
操作系统实验报告

实验一 可变分区存储管理

郑宇森

520021911173

电子信息与电气工程学院

zys0794@sjtu.edu.cn

2022 年 10 月 21 日

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

1. 加深对可变分区的存储管理的理解；
2. 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：指针和指针作为函数参数；
3. 掌握用指针实现链表和在链表上的基本操作。

3 实验要求

空闲存储区表可采用**结构数组**的形式（基本要求）或**双向链接表**（本实验中采取该数据结构）的形式（提高一步），建议采用的数据结构为：

```
1 # 结构数组的形式
2 struct map {
3     unsigned m_size;
4     char * m_addr;
5 };
6 struct map coremap[N];
```

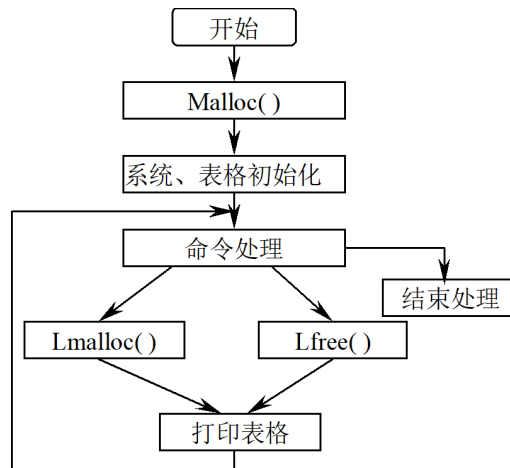


图 1: 系统基本框架

```

7 # 双向链接表的形式
8 struct map {
9     unsigned m_size;
10    char *m_addr;
11    struct map *next, *prior;
12 };
13 struct map *coremap;
  
```

要分配函数 `lmalloc` 的参数 `size` 和释放函数 `lfree` 的参数 `size`、`addr` 以键盘命令的形式输入，每次分配和释放后显示自己的空闲存储区表。系统基本框架如图 1 所示。

程序调试基本通过后，应进行全面的测试，采用白盒法的路径测试方法，测试路径包含 `lmalloc` 的“循环”、“首次”，`lfree` 的与邻近空闲分区联系的四种情况，还要包含必要的出错处理路径。

4 算法思想

本次实验中，我们使用**循环首次适应法**进行可变分区的存储管理。循环首次适应法将空闲表设计成顺序结构或链接结构的循环队列，各空闲区仍按照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，指向循环队列中的第一个空闲区表项。

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。

循环首次适应法的释放算法同首次适应法一样，分 4 种情况，如下所述。释放时当需要在顺序方法实现的空闲队列中插入一个表项时，根据该表项与起始查找指针之间的相对位置，可能需要修改指针值，使其仍旧指向原空闲表项。

- 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。
- 与前后空闲区均相连：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。
- 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。
- 与前后空闲区均不相连：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。

5 算法实现

本实验中我们主要实现了如下 5 个模块/接口：

- **初始化模块**：申请实验所需的模拟内存分区空间，并初始化空闲区结构体和起始查找指针。
- **内存分配/释放模块**：使用循环首次适应法分配/释放内存。该模块对非法输入和特殊情况有完备的处理逻辑。
- **打印模块**：打印内存中所有空闲分区的详细信息。
- **用户输入接口**：提供命令行用户交互接口，用户可以通过指令对内存空间进行操作。
- **测试模块**：对内存分配/释放模块的正确性和鲁棒性进行测试。

5.1 数据结构和初始化模块

我们采用**双向链接表**的数据结构记录空闲分区。模拟的内存分区大小定义在宏常量 `CORE_SIZE` 中。**空闲区结构体** `map` 包括四条记录，分别为该空闲分区大小 (`m_size`)、指向空闲分区地址的指针 (`m_addr`)、指向前一空

闲分区结构体的指针 (prior) 和指向后一空闲分区结构体的指针 (next)。循环首次适应法中需要维护一个起始查找指针，我们将其声明为 coremap。

```

1 #define CORE_SIZE 1000
2
3 struct map {
4     unsigned m_size;
5     char *m_addr;
6     struct map *next, *prior;
7 };
8 // Lookup pointer to the starting position
9 struct map *coremap;
10 // Memory start address
11 char *start_addr;

