操作系统 实验一 可变分区存储管理-循环首次适应法

摘要

在本次试验中，我使用双向链表的数据结构，用 C 语言成功实现了可变分区存储管理中的循环首次适应法，实现了对内存区的分配和释放管理。并且我考虑了很多分配和释放内存区时的错误，如分配时内存不足，释放越界，重复释放等问题，并给出了合适的解决办法。通过本次试验，我深刻地理解了可变分区存储管理，并练习了用指针和结构体实现双向链表和在链表上的基本操作。

算法思想和概要设计

算法思想

分配算法

程序采用循环首次适应法的思想，把空闲表设计成连接结构的循环队列，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，使其指向队伍中被分配的后面那块空闲区。下次分配时就从新指向的那块空闲区开始查找。

释放算法

循环首次适应法的释放算法针对以下四种情况采取四种策略：

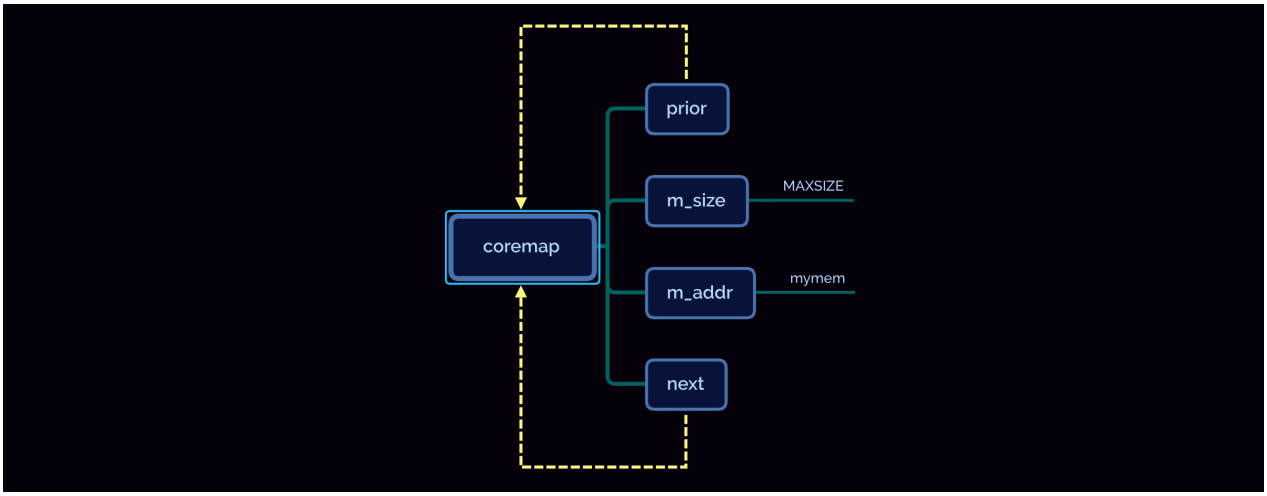
- **释放区与前空闲区相连：**合并前空闲区和释放区构成一块大的新空闲区，并修改前空闲区表项的地址和大小属性。
- **与前空闲区和后空闲区都相连：**将三块空闲区合并成一块空闲区，并修改前空闲区表项的地址和大小属性。
- **仅与后空闲区相连：**与后空闲区合并，并修改后空闲区表项的地址和大小属性。
- **与前、后空闲区皆不相连：**在前、后空闲区表项中间插入一个新的表项。

概要设计

主程序中，先向真实主存申请一块固定大小的内存，再用这块内存初始化空闲区双向链表。此时双向链表中只有一个表项，其前向指针与后向指针都指向自身，大小属性等于申请的内存的大小。之后用键盘输入指令来模拟程序向主存申请或释放内存。以下将以流程图来展示对该程序分配与释放内存的过程概要设计。

