操作系统实验一：可变分区存储管理

郭进尧 519030910124

一、实验说明

1.1 实验题目：

编写一个C语言程序，模拟UNIX的可变分区内存管理，使用循环首次适应法实现对一块内存区域的分配和释放管理。

1.2 实验目的：

1. 加深对可变分区存储管理的理解。
2. 考察使用C语言编写代码的能力，特别是C语言编程的难点之一：指针的使用。
3. 复习使用指针实现链表以及在链表上的基本操作。

1.3 实验要求：

1. 一次性向系统申请一块大内存空间，本次选取1000Byte。
2. 使用合适的数据结构实现空闲存储区。
3. 实现 `addr = (char *)lmalloc(unsigned size)` 和 `lfree(unsigned size, char *addr)` 两个内存分配函数，`size` 和 `addr` 均通过控制台窗口输入。
4. 执行完一次内存分配或释放后，将当前空闲存储区的情况打印出来。

二、算法思想

本节将简要介绍实验涉及到的课程知识

2.1 可变分区存储：

在可变分区存储管理系统中，系统并不预先将内存划分成固定分区，二是等待作业运行需要内存时向系统申请，系统从空闲内存区中分配大小等于作业所需的内存，如图1所示。每个分区对应一个 `map` 结构管理，分别有两个指针指向前后的两个空闲分区，可以避免产生内零头。

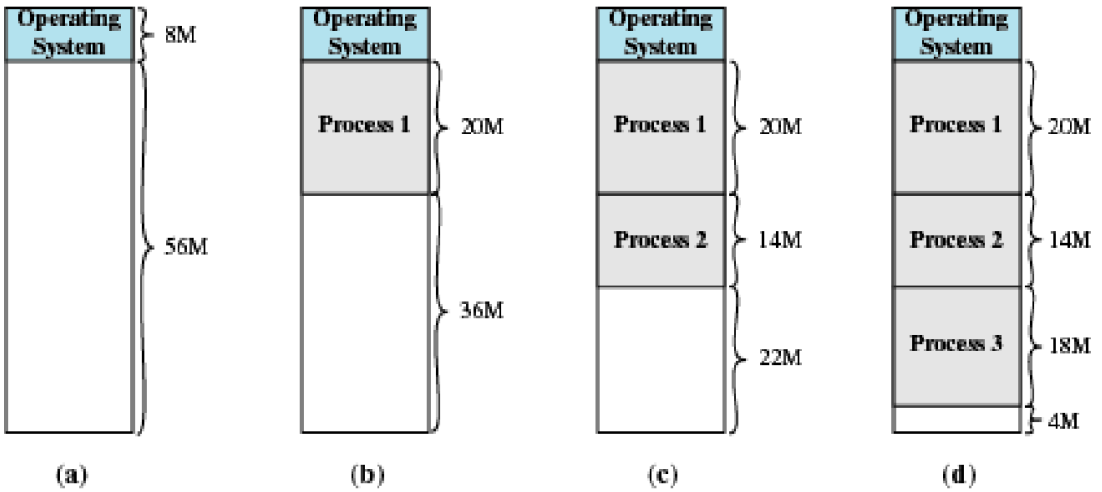

图1.可变分区存储示例

2.2 循环首次适应法

循环首次适应法将各空闲区按地址从低到高的次序组成循环链表，每次分配时从上次分配查到的空闲区的后面开始扫描，当找到一个满足要求的空闲区时，将指定大小的区域分配给该作业，如果遍历循环链表后仍未找到满足要求的空闲区，则返回报错信息。

