操作系统实验 1：可变分区存储管理

王靖康
515030910059
网络空间安全学院
wangjksjtu_01@sjtu.edu.cn

1 实验题目

编写一个 C 程序，用 char *malloc(unsigned size) 函数向系统申请一次内存空间（如 size=1000，单位为字节），用循环首次适应法 addr = (char*) lmalloc(unsigned size) 和 lfree(unsigned size,char *addr) 模拟 UNIX 可变分区内存管理，实现对该内存区的分配和释放管理。

2 实验目的

- 加深对可变分区存储管理的理解;
- 提高用 C 语言编制大型系统程序的能力，特别是掌握 C 语言编程的难点：指针和结构体作为函数参数;
- 掌握用指针实现链表和在链表上的基本操作。

3 算法思想

3.1 可变分区存储

与固定分区存储不同，可变分区存储管理并不预先将内存划分成分区，而是等到作业运行需要内存时就向系统申请，从空闲的内存区中申请占用空间，其大小等于作业所需内存大小, 如图 1 所示。

因此，可变分区存储不会产生“内零头”。每一个空闲分区使用一个 map 结构来进行管理，每个空闲分区按起始地址由低到高顺次登记在空闲分区存储表中。

3.2 管理方法

目前常用的几种管理方法有（如图 2 所示）：

- **首次适应算法 (First Fit)**：该算法从空闲分区链首开始查找，直至找到一个能满足其大小要求的空闲分区为止。然后再按照作业的大小，从该分区中划出一块内存分配给请求者，余下的空闲分区仍留在空闲分区链中。

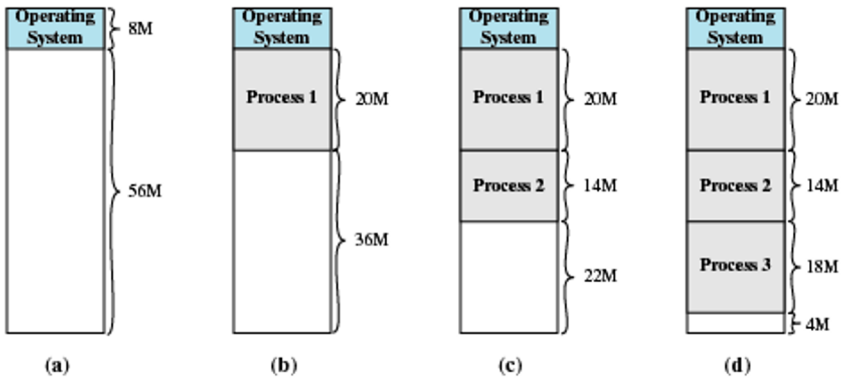

Figure 1: 动态分区示例 (64M Memory)

- **循环首次适应算法 (Next Fit)**：该算法是由首次适应算法演变而成的。在为进程分配内存空间时，不再每次从链首开始查找，直至找到一个能满足要求的空闲分区，并从中划出一块来分给作业。
- **最佳适应算法 (Best Fit)**：该算法总是把既能满足要求，又是最小的空闲分区分配给作业。为了加速查找，该算法要求将所有的空闲区按其大小排序后，以递增顺序形成一个空白链。这样每次找到的第一个满足要求的空闲区，必然是最优的。
- **最坏适应算法 (Worst Fit)**：该算法按大小递减的顺序形成空闲区链，分配时直接从空闲区链的第一个空闲区中分配（不能满足需要则不分配）。很显然，如果第一个空闲分区不能满足，那么再没有空闲分区能满足需要。这种分配方法初看起来不太合理，但它也有很强的直观吸引力：在大空闲区中放入程序后，剩下的空闲区常常也很大，于是还能装下一个较大的新程序。