```

在初始化 coremap 时,我们先为结构体分配大小为 sizeof(struct map) 内存空间,再为整个模拟内存分配大小为 CORE_SIZE 内存空间。并更新起始查找指针的表项。注意到在循环首次适应法中,我们需要把空闲表设计成顺序结构或链接结构的循环队列。因此初始时 coremap 的 prior 和 next 指针都应指向自身。完成初始化后,打印分配得到的模拟内存的信息。

```

1 void init_coremap(unsigned size) {
2     coremap = (struct map *)malloc(sizeof(struct map));
3     coremap->m_size = size;
4     coremap->m_addr = (char *)malloc(size);
5     coremap->next = coremap;
6     coremap->prior = coremap;
7     start_addr = coremap->m_addr;
8     printf("Simulation kernel space allocation succeeded!\n");
9     printf("kernel address: %p - %p, space "
10           "size: %u\n",
11           coremap->m_addr, coremap->m_addr + size, size);
12 }

```

5.2 分配算法

在该部分我们实现了 lmalloc(unsigned) 函数。

首先处理非法输入和无空闲分区的情况。

```

1 // wrong arguments
2 if (size <= 0 || size >= CORE_SIZE) {
3     printf(ANSI_COLOR_RED
4           "***** ERROR: wrong arguments! *****" ANSI_COLOR_RESET "\n");
5     return NULL;
6 }
7 // all memory has been allocated
8 if (coremap == NULL) {
9     printf(ANSI_COLOR_RED "***** ERROR: All kernel space has been used
10           ! "

```

```

10         "*****" ANSI_COLOR_RESET "\n");
11     return NULL;
12 }

```

分 3 种情况讨论。若当前分区空间大于作业所需空间，直接分配并修改剩余空闲分区大小；若当前分区空间等于作业所需空间，则分配全部内存空间，并释放当前的表项，将初始起始指针指向后一表项（此时需考虑一种特殊情况，即当前的表项为最后一个空闲分区时，需要将初始起始指针置为 NULL；若当前分区空间小于作业所需空间，则继续搜索空闲分区，直到找到合适的分区或回到初始位置（没有大小合适的空闲分区）。

```

1  char *addr;
2  struct map *p, *q;
3  p = q = coremap; // p move, q fix
4
5  do {
6      // current free partition size is appropriate
7      if (p->m_size >= size) {
8          addr = p->m_addr;
9          p->m_addr += size;
10
11         // the entire partition is allocated
12         if ((p->m_size -= size) == 0) {
13             // all memory has been allocated
14             if (p->next == p) {
15                 printf(ANSI_COLOR_YELLOW "***** WARNING: All kernel space
has been "
16                                     "used! *****" ANSI_COLOR_RESET "\n"
17             );
18             coremap = NULL;
19             } else {
20                 coremap = p->next;
21                 p->next->prior = p->prior;
22                 p->prior->next = p->next;
23             }
24             free(p);
25         }
26
27         printf(ANSI_COLOR_GREEN "Memory allocated succeeded, address: %p
- %p, "
28               "size: %u" ANSI_COLOR_RESET "\n",
29               addr, addr + size, size);
30         if (VERBOSE) {
31             print_free_mem();
32         }
33         return addr;
34     } else {
35         p = p->next;
36     }
37 }

```

```

36 } while (p != q);
37
38 printf(ANSI_COLOR_RED "***** ERROR: Required memory space [%d byte]
    is too "
39
    "large! *****" ANSI_COLOR_RESET "\n",
40
    size);
41 return NULL;

```

5.3 回收算法

在该部分我们实现了 `lfree(unsigned, char*)` 函数。
首先处理非法输入。

```

1 // wrong arguments
2 if (size <= 0 || size >= CORE_SIZE || addr == NULL) {
3     printf(ANSI_COLOR_RED
4
5         "***** ERROR: wrong arguments! *****" ANSI_COLOR_RESET "\n
6
7         ");
8     return;
9 }

