```
/*
    COMP3511 Fall 2024
    PA3: Simplified Memory Management (smm)

    Your name:LI,Yuntong
    Your ITSC email: ylino@connect.ust.hk

    Declaration:

    I declare that I am not involved in plagiarism
    I understand that both parties (i.e., students providing the codes and student

*/

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h> // use mmap, munmap system calls

// ==== About Heap Management in Per-process memory space =======
//
// Implementation notes:
// sbrk/brk is obselete (should not be used in future)
// mmap/munmap with some global constants/variables are used to define a heap segm
//
// Note: DO NOT MODIFY heap_start, heap_end, heap_current_break directly
// mm_sbrk is implemented to simulate the expected results of sbrk/brk system call
// Use mm_sbrk(). It provides a similar sbrk() function to adjust heap_current_bre
//
// Heap illustration:
// heap_end - heap_start = HEAP_SIZE bytes
//
// |-------------------| <------ heap_end (the upper limit of the heap)
// |                   |
// |                   |
// |-------------------| <------ heap_current_break (mm_sbrk(0) returns this addre
// |                   |
// |  Heap in used     |
// |                   |
// |                   |
// |-------------------| <------ heap_start (the lower limit of the heap)

const int HEAP_SIZE = 8000; // heap size in bytes
void *heap_start = NULL;
void *heap_end = NULL;
void *heap_current_break = NULL;

// Usage:
//   mm_sbrk(0) returns the current heap break point
//   if sz > 0, mm_sbrk(sz) moves up the current heap break point (i.e., enlarge t
//   if sz < 0, mm_sbrk(sz) moves down the current heap break point (i.e., shrink
void *mm_sbrk(int sz)
{
    if (heap_start == NULL || heap_end == NULL || heap_current_break == NULL)
        return MAP_FAILED; // error address: (void*) -1
    if (sz == 0)
        return heap_current_break;
    // Note: sz is positive
```

```c
    if (sz > 0 && heap_current_break + sz <= heap_end)
    {
        void *ret = heap_current_break;
        heap_current_break += sz;
        return ret;
    }
    // Note: sz is negative
    if (sz < 0 && heap_current_break + sz >= heap_start)
    {
        void *ret = heap_current_break;
        heap_current_break += sz;
        return ret;
    }
    return MAP_FAILED; // error address
}
// ==== End heap management =======

const int MAX_POINTERS = 26;
const int MAX_OPERATIONS = 100;

const char OPERATION_TYPE_MALLOC = 'M';
const char OPERATION_TYPE_FREE = 'F';
const char OPERATION_TYPE_COMBINE_NEARBY_FREE = 'C';

#define OPERATION_STR_MALLOC "malloc"
#define OPERATION_STR_FREE "free"
#define OPERATION_STR_COMBINE_NEARBY_FREE "combine_nearby_free"

const char META_DATA_STATUS_FREE = 'f';
const char META_DATA_STATUS_OCCUPIED = 'o';

// Data structure of MetaData
//
// The memory layout for this project assignment is:
//
// |--------------| <-- heap_current_break
// | Data N       |
// |--------------|
// | MetaData N   |
// |--------------|
// |    ...       |
// |    ...       |
// |--------------|
// | Data 1       |
// |--------------|
// | MetaData 1   |
// |--------------| <--- heap_start
struct
    __attribute__((__packed__)) // compiler directive, avoid "gcc" padding bytes to struct
    MetaData
{
    size_t size; // 8 bytes (in 64-bit OS)
    char status; // 1 byte ('f' or 'o')
};

// calculate the meta data size and store as a constant (exactly 9 bytes)
const size_t meta_data_size = sizeof(struct MetaData);

void mm_print()
{
    void *cur_heap_break = mm_sbrk(0);
    void *cur = heap_start;
    int i = 1;
    while (cur < cur_heap_break)
    {
        struct MetaData *md = (struct MetaData *)cur;
        printf("Block %02d: [%s] size = %4ld %s\n",
               i++,                                                     // block number -
               (md->status == META_DATA_STATUS_FREE) ? "FREE" : "OCCP", // free or occupie
               md->size,
               md->size == 1 ? "byte" : "bytes"); // size, in term of bytes

        // Advance to the next meta data
        cur += meta_data_size + md->size;
    }
}

void *mm_malloc(size_t size)
{
    // TODO: implement our own malloc function here
    void *cur = heap_start;
    void *cur_heap_break = mm_sbrk(0);

    // First-fit algorithm: scan through the heap to find a suitable free block
    while (cur < cur_heap_break)
    {
        struct MetaData *md = (struct MetaData *)cur;
        if (md->status == META_DATA_STATUS_FREE && md->size >= size)
        {
            // If the block is larger than needed, split it
            if (md->size > size + meta_data_size)
            {
                void *new_block = cur + meta_data_size + size;
                struct MetaData *new_md = (struct MetaData *)new_block;
                new_md->size = md->size - size - meta_data_size;
                new_md->status = META_DATA_STATUS_FREE;
                md->size = size;
            }
            md->status = META_DATA_STATUS_OCCUPIED;
            return cur + meta_data_size; // Return the address after MetaData
        }
        cur += meta_data_size + md->size;
    }

    // No suitable block found, request more memory from the heap
    struct MetaData *new_md = (struct MetaData *)mm_sbrk(meta_data_size + size);
    if (new_md == MAP_FAILED)
        return NULL; // Out of memory
    new_md->size = size;
    new_md->status = META_DATA_STATUS_OCCUPIED;
