《循环首次适应算法的内存调度程序》实验报告

姓名：翟拙存 学号：515030910141 班级： F1503006

一、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。

二、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。

三、实验环境

系统：Windows10

C 语言编译器：Microsoft Visual Studio Ultimate 2015

四、算法思想

1. 循环首次适应法的分配算法

把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。

循环首次适应法的实质是起始查找指针所指的空闲区和其后的空闲区群常为较长时间未被分割过的空闲区，它们已合并成为大的空闲区可能性较大，比起首次适应法，在没有增加多少代价的情况下却明显地提高了分配查找的速度。

2. 循环首次适应法的回收算法

释放算法基本同首次适应法一样。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或删除一个表项时，根据该表项与起始查找指针之间的相对位置，有可能需要修改指针值，使其仍旧指向原空闲表项。鉴于前后空闲区与新释放的空闲区的相邻情况有不同的合并方式，考虑

a.释放区仅与前空闲区相连；

b.释放区仅与后空闲区相连；

c.释放区与前、后空闲区都相连；

d.释放区与前、后空闲区都不相连。[本次实验中采用双链表，故与课本相比，在处理释放区与前后空闲区的关系的同时还需要考虑到双链表的特殊结构，进行相关处理]

五、概要设计

1. 系统基本框架如图 1 所示。本实验从内存中申请一块存储区，对这块存储区进行模拟的分配和回收活动。

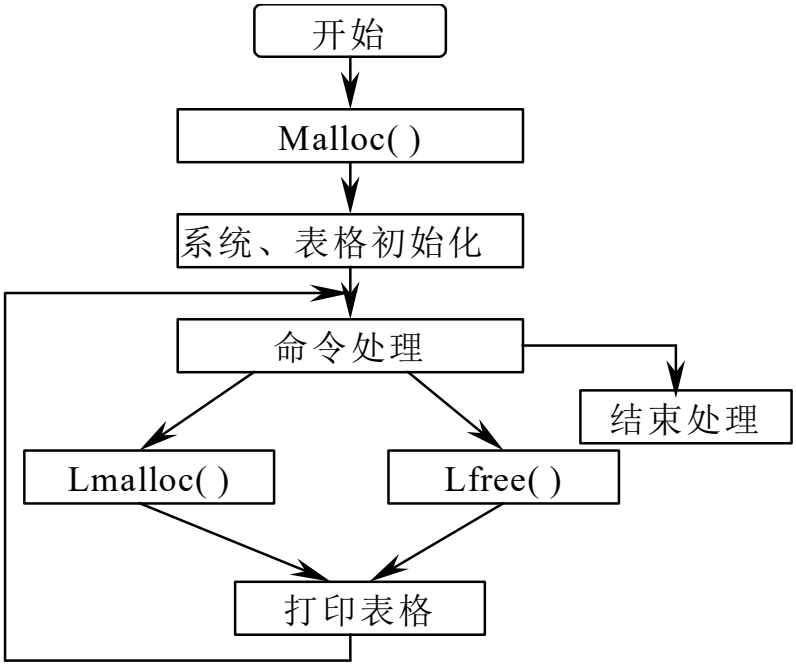

图 1

2. 程序主要模块

由图 1 可得，程序需要实现的功能为：识别键盘的命令，模拟对内存区的分配和释放管理。因此，设计程序分为几个主要的模块：初始化模块、处理命令模块、申请内存模块、释放内存模块、显示内存模块。

3. 主要数据结构

双向链表结构体 map，用于记录模拟内存当中的空闲内存块。每个结构 map，应该具有如下属性：空闲区大小，空闲区首址指针，上一个空闲区首址指针，下一个空闲区。

```
struct map {
unsigned m_size;                    //空闲区大小
```

```
char *m_addr;                    //空闲区首址指针
struct map *next, *prior;        //用于记录前后空闲区
};
```

六、重要模块的详细设计及功能及接口说明

（一）重要模块功能及其描述

1. void init_map(void)

该模块实现内存块的初始化。本次实验中，共定义了 20 个内存块用以模仿对内存空间的分配和释放管理。在对这 20 个内存块的首地址和大小进行随机定义后，在其基础上生成一个双向链表。链表中的每一个元素，即每一个结构体都代表了一个内存块，包括首地址(m_addr)、大小(m_size)、前一个连接的结构体(prior)、后一个连接的结构体(next)四个特征值。