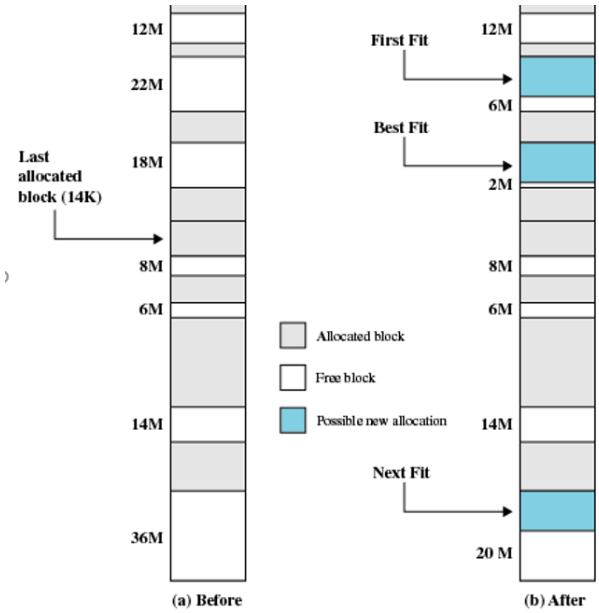

Figure 2: 几种常见的动态分区管理法

3.3 循环首次适应法

本次实验使用循环首次适应。即把空闲表设计成顺序结构或双向链接结构的循环队列，各空闲区仍按地址从低到高的次序登记在空闲区的管理队列中。同时需要设置一个起始查找指针，指向循环队列中的一个空闲区表项。循环首次适应法分配时总是从起始查找指针所指的表项开始查找。第一次找到满足要求的空闲区时，就分配所需大小的空闲区，修改表项，并调整起始查找指针，使其指向队列中被分配的后面的那块空闲区。下次分配时就从新指向的那块空闲区开始查找。

释放算法同首次适应法一样：第一次释放时，如果被释放的区域不与空闲分区相邻，此时需要修改空闲区的首表项，并将其加入空闲分区存储表中。如果被释放的区域与空闲分区相邻，则直接修改首表项的地址以及大小。非首次释放时，释放区与原空闲区相邻情况可归纳为四种情况（如图 3 所示）。

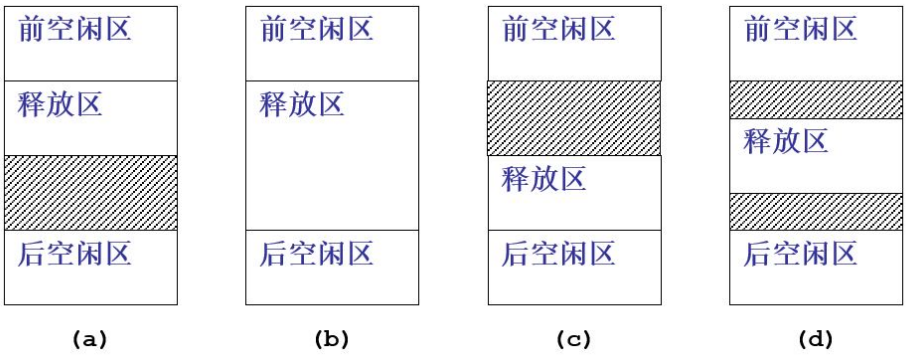

Figure 3: 释放区与原空闲区相邻的四种情况

(a) **仅与前空闲区相连**：合并前空闲区和释放区，构成一块大的新空闲区，并修改空闲区表项。该空闲区的 m_addr 不变，仍为原前空闲区的首地址，修改表项的长度域 m_size 为原 m_size 与释放区长度之和。

(b) **同时与前空闲区和后空闲区都相连**：将三块空闲区合并成一块空闲区。修改空闲区表中前空闲区表项，其始地址 m_addr 仍为原前空闲区始址，其大小 m_size 等于三个空闲区长度之和。在空闲区表中删除后项（释放空间）。

(c) **仅与后空闲区相连**：与后空闲区合并，使后空闲区表项的 m_addr 为释放区的始址，m_size 为释放区与后空闲区的长度之和。

(d) **与前、后空闲区皆不相连**：在前、后空闲区表项中间插入一个新的表项，其 m_addr 为释放区的始址，m_size 为释放区的长度。为此，先要将后项及以下表项都下移一个位置。

4 算法实现

本次实验程序分为算法实现和重定向测试两大部分。本节讨论算法实现部分。本实验全部使用 C 语言完成，主要分为三大模块：1）输入控制模块；2）内存分配、释放模块；3）打印模块。这三个模块分别实现了对输入流的控制，模拟循环首次适应发内存的分配及释放和打印进程列表、空闲列表等功能。具体来说：

