上海交通大學

实验一 可变分区存储管理

姓名：黄鹏程

学号：519030910206

班级：F1903601

一、 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。

*实验平台：操作系统：Windows 10；编辑器：VS code

二、 实验背景

a) 可变分区分配算法

可变分区存储管理不是预先把内存中的用户区域划分成若干固定分区，而是在作业要求装入内存时，根据用户作业的大小和当时内存空间使用情况决定是否为该作业分配一个分区。因此分区大小不是预先固定的，而是按作业需求量来划分的；分区的个数和位置也不是预先确定的。它有效地克服了固定分区方式中，由于分区内部剩余内存空置造成浪费的问题。

b) 循环首次适应算法（邻近适配）

- 算法思想：将各空闲区按地址从低到高的次序组成循环链表。每次扫描链表时从上次分配查到的那一块的后面开始扫描分配
- 算法特点：比首次适应算法具有更高的分配速度。经过长时间的运行后，系统中可能产生比较多的中等大小的空闲块；很少小细碎片，也很少有较大空闲块。

三、 实验目的

a) 加深对可变分区存储管理的理解
b) 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
c) 复习使用指针实现链表以及在链表上的基本操作

四、 实验要求

a) 一次性向系统申请一块大内存空间(e.g.1KB)
使用 malloc()函数进行内存申请，作为之后管理的内存空间
b) 使用合适的数据结构实现空闲存储区

- **结构数组**

```
struct map{
    unsigned m_size;
    char *m_addr;
};
```

- **双向链表**

```
struct map{
```

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

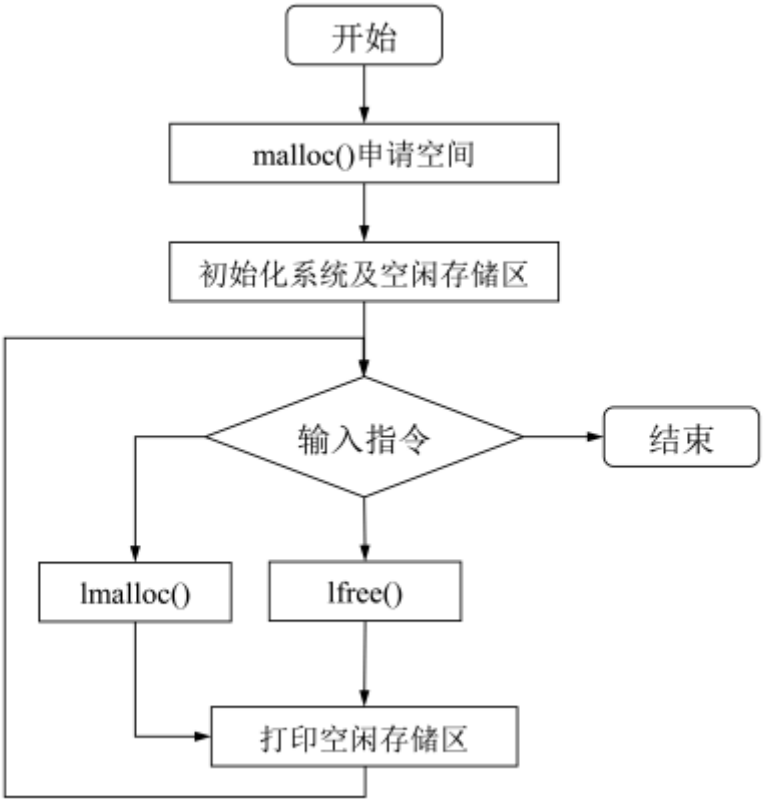
c) 主要实现如下两个内存分配函数

addr = (char *)lmalloc(unsigned size);

lfree(unsigned size,char*addr);

其中，参数 size 以及 addr 均通过控制台窗口输入，还可以传入其他需要的参数。执行完一次内存分配或释放后，需要将当前的空闲存储区情况打印出来。

d) 整体框架



五、 模块设计

a) 分配函数设计

函数声明：`struct map *lmalloc(struct map *cur, unsigned size)`

参数：cur：上次分配后的下一块空闲区指针，size：待分配内存块大小；

返回值：完成分配后的下一块空闲区指针

函数调用命令：“m int_x”，用于分配大小为 x 字节的内存块。

设计思路：首先检验 size 参数是否合法，其应该在(0,1000]的范围内，

若不在此范围则为不合法参数应当抛出错误退出函数。

```
    if ((size <= 0) || (size > 1000)){
        printf("malloc space size must be more than 0 and less than
1KB!\n");
        return cur;
    }
```