至此，所有的内存块以结构体 map 的形式描述完毕，且以双向链表的形式连接起来。除此之外，为了方便之后的函数功能，还定义了 coremap 这个结构体指针，指向第一个内存块。

2. void print(void)

该模块实现对内存块信息的打印。由于 init_map(void)模块的存在，打印内存块信息的工作实质上已经变成了打印整个双向链表信息的工作。通过临时指针 tmp 从 coremap 指针处（第一个内存块）开始，对整个双向链表的轮询，并在每一个指针处打印首地址和大小信息，可以完成要求。

3. unsigned lmalloc(unsigned size)

该模块实现对内存空间的分配。对于用户指定的大小(size)，bp 指针从当前

内存块处(current_map)出发轮询，一旦发现有一个内存块的大小可以包括用户要求的大小，便将其中对应大小的部分分配给用户，同时更新内存块信息。更新内存块的信息包括更改首地址和大小。特别地，如果整个内存块都被分配给了用户，则从链表中删除这个块。完成分配工作后，将 bp 指针停留的位置指定为新的当前内存块。

该模块实质上是对循环首次适应法的分配算法的实现。

4. void lmfree(unsigned size, unsigned free_addr)

该模块实现对内存空间的释放。对于用户指定的首地址(free_addr)和大小(size)，首先找到该地址在哪两个内存块之间，同样可以通过对链表的轮询实现。找到这两个内存块(bp 和 bp->prior)后，根据需要释放的内存空间的首地址和大小决定该释放操作属于四种释放情况中的哪一种。如果是第一种情况（释放空间与上一个内存块相连但不与下一个内存块相连），只需对上一个内存块进行大小上的扩充即可；如果是第二种情况（释放空间与上下内存块都相连），需要将这三块空间全部合并，成为一个新的内存块；如果是第三种情况（释放空间与下一个内存块相连但不与上一个内存块相连），需要对下一个内存块的首地址和大小都进行更改；如果是第四种情况（释放空间与上下内存块都不相连），则创建一个新的结构体，并将其插入双向链表中。

该模块实质上是对循环首次适应法的释放算法的实现。

5. void main()

该模块是算法的主函数，实现了对用户操作的提示以及获取用户命令，并根据命令让相应的函数工作。如果获取到的操作符为 m，则调用内存分配函数，并进一步获取用户指定的内存大小；如果获取到的操作符为 f，则调用内存释放

函数，并进一步获取用户指定的内存首地址和大小；如果获取到的操作符为 p，则打印内存块信息；如果获取到的操作符为 q，则退出。

（二）各函数接口

各函数接口		
函数	输入	输出
void init_map(void)	-	-
unsigned lmalloc(unsigned size)	unsigned size 表示申请的内存大小	unsigned 表示分配得到的内存首地址
void lmfree(unsigned size, unsigned free_addr)	unsigned size 表示需要释放的内存大小 free_addr 表示需要释放的内存首地址	-
void print(void)	-	-
int main( )	-	-

七、源代码

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
```

```
//memory block struction
typedef struct map
{
    unsigned m_size;
    unsigned m_addr;
    struct map *next, *prior;
}mp, *mps;

mps coremap; //defined as the first block

mps current_map; //defined as the current block

//initialize the memory
void init_map(void)
{
    //decide the size of each block randomly
    int coresize[20] = { 200, 1800, 1300, 700, 1100, 900, 1600, 400, 1200,
800,
        100, 1900, 300, 1700, 1500, 500, 600, 1400, 1000, 66 };
    int coreaddress[20];
    int i;
    for (i = 0; i<20; i++)
```

```
{
    coreaddress[i] = 2000 * (i + 1);
}

//form chain
mps pHead = (mps)malloc(sizeof(mp));
mps pTail = NULL, p_new = NULL;

pHead->m_size = coresize[0];
pHead->m_addr = coreaddress[0];
pHead->prior = NULL;
pHead->next = NULL;
pTail = pHead;
for (i = 1; i<20; i++)
{
    p_new = (mps)malloc(sizeof(mp));

    p_new->m_size = coresize[i];
    p_new->m_addr = coreaddress[i];
    p_new->next = NULL;
    p_new->prior = pTail;
    pTail->next = p_new;
```