- 输入控制模块实现了和用户的交互，对不同输入使用分支控制语句（case）和多重条件语句进行控制；
- 内存分配、释放模块实现了循环首次适配算法的模拟，包括模拟进程的释放，空闲块的合并等（四种情况）；
- 打印模块实现了常见字符串打印帮助函数和对进程管理列表和空闲列表的打印（id、地址、大小）。

4.1 数据结构

本次实验主要数据结构有：1）空闲块结构体 (map): 包含每个空闲块的大小、起始地址、指向前空闲块和后空闲块的指针。2）进程块结构体 (process): 包含每个分配进程的 id（释放时需要指明 id）、起始地址和每个模拟进程所占大小。程序中结构体定义如下所示：

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

struct process {
    unsigned id;
    char* m_addr;
    unsigned m_size;
};
```

4.2 模块与接口设计

本小节简要介绍程序中除循环首次适应算法分配和释放模块外的其他模块。包括：1）初始化模块：初始化变量、模拟申请 1000 个字节的空间。2）输入控制模块：使用分支语句 case 和多层条件语句完成对程序输入流的控制（包括算法选择，操作选择）。3）打印模块：打印进程列表和空闲块列表（id，起始地址，内存大小）。

初始化模块

```
250 void initialize(struct map **coremap_out) {
251     // static struct map* coremap;
252     coremap = malloc(sizeof(struct map));
253     coremap->m_size = 1000;
254     coremap->m_addr = (char*) malloc(sizeof(int) * 1000);
255     coremap->prior = coremap;
256     coremap->next = coremap;
257     *coremap_out = coremap;
258     pointer = coremap;
259     // printf("%p\n", &coremap);
260     printf("Hello, nice to meet you!\n");
261     printf("malloc 1000: %p\n", coremap->m_addr);
262 }
```

Figure 4: 初始化模块：实现初始空间申请，指针初始化

输入控制模块

```
198 void malloc_or_free(struct map **coremap, struct map **pointer) {
199     printf("1->malloc(FF); 2->free\n");
200     int choice, size, id;
201     if (scanf("%d", &choice)) {
202         if (choice == 1) {
203             printf ("malloc memory size: ");
204             scanf("%d", &size);
205             if (!lmalloc(size, pointer, choice))
206                 printf ("No enough memory!\n");
207         }
208         else {
209             printf ("free process id: ");
210             scanf("%d", &id);
211             if (!lfree(id, coremap, choice)) {
212                 printf ("The process is not existant!\n");
213             }
214         }
215     }
216     print_map(*coremap);
217     // printf("%p\n", (*coremap)->m_addr);
218     print_process(p_manager);
219 }
```

Figure 5: 输入控制模块: 实现输入控制（算法、操作）

打印模块

```
39  void print_map(struct map* coremap) {
40      struct map *ptr = coremap;
41      printf ("Free memory\n");
42      print_line(28, '-');
43      int cnt = 0;
44      do {
45          printf("%d\t%p\t%d\n", cnt, ptr->m_addr, ptr->m_size);
46          ptr = ptr->next;
47          ++cnt;
48      } while (ptr != coremap);
49      print_line(28, '-');
50  };
51
52  void print_process(struct process* p_manager) {
53      printf ("Process\n");
54      print_line(28, '-');
55      for (int i = 0; i < count_process; i++) {
56          struct process proc = p_manager[i];
57          printf("%d\t%p\t%d\n", proc.id, proc.m_addr,
            proc.m_size);
58      }
59      print_line(28, '-');
60  }
```

Figure 6: 打印模块: 打印进程列表和空闲块列表

4.3 循环首次适应算法实现

内存分配和释放算法的实现是本次实验的核心部分，也是区别首次适应法、循环首次适应法、最佳适应发、最差适应法的关键。由于时间有限，本次实验仅实现首次适应法和循环首次适应法 (二者的唯一区别在于分配的起始位置)，而为最佳适应法和最差适应法保留接口，便于之后加入。

分配内存 (lmalloc): 当进程请求分配内存时，从上次分配内存位置开始，循环查找空闲块，当空闲块大小大于进程请求空间时，使用 malloc 函数为进程分配内存，设定进程的起始地址、大小，同时更新空闲块的起始地址大小等，如图 7 所示。如未找到符合要求的块，则返回 false，交给控制模块处理。