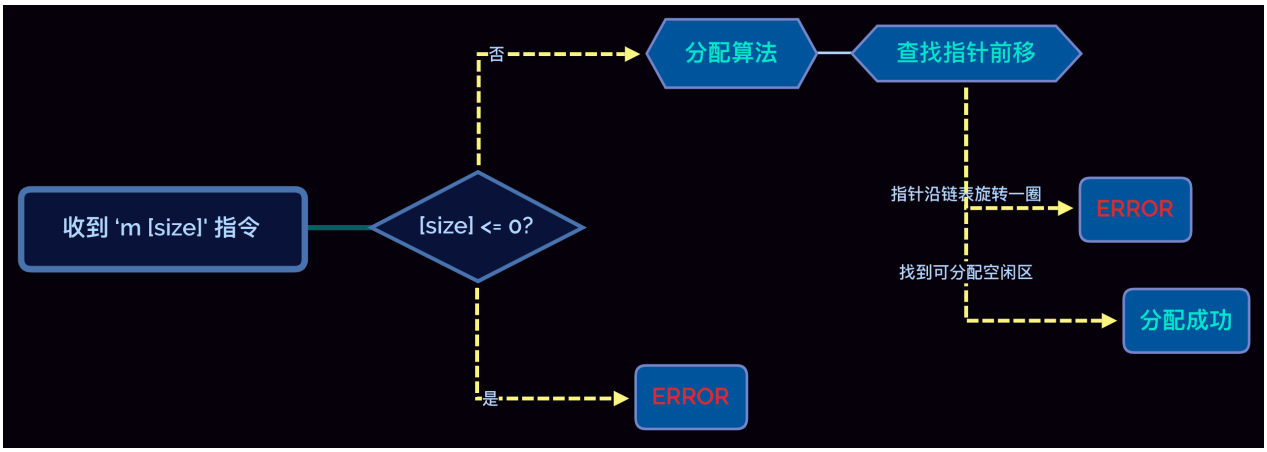
初始化

双向链表中只有一个表项，其前向指针与后向指针都指向自身，大小属性等于申请的内存的大小，地址属性等于申请的内存的基地址。



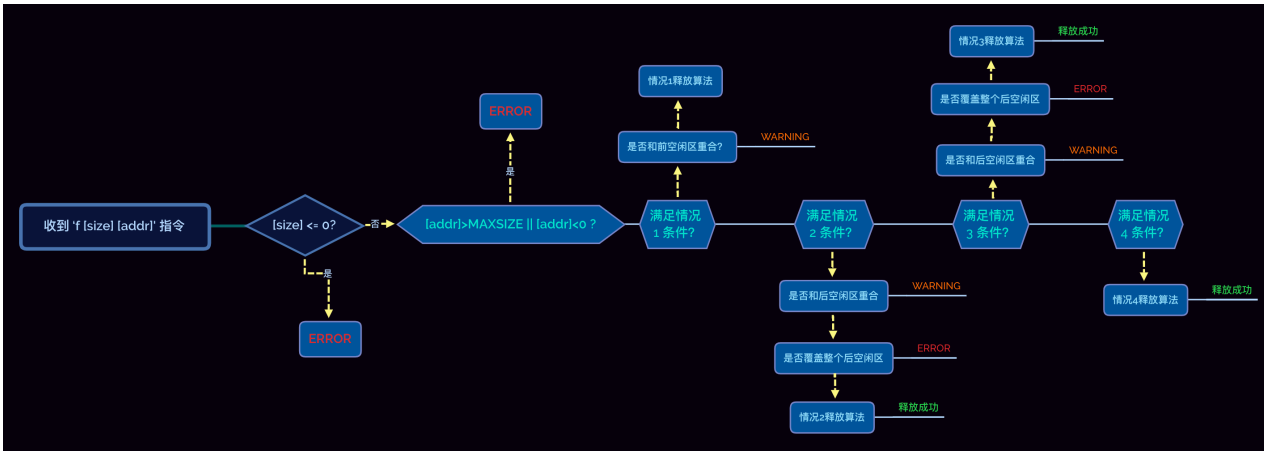
分配

查找指针不断前移直到找到合适的空闲区。如果沿双向链表找了一圈仍未找到，就认为不存在合适的空闲区，拒绝分配并报错。



释放

由于使用了基地址，所以指令中输入相对地址进行释放。值得一提的是，释放时考虑了多个可能出现的错误，如**重复释放已空闲区域，释放区位置与已存在的空闲区部分重合**等问题。