```

若所有内存都被使用，此时释放空间后需要创建一个新的表项。

```

1 // all memory has been allocated
2 if (coremap == NULL) {
3     coremap = (struct map *)malloc(sizeof(struct map));
4     coremap->m_addr = addr;
5     coremap->m_size = size;
6     coremap->prior = coremap;
7     coremap->next = coremap;
8     if (VERBOSE) {
9         print_free_mem();
10    }
11    return;
12 }

```

找到一个合适的空闲分区 `p`，使得要释放的空间地址位于 `p` 与 `p->next` 指向的空闲分区的地址之间。此时需分 3 种情况讨论，见下方程序中的逻辑判断部分。

```

1 // find the right address, let addr between p->m_addr & p->next->
2 // m_addr
3 struct map *p = coremap;
4 while (!((p->m_addr < addr &&
5
6     p->next->m_addr > addr) || // e.g. .. 100 .. [250]
7
8     .. 300 ..
9
10    ((p->m_addr < addr || // e.g. [50] .. 100 ..
11
12    p->next->m_addr > addr) && // e.g. .. 800 [900]
13
14    p->m_addr >= p->next->m_addr))) { // = for only one node
15
16    p = p->next;
17 }

```

释放的空闲分区地址与模拟内存中的空闲分区地址间存在 4 种情况。
当释放分区与前后空闲分区均相邻时：

```

1  p->m_size += (size + p->next->m_size);
2  struct map *q = p->next;
3  p->next = q->next;
4  q->next->prior = p;
5  if (coremap == q) { // after free, need to change coremap pointer
6      val
7      coremap = p;
8  }
9  free(q);

```

当释放分区只与前空闲分区相邻时：

```

1  p->m_size += size;

```

当释放分区只与后空闲分区相邻时：

```

1  p->next->m_addr -= size;
2  p->next->m_size += size;

```

当释放分区与前后空闲分区均不相邻时：

```

1  struct map *newCoremap = (struct map *)malloc(sizeof(struct map))
2  ;
3  newCoremap->m_addr = addr;
4  newCoremap->m_size = size;
5  newCoremap->next = p->next;
6  newCoremap->prior = p;
7  p->next->prior = newCoremap;
8  p->next = newCoremap;

```

5.4 空闲分区状态打印模块

print_free_mem() 函数从起始查找指针（即 coremap）开始，遍历打印内存中所有的空闲分区。该函数也处理了内存分区被全部占用的特殊情况。

```

1 void print_free_mem() {
2     printf(ANSI_COLOR_BLUE
3         "*****Start Print*****" ANSI_COLOR_RESET
4         "\n");
5     // all memory has been allocated
6     if (coremap == NULL) {
7         printf("None available memory partition.\n");
8         return;
9     }
10    struct map *p = coremap;
11    printf("Available memory partition:\n");
12    do {

```

```

12     printf("memory address: %p - %p, size: %u\n", p->m_addr,
13           p->m_addr + p->m_size, p->m_size);
14     p = p->next;
15 } while (p != coremap);
16 printf(ANSI_COLOR_BLUE
17        "*****End Print*****" ANSI_COLOR_RESET "
18        \n");
19 return;
20 }

```

5.5 用户输入处理接口

input() 函数处理用户输入。进入该函数时会输出交互说明，用户输入 m <#size> 时进行内存分配，输入 f <#size> <addr> 时进行内存回收，输入 h 时打印帮助命令，输入 q 时结束输入。函数自动过滤掉空白字符和非法字符。

```

1 void input() {
2     char cmdchar;
3     unsigned size, addr; // addr is offset address
4     do {
5         printf("Input command, h help, q quit.\n");
6         do {
7             cmdchar = getchar();
8         } while (cmdchar == ' ' || cmdchar == '\t' || cmdchar == '\n');
9         switch (cmdchar) {
10            case 'm':
11                scanf("%u", &size);
12                lmalloc(size);
13                break;
14            case 'f':
15                scanf("%u%u", &size, &addr);
16                lfree(size, start_addr + addr);
17                break;
18            case 'h':
19                printf("memory allocate: m <#size>, memory free: f <#size> <addr>\n");
20                break;
21            case 'q':
22                break;
23            default:
24                continue;
25        }
26    } while (cmdchar != 'q');
27 }

