `if (result->size>required_size && result->size - required_size > 4)` if the block size is greater than required, I will split the block into two blocks. The first block has `size = required_size` and the second block has `size = result->size - required_size`.
- Insert second block (if existed) into free block tree after merging adjacent free blocks and insert first block into used block tree.
- Return the first block's `void* start`
- Time complexity: O(log n)