对于合法的 size 参数，从 cur 指向的当前分区快开始遍历查找，寻找大小大于等于 size 的空闲区进行分配。在分配后，若空闲区被完全分配(size 减为 0)则应该在链表中删去此块。

若完全遍历后没有满足条件的空闲区，则表示没有符合条件的空闲区，需报出异常、结束函数调用。

```
    //there is enough space to malloc
    if (cur->m_size >= size){
        cur->m_addr += size;
        cur->m_size -= size;
        //all space malloced,update list
        if (cur->m_size == 0){
            cur->prior->next = cur->next;
            cur->next->prior = cur->prior;
            cur = cur->next;
            return cur;
        }
        return cur->next;
    }
    //no enough space,traverse to find one
    else{
        struct map *tmp = cur;//tmp to judge traverse end
        cur = cur->next;
        while (cur != tmp)
        {
            if (cur->m_size >= size)
            {
                cur->m_addr += size;
                cur->m_size -= size;
                if (cur->m_size == 0){
                    cur->prior->next = cur->next;
                    cur->next->prior = cur->prior;
                    cur = cur->next;
                    return cur;
                }
                return cur->next;
            }
            else cur = cur->next;
        }
```

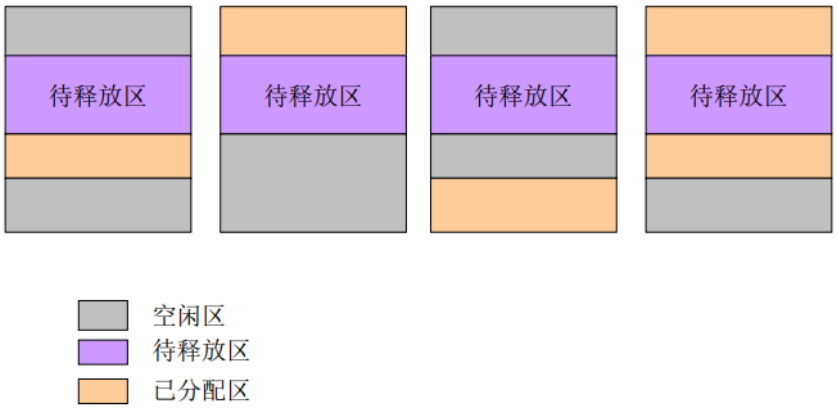
```
        printf("Failed to malloc because no matching free space\n");
    }
    return cur->next;
```

b）释放函数设计

函数声明：`char *lfree(struct map *cur, unsigned size, char *addr)`
参数：cur:上次分配后的下一块空闲区指针，size：待释放内存块大小，addr：待释放内存块起始地址；返回值：完成释放后的下一块空闲区指针
函数调用命令：“f int_x int_addr”，用于释放起始地址为 addr，大小为 x 字节的内存块。
设计思路：
同分配函数，首先进行 size 参数的合法性检查，然后进行依照四种分布情况进行释放操作。
在程序接收到用户空间释放指令后，首先遍历链表找到待释放区后第一块空闲区的指针：

```
    do{
        tmp = tmp->next;
        if (((tmp->prior->m_addr <= a) && (tmp->m_addr >= a)) ||
(tmp->prior == tmp)
            || ((tmp->prior->m_addr > a) &&
(tmp->prior->prior->m_addr >= tmp->prior->m_addr)))
            break;
    } while (tmp != cur);
```