释放一个占用的区域时，将目标内存块标记为空闲区，并将其与相邻的空闲区合并，具体可分为四种情况，如图2所示。

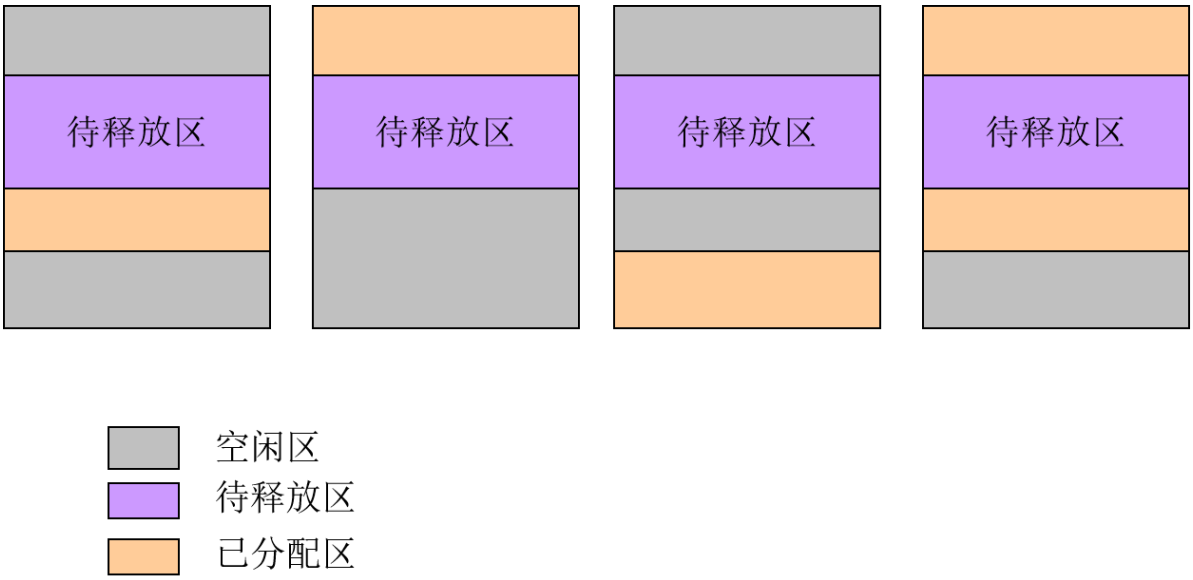

图2.释放区域与相邻空闲区的四种情况

1. **仅前相邻区域为空闲区**：合并前空闲区与释放区，修改前空闲区 `map` 中 `size` 的值为两个区域之和，并令前空闲区指向后一节点的指针 `next` 指向释放区后一节点，删去释放区节点。
2. **仅后相邻区域为空闲区**：合并后空闲区与释放区，修改释放区 `map` 中 `size` 的值为两个区域之和，将释放区的使用位改为0，并令释放区指向后一节点的指针 `prior` 指向释放区后一节点，删去后空闲区节点。
3. **前、后相邻区域为空闲区**：与前、后空闲区合并，修改前空闲区 `map` 中 `size` 的值为三个区域之和，并令前空闲区指向后一节点的指针 `next` 指向后空闲区后一节点，删去释放区节点和后空闲区节点。
4. **前、后相邻区域均不为空闲区**：将释放区的使用位改为0。

由于在一片连续地址中实现循环链表，会出现前一节点实际在当前节点之后的情况（地址更大），此时要进行边界检查。

三、算法实现

本节主要介绍实验中主要功能 `lmalloc` 和 `lfree` 的实现方法，实现循环链表所采用的数据结构，以及与用户交互的实现，包括接收输入和打印空闲存储区。具体来说，程序主要包括以下几个部分：

- 初始化模块 `initialize` 向内存申请1000bytes大小的空间，并设置第一个节点信息。
- 输入处理模块 `handle_input` 处理用户通过命令行输入的指令，通过 `switch` 语句进行控制。
- 内存分配模块 `lmalloc` 负责模拟内存分配的场景。
- 内存释放模块 `lfree` 负责模拟内存释放的场景。
- 打印模块 `print_freemap` 和 `print_help` 负责打印空闲区情况和帮助文档。

3.1 数据结构

本次实验中，通过双向链表来实现所有区块的循环链表，每个区块通过一个双向链表中的节点 `map` 控制，`map` 中包含该区块的大小 `m_size`，区块的首地址 `m_addr`，指向该区块前后区域的指针 `next` 和 `prior`，并且设置使用位标识该块为空闲区还是使用区，空闲区 `isUsed` 为0，使用区 `isUsed` 为1。