```
        pTail = p_new;
    }
    pTail->next = pHead;
    pHead->prior = pTail;

    coremap = pHead;
}


//print info of the memory block
void print(void)
{
    mps tmp1, tmp2;

    tmp1 = coremap->prior;

    for (tmp2 = coremap; tmp2 != tmp1; tmp2 = tmp2->next)
    {
        printf("address: %d,size: %d\n", tmp2->m_addr, tmp2->m_size);
    }
    printf("address: %d,size: %d\n", tmp1->m_addr, tmp1->m_size);
}
```

```
//allocate memory space
unsigned lmalloc(unsigned size)
{
    unsigned malloc_addr, current_addr;
    mps bp;

    if (current_map == NULL)
        current_map = coremap;

    current_addr = current_map->m_addr;

    bp = current_map;
    if (bp->m_size >= size)
    {
        malloc_addr = bp->m_addr;
        bp->m_addr += size;
        if ((bp->m_size -= size) == 0)
        {
            bp->prior->next = bp->next;
            bp->next->prior = bp->prior;
```

```
            current_map = bp->next;
            free(bp);
        }
        else
            current_map = bp;
        return (malloc_addr);
    }

    for (bp = current_map->next; bp->m_addr != current_addr; bp = bp->next) //search for the whole space
    {
        if (bp->m_size >= size)
        {
            malloc_addr = bp->m_addr;
            bp->m_addr += size;
            if ((bp->m_size -= size) == 0)
            {
                bp->prior->next = bp->next;
                bp->next->prior = bp->prior;
                current_map = bp->next;
                free(bp);
            }
```

```
            else
                current_map = bp;
            return (malloc_addr);
        }
    }
    return 0;
}

//free memory space
void lmfree(unsigned size, unsigned free_addr)
{
    mps bp, new_map;

    unsigned addr;
    addr = free_addr;

    //search for two adjacent blocks between which free_addr is
    for (bp = coremap; !((bp->m_addr > addr && bp->prior->m_addr <
addr) ||
        (bp->m_addr < addr && bp->prior->m_addr < addr &&
bp->m_addr < bp->prior->m_addr) ||
        (bp->m_addr > addr && bp->prior->m_addr > addr &&
```

```
bp->m_addr < bp->prior->m_addr)); bp = bp->next);


    if (bp->prior->m_addr + bp->prior->m_size == addr) // case 1&2
    {
        bp->prior->m_size += size;   // case 1
        if (addr + size == bp->m_addr)   // case 2
        {
            bp->prior->m_size += bp->m_size;
            bp->prior->next = bp->next;
            bp->next->prior = bp->prior;
            //free(bp);
        }
    }
    else if (addr + size == bp->m_addr) // case 3
    {
        bp->m_addr -= size;
        bp->m_size += size;
    }
    //case 4
    else if ((addr + size < bp->m_addr && bp->prior->m_addr +
bp->prior->m_size < addr) ||
        (addr + size > bp->m_addr && bp->prior->m_addr +
```

```
bp->prior->m_size < addr && bp->m_addr < bp->prior->m_addr) ||
        (addr + size < bp->m_addr && bp->prior->m_addr +
bp->prior->m_size > addr && bp->m_addr < bp->prior->m_addr))
    {
        new_map = (mps)malloc(sizeof(mp));
        new_map->m_size = size;
        new_map->m_addr = free_addr;
        new_map->next = bp;
        new_map->prior = bp->prior;
        bp->prior->next = new_map;
        bp->prior = new_map;
    }
    else printf("Memory release error!!!\n");
}

void main()
{
    char op;
    unsigned arg1, arg2;
    unsigned allo_addr; //defined as the allocated address

    init_map();
```

```
    printf("Current info of the memory space:\n");
    print();

    while (1)
    {
        printf("Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>>
");
        do
        op = getchar();
        while (op == '\n' || op == '\t' || op == ' ');
        if (op == 'm')
        {
            printf("Please input the argument(size) >>> ");
            scanf("%u", &arg1);
            allo_addr = lmalloc(arg1);
            if (allo_addr != 0)
                printf("The address is %u\n", allo_addr);
            else printf("Allocation false!\n");
        }
        else if (op == 'f')
        {
            printf("Please input the arguments(size && address) >>> ");
```