数据结构与变量说明

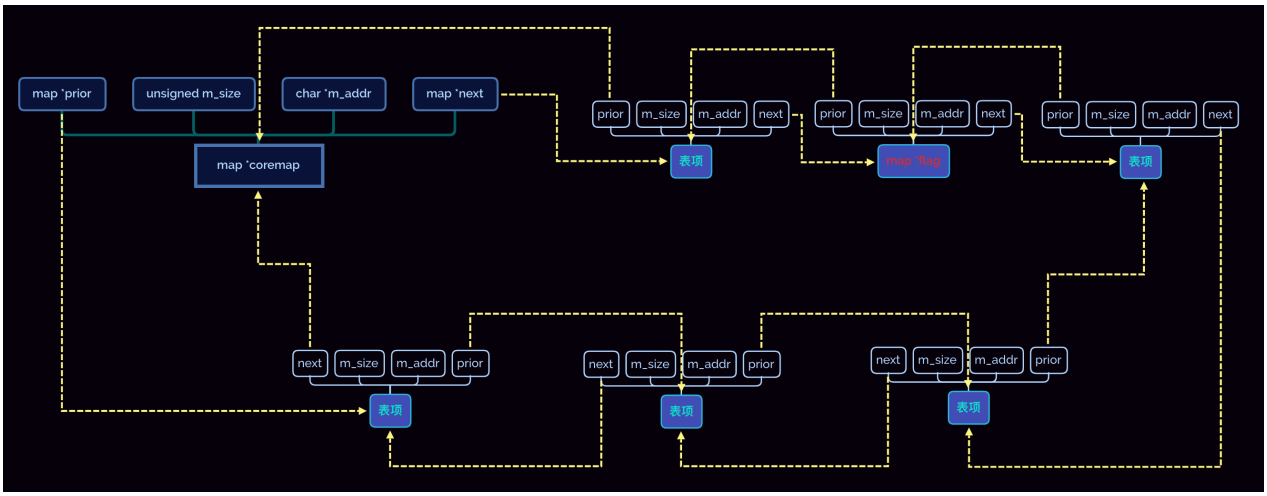
双向链表的表项：

```
/*双向链表的定义*/
struct map
{
    unsigned m_size;           //空闲区大小
    char *m_addr;              //空闲区首地址
    struct map *next, *prior; //后向指针与前向指针
};
```

全局指针

```
struct map *coremap;     //链表起始指针
struct map *flag;        //起始查找指针
```

以下为本程序的双向链表的示意图。



源程序

链表结构体与全局变量的定义

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
//#include <malloc.h>

#define MEMSIZE 1000

/*双向链表的定义*/
struct map
{
    unsigned m_size;
    char *m_addr;
    struct map *next, *prior;
};

struct map *coremap;    //链表起始指针
struct map *flag;       //起始查找指针
```

循环首次适应法的分配函数

```
/*循环适应的分配函数*/
char *lmalloc(unsigned size)
{
    register char *a;
    register struct map *bp;
    bp = flag;
    do
    {
      if(bp->m_size >= size)
        {
            a = bp->m_addr;
```

```
            bp->m_addr += size;
            flag = bp->next;
            if((bp->m_size -= size) == 0)    // 删除用光的空闲区项
            {
                bp->prior->next = bp->next;
                bp->next->prior = bp->prior;
                free(bp);
            }
            printf("Success: lmalloc size: %d, addr:%p\n", size, a);    //分配空间成功
            return(a);
        }
        bp = bp->next;
    }while(bp != flag);
    printf("Error: There is no appropriate memory to be allocated for the size!\n");        //分配空间失败
    return(0);
}
```

循环首次适应法的释放函数