设置全局指针 `pointer` 来进行遍历，`pointer` 指向当前的空闲区块。

```
struct map
{
    unsigned m_size;
    int isUsed;
    char *m_addr;
    struct map *next, *prior;
};

struct map *pointer;                    //指向当前空闲区的指针
```

3.2 各部分实现

初始化：

初始化函数通过 `malloc` 向内存申请1000bytes大小的空间，将其作为初始节点，令它的前后指针都指向自己。

```
void initialize() {
    initmap = malloc(sizeof(struct map));
    initmap->m_size = 1000;
    initmap->isUsed = 0;
    initmap->m_addr = (char*) malloc(sizeof(char) * 1000);
    initmap->prior = initmap;
    initmap->next = initmap;
    pointer = initmap;
    printf("Environment Initialized Successly\n");
    printf("malloc 1000: %p\n", initmap->m_addr);
}
```

用户输入处理：

用户输入处理函数通过 `scanf` 获取用户输入的第字符，根据不同字符采取不同的处理逻辑：

- `m`：分配内存，具体为 `m [size]`
- `f`：释放内存，具体为 `f [size] [addr]`
- `q`：退出程序

同时，在读取输入指令时略去 `'\n', '\t', ' '` 字符。

```
void handle_input(){
    char choice;
    unsigned addr,size;
    do{
        scanf("%c",&choice);
        switch (choice){
        case 'm':
            scanf("%u",&size);
```

```
            printf("alloc size %u \n",size);
            lmalloc(size);
            print_freemap();
            break;
        case 'f':
            scanf("%u %u", &size, &addr);
            printf("free size:%u, addr:%u\n",size,addr);
            bool isFree = lfree(size, (char*) addr);
            if(isFree){
                print_freemap();
            }
            break;
        case 'q':
            printf("Bye~\n");
            exit(0);
            break;
        default:
            if(choice != '\n' && choice != '\t' && choice != ' ')
                print_help();
            break;
        }
    }while(true);
}
```

内存分配：

当请求分配内存时，从上次分配内存的位置开始，循环查找空闲块，当空闲块大小大于等于请求空间时，为其新建一个 `map` 节点加入循环列表，标识分配出去的内存块，返回该内存块的起始地址。如果遍历所有空闲块后均不满足条件，则分配失败，输出报错信息：“err: Failed to alloc memory--space is not enough.”

```
char* lmalloc(unsigned size) {
    bool flag = false;
    struct map *ptr = pointer;
    struct map *new_map = malloc(sizeof(struct map));
    do {
        if (pointer->m_size >= size) {
            new_map->m_size = size;
            new_map->m_addr = ptr->m_addr;
            new_map->isUsed = 1;
            new_map->prior = ptr->prior;
            new_map->next = ptr;

            pointer->m_size -= size;
            pointer->m_addr = (char*) ((unsigned) pointer->m_addr + size);
            ptr->prior->next = new_map;
            pointer->prior = new_map;
            flag = true;
            break;
        }
        else pointer = pointer->next;
    } while (pointer != ptr);
    if(flag)
        return new_map->m_addr;
    else{
        printf("err: Failed to alloc memory--space is not enough.\n");
        free(new_map);
```

```
        return NULL;
    }
}
```

内存释放

内存释放与空闲区归并是所有函数中最为复杂的，除了要对2.2节提到的四种情况进行处理外，还要考虑实际物理内存的边界条件限制：当释放的内存块为地址最小的内存块时，即使它的前内存块为空闲，也不能对其进行合并，因为这2个区域的物理内存并不连续；在与后相邻空闲区合并时，也要考虑释放区域是否为地址最大的内存块，如果是则不能合并。