```
            scanf("%u%u", &arg1, &arg2);
            lmfree(arg1, arg2);
        }
        else if (op == 'p') print();
        else if (op == 'q') break;
        else printf("Check your input!!!\n");
    }
}
```

八、测试及结果分析

输入和输出文本中注释数字相同的地方表示该输入对应的输出结果。

1. 测试输入文本分析

```
a       //用于测试用户输入错误操作符时的情况  //1
m
50      //申请分配大小为 50 的内存空间
m
500         //申请分配大小为 500 的内存空间
m
1400        //申请分配大小为 1400 的内存空间
m
100         //申请分配大小为 100 的内存空间
m
```

100　　//申请分配大小为 100 的内存空间

//该次分配测试了某个内存块全部分配完了的情况

m

1800　　//申请分配大小为 1800 的内存空间

m

1000　　//申请分配大小为 1000 的内存空间

m

1000　　//申请分配大小为 1000 的内存空间

m

1000　　//申请分配大小为 1000 的内存空间

m

1250　　//申请分配大小为 1250 的内存空间

//该次分配测试了链表的循环性

m

100000　　//用于测试用户申请内存空间过大 没有合适内存块时的情况 **//2**

p　　//打印经过以上 10 次分配后的内存块情况 **//3**

f

100 15400　　//释放大小为 100 首地址为 15400 处的内存空间

//测试了第四种释放情况（均不相连）

p　　//打印经过该次释放后的内存块情况 **//4**

f

100 15500　　//释放大小为 100 首地址为 15500 处的内存空间

//测试了第一种释放情况（仅与上一个内存块相连）

p　　//打印经过该次释放后的内存块情况 **//5**

f

1400 14000　　//释放大小为 1400 首地址为 14000 处的内存空间

//测试了第三种释放情况（仅与下一个内存块相连）

p　　//打印经过该次释放后的内存块情况 **//6**

f

1100 12900　　//释放大小为 1100 首地址为 12900 处的内存空间

//测试了第二种释放情况（与上下内存块都相连）

p　　//打印经过该次释放后的内存块情况 **//7**

q　　//测试退出功能

2. 测试输出文本分析

Current info of the memory space:

address: 2000,size: 200

address: 4000,size: 1800

address: 6000,size: 1300

address: 8000,size: 700

address: 10000,size: 1100

address: 12000,size: 900

address: 14000,size: 1600

address: 16000,size: 400

address: 18000,size: 1200

address: 20000,size: 800

address: 22000,size: 100

address: 24000,size: 1900

address: 26000,size: 300

address: 28000,size: 1700

address: 30000,size: 1500

address: 32000,size: 500

address: 34000,size: 600

address: 36000,size: 1400

address: 38000,size: 1000

address: 40000,size: 66 **//打印初始内存块信息**

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Check your input!!!

//1

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> The address is 2000

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> The address is 4000

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> The address is 14000

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> The address is 15400

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> The address is 15500

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> The address is 24000

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> The address is 28000

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> The address is 30000

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> The address is 36000

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> The address is 4500

//以上 10 条记录分别输出了 10 次分配中的内存空间首地址

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the argument(size) >>> Allocation false! **//2**

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>>

address: 2050,size: 150

address: 5750,size: 50

address: 6000,size: 1300

address: 8000,size: 700

address: 10000,size: 1100

address: 12000,size: 900

address: 16000,size: 400

address: 18000,size: 1200

address: 20000,size: 800

address: 22000,size: 100

address: 25800,size: 100

address: 26000,size: 300

address: 29000,size: 700

address: 31000,size: 500

address: 32000,size: 500

address: 34000,size: 600

address: 37000,size: 400

address: 38000,size: 1000

address: 40000,size: 66 **//3**

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the arguments(size && address) >>> Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>>

address: 2050,size: 150

address: 5750,size: 50