```
/* 循环首次适应的释放函数 */
char *lfree(unsigned size, char *aa)
{
    struct map *bp;
    char *a;
    a = aa;
    for(bp = coremap; bp->m_addr <= a; bp = bp->next);
    if (bp->prior->m_addr + bp->prior->m_size >= a && a >= bp->prior->m_addr && bp->next != bp) /* 情况1,2 */
    {
        if(a + size > bp->m_addr + bp->m_size)  // 防止向下溢出一整个空闲项
        {
            printf("Error: Release area out of bounds!\n");
            return(0);
        }
        else if(bp->prior->m_addr + bp->prior->m_size > a + size)   // 防止重复释放已空闲的区域
        {
            printf("Error: Already released!\n");
            return(0);
        }
        else
        {
            if(bp->prior->m_addr + bp->prior->m_size > a)       // 警告：所释放的区域与上一个空闲项的后部有重叠，但仍将执行
            {
                printf("Warning: Release area out of bounds!\n");
                bp->prior->m_size = a + size - bp->prior->m_addr;
```

```
            }
            else bp->prior->m_size += size;     /* 情况1 */
        }

        if (a + size >= bp->m_addr)     /* 情况2 */
        {
            if(a + size > bp->m_addr)       // 警告：所释放的区域与下一个空闲项的前部有重叠，但仍将执行
            {
                printf("Warning: Release area out of bounds!\n");
                bp->prior->m_size = bp->m_addr + bp->m_size - bp->prior->m_addr;
            }
            else bp->prior->m_size += bp->m_size;

            bp->prior->next = bp->next;
            bp->next->prior = bp->prior;
            if(bp == flag)
                flag = bp->next;
            free(bp);
        }
      }
    else
    {
        if(a + size > bp->m_addr + bp->m_size)      // 防止向下溢出一整个空闲项
        {
            printf("Error: Release area out of bounds!\n");
            return(0);
        }
        else if (a+size == bp->m_addr) /* 情况3 */
        {
            bp->m_addr -= size;
            bp->m_size += size;
        }
        else if(a+size > bp->m_addr)        // 警告：所释放的区域与下一个空闲项的前部有重叠，但仍将执行
        {
            printf("Warning: Release area out of bounds!\n");
            bp->m_size = bp->m_addr + bp->m_size - a;
            bp->m_addr = a;
        }
        else/* 情况4 */
        {
            struct map *novel;
            novel = (struct map *)malloc(sizeof(struct map)); // 新建节点来表示新的独立空闲区
            novel->m_addr = a;
            novel->m_size = size;
```

```
            novel->prior = bp->prior;
            novel->next = bp;
            bp->prior->next = novel;
            bp->prior = novel;
            if(coremap->m_addr > novel->m_addr)
                coremap = novel;
        }
      }
    printf("Success: lfree mem size=%u, addr=%p\n", size, aa);
    return(0);
}
```

coremap链表的初始化程序

```
/*  coremap链表的初始化程序 */
void initcoremap(char *addr, unsigned size)
{
    printf("init coremap, first addr: %p\n", addr);
    coremap = (struct map *)malloc(sizeof(struct map));
    coremap->m_size = size;
    coremap->m_addr = addr;
    coremap->next = coremap;
    coremap->prior = coremap;
    flag = coremap;
  // 初始化coremap双向链表
}
```

输出链表的内容

```
/*  输出链表的内容 */
void printcoremap()
{
    struct map *fg;
    fg = coremap;
    do                          // 从表头开始打印整个链表
    {
        printf("flag->m_addr = %p  ", fg->m_addr);
        printf("flag->m_size = %d\n", fg->m_size);
        fg = fg->next;
    }while(fg != coremap);
    /* Function body:  打印coremap链表中各项的m_size和m_addr  */
}
```

主程序