```


5.6 测试模块

在通过 `input()` 函数处理用户输入进行测试外,我还直接提供了 `test()` 函数。通过修改该函数的内容可直接在程序中进行测试。

```

1 void test() {
2     // change the func to test memory allocation and free
3     char *m1 = malloc(100);
4     char *m2 = malloc(100);
5     char *m3 = malloc(100);
6     free(100, m1);
7     char *m4 = malloc(200);
8     char *m5 = malloc(300);
9     char *m6 = malloc(300); // no space
10    char *m7 = malloc(200);
11    free(100, m2);
12    free(300, m5);
13    free(200, NULL);           // wrong arguments
14    char *m8 = malloc(-100); // wrong arguments
15    free(200, m7);
16    char *m9 = malloc(100); // allocation after free
17    free(200, m4);
18    free(100, m9);
19    free(100, m3);
20 }

```

5.7 杂项

为了便于调试和观察 `malloc()`, `free()`, `print_free_mem()` 函数的工作情况,我对输出内容进行了着色区分。正确分配、回收内存时输出内容为绿色、打印空闲分区时输出为蓝色、warning 为黄色、error 为红色。我也提供了一个名为 `VERBOSE` 的常量,当其值为 1 时表明在输出过程中打印详细信息。这些内容都定义在宏中。

```

1 #define ANSI_COLOR_RED "\x1b[31m"
2 #define ANSI_COLOR_GREEN "\x1b[32m"
3 #define ANSI_COLOR_YELLOW "\x1b[33m"
4 #define ANSI_COLOR_BLUE "\x1b[34m"
5 #define ANSI_COLOR_RESET "\x1b[0m"
6
7 #define VERBOSE 1

```

6 程序测试

6.1 测试环境

主机: Windows 11

虚拟机：Ubuntu 20.04LTS, WSL(Windows Subsystem for Linux)

6.2 测试方法

本实验实现了两种测试方法：

- 调用 `input()` 函数，通过命令行交互进行测试（此时释放空间时输入偏移地址）。此方法支持使用 I/O 重定向，如 `./main < ./input.txt`。
- 修改并调用 `test()` 函数进行测试（此时释放空间时输入绝对地址）。

6.3 测试路径

本节选择一个典型测试路径进行介绍，我绘制了该用例的示意图如图 2。注意到，该测试路径包含了 `malloc` 的“循环”、“首次”，`free` 的与邻近空闲分区联系的四种情况，还包含了一些必要的出错处理路径，具有一般性。该用例的输出见图 4。