```
62  bool lmalloc(size_t size, struct map **pointer, int choice) {
63      bool flag = false;
64      struct map *ptr = *pointer;
65      do {
66          if ((*pointer)->m_size >= size) {
67              struct process* new_proc = malloc(sizeof(struct process));
68              new_proc->m_size = size;
69              new_proc->m_addr = ptr->m_addr;
70              new_proc->id = current_id;
71              p_manager[count_process] = *new_proc;
72              current_id++;
73              count_process++;
74              flag = true;
75              (*pointer)->m_size -= size;
76              (*pointer)->m_addr = (char*) ((unsigned)(*pointer)->m_addr + size);
77              break;
78          }
79          else (*pointer) = (*pointer)->next;
80      } while ((*pointer) != ptr);
81      return flag;
82  }
```

Figure 7: 循环首次适应内存分配实现

值得说明的是，对于首次适应法，只需要将全局指针 pointer（指向上一次内存分配的位置）替换为全局指针 coremap（指向地址最低的空闲块）即可。

释放内存 (lfree): 如图 3 所示，内存释放较为复杂，分为四种情况。每种情况对指针的处理均不同。为了简化四种情况的判定和处理，首先对空闲块列表扫查一遍确定待释放进程相邻的空闲块（称为前空闲块、后空闲块）。因此，释放区与原空闲区相邻的四种情况可以根据前空闲块是否为空分成两大类（如图 9，10 所示）。

5 测试方法

本次实验采用了两种测试方法，分别是交互式测试（即用户每次选择算法、选择释放或申请内存，选择申请内存大小或释放进程 ID）和文件测试（在 Linux 下使用重定向实现，更全面的测试用例）。

在测试前需要编译 c 程序，相关命令为：$ gcc memory_alloc.c -o memory -Wall

交互式测试的内存分配和释放流程如图 11 所示，结果见附录 7:

文件测试的相关命令为: $ ./memory < test > result，即将 test 文件重定向到标准输入中，再将标准输出重定向到 result 文件。

```
 99  bool lfree(int id, struct map **coremap, int choice) {
100      struct process *free_proc;
101      if (id > current_id || get_index(id) == -1) return false;
102      else free_proc = search_process(id);
103
104      // printf ("%d\n", free_proc->m_size);
105      struct map *ptr = *coremap;
106
107      struct map *upper = NULL;
108      struct map *lower = NULL;
109      unsigned addr_proc = free_proc->m_addr;
110
111
112      do {
113          unsigned addr_ptr =  ptr->m_addr;
114          if (addr_ptr < addr_proc) lower = ptr;
115          if (addr_ptr > addr_proc && upper == NULL) upper = ptr;
116          ptr = ptr->next;
117      } while (ptr != *coremap);
```

Figure 8: 查找释放进程的所在位置相邻两块空闲块

```
122      if (upper == NULL || lower == NULL) {
123          if (upper == NULL) {
124              // printf ("upper is empty\n");
125              if ((unsigned) lower->m_addr + lower->m_size == addr_proc) {
126                  lower->m_size += free_proc->m_size;
127              }
128              else {
129                  struct map* newmap = malloc(sizeof(struct map));
130                  lower->next = newmap;
131                  newmap->prior = lower;
132                  newmap->next = lower->next;
133                  newmap->m_addr = addr_proc;
134                  newmap->m_size = free_proc->m_size;
135              }
136
137          }
```

Figure 9: 循环首次适应内存释放（前无空闲块）