```
/*  主程序 */
int main()
{
```

```
    char *mymem;
    int size;
    int addr;
    char cmdchar;
    char c;
    if ((mymem = malloc(MEMSIZE)) == NULL)      //在真实主存中申请一块作业空间
    {
      printf("Not enough memory to allocate buffer\n");
      exit(1);
    }
    initcoremap(mymem, MEMSIZE);    //初始化空闲双向链表

    while(c!='q')
    {
        do
        {
            c = getchar();
        }while(c=='\n'||c=='\t'||c==' ');

        cmdchar = c;
        switch (cmdchar)
        {
        case 'm':                           // 分配
            scanf("%u", &size);
            if(size <= 0)
            {
                printf("Error: Wrong size!\n");
                break;
            }
            lmalloc(size);
            break;
        case 'f':                           // 释放
            scanf("%u %u", &size, &addr);
            if(size <= 0)
            {
                printf("Error: Wrong size!\n");
                break;
            }
            if(addr > MEMSIZE || addr < 0)
            {
                printf("Error: Wrong address!\n");
                break;
            }
            lfree(size, mymem + addr);
            break;
        case 'p':                           // 打印整个空闲区链表
            printcoremap();
            break;
        default:
```

```
            break;
        }
    }
    free(mymem);
    return 0;
}
```

测试

测试样例设计

基本功能测试设计

```
p                   // 打印初始空闲区链表
m 600               // 分配功能测试：申请分配大小为600的空间
p                   // 观察分配后的空闲区链表
m 100               // 分配功能测试：申请分配大小为100的空间
p                   // 观察分配后的空闲区链表
f 200 0             // 释放功能测试：释放大小为200、相对地址为0的空间（该空间目前非空闲）
p                   // 观察释放后的空闲区链表
f 100 300           // 释放功能测试：情况四测试，释放大小为100、相对地址为300的空间
p                   // 观察释放结果
f 50 200            // 释放功能测试：情况一测试，释放大小为50、相对地址为200的空间
p                   // 观察释放结果
f 50 250            // 释放功能测试：情况二测试，释放大小为50、相对地址为250的空间
p                   // 观察释放结果
f 100 600           // 释放功能测试：情况三测试，释放大小为100、相对地址为600的空间
p                   // 观察释放结果
f 50 500            // 情况三，释放大小为50、相对地址为500的空间，为分配功能测试做准备
p                   // 观察释放结果
m 50                // 分配功能测试：申请分配大小为50的空间
p                   // 观察释放结果
m 50                // 分配功能测试：申请分配大小为50的空间
p                   // 观察释放结果
m 50                // 分配功能测试：申请分配大小为50的空间
p                   // 观察释放结果
m 50                // 分配功能测试：申请分配大小为50的空间
p                   // 观察释放结果
m 50                // 分配功能测试：申请分配大小为50的空间
p                   // 观察释放结果
q                   // 结束程序
```

出错处理测试设计

```
p                   // 打印初始空闲区链表
m 1200              // 分配错误测试：申请分配大于MAXSIZE的空间（MAXSIZE=1000）
m 0                 // 分配错误测试：申请分配大小为0的空间
```

```
m -100                  // 分配错误测试：申请分配小于0的空间
p                       // 打印空闲区链表
m 800                   // 构造测试用链表
f 200 0
f 200 400               // 此时有三个空闲区，起始地址分别为0、400、800，大小均为200
p                       // 打印空闲区链表
m 300                   // 分配错误测试：请求分配大小为300的空间（没有足够大的空闲区）
p                       // 打印空闲区链表
f 0 100                 // 释放错误测试：释放区大小为0（拒绝释放）
f -50 100               // 释放错误测试：释放区大小小于0（拒绝释放）
f 100 -100              // 释放错误测试：释放区相对起始地址小于0（拒绝释放）
f 100 1200              // 释放错误测试：释放区相对起始地址大于MAXSIZE（拒绝释放）
p                       // 打印空闲区链表
f 100 50                // 释放错误测试：释放已完全空闲的区域（拒绝释放）
p                       // 打印空闲区链表
f 200 50                // 释放错误测试：释放区与前空闲区部分重合（仍作为情况一释放，但警告）
p                       // 打印空闲区链表
f 200 350               // 释放错误测试：释放区与后空闲区部分重合（仍作为情况三释放，但警告）
p                       // 打印空闲区链表
f 400 300               // 释放错误测试：释放区完全包含了后空闲区（拒绝释放）
p                       // 打印空闲区链表
f 200 200               // 释放错误测试：释放区与前后空闲区均有部分重合（仍作为情况二释放，但警告）
p                       // 打印空闲区链表
q                       // 结束程序
```

测试结果

基本功能测试结果