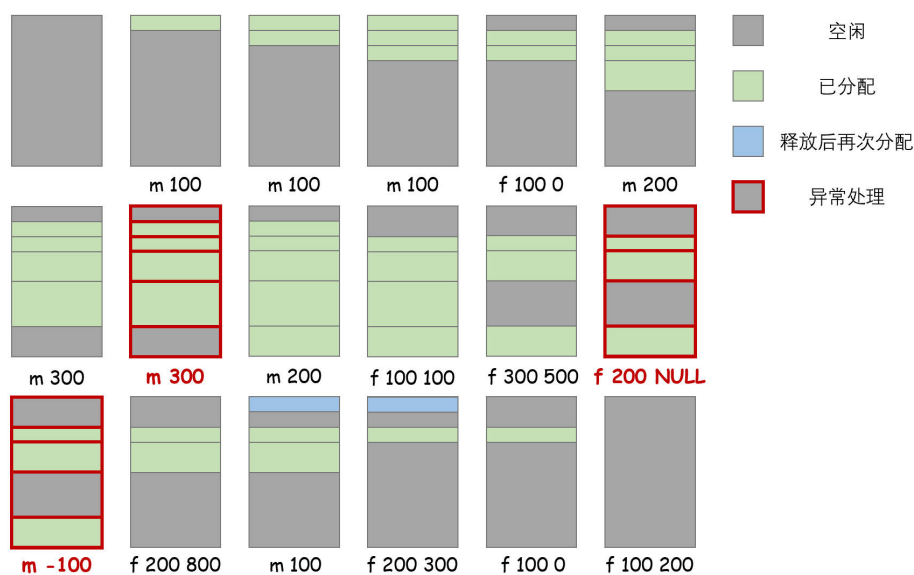


图 2: 测试用例示意图

6.4 测试结果

使用许多测试用例进行测试，程序各模块和异常处理部分均正常运行。两个典型用例（分别是命令行输入和调用测试函数）的测试结果见附录 A 图 3, 4。

7 总结

本次实验难度不大，在过程中我遇到的错误主要是处理复杂逻辑中会遗漏某些细节。如在 `malloc(unsigned)` 函数中，在 `free(p)` 时忘记判断 `coremap` 与 `p` 是否相等，从而在一次测试中导致起始查找指针指向的空闲区表被错误释放。经过多轮测试和修改，最终的程序更加完善和鲁棒。这也提醒我今后在完成类似任务时，需要注重细节，完备考虑各种可能的情况。

最后要感谢薛质老师的详细讲解，让我对可变分区存储管理有了更为深入的理解。

A 程序输出图片

```

lab/lab1 on ʘ NIS3325 [!?] via C v9.4.0-gcc took 5s
> ./main < ./input.txt
Simulation kernel space allocation succeeded!
kernel address: 0x55d17e71a2d0 - 0x55d17e71a6b8, space size: 1000
Input command, h help, q quit.
Memory allocated succeeded, address: 0x55d17e71a2d0 - 0x55d17e71a334, size: 100
*****Start Print*****
Avaliable memory partition:
memory address: 0x55d17e71a334 - 0x55d17e71a6b8, size: 900
*****End Print*****
Input command, h help, q quit.
Input command, h help, q quit.
Memory allocated succeeded, address: 0x55d17e71a334 - 0x55d17e71a3fc, size: 200
*****Start Print*****
Avaliable memory partition:
memory address: 0x55d17e71a3fc - 0x55d17e71a6b8, size: 700
*****End Print*****
Input command, h help, q quit.
Input command, h help, q quit.
Input command, h help, q quit.
Input command, h help, q quit.
***** WARNING: All kernel space has been used! *****
Memory allocated succeeded, address: 0x55d17e71a4c4 - 0x55d17e71a6b8, size: 500
*****Start Print*****
None avaliable memory partition.
Input command, h help, q quit.
Input command, h help, q quit.
***** ERROR: All kernel space has been used! *****
Input command, h help, q quit.
Input command, h help, q quit.
*****Start Print*****
Avaliable memory partition:
memory address: 0x55d17e71a2d0 - 0x55d17e71a334, size: 100
*****End Print*****
Input command, h help, q quit.
Input command, h help, q quit.
Memory free succeeded, address: 0x55d17e71a4c4 - 0x55d17e71a6b8, size: 500
*****Start Print*****
Avaliable memory partition:
memory address: 0x55d17e71a2d0 - 0x55d17e71a334, size: 100
memory address: 0x55d17e71a4c4 - 0x55d17e71a6b8, size: 500
*****End Print*****
Input command, h help, q quit.
Input command, h help, q quit.
Memory free succeeded, address: 0x55d17e71a3fc - 0x55d17e71a4c4, size: 200
*****Start Print*****
Avaliable memory partition:
memory address: 0x55d17e71a2d0 - 0x55d17e71a334, size: 100
memory address: 0x55d17e71a3fc - 0x55d17e71a6b8, size: 700
*****End Print*****
Input command, h help, q quit.
Input command, h help, q quit.
Memory free succeeded, address: 0x55d17e71a334 - 0x55d17e71a3fc, size: 200
*****Start Print*****
Avaliable memory partition:
memory address: 0x55d17e71a2d0 - 0x55d17e71a6b8, size: 1000
*****End Print*****
Input command, h help, q quit.
Input command, h help, q quit.