在实现时，释放函数会通过以下两点规则检查输入的参数是否有效，若输入参数无效，则不会执行free操作并返回报错信息。

- 地址 `addr` 必须是一个占用块的首地址。
- 释放大小 `size` 小于等于要释放的内存块的大小

在归并空闲区时，根据四种相邻情况和目标空闲区的位置来选择不同的归并方法，详见代码注释。

```
bool lfree(unsigned size, char *addr){
    //检查addr是否为一个区块的地址
    struct map *ptr = initmap;
    bool isFind = false;
    do {
        if(ptr->m_addr == addr && ptr->isUsed == 1){
            isFind = true;
            break;
        }
        ptr = ptr->next;
    } while (ptr != initmap);
    //如果free地址不是map地址，则报错
    if(!isFind){
        printf("err : Try to free invalid address!\n");
        return false;
    }
    //如果free大小大于目标map的大小，报错
    if(size > ptr->m_size){
        printf("err: Invalid size, too big!\n");
        return false;
    }
    //free大小与目标map大小相等
    if(size == ptr->m_size){
        struct map *front = ptr->prior;
        struct map *back = ptr->next;
        //如果后为占用区或ptr为队尾
        if(back->isUsed == 1 || back->m_addr < ptr->m_addr){
            if(front->isUsed == 1){
                ptr->isUsed = 0;
            }
            else{
                front->m_size += size;
                front->next = back;
                back->prior = front;
                free(ptr);
            }
        }
        //后为空闲区且ptr不在队尾
```

```
        else{
            //前为占用区或ptr在队首
            if(front->isUsed == 1 || front->m_addr > ptr->m_addr){
                    ptr->m_size += back->m_size;
                    ptr->isUsed = 0;
                    ptr->next = back->next;
                    back->next->prior = ptr;
                    free(back);
                    pointer = ptr;
                }
            else{
                front->m_size = front->m_size + size + back->m_size;
                front->next = back->next;
                back->next->prior = front;
                free(ptr);
                free(back);
                pointer = front;
            }
        }
    }
    //free大小小于目标map大小
    else{
        struct map *front = ptr->prior;
        //前为占用区或者ptr为头部
        if(front->isUsed == 1 || front->m_addr > ptr->m_addr){
            struct map *new_map = malloc(sizeof(struct map));
            new_map->isUsed = 0;
            new_map->m_addr = ptr->m_addr;
            new_map->m_size = size;
            new_map->next = ptr;
            new_map->prior = front;

            front->next = new_map;
            ptr->m_size -= size;
            ptr->m_addr = (char*) ((unsigned) ptr->m_addr + size);
            ptr->prior = new_map;
        }
        //前为空闲区且ptr不是头部
        else{
            front->m_size += size;
            ptr->m_size -= size;
            ptr->m_addr = (char*) ((unsigned) ptr->m_addr + size);
        }
    }
    printf("Freed successfully\n");
    return true;
}
```

空闲块打印

遍历方法与分配内存时相似，从当前 `pointer` 指向的块开始，打印出所有空闲块的地址和大小，当再次遍历到 `pointer` 指向的内存块时结束。

```
void print_freemap() {
    struct map *ptr = pointer;
    printf ("-------------------------------------------\n");
    int cnt = 1;
```

```
    printf("Current Free Memory: num address size\n");
    do {
        if(ptr->isUsed == 0){
            printf("Current Free Memory: %d\t%u\t%d\n", cnt, ptr->m_addr,
ptr->m_size);
            ++cnt;
        }
        ptr = ptr->next;
    } while (ptr != pointer);
    printf ("-----------------------------------------\n");
};
```