```
    else {
        // printf ("lower is empty\n");
        if ((unsigned) upper->m_addr - free_proc->m_size == addr_proc) {
            upper->m_size += free_proc->m_size;
            upper->m_addr = (char*) ((unsigned)upper->m_addr - free_proc->m_size);
        }
        else {
            // printf ("new map\n");
            struct map* newmap = malloc(sizeof(struct map));
            upper->prior = newmap;
            if (upper->next == upper) upper->next = newmap;
            newmap->next = upper;
            newmap->prior = upper->prior;
            newmap->m_addr = addr_proc;
            newmap->m_size = free_proc->m_size;
            *coremap = newmap;
            // printf ("%d\t%d\t%p\n", (*coremap)->m_size, (*coremap)->next->m_size,
            // printf ("%p\t%p\n", (*coremap)->next->m_addr, (*coremap)->next->next-
        }
    }
```

Figure 10: 循环首次适应内存释放（前有空闲块）

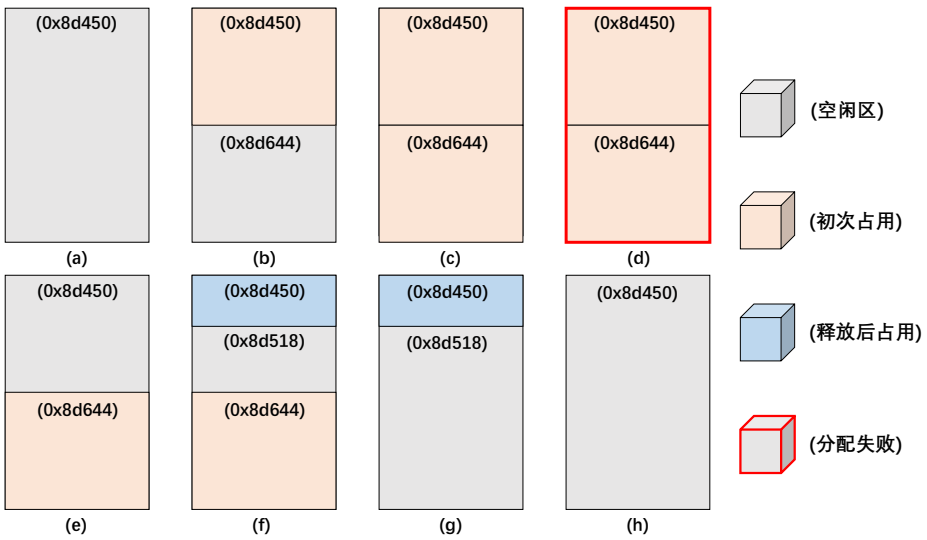

Figure 11: 交互式测试的测试流程

6 总结与思考

6.1 错误分析

在本次实验中遇到了较多的问题，其中绝大多数均与程序语法相关。这主要是由于对 C 语言的不熟悉以及 C++ 的混淆造成的，因此在今后的学习中，需要加强对 C 语言的理解。通过不断的发现错误，解决错误，提升开发大型程序的能力。

指针的使用：本次实验很重要的一部分就是指针的使用，需要构造两个个双向链表，完成对进程列表和内存空闲区域的模拟。在实际编程过程中，对指针使用的要求十分灵活，需要熟练掌握链表的插入，删除，遍历等操作。

另外，程序中对 coremap 变量和 pointer 变量还使用了二级指针。这是因为在分配和释放内存需要对 *coremap，*pointer 指针的地址进行修改，故需要将二级指针作为函数形参，将 *coremap 取地址作为实参，从而实现对指针地址的修改。对指针以及地址的理解十分重要，直接决定程序能否正常执行。**指针的值是地址，指针的解引用是指针地址指向的内容**。

C 语言与 C++ 的差异：由于并未系统学习过 C 语言，这使得在初次编程过程中容易

6.2 程序改进

- 优化 IO 交互过程，使得可以从文件批量输入和将内存分配过程批量输出到指定文件，增加自动化测试函数。
- 对全局变量和无用变量的剔除和改写，完善函数调用接口。
- 支持最佳适配和最差适配分配法。
- 考虑更完善，更通用的的内存释放（即进程在运行中动态申请和释放内存），并完善异常情况处理代码部分。

6.3 个人总结

本次实验总体来说难度适中，在规定的时间内可以较为顺利的完成，同时该实验也是很有意义的。首先，实验加深对可变分区存储管理的理解；其次，本次实验提高了我们用 C 语言编制大型系统程序的能力，特别是掌握 C 语言编程的难点：指针和结构体作为函数参数；最后，实验也促使我们掌握用指针实现链表和在链表上的基本操作，提高了编写 C 语言程序的信心。在本次实验后，我也紧接着又完成计算机病毒 C++ 的 qt5 编程。

在本次实验中，我特别要感谢的是《C++ Primer》书籍的指导和室友吕正同学（软件学院）的帮助。我对指针的正确理解和应用是在自学和讨论中进一步深化的，吕同学为我指出了 C 和 C++ 的常见区别。

最后，感谢刘老师的指导和帮助，算法的正确理解和实验的顺利进行都与他的帮助是分不开的。

7 附录

7.1 程序源代码

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

#define MAX 100

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

struct process {
    unsigned id;
    char* m_addr;
    unsigned m_size;
};

struct map *coremap;
struct map *pointer;
struct process p_manager[MAX];
int count_process = 0;
int current_id = 0;

void print_choice(void) {
    printf("1:First Fit(FF); 2:Next Fit(NF); \
3:Best Fit(BF); 4:Worst Fit(WF); Others:Quit\n");
}

void print_line(int count, char c) {
```

```
31      char line[MAX];
32      for (int i = 0; i < count; ++i) {
33          line[i] = c;
34      }
35      line[count] = '\0';
36      printf("%s\n", line);
37  }
38
39  void print_map(struct map* coremap) {
40      struct map *ptr = coremap;
41      printf ("Free memory\n");
42      print_line(28, '-');
43      int cnt = 0;
44      do {
45          printf("%d\t%p\t%d\n", cnt, ptr->m_addr, ptr->m_size);
46          ptr = ptr->next;
47          ++cnt;
48      } while (ptr != coremap);
49      print_line(28, '-');
50  };
51
52  void print_process(struct process* p_manager) {
53      printf ("Process\n");
54      print_line(28, '-');
55      for (int i = 0; i < count_process; i++) {
56          struct process proc = p_manager[i];
57          printf("%d\t%p\t%d\n", proc.id, proc.m_addr, proc.m_size);
58      }
59      print_line(28, '-');
60  }
61
62  bool lmalloc(size_t size, struct map **pointer, int choice) {
63      bool flag = false;
64      struct map *ptr = *pointer;
65      do {
66          if ((*pointer)->m_size >= size) {
67              struct process* new_proc = malloc(sizeof(struct process));
68              new_proc->m_size = size;
69              new_proc->m_addr = ptr->m_addr;
70              new_proc->id = current_id;
71              p_manager[count_process] = *new_proc;
