

# 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 *)malloc(unsigned size)` 和 `free(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。

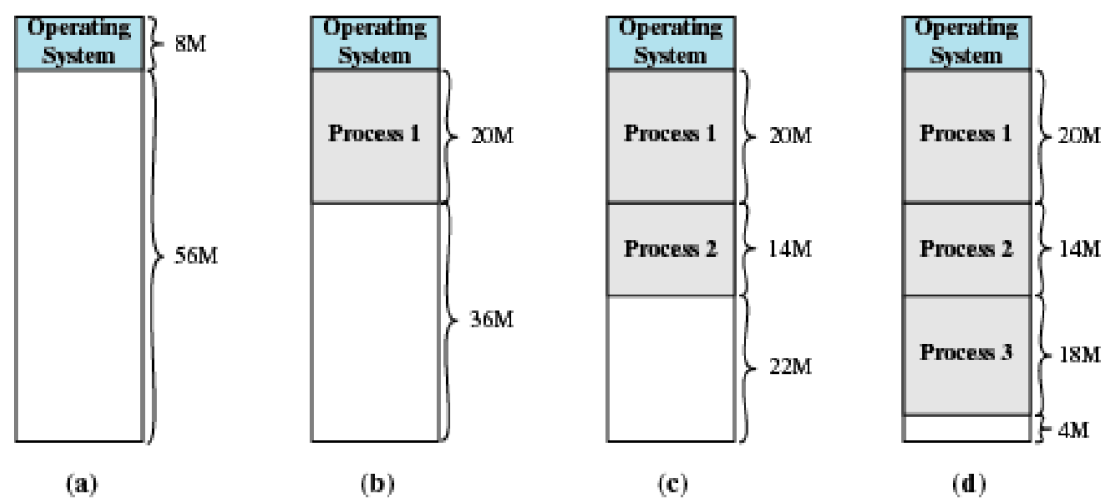


图1.可变分区存储示例

## 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。

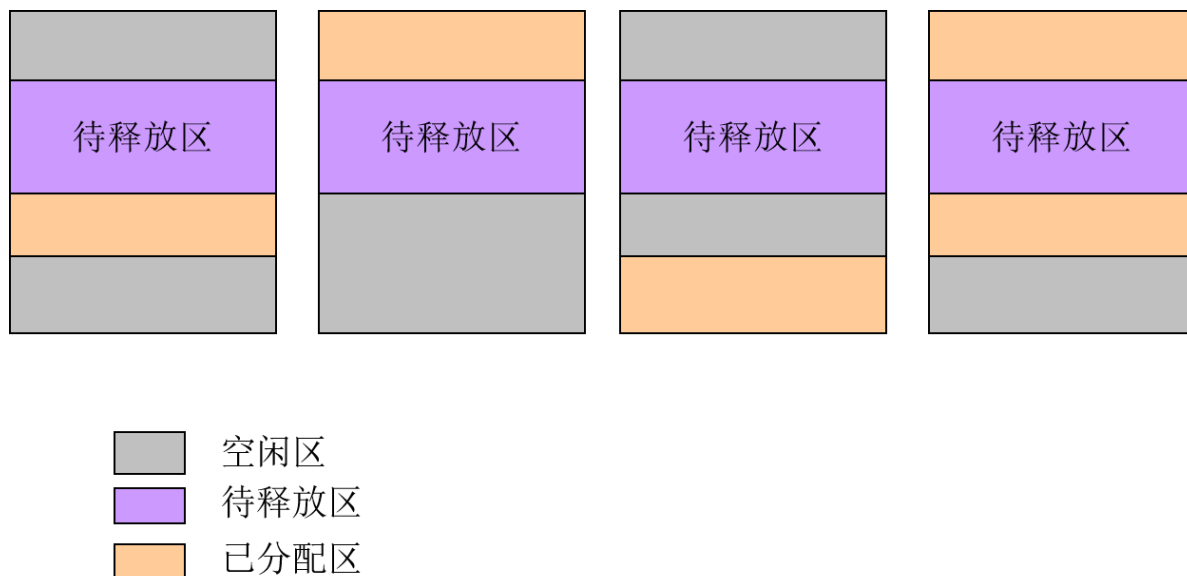


图2.释放区域与相邻空闲区的四种情况

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。

```
1 struct map
2 {
3     unsigned m_size;
4     int isused;
5     char *m_addr;
6     struct map *next, *prior;
7 };
8
9 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。