再依照下列四种情况进行释放操作：



- **待释放区与前空闲区相连**

  在这种情况下，程序会合并释放区与前空闲区，构成新的空闲区。前空闲区的起始地址不需要改变，只需要将其大小增加 size 即可。

- **待释放区与后空闲区相连**

  在这种情况下，程序会合并释放区与后空闲区，构成新的空闲区。将

后空闲区的起始地址减去 size，再将其大小增加 size 即可。

- **待释放区与前后空闲区均相连**

  在这种情况下，程序需要将前空闲区、后空闲区及待释放区合并为一个新的空闲区。此空闲区的起始地址是前空闲区的起始地址，空间大小是三者空间大小之和。注意需要修改此新空闲区的前驱结点为前空闲区的前驱结点，后继节点为后空闲区的后继节点。

- **待释放区与前后空闲区均不相连**

  在这种情况下，程序需要将释放区建立成为一个新的空闲区。其起始地址为 addr，空间大小为 size，前驱节点是它的前一个空闲区，后继节点是它的后一个空闲区。相应的，要将前一个空闲区的后继节点设置为这个新的空闲区，将后一个空闲区的前驱结点设置为这个新的空闲区。

c）空闲区显示函数设计

函数声明：`void show(struct map*cur)`
调用命令：“d”，用于执行完一次内存分配或释放后，将当前的空闲存储区情况打印出来。遍历空闲区块链表，将各空闲区信息打印。

六、 程序接口-main()函数

程序初始化、接收命令、执行操作操作对外接口为 int main()函数。
首先进行初始内存区的申请，用于后续内存分配：

```
    char *memory = (char *)malloc(sizeof(char) * 1000);
    if (memory == NULL){
        printf("Failed to malloc! \n");
        exit(-1);
    }
```

给出使用命令说明：

```
    printf("malloc successfuly,1KB space in total\n\n");
    printf("-----##COMMAND LIST##-----\n");
    printf("1.'m x'  :apply 'x' Bytes space. eg.'m 100'\n");
    printf("2.'f x address'  free space starts from address and size is
x. eg.'f 100 17650'\n");
    printf("3.quit:'e'\n\n");
```

从命令行读入操作指令，依照指令进行对应函数操作，完成内存块的分配、释放以及空闲区情况打印。

命令读取：

```
input = getchar();
scanf(" %u", &size);
```

然后依照指令进行特定传输调用，完成操作。

七、 异常处理

a）输入 size 大小不合法

size 参数大小必须在(0,1000]范围，否则为非法大小，报错

```
 printf("malloc/free space size must be more than 0 and less than
1KB!\n");
```

b）空闲区不满足分配要求，空间不足

在遍历完所有空闲区块后，仍未完成分配，则说明不存在满足分配要求的空闲区块，报错

```
printf("Failed to malloc because no matching free space\n");
```

c）释放区与空闲区重叠，无法释放

释放区由起始地址与大小定义，若与空闲区有重叠（包括部分重叠与待释放区即为空闲区），则无法完成释放操作，报错

```
printf("The zone needn't freeing\n");
```