72              current_id++;
73              count_process++;
74              flag = true;
75              (*pointer)->m_size -= size;
76              (*pointer)->m_addr = (char*) ((unsigned)(*pointer)->m_addr +
    size);
77              break;
```

```
78          }
79          else (*pointer) = (*pointer)->next;
80      } while ((*pointer) != ptr);
81      return flag;
82  }
83
84  struct process* search_process(int id) {
85      for (int i = 0; i < count_process; i++) {
86          if (p_manager[i].id == id)
87              return &(p_manager[i]);
88      }
89  }
90
91  int get_index(int id) {
92      for (int i = 0; i < count_process; i++) {
93          if (p_manager[i].id == id)
94              return i;
95      }
96      return -1;
97  }
98
99  bool lfree(int id, struct map **coremap, int choice) {
100     struct process *free_proc;
101     if (id > current_id || get_index(id) == -1) return false;
102     else free_proc = search_process(id);
103
104     // printf ("%d\n", free_proc->m_size);
105     struct map *ptr = *coremap;
106
107     struct map *upper = NULL;
108     struct map *lower = NULL;
109     unsigned addr_proc = free_proc->m_addr;
110
111
112     do {
113         unsigned addr_ptr =  ptr->m_addr;
114         if (addr_ptr < addr_proc) lower = ptr;
115         if (addr_ptr > addr_proc && upper == NULL) upper = ptr;
116         ptr = ptr->next;
117     } while (ptr != *coremap);
118
119     // if (upper) printf("%d\n", upper->m_size);
120     // if (lower) printf("%d\n", lower->m_size);
121
122     if (upper == NULL || lower == NULL) {
123         if (upper == NULL) {
124             // printf ("upper is empty\n");
125             if ((unsigned) lower->m_addr + lower->m_size == addr_proc) {
```

```
126                 lower->m_size += free_proc->m_size;
127             }
128             else {
129                 struct map* newmap = malloc(sizeof(struct map));
130                 lower->next = newmap;
131                 newmap->prior = lower;
132                 newmap->next = lower->next;
133                 newmap->m_addr = addr_proc;
134                 newmap->m_size = free_proc->m_size;
135             }
136 
137         }
138         else {
139             // printf ("lower is empty\n");
140             if ((unsigned) upper->m_addr - free_proc->m_size == addr_proc
    ) {
141                 upper->m_size += free_proc->m_size;
142                 upper->m_addr = (char*) ((unsigned)upper->m_addr -
    free_proc->m_size);
143             }
144             else {
145                 // printf ("new map\n");
146                 struct map* newmap = malloc(sizeof(struct map));
147                 upper->prior = newmap;
148                 if (upper->next == upper) upper->next = newmap;
149                 newmap->next = upper;
150                 newmap->prior = upper->prior;
151                 newmap->m_addr = addr_proc;
152                 newmap->m_size = free_proc->m_size;
153                 *coremap = newmap;
154                 // printf ("%d\t%d\t%p\n", (*coremap)->m_size, (*coremap)
    ->next->m_size, (*coremap)->m_addr);
155                 // printf ("%p\t%p\n", (*coremap)->next->m_addr, (*
    coremap)->next->next->m_addr);
156             }
157         }
158     }
159     else {
160         // printf ("both upper and lower are not empty\n");
161         // printf ("%p\t%p\t%p\t\n", lower->m_addr, addr_proc, upper->
    m_addr);
162         if ((unsigned) lower->m_addr + lower->m_size == addr_proc) {
163             if (addr_proc + free_proc->m_size == (unsigned) upper->m_addr
    ) {
164                 // printf ("merge!");
165                 lower->m_size += free_proc->m_size + upper->m_size;
166                 lower->next = upper->next;