```
1 void initialize() {
2     initmap = malloc(sizeof(struct map));
3     initmap->m_size = 1000;
4     initmap->isused = 0;
5     initmap->m_addr = (char*) malloc(sizeof(char) * 1000);
6     initmap->prior = initmap;
7     initmap->next = initmap;
8     pointer = initmap;
9     printf("Environment Initialized Successfully\n");
10    printf("malloc 1000: %p\n", initmap->m_addr);
11 }
```

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。

```
1 void handle_input(){
2     char choice;
3     unsigned addr,size;
4     do{
5         scanf("%c",&choice);
6         switch (choice){
7             case 'm':
8                 scanf("%u",&size);
```

```

9         printf("alloc size %u \n",size);
10        lmalloc(size);
11        print_freemap();
12        break;
13    case 'f':
14        scanf("%u %u", &size, &addr);
15        printf("free size:%u, addr:%u\n",size,addr);
16        bool isFree = lfree(size, (char*) addr);
17        if(isFree){
18            print_freemap();
19        }
20        break;
21    case 'q':
22        printf("Bye~\n");
23        exit(0);
24        break;
25    default:
26        if(choice != '\n' && choice != '\t' && choice != ' ')
27            print_help();
28        break;
29    }
30 }while(true);
31 }

```

## 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."

```

1 char* lmalloc(unsigned size) {
2     bool flag = false;
3     struct map *ptr = pointer;
4     struct map *new_map = malloc(sizeof(struct map));
5     do {
6         if (pointer->m_size >= size) {
7             new_map->m_size = size;
8             new_map->m_addr = ptr->m_addr;
9             new_map->isUsed = 1;
10            new_map->prior = ptr->prior;
11            new_map->next = ptr;
12
13            pointer->m_size -= size;
14            pointer->m_addr = (char*) ((unsigned) pointer->m_addr + size);
15            ptr->prior->next = new_map;
16            pointer->prior = new_map;
17            flag = true;
18            break;
19        }
20        else pointer = pointer->next;
21    } while (pointer != ptr);
22    if(flag)
23        return new_map->m_addr;
24    else{
25        printf("err: Failed to alloc memory--space is not enough.\n");
26        free(new_map);

```

```

27     return NULL;
28 }
29 }

```

## 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。

```

1  bool lfree(unsigned size, char *addr){
2      //检查addr是否为一个区块的地址
3      struct map *ptr = initmap;
4      bool isFind = false;
5      do {
6          if(ptr->m_addr == addr && ptr->isUsed == 1){
7              isFind = true;
8              break;
9          }
10         ptr = ptr->next;
11     } while (ptr != initmap);
12     //如果free地址不是map地址，则报错
13     if(!isFind){
14         printf("err : Try to free invalid address!\n");
15         return false;
16     }
17     //如果free大小大于目标map的大小，报错
18     if(size > ptr->m_size){
19         printf("err: Invalid size, too big!\n");
20         return false;
21     }
22     //free大小与目标map大小相等
23     if(size == ptr->m_size){
24         struct map *front = ptr->prior;
25         struct map *back = ptr->next;
26         //如果后为占用区或ptr为队尾
27         if(back->isUsed == 1 || back->m_addr < ptr->m_addr){
28             if(front->isUsed == 1){
29                 ptr->isUsed = 0;
30             }
31             else{
32                 front->m_size += size;
33                 front->next = back;
34                 back->prior = front;
35                 free(ptr);
36             }
37         }
38         //后为空闲区且ptr不在队尾

```