```
init coremap, first addr: 0x1006028a0
p
flag->m_addr = 0x1006028a0  flag->m_size = 1000
m 600
Success: lmalloc size: 600, addr:0x1006028a0
p
flag->m_addr = 0x100602af8  flag->m_size = 400
m 100
Success: lmalloc size: 100, addr:0x100602af8
p
flag->m_addr = 0x100602b5c  flag->m_size = 300
f 200 0
Success: lfree mem size=200, addr=0x1006028a0
p
flag->m_addr = 0x1006028a0  flag->m_size = 200
flag->m_addr = 0x100602b5c  flag->m_size = 300
f 100 300
```

```
Success: lfree mem size=100, addr=0x1006029cc
p
flag->m_addr = 0x1006028a0  flag->m_size = 200
flag->m_addr = 0x1006029cc  flag->m_size = 100
flag->m_addr = 0x100602b5c  flag->m_size = 300
f 50 200
Success: lfree mem size=50, addr=0x100602968
p
flag->m_addr = 0x1006028a0  flag->m_size = 250
flag->m_addr = 0x1006029cc  flag->m_size = 100
flag->m_addr = 0x100602b5c  flag->m_size = 300
f 50 250
Success: lfree mem size=50, addr=0x10060299a
p
flag->m_addr = 0x1006028a0  flag->m_size = 400
flag->m_addr = 0x100602b5c  flag->m_size = 300
f 100 600
Success: lfree mem size=100, addr=0x100602af8
p
flag->m_addr = 0x1006028a0  flag->m_size = 400
flag->m_addr = 0x100602af8  flag->m_size = 400
f 50 500
Success: lfree mem size=50, addr=0x100602a94
p
flag->m_addr = 0x1006028a0  flag->m_size = 400
flag->m_addr = 0x100602a94  flag->m_size = 50
flag->m_addr = 0x100602af8  flag->m_size = 400
m 50
Success: lmalloc size: 50, addr:0x100602af8
p
flag->m_addr = 0x1006028a0  flag->m_size = 400
flag->m_addr = 0x100602a94  flag->m_size = 50
flag->m_addr = 0x100602b2a  flag->m_size = 350
m 50
Success: lmalloc size: 50, addr:0x1006028a0
p
flag->m_addr = 0x1006028d2  flag->m_size = 350
flag->m_addr = 0x100602a94  flag->m_size = 50
flag->m_addr = 0x100602b2a  flag->m_size = 350
m 50
Success: lmalloc size: 50, addr:0x100602a94
p
flag->m_addr = 0x1006028d2  flag->m_size = 350
flag->m_addr = 0x100602b2a  flag->m_size = 350
m 50
Success: lmalloc size: 50, addr:0x100602b2a
p
flag->m_addr = 0x1006028d2  flag->m_size = 350
flag->m_addr = 0x100602b5c  flag->m_size = 300
```

```
m 50
Success: lmalloc size: 50, addr:0x1006028d2
p
flag->m_addr = 0x100602904  flag->m_size = 300
flag->m_addr = 0x100602b5c  flag->m_size = 300
q
Program ended with exit code: 0
```

出错处理测试结果

