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 20 日

1　实验题目

编写一个 C 程序，用 `char *malloc(unsignedsize)` 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：

```
# 结构数组的形式
struct map {
    unsigned m_size;
    char * m_addr;
};
struct map coremap[N];
```

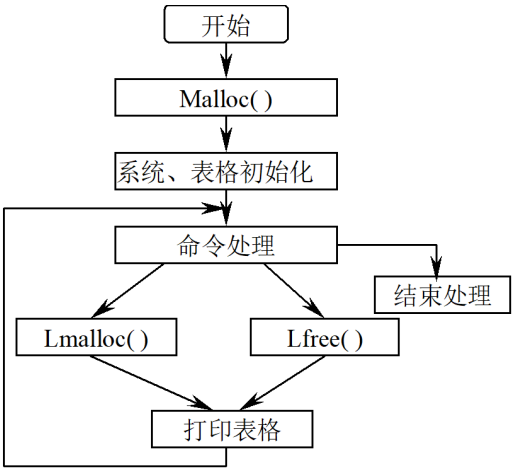

图 1: 系统基本框架

```
# 双向链接表的形式
struct map {
    unsigned m_size;
    char *m_addr;
    struct map *next, *prior;
};
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。

```
#define CORE_SIZE 1000

struct map {
  unsigned m_size;
  char *m_addr;
  struct map *next, *prior;
};
// Lookup pointer to the starting position
struct map *coremap;
// Memory start address
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。

```
void init_coremap(unsigned size) {
  coremap = (struct map *)malloc(sizeof(struct map));
  coremap->m_size = size;
  coremap->m_addr = (char *)malloc(size);
  coremap->next = coremap;
  coremap->prior = coremap;
  start_addr = coremap->m_addr;
  printf("Simulation kernel space allocation succeeded!\n");
  printf("kernel address: %p - %p, space "
         "size: %u\n",
         coremap->m_addr, coremap->m_addr + size, size);
}
```

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。

```
    // wrong arguments
    if (size <= 0 || size >= CORE_SIZE) {
        printf(ANSI_COLOR_RED
       "****** ERROR: wrong arguments! ******" ANSI_COLOR_RESET "\n");
    return NULL;
    }
    // all memory has been allocated
    if (coremap == NULL) {
    printf(ANSI_COLOR_RED "****** ERROR: All kernel space has been used
    ! "
```

```
                    "******" ANSI_COLOR_RESET "\n");
  return NULL;
  }
```

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）。

```
char *addr;
struct map *p, *q;
p = q = coremap; //  p move, q fix

do {
  // current free partition size is appropriate
  if (p->m_size >= size) {
    addr = p->m_addr;
    p->m_addr += size;

    // the entire partition is allocated
    if ((p->m_size -= size) == 0) {
      // all memory has been allocated
      if (p->next == p) {
        printf(ANSI_COLOR_YELLOW "****** WARNING: All kernel space has been "
                                 "used! ******" ANSI_COLOR_RESET "\n");
        coremap = NULL;
      } else {
        coremap = p->next;
        p->next->prior = p->prior;
        p->prior->next = p->next;
      }
      free(p);
    }

    printf(ANSI_COLOR_GREEN "Memory allocated succeeded, address: %p - %p, "
                            "size: %u" ANSI_COLOR_RESET "\n",
           addr, addr + size, size);
    if (VERBOSE) {
      print_free_mem();
    }
    return addr;
  } else {
    p = p->next;
  }
```

```
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           "****** ERROR: wrong arguments! ******" ANSI_COLOR_RESET "\n
       ");
5    return;
6  }
```

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->
     m_addr
2  struct map *p = coremap;
3  while (!((p->m_addr < addr &&
4            p->next->m_addr > addr) ||         // e.g. .. 100 .. [250]
     .. 300 ..
5           ((p->m_addr < addr ||               // e.g. [50] .. 100 ..
6             p->next->m_addr > addr) &&        // e.g. .. 800 [900]
7            p->m_addr >= p->next->m_addr))) { // = for only one node
8    p = p->next;
9  }
```

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：

```
      p->m_size += (size + p->next->m_size);
      struct map *q = p->next;
      p->next = q->next;
      q->next->prior = p;
      if (coremap == q) { // after free, need to change coremap pointer
     val
        coremap = p;
      }
      free(q);
```

当释放分区只与前空闲分区相邻时：

```
      p->m_size += size;
```

当释放分区只与后空闲分区相邻时：

```
      p->next->m_addr -= size;
      p->next->m_size += size;
```

当释放分区与前后空闲分区均不相邻时：