```

39         else{
40             //前为占用区或ptr在队首
41             if(front->isUsed == 1 || front->m_addr > ptr->m_addr){
42                 ptr->m_size += back->m_size;
43                 ptr->isused = 0;
44                 ptr->next = back->next;
45                 back->next->prior = ptr;
46                 free(back);
47                 pointer = ptr;
48             }
49             else{
50                 front->m_size = front->m_size + size + back->m_size;
51                 front->next = back->next;
52                 back->next->prior = front;
53                 free(ptr);
54                 free(back);
55                 pointer = front;
56             }
57         }
58     }
59     //free大小小于目标map大小
60     else{
61         struct map *front = ptr->prior;
62         //前为占用区或者ptr为头部
63         if(front->isUsed == 1 || front->m_addr > ptr->m_addr){
64             struct map *new_map = malloc(sizeof(struct map));
65             new_map->isUsed = 0;
66             new_map->m_addr = ptr->m_addr;
67             new_map->m_size = size;
68             new_map->next = ptr;
69             new_map->prior = front;
70
71             front->next = new_map;
72             ptr->m_size -= size;
73             ptr->m_addr = (char*) ((unsigned) ptr->m_addr + size);
74             ptr->prior = new_map;
75         }
76         //前为空闲区且ptr不是头部
77         else{
78             front->m_size += size;
79             ptr->m_size -= size;
80             ptr->m_addr = (char*) ((unsigned) ptr->m_addr + size);
81         }
82     }
83     printf("Freed successfully\n");
84     return true;
85 }

```

## 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。

```

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

```

```

5     printf("Current Free Memory: num address size\n");
6     do {
7         if(ptr->isused == 0){
8             printf("Current Free Memory: %d\t%u\t%d\n", cnt, ptr->m_addr,
ptr->m_size);
9             ++cnt;
10        }
11        ptr = ptr->next;
12    } while (ptr != pointer);
13    printf ("-----\n");
14 };

```

## 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

```
1 | m [size] #size为申请内存块的大小
```

- 释放内存

```
1 | 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 Successfully
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。



```

Environment Initialized Successfully
malloc 1000: 00000000001D1450
Initial address: 1905744
-----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 1906044 700
-----

m 800
alloc size 800
err: Failed to alloc memory--space is not enough.
-----

Current Free Memory: num address size
Current Free Memory: 1 1906044 700
-----

f 400 1905744
free size:400, addr:1905744
err: Invalid size, too big!
f 300 2000000
free size:300, addr:2000000
err : Try to free invalid address!
f 300 1905744
free size:300, addr:1905744
Freed successfully
-----

Current Free Memory: num address size
Current Free Memory: 1 1905744 1000
-----

q
Bye~
PS D:\GuoJinyao\大三下学期\操作系统\lab1>

```

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
5  struct map
6  {
7      unsigned m_size;           //大小
8      int isused;                //是否空闲 0为空闲, 1为使用中
9      char *m_addr;              //首地址
10     struct map *next, *prior;   //指向前、后空闲块的指针
11 };
12
13 struct map *pointer;            //指向当前空闲区的指针
14 struct map *initmap;           //初始分配的区域
15
16 void initialize();
17 void print_freemap();
18 void print_help();
19 void handle_input();
20 char* lmalloc(unsigned size);
21 bool lfree(unsigned size, char *addr);
22
23 //程序初始化
24 void initialize() {
25     // static struct map* coremap;
26     initmap = malloc(sizeof(struct map));
27     initmap->m_size = 1000;
28     initmap->isused = 0;
29     initmap->m_addr = (char*) malloc(sizeof(char) * 1000);
30     initmap->prior = initmap;
31     initmap->next = initmap;
32     pointer = initmap;
33     printf("Environment Initialized Successfully\n");
34     printf("malloc 1000: %p\n", initmap->m_addr);
35 }
36
37 //打印当前空闲存储区
38 void print_freemap() {
39     struct map *ptr = pointer;
40     printf ("-----\n");
41     int cnt = 1;
42     printf("Current Free Memory: num address size\n");
```

```

43     do {
44         if(ptr->isused == 0){
45             printf("Current Free Memory: %d\t%u\t%d\n", cnt, ptr->m_addr,
ptr->m_size);
46             ++cnt;
47         }
48         ptr = ptr->next;
49     } while (ptr != pointer);
50     printf ("-----\n");
51 };
52
53 //程序说明
54 void print_help(){
55     printf("-----Help-----\n");
56     printf("alloc memory:  \m [size] \ ' \n");
57     printf("free memory:  \f [size] [address]\ ' \n");
58     printf("quit:  \q\ ' \n");
59     printf("-----end-----\n");
60 }
61
62 //接收输入
63 void handle_input(){
64     char choice;
65     unsigned addr,size;
66     do{
67         scanf("%c",&choice);
68         switch (choice){
69             case 'm':
70                 scanf("%u",&size);
71                 printf("alloc size %u \n",size);
72                 lmalloc(size);
73                 print_freemap();
74                 break;
75             case 'f':
76                 scanf("%u %u", &size, &addr);
77                 printf("free size:%u, addr:%u\n",size,addr);
78                 bool isFree = lfree(size, (char*) addr);
79                 if(isFree){
80                     print_freemap();
81                 }
82                 break;
83             case 'q':
84                 printf("Bye~\n");
85                 exit(0);
86                 break;
87             default:
88                 if(choice != '\n' && choice != '\t' && choice != ' ')
89                     print_help();
90                 break;
91         }
92     }while(true);
93 }
94
95 char* lmalloc(unsigned size) {
96     bool flag = false;
97     struct map *ptr = pointer;
98     struct map *new_map = malloc(sizeof(struct map));
99     do {

```