d）非法输入命令

定义的合法命令有“m int_x”“f int_x int_addr” “d” “e”其余输出为非法命令，报错

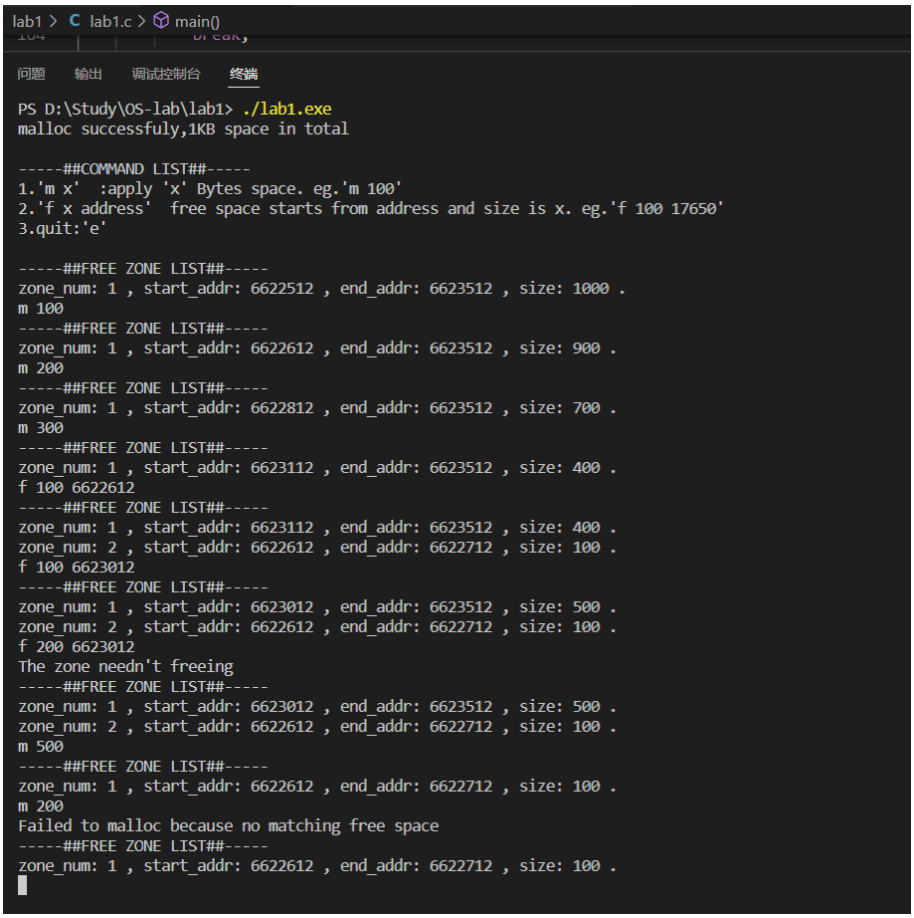
```
printf("Illegal command\n");
```

八、 实验测试

运行程序进行多次测试，对内存先进行连续分配，然后在已分配区进行释放，产生间断空闲区块，进行测试操作。

当待释放区与空闲区存在重叠时，程序报出错误信息并拒绝释放；当内存不足时，程序报出错误信息并拒绝分配；当 size 参数非法时程序报出错误信息等待下一指令。

测试命令及结果截图如下：



经多次测试，程序运行正常，符合循环首次适应分配算法要求，实验成功。

九、 个人总结

本次实验为操作系统课程可变分区存储管理算法实现作业，在实验中完整实现了循环首次适应算法。

在我看来，操作系统是很抽象、很模糊的概念，上课所听到的名词都仅停留于书本介绍、老师讲解，看不见摸不着很是空洞，对于它的理解总是很浅显而不生动。实践课便能够让我亲手操作、自行实现操作系统中涉及的算法，对算法本身就会有了更进一步的理解与体会。“纸上得来终觉浅，绝知此事要躬行”，实践是知识内化的重要一步，也是改善学习效果的有力助手，对知识概念的复现，能帮助我更好掌握、运用知识、增强个人能力。

此外，本次实验也在很大程度上锻炼了我的 C 语言编程能力，我尤其是在指针链表的使用方面上收益颇丰。C 语言是较为基础而广泛应用的语言，对于未来学业或是工作开发都十分重要，通过课程 lab 进行实践锻炼效果显著，大有裨益。

同时，也在此向辛勤教学的刘老师、热心指导的助教老师致以真诚谢意！疫情之下，教学工作多有困难，你们辛苦了！

十、 代码附录

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

//Memory block data structure model
struct map{
    unsigned m_size;
    char *m_addr;
    struct map *next,*prior;
};

// struct map *lmalloc(struct map *cur, unsigned size)
// cur:last alloc location,size:size of needed memory
struct map *lmalloc(struct map *cur, unsigned size){
    if ((size <= 0) || (size > 1000)){
        printf("malloc space size must be more than 0 and less than
1KB!\n");
        return cur;
    }
    //there is enough space to malloc
    if (cur->m_size >= size){
        cur->m_addr += size;
        cur->m_size -= size;
        //all space malloced,update list
        if (cur->m_size == 0){
            cur->prior->next = cur->next;
            cur->next->prior = cur->prior;
            cur = cur->next;
            return cur;
        }
        return cur->next;
    }
    //no enough space,traverse to find one
    else{
        struct map *tmp = cur;//tmp to judge traverse end
        cur = cur->next;
        while (cur != tmp)
        {
            if (cur->m_size >= size)
            {
                cur->m_addr += size;
                cur->m_size -= size;
                if (cur->m_size == 0){
                    cur->prior->next = cur->next;
```