```
      struct map *newCoremap = (struct map *)malloc(sizeof(struct map))
    ;
      newCoremap->m_addr = addr;
      newCoremap->m_size = size;
      newCoremap->next = p->next;
      newCoremap->prior = p;
      p->next->prior = newCoremap;
      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。

```
void print_free_mem() {
  printf(ANSI_COLOR_BLUE
         "****************Start Print****************" ANSI_COLOR_RESET
     "\n");
  // all memory has been allocated
  if (coremap == NULL) {
    printf("None avaliable memory partition.\n");
    return;
  }
  struct map *p = coremap;
  printf("Avaliable memory partition:\n");
  do {
```

```
    printf("memory address: %p - %p, size: %u\n", p->m_addr,
           p->m_addr + p->m_size, p->m_size);
    p = p->next;
  } while (p != coremap);
  printf(ANSI_COLOR_BLUE
         "****************End Print****************" ANSI_COLOR_RESET "
    \n");
  return;
}
```

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。

```
void input() {
  char cmdchar;
  unsigned size, addr; // addr is offset address
  do {
    printf("Input command, h help, q quit.\n");
    do {
      cmdchar = getchar();
    } while (cmdchar == ' ' || cmdchar == '\t' || cmdchar == '\n');
    switch (cmdchar) {
    case 'm':
      scanf("%u", &size);
      lmalloc(size);
      break;
    case 'f':
      scanf("%u%u", &size, &addr);
      lfree(size, start_addr + addr);
      break;
    case 'h':
      printf("memory allocate: m <#size>, memory free: f <#size> <addr
    >\n");
      break;
    case 'q':
      break;
    default:
      continue;
    }
  } while (cmdchar != 'q');
}
```

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。

```
void test() {
  // change the func to test memory allocation and free
  char *m1 = lmalloc(100);
  char *m2 = lmalloc(100);
  char *m3 = lmalloc(100);
  lfree(100, m1);
  char *m4 = lmalloc(200);
  char *m5 = lmalloc(300);
  char *m6 = lmalloc(300); // no space
  char *m7 = lmalloc(200);
  lfree(100, m2);
  lfree(300, m5);
  lfree(200, NULL);          // wrong arguments
  char *m8 = lmalloc(-100); // wrong arguments
  lfree(200, m7);
  char *m9 = lmalloc(100); // allocation after free
  lfree(200, m4);
  lfree(100, m9);
  lfree(100, m3);
}
```