四、测试方法与结果展示

本次实验采用交互式测试，用户通过命令行输入指令来完成内存申请和释放，主要命令如下：

- 申请内存

```
m [size]  #size为申请内存块的大小
```

- 释放内存

```
m [size] [addr]  #size为释放的大小 addr是释放内存块的起始地址
```

- 退出程序 `q`

测试结果展示：

正常操作结果如图3所示，能顺利完成内存分配和释放，并打印空闲区。

```
Environment Initialized Successly
malloc 1000: 0000000000C51450
Initial address: 12915792
--------Help-----------
alloc memory:  'm [size] '
free memory:  'f [size] [address]'
quit:  'q'
---------end-----------
m 300
alloc size 300
-----------------------------------------
Current Free Memory: num address size
Current Free Memory: 1  12916092        700
-----------------------------------------
f 300 12915792
free size:300, addr:12915792
Freed successfully
-----------------------------------------
Current Free Memory: num address size
Current Free Memory: 1  12915792        1000
-----------------------------------------
m 400
alloc size 400
-----------------------------------------
Current Free Memory: num address size
Current Free Memory: 1  12916192        600
-----------------------------------------
m 200
alloc size 200
-----------------------------------------
Current Free Memory: num address size
Current Free Memory: 1  12916392        400
-----------------------------------------
f 160 12916192
free size:160, addr:12916192
Freed successfully
-----------------------------------------
Current Free Memory: num address size
Current Free Memory: 1  12916392        400
Current Free Memory: 2  12916192        160
-----------------------------------------
q
Bye~
PS D:\GuoJinyao\大三下学期\操作系统\lab1>
```