```

100         if (pointer->m_size >= size) {
101             new_map->m_size = size;
102             new_map->m_addr = ptr->m_addr;
103             new_map->isUsed = 1;
104             new_map->prior = ptr->prior;
105             new_map->next = ptr;
106
107             pointer->m_size -= size;
108             pointer->m_addr = (char*) ((unsigned) pointer->m_addr + size);
109             ptr->prior->next = new_map;
110             pointer->prior = new_map;
111             flag = true;
112             break;
113         }
114         else pointer = pointer->next;
115     } while (pointer != ptr);
116     if(flag)
117         return new_map->m_addr;
118     else{
119         printf("err: Failed to alloc memory--space is not enough.\n");
120         free(new_map);
121         return NULL;
122     }
123 }
124
125 bool lfree(unsigned size, char *addr){
126     //检查size是否为一个区块的地址
127     struct map *ptr = initmap;
128     bool isFind = false;
129     do {
130         if(ptr->m_addr == addr && ptr->isUsed == 1){
131             isFind = true;
132             break;
133         }
134         ptr = ptr->next;
135     } while (ptr != initmap);
136     //如果free地址不是map地址，则报错
137     if(!isFind){
138         printf("err : Try to free invalid address!\n");
139         return false;
140     }
141     //如果free大小大于目标map的大小，报错
142     if(size > ptr->m_size){
143         printf("err: Invalid size, too big!\n");
144         return false;
145     }
146     //free大小与目标map大小相等
147     if(size == ptr->m_size){
148         struct map *front = ptr->prior;
149         struct map *back = ptr->next;
150         //如果后为占用区或ptr为队尾
151         if(back->isused == 1 || back->m_addr < ptr->m_addr){
152             if(front->isUsed == 1){
153                 ptr->isUsed = 0;
154             }
155             else{
156                 front->m_size += size;
157                 front->next = back;

```

```

158         back->prior = front;
159         free(ptr);
160     }
161 }
162 //后为空闲区且ptr不在队尾
163 else{
164     //前为占用区或ptr在队首
165     if(front->isUsed == 1 || front->m_addr > ptr->m_addr){
166         ptr->m_size += back->m_size;
167         ptr->isUsed = 0;
168         ptr->next = back->next;
169         back->next->prior = ptr;
170         free(back);
171         pointer = ptr;
172     }
173     else{
174         front->m_size = front->m_size + size + back->m_size;
175         front->next = back->next;
176         back->next->prior = front;
177         free(ptr);
178         free(back);
179         pointer = front;
180     }
181 }
182 }
183 //free大小小于目标map大小
184 else{
185     struct map *front = ptr->prior;
186     //前为占用区或者ptr为头部
187     if(front->isUsed == 1 || front->m_addr > ptr->m_addr){
188         struct map *new_map = malloc(sizeof(struct map));
189         new_map->isUsed = 0;
190         new_map->m_addr = ptr->m_addr;
191         new_map->m_size = size;
192         new_map->next = ptr;
193         new_map->prior = front;
194
195         front->next = new_map;
196         ptr->m_size -= size;
197         ptr->m_addr = (char*) ((unsigned) ptr->m_addr + size);
198         ptr->prior = new_map;
199     }
200     //前为空闲区且ptr不是头部
201     else{
202         front->m_size += size;
203         ptr->m_size -= size;
204         ptr->m_addr = (char*) ((unsigned) ptr->m_addr + size);
205     }
206 }
207 printf("Freed successfully\n");
208 return true;
209 }
210
211 int main(){
212     initialize();
213     printf("Initial address: %u\n", initmap->m_addr);
214     print_help();
215     handle_input();

```

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