5.7 杂项

为了便于调试和观察 malloc(), lfree(), 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。

```
#define ANSI_COLOR_RED "\x1b[31m"
#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_YELLOW "\x1b[33m"
#define ANSI_COLOR_BLUE "\x1b[34m"
#define ANSI_COLOR_RESET "\x1b[0m"

#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 `lmalloc` 的“循环”、“首次”，`lfree` 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.

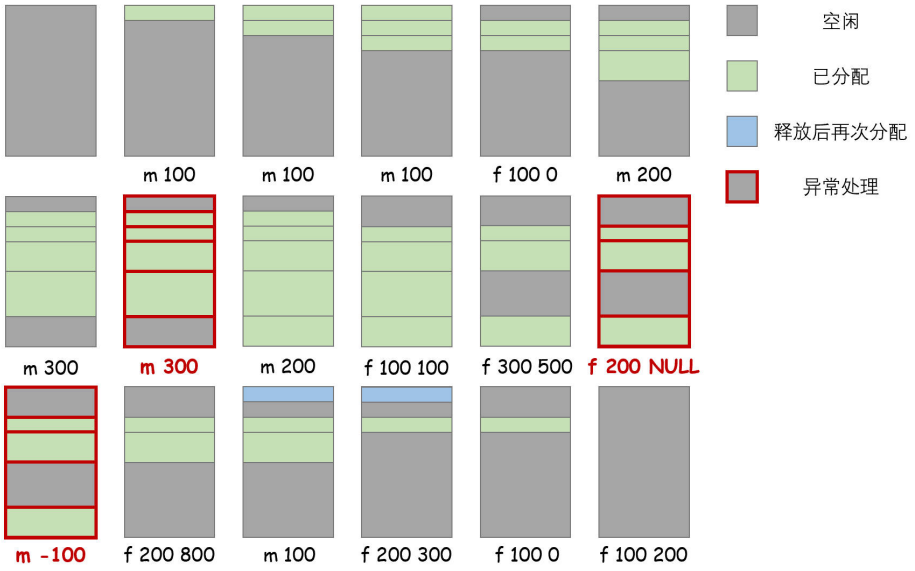

图 2: 测试用例示意图

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  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: 0x561c5fefc2d0 - 0x561c5fefc6b8, space size: 1000
Memory allocated succeeded, address: 0x561c5fefc2d0 - 0x561c5fefc334, size: 100
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc334 - 0x561c5fefc6b8, size: 900
****************End Print****************
Memory allocated succeeded, address: 0x561c5fefc334 - 0x561c5fefc398, size: 100
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc398 - 0x561c5fefc6b8, size: 800
****************End Print****************
Memory allocated succeeded, address: 0x561c5fefc398 - 0x561c5fefc3fc, size: 100
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc3fc - 0x561c5fefc6b8, size: 700
****************End Print****************
Memory free succeeded, address: 0x561c5fefc2d0 - 0x561c5fefc334, size: 100
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc3fc - 0x561c5fefc6b8, size: 700
memory address: 0x561c5fefc2d0 - 0x561c5fefc334, size: 100
****************End Print****************
****** ERROR: Required memory space [300 byte] is too large! ******
Memory allocated succeeded, address: 0x561c5fefc5f0 - 0x561c5fefc6b8, size: 200
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc2d0 - 0x561c5fefc334, size: 100
****************End Print****************
Memory free succeeded, address: 0x561c5fefc334 - 0x561c5fefc398, size: 100
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc2d0 - 0x561c5fefc398, size: 200
****************End Print****************
Memory free succeeded, address: 0x561c5fefc4c4 - 0x561c5fefc5f0, size: 300
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc2d0 - 0x561c5fefc398, size: 200
memory address: 0x561c5fefc4c4 - 0x561c5fefc5f0, size: 300
****************End Print****************
****** ERROR: wrong arguments! ******
****** ERROR: wrong arguments! ******
Memory free succeeded, address: 0x561c5fefc5f0 - 0x561c5fefc6b8, size: 200
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc2d0 - 0x561c5fefc398, size: 200
memory address: 0x561c5fefc4c4 - 0x561c5fefc6b8, size: 500
****************End Print****************
Memory allocated succeeded, address: 0x561c5fefc2d0 - 0x561c5fefc334, size: 100
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc334 - 0x561c5fefc398, size: 100
memory address: 0x561c5fefc4c4 - 0x561c5fefc6b8, size: 500
****************End Print****************
Memory free succeeded, address: 0x561c5fefc3fc - 0x561c5fefc4c4, size: 200
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc334 - 0x561c5fefc398, size: 100
memory address: 0x561c5fefc3fc - 0x561c5fefc6b8, size: 700
****************End Print****************
Memory free succeeded, address: 0x561c5fefc2d0 - 0x561c5fefc334, size: 100
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc2d0 - 0x561c5fefc398, size: 200
memory address: 0x561c5fefc3fc - 0x561c5fefc6b8, size: 700
****************End Print****************
Memory free succeeded, address: 0x561c5fefc398 - 0x561c5fefc3fc, size: 100
****************Start Print****************
Avaliable memory partition:
memory address: 0x561c5fefc2d0 - 0x561c5fefc6b8, size: 1000
****************End Print****************
```