```

图 3: 测试 input() 函数

```

lab/lab1 on NIS3325 [!] via C v9.4.0-gcc
> ./main
Simulation kernel space allocation succeeded!
kernel address: 0x561c5fetc2d0 - 0x561c5fetc6b8, space size: 1000
Memory allocated succeeded, address: 0x561c5fetc2d0 - 0x561c5fetc334, size: 100
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc334 - 0x561c5fetc6b8, size: 900
*****End Print*****
Memory allocated succeeded, address: 0x561c5fetc334 - 0x561c5fetc398, size: 100
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc398 - 0x561c5fetc6b8, size: 800
*****End Print*****
Memory allocated succeeded, address: 0x561c5fetc398 - 0x561c5fetc3fc, size: 100
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc3fc - 0x561c5fetc6b8, size: 700
*****End Print*****
Memory free succeeded, address: 0x561c5fetc2d0 - 0x561c5fetc334, size: 100
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc3fc - 0x561c5fetc6b8, size: 700
memory address: 0x561c5fetc2d0 - 0x561c5fetc334, size: 100
*****End Print*****
***** ERROR: Required memory space [300 byte] is too large! *****
Memory allocated succeeded, address: 0x561c5fetc5f0 - 0x561c5fetc6b8, size: 200
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc2d0 - 0x561c5fetc334, size: 100
*****End Print*****
Memory free succeeded, address: 0x561c5fetc334 - 0x561c5fetc398, size: 100
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc2d0 - 0x561c5fetc398, size: 200
*****End Print*****
Memory free succeeded, address: 0x561c5fetc4c4 - 0x561c5fetc5f0, size: 300
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc2d0 - 0x561c5fetc398, size: 200
memory address: 0x561c5fetc4c4 - 0x561c5fetc5f0, size: 300
*****End Print*****
***** ERROR: wrong arguments! *****
***** ERROR: wrong arguments! *****
Memory free succeeded, address: 0x561c5fetc5f0 - 0x561c5fetc6b8, size: 200
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc2d0 - 0x561c5fetc398, size: 200
memory address: 0x561c5fetc4c4 - 0x561c5fetc6b8, size: 500
*****End Print*****
Memory allocated succeeded, address: 0x561c5fetc2d0 - 0x561c5fetc334, size: 100
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc334 - 0x561c5fetc398, size: 100
memory address: 0x561c5fetc4c4 - 0x561c5fetc6b8, size: 500
*****End Print*****
Memory free succeeded, address: 0x561c5fetc3fc - 0x561c5fetc4c4, size: 200
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc334 - 0x561c5fetc398, size: 100
memory address: 0x561c5fetc3fc - 0x561c5fetc6b8, size: 700
*****End Print*****
Memory free succeeded, address: 0x561c5fetc2d0 - 0x561c5fetc334, size: 100
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc2d0 - 0x561c5fetc398, size: 200
memory address: 0x561c5fetc3fc - 0x561c5fetc6b8, size: 700
*****End Print*****
Memory free succeeded, address: 0x561c5fetc398 - 0x561c5fetc3fc, size: 100
*****Start Print*****
Avaliable memory partition:
memory address: 0x561c5fetc2d0 - 0x561c5fetc6b8, size: 1000
*****End Print*****