```
                    cur->next->prior = cur->prior;
                    cur = cur->next;
                    return cur;
                }
                return cur->next;
            }
            else cur = cur->next;
        }
        printf("Failed to malloc because no matching free space\n");
    }
    return cur->next;
}
//char *lfree(struct map *cur, unsigned size, char *addr)
char *lfree(struct map *cur, unsigned size, char *addr){
    struct map *tmp = cur;
    char *a = addr;
    if ((size <= 0) || (size > 1000)){
        printf("Free space size must be more than 0 and less than
1KB!\n");
        return a;
    }

    do{
        tmp = tmp->next;
        if (((tmp->prior->m_addr <= a) && (tmp->m_addr >= a)) ||
(tmp->prior == tmp)
            || ((tmp->prior->m_addr > a) &&
(tmp->prior->prior->m_addr >= tmp->prior->m_addr)))
            break;
    } while (tmp != cur);

    if (tmp->prior == tmp){
        if ((tmp->m_addr <= a) && (tmp->m_addr + tmp->m_size >= a)){
            printf("The zone needn't free\n");
            return a;
        }
    }
    else if (((tmp->prior->m_addr + tmp->prior->m_size) > a) || (a +
size > tmp->m_addr)){
        printf("The zone needn't freeing\n");
        return a;
    }

    //1.there is free zone front
```

```
    if ((tmp->prior->m_addr + tmp->prior->m_size) == a){
        tmp->prior->m_size += size;
        //2.and there is free zone behind
        if (a + size == tmp->m_addr)
        {
            tmp->prior->m_size += tmp->m_size;
            tmp->prior->next = tmp->next;
            tmp->next->prior = tmp->prior;
        }
        return a;
    }
    //3.there is free zone behind
    else if (a + size == tmp->m_addr){
        tmp->m_addr -= size;
        tmp->m_size += size;
        return a;
    }
    //4.there is no free zone front or behind
    else{
        tmp->prior->next = (struct map *)malloc(sizeof(struct map));
        tmp->prior->next->m_addr = a;
        tmp->prior->next->m_size = size;
        tmp->prior->next->prior = tmp->prior;
        tmp->prior->next->next = tmp;
        tmp->prior = tmp->prior->next;
        return a;
    }
}

//void show(struct map*cur),print free zone list
void show(struct map*cur){
    printf("-----##FREE ZONE LIST##-----\n");
    int num = 1;
    struct map *tmp = cur;
    do{
        printf("zone_num: %u , start_addr: %u , end_addr: %u ,
size: %u .\n",num,cur->m_addr,cur->m_addr+cur->m_size,cur->m_size);
        num++;
        cur = cur->next;
    }while(cur != tmp);
}

int main(){
    char *memory = (char *)malloc(sizeof(char) * 1000);
```

```
    if (memory == NULL){
        printf("Failed to malloc! \n");
        exit(-1);
    }
    struct map *init = (struct map *)malloc(sizeof(struct map));
    init->m_addr = memory;
    init->m_size = 1000;
    init->next = init;
    init->prior = init;
    printf("malloc successfuly,1KB space in total\n\n");
    printf("-----##COMMAND LIST##-----\n");
    printf("1.'m x'  :apply 'x' Bytes space. eg.'m 100'\n");
    printf("2.'f x address'  free space starts from address and size is
x. eg.'f 100 17650'\n");
    printf("3.quit:'e'\n\n");

    char input;
    int size;
    char *addr;
    struct map *cur, *search;
    cur = init;
    search = init;
    show(init);
    do{
        input = getchar();
        if (input == 'm'){
            scanf(" %u", &size);
            cur = lmalloc(cur, size);
            show(cur);
        }
        else if (input == 'f'){
            scanf(" %u %u", &size, &addr);
            search = init;
            lfree(search, size, addr);
            show(search);
        }
        else if (input == 'e'){
            printf("Bye!\n");
            break;
        }
        else if (input == '\n' || input == '\t' || input == ' '){
            continue;
        }
        else{
```

```
            printf("Illegal command\n");
        }
    } while (true);
    free(memory);
    return 0;
}
```