```
init coremap, first addr: 0x100545ce0
p
flag->m_addr = 0x100545ce0  flag->m_size = 1000
m 1200
'Error: There is no appropriate memory to be allocated for the size!'
m 0
'Error: Wrong size!'
m -100
'Error: Wrong size!'
p
flag->m_addr = 0x100545ce0  flag->m_size = 1000
m 800
Success: lmalloc size: 800, addr:0x100545ce0
f 200 0
Success: lfree mem size=200, addr=0x100545ce0
f 200 400
Success: lfree mem size=200, addr=0x100545e70
p
flag->m_addr = 0x100545ce0  flag->m_size = 200
flag->m_addr = 0x100545e70  flag->m_size = 200
flag->m_addr = 0x100546000  flag->m_size = 200
m 300
'Error: There is no appropriate memory to be allocated for the size!'
p
flag->m_addr = 0x100545ce0  flag->m_size = 200
flag->m_addr = 0x100545e70  flag->m_size = 200
flag->m_addr = 0x100546000  flag->m_size = 200
f 0 100
'Error: Wrong size!'
f -50 100
'Error: Wrong size!'
f 100 -100
'Error: Wrong address!'
f 100 1200
'Error: Wrong address!'
p
flag->m_addr = 0x100545ce0  flag->m_size = 200
flag->m_addr = 0x100545e70  flag->m_size = 200
flag->m_addr = 0x100546000  flag->m_size = 200
```

```
f 100 50
'Error: Already released!'
p
flag->m_addr = 0x100545ce0  flag->m_size = 200
flag->m_addr = 0x100545e70  flag->m_size = 200
flag->m_addr = 0x100546000  flag->m_size = 200
f 200 50
'Warning: Release area out of bounds!'
Success: lfree mem size=200, addr=0x100545d12
p
flag->m_addr = 0x100545ce0  flag->m_size = 250
flag->m_addr = 0x100545e70  flag->m_size = 200
flag->m_addr = 0x100546000  flag->m_size = 200
f 200 350
'Warning: Release area out of bounds!'
Success: lfree mem size=200, addr=0x100545e3e
p
flag->m_addr = 0x100545ce0  flag->m_size = 250
flag->m_addr = 0x100545e3e  flag->m_size = 250
flag->m_addr = 0x100546000  flag->m_size = 200
f 400 300
'Error: Release area out of bounds!'
p
flag->m_addr = 0x100545ce0  flag->m_size = 250
flag->m_addr = 0x100545e3e  flag->m_size = 250
flag->m_addr = 0x100546000  flag->m_size = 200
f 200 200
'Warning: Release area out of bounds!'
'Warning: Release area out of bounds!'
Success: lfree mem size=200, addr=0x100545da8
p
flag->m_addr = 0x100545ce0  flag->m_size = 600
flag->m_addr = 0x100546000  flag->m_size = 200
q
Program ended with exit code: 0
```

可见，测试结果完全正确，程序能正确运行。

改进与体会

改进

实现程序时，发现仅仅实现循环首次适应法的基本功能还是比较容易的。但在测试时发现程序不能处理如**重复释放已空闲区域、释放区域与空闲区域有部分重合**的情况；如果出现这种情况，空闲区链表将会加入重复项、无效项与坏项。而实际情况中应该要考虑这些问题。所以我在循环首次适应法的基础上加入了各种情况的出错处理机制，使程序能处理**几乎所有可能的情况，大大增强了程序的容错能力**。

体会

由于上个学期在编译原理课程上使用过单链表的数据结构实现了语法分析程序，所以这次用双向链表实现循环首次适应法对我来说并不难，在透彻理解循环首次适应法的思想后，程序也很快就写好了。

这次实验让我深刻理解了循环首次适应法的算法过程与思想。除了惊叹于其巧妙与精练之外，也让我对操作系统的内存管理机制有了更深、更好的理解。另外，这次实验让我的 *C* 语言熟练度也有一些提升。

by 刘浩文 517021911065