图 4: 测试 test() 函数

B 程序源代码

```c
/*
 * Yusen Zheng
 * 2022-10-19
 */
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#define ANSI_COLOR_RED "\x1b[31m"
#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_YELLOW "\x1b[33m"
#define ANSI_COLOR_BLUE "\x1b[34m"
#define ANSI_COLOR_RESET "\x1b[0m"

#define CORE_SIZE 1000
#define VERBOSE 1

struct map {
  unsigned m_size;
  char *m_addr;
  struct map *next, *prior;
};
// Lookup pointer to the starting position
struct map *coremap;
// Memory start address
char *start_addr;

// function declarations
void init_coremap(unsigned);
char *lmalloc(unsigned);
void lfree(unsigned, char *);
void print_free_mem();
void input();
void test();

// function definitions
void init_coremap(unsigned size) {
  coremap = (struct map *)malloc(sizeof(struct map));
  coremap->m_size = size;
  coremap->m_addr = (char *)malloc(size);
  coremap->next = coremap;
  coremap->prior = coremap;
  start_addr = coremap->m_addr;
  printf("Simulation kernel space allocation succeeded!\n");
  printf("kernel address: %p - %p, space "
         "size: %u\n",
         coremap->m_addr, coremap->m_addr + size, size);
```

```
}

char *lmalloc(unsigned size) {
  // wrong arguments
  if (size <= 0 || size >= CORE_SIZE) {
    printf(ANSI_COLOR_RED
           "****** ERROR: wrong arguments! ******" ANSI_COLOR_RESET "\n
    ");
    return NULL;
  }
  // all memory has been allocated
  if (coremap == NULL) {
    printf(ANSI_COLOR_RED "****** ERROR: All kernel space has been used
    ! "
                          "******" ANSI_COLOR_RESET "\n");
    return NULL;
  }

  char *addr;
  struct map *p, *q;
  p = q = coremap; //  p move, q fix

  do {
    // current free partition size is appropriate
    if (p->m_size >= size) {
      addr = p->m_addr;
      p->m_addr += size;

      // the entire partition is allocated
      if ((p->m_size -= size) == 0) {
        // all memory has been allocated
        if (p->next == p) {
          printf(ANSI_COLOR_YELLOW "****** WARNING: All kernel space
    has been "
                                   "used! ******" ANSI_COLOR_RESET "\n"
    );
          coremap = NULL;
        } else {
          coremap = p->next;
          p->next->prior = p->prior;
          p->prior->next = p->next;
        }
        free(p);
      }

      printf(ANSI_COLOR_GREEN "Memory allocated succeeded, address: %p
    - %p, "
                              "size: %u" ANSI_COLOR_RESET "\n",
             addr, addr + size, size);
      if (VERBOSE) {
        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]
        is too "
103                         "large! ******" ANSI_COLOR_RESET "\n",
104          size);
105   return NULL;
106 }
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     return;
114   }
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->
        m_addr
130   struct map *p = coremap;
131   while (!((p->m_addr < addr &&
132             p->next->m_addr > addr) ||         // e.g. .. 100 .. [250]
        .. 300 ..
133            ((p->m_addr < addr ||               // e.g. [50] .. 100 ..
134              p->next->m_addr > addr) &&        // e.g. .. 800 [900]
135             p->m_addr >= p->next->m_addr))) { // = for only one node
136     p = p->next;
137   }
138
139   if (p->m_addr + p->m_size == addr) {
140     if (p->next->m_addr == addr + size) { // case 1: adjacent to front
        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
          val
146         coremap = p;
147       }
148       free(q);
149     } else { // case 2: adjacent to front
150       p->m_size += size;
151     }
152   } else {
153     if (p->next->m_addr == addr + size) { // case 3: adjacent to rear
154       p->next->m_addr -= size;
155       p->next->m_size += size;
156     } else { // case 4: not adjacent
157       struct map *newCoremap = (struct map *)malloc(sizeof(struct map))
          ;
158       newCoremap->m_addr = addr;
159       newCoremap->m_size = size;
160       newCoremap->next = p->next;
161       newCoremap->prior = p;
162       p->next->prior = newCoremap;
163       p->next = newCoremap;
164     }
165   }
166
167   printf(ANSI_COLOR_GREEN
168          "Memory free succeeded, address: %p - %p, size: %u"
         ANSI_COLOR_RESET
169          "\n",
170          addr, addr + size, size);
171   if (VERBOSE) {
172     print_free_mem();
173   }
174   return;
175 }
176
177 // print free memory partition
178 void print_free_mem() {
179   printf(ANSI_COLOR_BLUE
180          "****************Start Print****************" ANSI_COLOR_RESET
         "\n");
181   // all memory has been allocated
182   if (coremap == NULL) {
183     printf("None avaliable memory partition.\n");
184     return;
185   }
186   struct map *p = coremap;
187   printf("Avaliable 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 "
      \n");
195   return;
196 }
197
198 void input() {
199   char cmdchar;
200   unsigned size, addr; // addr is offset address
201   do {
202     printf("Input command, h help, q quit.\n");
203     do {
204       cmdchar = getchar();
205     } while (cmdchar == ' ' || cmdchar == '\t' || cmdchar == '\n');
206
207     switch (cmdchar) {
208     case 'm':
209       scanf("%u", &size);
210       lmalloc(size);
211       break;
212
213     case 'f':
214       scanf("%u%u", &size, &addr);
215       lfree(size, start_addr + addr);
216       break;
217
218     case 'h':
219       printf("memory allocate: m <#size>, memory free: f <#size> <addr
      >\n");
220       break;
221
222     case 'q':
223       break;
224
225     default:
226       continue;
227     }
228   } while (cmdchar != 'q');
229 }
230
231 // test memory allocation and free
232 void test() {
233   char *m1 = lmalloc(100);
234   char *m2 = lmalloc(100);
235   char *m3 = lmalloc(100);
236   lfree(100, m1);
```

```
237   char *m4 = lmalloc(200);
238   char *m5 = lmalloc(300);
239   char *m6 = lmalloc(300); // no space
240   char *m7 = lmalloc(200);
241   lfree(100, m2);
242   lfree(300, m5);
243   lfree(200, NULL);          // wrong arguments
244   char *m8 = lmalloc(-100); // wrong arguments
245   lfree(200, m7);
246   char *m9 = lmalloc(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 }
```