address: 6000,size: 1300

address: 8000,size: 700

address: 10000,size: 1100

address: 12000,size: 900

address: 15400,size: 100

address: 16000,size: 400

address: 18000,size: 1200

address: 20000,size: 800

address: 22000,size: 100

address: 25800,size: 100

address: 26000,size: 300

address: 29000,size: 700

address: 31000,size: 500

address: 32000,size: 500

address: 34000,size: 600

address: 37000,size: 400

address: 38000,size: 1000

address: 40000,size: 66 **//4**

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the arguments(size && address) >>> Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>>

address: 2050,size: 150

address: 5750,size: 50

address: 6000,size: 1300

address: 8000,size: 700

address: 10000,size: 1100

address: 12000,size: 900

address: 15400,size: 200

address: 16000,size: 400

address: 18000,size: 1200

address: 20000,size: 800

address: 22000,size: 100

address: 25800,size: 100

address: 26000,size: 300

address: 29000,size: 700

address: 31000,size: 500

address: 32000,size: 500

address: 34000,size: 600

address: 37000,size: 400

address: 38000,size: 1000

address: 40000,size: 66 **//5**

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the arguments(size && address) >>> Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>>

address: 2050,size: 150

address: 5750,size: 50

address: 6000,size: 1300

address: 8000,size: 700

address: 10000,size: 1100

address: 12000,size: 900

address: 14000,size: 1600

address: 16000,size: 400

address: 18000,size: 1200

address: 20000,size: 800

address: 22000,size: 100

address: 25800,size: 100

address: 26000,size: 300

address: 29000,size: 700

address: 31000,size: 500

address: 32000,size: 500

address: 34000,size: 600

address: 37000,size: 400

address: 38000,size: 1000

address: 40000,size: 66 **//6**

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> Please input the arguments(size && address) >>> Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>>

address: 2050,size: 150

address: 5750,size: 50

address: 6000,size: 1300

address: 8000,size: 700

address: 10000,size: 1100

address: 12000,size: 3600

address: 16000,size: 400

address: 18000,size: 1200

address: 20000,size: 800

address: 22000,size: 100

address: 25800,size: 100

address: 26000,size: 300

address: 29000,size: 700

address: 31000,size: 500

address: 32000,size: 500

address: 34000,size: 600

address: 37000,size: 400

address: 38000,size: 1000

address: 40000,size: 66 **//7**

Choose the operation(m[alloc]/f[ree]/p[rint]/q[uit]) >>> **//quit**

3. 测试总结

（1）对内存分配的“循环”及“首次”性进行了测试，验证了循环首次适应法算法的实现。

（2）对内存释放的四种情况进行了测试，验证了内存释放后内存块的有序性，

仍旧保持有序、正确的双向链表结构。

（3）对一些出错处理进行了测试，如用户键入操作符错误，索要内存空间过大导致没有合适的内存块分配等等。

（4）测试内容写入 input.txt 后，在 DOS 环境下使用 malloc.exe < input.txt > result.txt 命令将输出写入 result.txt 中，再使用 copy malloc.c+input.txt+result.txt progress.txt 命令将源码、输入、输出合并到一个文件中。

九、实验心得体会

由于很长时间没有使用 C 语言进行编程，加之以前对于指针以及双向链表的学习不是很深入，我没有很熟练地掌握相关的编程要领。在开始这次实验之前，我重新温习了 C 指针、链表的相关内容，了解到指针和链表在深入了解数据结构、操作系统的过程中非常重要。

在操作系统课上我们学习了循环首次适应法，其优点在于，比起首次适应法而言并没有增加过多代价却明显的提高了分配查找的速度，因为起始查找指针所指的空闲区和其后的空闲区群常为较长时间未被分割过的空闲区，它们已合并成为大的空闲区可能性更大，从而更可能符合新的分配需求——总的来说，便是使内存中的空闲分区分布得更为均匀，减少了查找时的系统开销，具有较强的实用意义。尽管能够较为清晰地阐述出关于循环首次适应法简单的算法思想，但其具体的实现和测试要更为复杂和细节化，在本次实验的过程当中我也在程序调试上花了大量的时间。用双向链表代替简单的结构体有许多优越性，但同时对编程的要求也更高。因此，程序的逻辑设计完整性对于我而言是一个比较大的挑战。

本次实验中的不足之处在于，尽管基本实现了本次实验所要求的功能，但是逻辑不够简洁，代码有一定的冗余。

这次实验让我对于可变分区存储管理有了更深入的了解，也增强了我的逻辑思考能力和编程能力。同时，我认识到熟悉一些经典数据结构的编程对于我们今后学习计算机相关课程具有重要的意义，非常感谢老师课上给予的耐心指导！