    return (void *)new_md + meta_data_size;  // you should return a suitable address here
}

void mm_free(void *p)
{
    // TODO: implement our own free function here
    if (p == NULL)
        return;

    struct MetaData *md = (struct MetaData *)(p - meta_data_size);
    md->status = META_DATA_STATUS_FREE;
}

void mm_combine_nearby_free()
{
    // TODO: implement the algorithm to combine nearby free blocks
    void *cur = heap_start;
    void *cur_heap_break = mm_sbrk(0);

    while (cur < cur_heap_break)
    {
        struct MetaData *md = (struct MetaData *)cur;
        if (md->status == META_DATA_STATUS_FREE)
        {
            void *next = cur + meta_data_size + md->size;
            if (next < cur_heap_break)
            {
                struct MetaData *next_md = (struct MetaData *)next;
                if (next_md->status == META_DATA_STATUS_FREE)
                {
                    // Combine the two blocks
                    md->size += meta_data_size + next_md->size;
                    continue; // Check the combined block with the next block
                }
            }
        }
        cur += meta_data_size + md->size;
    }
}

int main()
{
    char operation_types[MAX_OPERATIONS];
    char pointer_chars[MAX_OPERATIONS];
    int malloc_sizes[MAX_OPERATIONS];
    int sz_operations;
    int i, j;

    // Assume there are at most 26 different malloc/free
    // Here is the rule to map the block_name to pointers index
    // a=>0, b=>1, ..., z=>25
    void *pointers[MAX_POINTERS];
    for (i = 0; i < MAX_POINTERS; i++)
        pointers[i] = NULL;
    char *target = NULL;

    char command[30];  // malloc/free/combine_nearby_free
    char block_name;   // a-z
    size_t block_size; // a non-negative integer

    // Part 1: read and store the input
    scanf("%d", &sz_operations); // read the number of operations
    for (i = 0; i < sz_operations; i++)
    {
        scanf("%s", command);
        if (strcmp(command, OPERATION_STR_MALLOC) == 0)
        {
            scanf(" %c %ld", &block_name, &block_size);
            operation_types[i] = OPERATION_TYPE_MALLOC;
            pointer_chars[i] = block_name;
            malloc_sizes[i] = block_size;
        }
        else if (strcmp(command, OPERATION_STR_FREE) == 0)
        {
            scanf(" %c", &block_name);
            operation_types[i] = OPERATION_TYPE_FREE;
            pointer_chars[i] = block_name;
        }
        else if (strcmp(command, OPERATION_STR_COMBINE_NEARBY_FREE) == 0)
        {
            operation_types[i] = OPERATION_TYPE_COMBINE_NEARBY_FREE;
        }
    }

    // Part 2: Allocate the HEAP_SIZE memory from OS and
    // setup heap_start, heap_end, and heap_current_break pointers
    // On success, heap_start points to the starting address of the heap region
    // At the beginning, heap_current_break is pointing to heap_start (heap is used in

    heap_start = mmap(NULL, HEAP_SIZE, PROT_READ | PROT_WRITE,
                      MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    if (heap_start == MAP_FAILED)
    {
        printf("Error in creating heap using mmap\n");
        exit(-1);
    }
    heap_current_break = heap_start;
    heap_end = heap_start + HEAP_SIZE;


    // Part 3: Do the simulation
    for (i = 0; i < sz_operations; i++)
    {
        if (operation_types[i] == OPERATION_TYPE_MALLOC)
        {
            block_name = pointer_chars[i];
            block_size = malloc_sizes[i];
            if (pointers[block_name - 'a'] != NULL)
            {
                printf("=== %s %c %ld ===\n", OPERATION_STR_MALLOC, block_name, block_size
                printf("malloc Error: %c is pointing to some memory address\n", block_name
            }
            else
            {
                target = mm_malloc(block_size);
                if (target != NULL)
                {
                    // This operation ensures that the returned pointer is correct
                    // As we only fill characters up to the block size,
                    // no meta data should be erased
                    for (j = 0; j < block_size; j++)
                        target[j] = ' '; // 2024-Nov-19: Fixed this line
                }
                pointers[block_name - 'a'] = target;
                printf("=== %s %c %ld ===\n", OPERATION_STR_MALLOC, block_name, block_size
                mm_print();
            }
        }
        else if (operation_types[i] == OPERATION_TYPE_FREE)
        {
            block_name = pointer_chars[i];
            if (pointers[block_name - 'a'] == NULL)
            {
                printf("=== %s %c ===\n", OPERATION_STR_FREE, block_name);
                printf("free Error: %c is pointing to NULL\n", block_name);
            }
            else
            {
                mm_free(pointers[block_name - 'a']);
                pointers[block_name - 'a'] = NULL;
                printf("=== %s %c ===\n", OPERATION_STR_FREE, block_name);
                mm_print();
            }
        }
        else if (operation_types[i] == OPERATION_TYPE_COMBINE_NEARBY_FREE)
        {
            mm_combine_nearby_free();
            printf("=== Combine nearby free blocks ===\n");
            mm_print();
        }
    }

    // Part 4: return HEAP_SIZE memory to the OS
    if (munmap(heap_start, HEAP_SIZE))
    {
        // failure case
        printf("Error in munmap\n");
        exit(-1);
    }

    return 0;
}
```