```

图 4: 测试 test() 函数

B 程序源代码

```

1  /*
2   * Yusen Zheng
3   * 2022-10-19
4   */
5  #include <stdbool.h>
6  #include <stdint.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 #define ANSI_COLOR_RED "\x1b[31m"
11 #define ANSI_COLOR_GREEN "\x1b[32m"
12 #define ANSI_COLOR_YELLOW "\x1b[33m"
13 #define ANSI_COLOR_BLUE "\x1b[34m"
14 #define ANSI_COLOR_RESET "\x1b[0m"
15
16 #define CORE_SIZE 1000
17 #define VERBOSE 1
18
19 struct map {
20     unsigned m_size;
21     char *m_addr;
22     struct map *next, *prior;
23 };
24 // Lookup pointer to the starting position
25 struct map *coremap;
26 // Memory start address
27 char *start_addr;
28
29 // function declarations
30 void init_coremap(unsigned);
31 char *lmalloc(unsigned);
32 void lfree(unsigned, char *);
33 void print_free_mem();
34 void input();
35 void test();
36
37 // function definitions
38 void init_coremap(unsigned size) {
39     coremap = (struct map *)malloc(sizeof(struct map));
40     coremap->m_size = size;
41     coremap->m_addr = (char *)malloc(size);
42     coremap->next = coremap;
43     coremap->prior = coremap;
44     start_addr = coremap->m_addr;
45     printf("Simulation kernel space allocation succeeded!\n");
46     printf("kernel address: %p - %p, space "
47           "size: %u\n",
48           coremap->m_addr, coremap->m_addr + size, size);

```

```

49 }
50
51 char *lmalloc(unsigned size) {
52     // wrong arguments
53     if (size <= 0 || size >= CORE_SIZE) {
54         printf(ANSI_COLOR_RED
55             "***** ERROR: wrong arguments! *****" ANSI_COLOR_RESET "\n
56             ");
57         return NULL;
58     }
59     // all memory has been allocated
60     if (coremap == NULL) {
61         printf(ANSI_COLOR_RED "***** ERROR: All kernel space has been used
62             ! "
63             "*****" ANSI_COLOR_RESET "\n");
64         return NULL;
65     }
66
67     char *addr;
68     struct map *p, *q;
69     p = q = coremap; // p move, q fix
70
71     do {
72         // current free partition size is appropriate
73         if (p->m_size >= size) {
74             addr = p->m_addr;
75             p->m_addr += size;
76
77             // the entire partition is allocated
78             if ((p->m_size -= size) == 0) {
79                 // all memory has been allocated
80                 if (p->next == p) {
81                     printf(ANSI_COLOR_YELLOW "***** WARNING: All kernel space
82                         has been "
83                         "used! *****" ANSI_COLOR_RESET "\n"
84                     );
85                     coremap = NULL;
86                 } else {
87                     coremap = p->next;
88                     p->next->prior = p->prior;
89                     p->prior->next = p->next;
90                 }
91                 free(p);
92             }
93
94             printf(ANSI_COLOR_GREEN "Memory allocated succeeded, address: %p
95                 - %p, "
96                 "size: %u" ANSI_COLOR_RESET "\n",
97                 addr, addr + size, size);
98             if (VERBOSE) {
99                 print_free_mem();

```

```

95     }
96     return addr;
97 } else {
98     p = p->next;
99 }
100 } while (p != q);
101
102 printf(ANSI_COLOR_RED "***** ERROR: Required memory space [%d byte]
103         is too "
104         "large! *****" ANSI_COLOR_RESET "\n",
105         size);
106 return NULL;
107 }
108 void lfree(unsigned size, char *addr) {
109     // wrong arguments
110     if (size <= 0 || size >= CORE_SIZE || addr == NULL) {
111         printf(ANSI_COLOR_RED
112             "***** ERROR: wrong arguments! *****" ANSI_COLOR_RESET "\n
113             ");
114         return;
115     }
116     // all memory has been allocated
117     if (coremap == NULL) {
118         coremap = (struct map *)malloc(sizeof(struct map));
119         coremap->m_addr = addr;
120         coremap->m_size = size;
121         coremap->prior = coremap;
122         coremap->next = coremap;
123         if (VERBOSE) {
124             print_free_mem();
125         }
126         return;
127     }
128
129     // find the right address, let addr between p->m_addr & p->next->
130     m_addr
131     struct map *p = coremap;
132     while (!((p->m_addr < addr &&
133         p->next->m_addr > addr) || // e.g. .. 100 .. [250]
134         .. 300 ..
135         ((p->m_addr < addr || // e.g. [50] .. 100 ..
136         p->next->m_addr > addr) && // e.g. .. 800 [900]
137         p->m_addr >= p->next->m_addr))) { // = for only one node
138         p = p->next;
139     }
140
141     if (p->m_addr + p->m_size == addr) {
142         if (p->next->m_addr == addr + size) { // case 1: adjacent to front
143             and rear