167                 if (lower->prior == upper) lower->prior = lower;
```

```
168                     free(upper);
169                 }
170                 else lower->m_size += free_proc->m_size;
171             }
172             else {
173                 if ((unsigned)upper->m_addr - free_proc->m_size == addr_proc)
     {
174                     upper->m_size += free_proc->m_size;
175                     upper->m_addr = (char*) upper->m_addr - free_proc->m_size
    ;
176                 }
177                 else {
178                     struct map* newmap = malloc(sizeof(struct map));
179                     newmap->m_addr = addr_proc;
180                     newmap->m_size = free_proc->m_size;
181                     newmap->next = upper;
182                     newmap->prior = lower;
183                     lower->next = newmap;
184                     upper->prior = newmap;
185                 }
186 
187             }
188             // exit(0);
189         }
190         int free_id = get_index(id);
191         for (int i = free_id; i < count_process - 1; i++) {
192             p_manager[i] = p_manager[i+1];
193         }
194         count_process -= 1;
195         return true;
196 }
197 
198 void malloc_or_free(struct map **coremap, struct map **pointer) {
199     printf("1->malloc(FF); 2->free\n");
200     int choice, size, id;
201     if (scanf("%d", &choice)) {
202         if (choice == 1) {
203             printf ("malloc memory size: ");
204             scanf("%d", &size);
205             if (!lmalloc(size, pointer, choice))
206                 printf ("No enough memory!\n");
207         }
208         else {
209             printf ("free process id: ");
210             scanf("%d", &id);
211             if (!lfree(id, coremap, choice)) {
212                 printf ("The process is not existant!\n");
213             }
```

```
214         }
215     }
216     print_map(*coremap);
217     // printf("%p\n", (*coremap)->m_addr);
218     print_process(p_manager);
219 }
220
221 void control_switch(int choice, struct map **coremap, struct map **
        pointer) {
222     while (true) {
223         switch(choice) {
224             case 1:
225                 printf("First Fit\n");
226                 // first_fit(sizeof(int), coremap);
227                 break;
228             case 2:
229                 // printf("Next Fit\n");
230                 print_map(*coremap);
231                 malloc_or_free(coremap, pointer);
232                 // printf("%p\n", (*coremap)->m_addr);
233                 printf("\n");
234                 // print_choice();
235                 break;
236             case 3:
237                 printf("Best Fit\n");
238                 break;
239             case 4:
240                 printf("Worst Fit\n");
241                 break;
242             default:
243                 printf("Byebye~\n");
244                 exit(0);
245         }
246         scanf("%d", &choice);
247     }
248 }
249
250 void initialize(struct map **coremap_out) {
251     // static struct map* coremap;
252     coremap = malloc(sizeof(struct map));
253     coremap->m_size = 1000;
254     coremap->m_addr = (char*) malloc(sizeof(int) * 1000);
255     coremap->prior = coremap;
256     coremap->next = coremap;
257     *coremap_out = coremap;
258     pointer = coremap;
259     // printf("%p\n", &coremap);
260     printf("Hello, nice to meet you!\n");
```

```
261     printf("malloc 1000: %p\n", coremap->m_addr);
262 }
263
264 void test(int* p) {
265     printf("%p\n", &p);
266
267 }
268
269 int main() {
270     printf("%p\n", &coremap);
271     initialize(&coremap);
272     print_choice();
273     int choice;
274     scanf("%d", &choice);
275     control_switch(choice, &coremap, &pointer);
276     return 0;
277 }
```