图3.正常运行结果

程序收到非法输入时，会返回相应报错信息，并打印帮助文档，如图4所示。

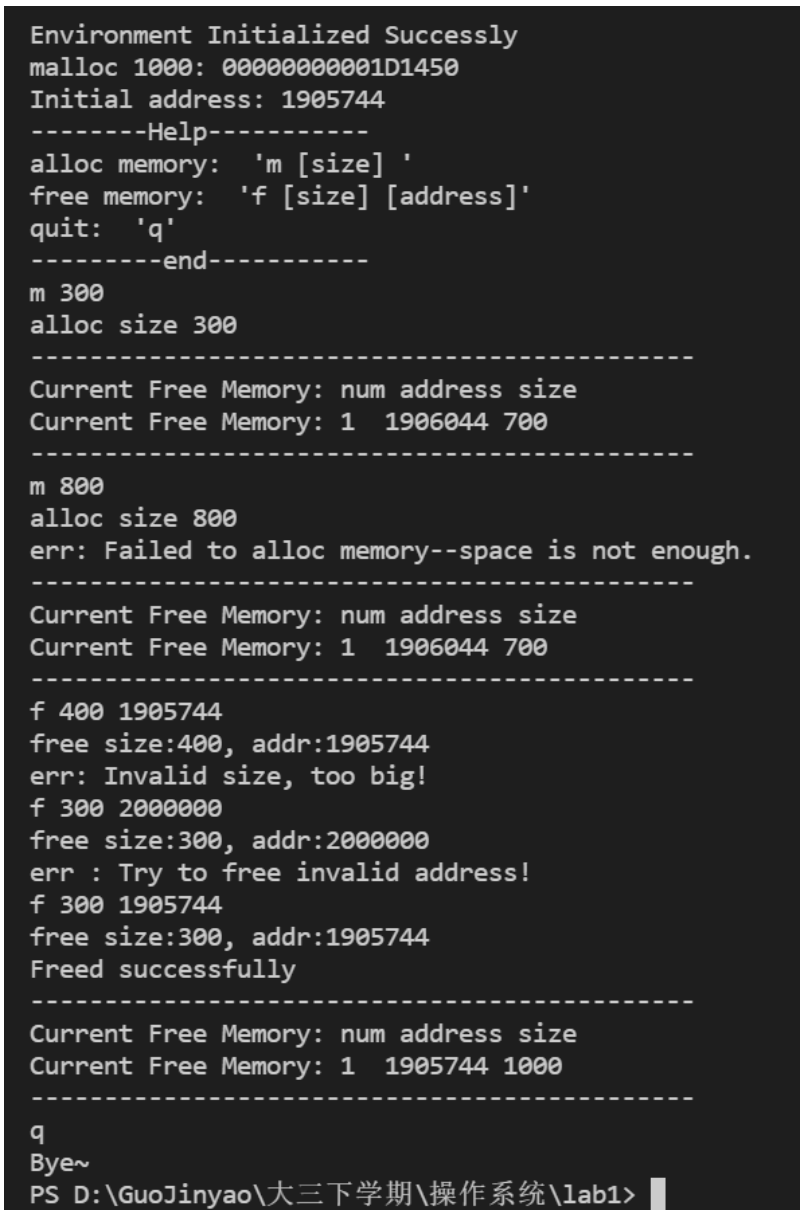

图4.非法输入的运行结果

五、总结与思考

错误分析：

在本次实验中，我遇到的难点主要是对C语言语法不熟悉以及对边界条件的检查。

首先，由于之前C语言使用较少，实验过程中对C语言语法有些生疏，且容易和C++混淆，使得在编写代码时效率较低，尤其在实验初期，我用了接近一个小时才写好输入处理函数，耗时远高于预期。本次实验过程也是我重温C语言的过程，在完成输入处理和打印函数的热身后，我的开发效率也有效回升，在预期时间内完成了实验。

另外一个逻辑上的难点在于对边界条件的检查，实验申请到的内存是一块1000Byte的线性空间，而实现循环链表时并没有考虑实际的地址是否连续，因此在合并空闲区时要进行额外的边界检查，对于循环链表中连续而实际内存中不连续的区块不能进行合并。一开始我并没有考虑到这点，在测试时出现了明显的越界Bug，打印出的空闲区大小甚至会超过1000，在后续修正过程中，我又重写了内存释放函数，添加了关于边界条件的检查。

个人总结

通过本次实验，我收获良多：一方面重温了C语言语法知识，一方面通过自己动手实现内存管理加深了对知识的理解。之前，我通过CSAPP和其他操作系统书籍了解过一些内存分配管理的知识，本次实验中，我借鉴了堆的管理方法和 `chunk` 结构，通过双向链表来管理内存，将每个内存块作为双向链表中的节点，设置使用位来标识内存块的类型。通过借鉴和实现，让我对内存管理的理解更加深刻，通过调试错误，让我更加重视代码逻辑的严密性。

最后，感谢刘老师的指导和同学的帮助！

六、附录

程序源代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct map
{
    unsigned m_size;                //大小
    int isUsed;                     //是否空闲 0为空闲，1为使用中
    char *m_addr;                   //首地址
    struct map *next, *prior;       //指向前、后空闲块的指针
};

struct map *pointer;                //指向当前空闲区的指针
struct map *initmap;                //初始分配的区域

void initialize();
void print_freemap();
void print_help();
void handle_input();
char* lmalloc(unsigned size);
bool lfree(unsigned size, char *addr);

//程序初始化
void initialize() {
    // static struct map* coremap;
    initmap = malloc(sizeof(struct map));
    initmap->m_size = 1000;
    initmap->isUsed = 0;
    initmap->m_addr = (char*) malloc(sizeof(char) * 1000);
    initmap->prior = initmap;
    initmap->next = initmap;
    pointer = initmap;
    printf("Environment Initialized Successly\n");
    printf("malloc 1000: %p\n", initmap->m_addr);
}

//打印当前空闲存储区
void print_freemap() {
    struct map *ptr = pointer;
    printf ("---------------------------------------------\n");
    int cnt = 1;
    printf("Current Free Memory: num address size\n");
```

```
    do {
        if(ptr->isUsed == 0){
            printf("Current Free Memory: %d\t%u\t%d\n", cnt, ptr->m_addr,
ptr->m_size);
            ++cnt;
        }
        ptr = ptr->next;
    } while (ptr != pointer);
    printf ("-----------------------------------------------\n");
};

//程序说明
void print_help(){
    printf("--------Help-----------\n");
    printf("alloc memory:  \'m [size] \' \n");
    printf("free memory:  \'f [size] [address]\' \n");
    printf("quit:  \'q\' \n");
    printf("---------end-----------\n");
}