```



```

141     p->m_size += (size + p->next->m_size);
142     struct map *q = p->next;
143     p->next = q->next;
144     q->next->prior = p;
145     if (coremap == q) { // after free, need to change coremap pointer
146         val
147         coremap = p;
148     }
149     free(q);
150 } else { // case 2: adjacent to front
151     p->m_size += size;
152 }
153 } else {
154     if (p->next->m_addr == addr + size) { // case 3: adjacent to rear
155         p->next->m_addr -= size;
156         p->next->m_size += size;
157     } else { // case 4: not adjacent
158         struct map *newCoremap = (struct map *)malloc(sizeof(struct map))
159         ;
160         newCoremap->m_addr = addr;
161         newCoremap->m_size = size;
162         newCoremap->next = p->next;
163         newCoremap->prior = p;
164         p->next->prior = newCoremap;
165         p->next = newCoremap;
166     }
167 }
168
169 printf(ANSI_COLOR_GREEN
170        "Memory free succeeded, address: %p - %p, size: %u"
171        ANSI_COLOR_RESET
172        "\n",
173        addr, addr + size, size);
174 if (VERBOSE) {
175     print_free_mem();
176 }
177 return;
178 }
179
180 // print free memory partition
181 void print_free_mem() {
182     printf(ANSI_COLOR_BLUE
183            "*****Start Print*****" ANSI_COLOR_RESET
184            "\n");
185     // all memory has been allocated
186     if (coremap == NULL) {
187         printf("None available memory partition.\n");
188         return;
189     }
190     struct map *p = coremap;
191     printf("Available memory partition:\n");

```

```

188     do {
189         printf("memory address: %p - %p, size: %u\n", p->m_addr,
190             p->m_addr + p->m_size, p->m_size);
191         p = p->next;
192     } while (p != coremap);
193     printf(ANSI_COLOR_BLUE
194         "*****End Print*****" ANSI_COLOR_RESET "
195         \n");
196     return;
197 }
198
199 void input() {
200     char cmdchar;
201     unsigned size, addr; // addr is offset address
202     do {
203         printf("Input command, h help, q quit.\n");
204         do {
205             cmdchar = getchar();
206         } while (cmdchar == ' ' || cmdchar == '\t' || cmdchar == '\n');
207
208         switch (cmdchar) {
209             case 'm':
210                 scanf("%u", &size);
211                 lmalloc(size);
212                 break;
213
214             case 'f':
215                 scanf("%u%u", &size, &addr);
216                 lfree(size, start_addr + addr);
217                 break;
218
219             case 'h':
220                 printf("memory allocate: m <#size>, memory free: f <#size> <addr>\n");
221                 break;
222
223             case 'q':
224                 break;
225
226             default:
227                 continue;
228         }
229     } while (cmdchar != 'q');
230 }
231
232 // test memory allocation and free
233 void test() {
234     char *m1 = lmalloc(100);
235     char *m2 = lmalloc(100);
236     char *m3 = lmalloc(100);
237     lfree(100, m1);

```

```
237 char *m4 = malloc(200);
238 char *m5 = malloc(300);
239 char *m6 = malloc(300); // no space
240 char *m7 = malloc(200);
241 lfree(100, m2);
242 lfree(300, m5);
243 lfree(200, NULL); // wrong arguments
244 char *m8 = malloc(-100); // wrong arguments
245 lfree(200, m7);
246 char *m9 = malloc(100); // allocation after free
247 lfree(200, m4);
248 lfree(100, m9);
249 lfree(100, m3);
250 }
251
252 int main() {
253     init_coremap(CORE_SIZE);
254
255     input();
256     // test();
257
258     free(coremap);
259
260     return 0;
261 }
```