7.2 程序运行及测试截图

```
Hello, nice to meet you!
malloc 1000: 0xc8d450
1:First Fit(FF); 2:Next Fit(NF); 3:Best Fit(BF); 4:Worst Fit(WF); Others:Quit
2 1 500
Free memory
----------------------------
0       0xc8d450        1000
----------------------------
1->malloc(FF); 2->free
malloc memory size: Free memory
----------------------------
0       0xc8d644        500
----------------------------
Process
----------------------------
0       0xc8d450        500
----------------------------

2 1 500
Free memory
----------------------------
0       0xc8d644        500
----------------------------
1->malloc(FF); 2->free
malloc memory size: Free memory
----------------------------
0       0xc8d838        0
----------------------------
Process
----------------------------
0       0xc8d450        500
1       0xc8d644        500
----------------------------
```

```
2 1 200
Free memory
----------------------------
0       0xc8d838        0
----------------------------
1->malloc(FF); 2->free
malloc memory size: No enough memory!
Free memory
----------------------------
0       0xc8d838        0
----------------------------
Process
----------------------------
0       0xc8d450        500
1       0xc8d644        500
----------------------------

2 2 0
Free memory
----------------------------
0       0xc8d838        0
----------------------------
1->malloc(FF); 2->free
free process id: lower is empty
new map
Free memory
----------------------------
0       0xc8d450        500
1       0xc8d838        0
----------------------------
Process
----------------------------
1       0xc8d644        500
----------------------------
```

```
2 1 200
Free memory
----------------------------
0       0xc8d450        500
1       0xc8d838        0
----------------------------
1->malloc(FF); 2->free
malloc memory size: Free memory
----------------------------
0       0xc8d518        300
1       0xc8d838        0
----------------------------
Process
----------------------------
1       0xc8d644        500
2       0xc8d838        200
----------------------------

2 2 1
Free memory
----------------------------
0       0xc8d518        300
1       0xc8d838        0
----------------------------
1->malloc(FF); 2->free
free process id: both upper and lower are not empty
0xc8d518        0xc8d644        0xc8d838
merge!Free memory
----------------------------
0       0xc8d518        800
----------------------------
Process
----------------------------
2       0xc8d838        200
----------------------------
```

```
2 2 2
Free memory
----------------------------
0       0xc8d518        800
----------------------------
1->malloc(FF); 2->free
free process id: upper is empty
here!Free memory
----------------------------
0       0xc8d518        1000
----------------------------
Process
----------------------------
----------------------------
```