//接收输入
void handle_input(){
    char choice;
    unsigned addr,size;
    do{
        scanf("%c",&choice);
        switch (choice){
        case 'm':
            scanf("%u",&size);
            printf("alloc size %u \n",size);
            lmalloc(size);
            print_freemap();
            break;
        case 'f':
            scanf("%u %u", &size, &addr);
            printf("free size:%u, addr:%u\n",size,addr);
            bool isFree = lfree(size, (char*) addr);
            if(isFree){
                print_freemap();
            }
            break;
        case 'q':
            printf("Bye~\n");
            exit(0);
            break;
        default:
            if(choice != '\n' && choice != '\t' && choice != ' ')
                print_help();
            break;
        }
    }while(true);
}

char* lmalloc(unsigned size) {
    bool flag = false;
    struct map *ptr = pointer;
    struct map *new_map = malloc(sizeof(struct map));
    do {
```

```
        if (pointer->m_size >= size) {
            new_map->m_size = size;
            new_map->m_addr = ptr->m_addr;
            new_map->isUsed = 1;
            new_map->prior = ptr->prior;
            new_map->next = ptr;

            pointer->m_size -= size;
            pointer->m_addr = (char*) ((unsigned) pointer->m_addr + size);
            ptr->prior->next = new_map;
            pointer->prior = new_map;
            flag = true;
            break;
        }
        else pointer = pointer->next;
    } while (pointer != ptr);
    if(flag)
        return new_map->m_addr;
    else{
        printf("err: Failed to alloc memory--space is not enough.\n");
        free(new_map);
        return NULL;
    }
}

bool lfree(unsigned size, char *addr){
    //检查size是否为一个区块的地址
    struct map *ptr = initmap;
    bool isFind = false;
    do {
        if(ptr->m_addr == addr && ptr->isUsed == 1){
            isFind = true;
            break;
        }
        ptr = ptr->next;
    } while (ptr != initmap);
    //如果free地址不是map地址，则报错
    if(!isFind){
        printf("err : Try to free invalid address!\n");
        return false;
    }
    //如果free大小大于目标map的大小，报错
    if(size > ptr->m_size){
        printf("err: Invalid size, too big!\n");
        return false;
    }
    //free大小与目标map大小相等
    if(size == ptr->m_size){
        struct map *front = ptr->prior;
        struct map *back = ptr->next;
        //如果后为占用区或ptr为队尾
        if(back->isUsed == 1 || back->m_addr < ptr->m_addr){
            if(front->isUsed == 1){
                ptr->isUsed = 0;
            }
            else{
                front->m_size += size;
                front->next = back;
```

```
                back->prior = front;
                free(ptr);
            }
        }
        //后为空闲区且ptr不在队尾
        else{
            //前为占用区或ptr在队首
            if(front->isUsed == 1 || front->m_addr > ptr->m_addr){
                    ptr->m_size += back->m_size;
                    ptr->isUsed = 0;
                    ptr->next = back->next;
                    back->next->prior = ptr;
                    free(back);
                    pointer = ptr;
                }
            else{
                front->m_size = front->m_size + size + back->m_size;
                front->next = back->next;
                back->next->prior = front;
                free(ptr);
                free(back);
                pointer = front;
            }
        }
    }
    //free大小小于目标map大小
    else{
        struct map *front = ptr->prior;
        //前为占用区或者ptr为头部
        if(front->isUsed == 1 || front->m_addr > ptr->m_addr){
            struct map *new_map = malloc(sizeof(struct map));
            new_map->isUsed = 0;
            new_map->m_addr = ptr->m_addr;
            new_map->m_size = size;
            new_map->next = ptr;
            new_map->prior = front;

            front->next = new_map;
            ptr->m_size -= size;
            ptr->m_addr = (char*) ((unsigned) ptr->m_addr + size);
            ptr->prior = new_map;
        }
        //前为空闲区且ptr不是头部
        else{
            front->m_size += size;
            ptr->m_size -= size;
            ptr->m_addr = (char*) ((unsigned) ptr->m_addr + size);
        }
    }
    printf("Freed successfully\n");
    return true;
}

int main(){
    initialize();
    printf("Initial address: %u\n", initmap->m_addr);
    print_help();
    handle_input();
```

```
216     return 0;
217 }
```