操作系统实验一：可变分区存储管理

姓名：吴嘉锐

班级：F1803601

学号：518021910082

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。

实验目的

- 加深对可变分区的存储管理的理解；

- 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；

- 掌握用指针实现链表和在链表上的基本操作。

算法思想

可变分区存储

- 可变分区存储管理法并不预先将内存划分成分区，而是等到作业运行需要内存时就向系统申请，从空闲的内存区中“挖”一块出来，其大小等于作业所需内存大小，这样就不会产生“内零头”。

![avatar](./空闲表.png)

Figure1：空闲表

- 管理空闲内存区的数据结构可采用链接法和连续线性表格法。本实验采用循环双链表。

循环首次适应法

- 把空闲表设计成顺序结构或双向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。
- 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。比起首次适应法来，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。
- 释放算法基本同首次适应法一样。
- 首次适应法和循环首次适应法不仅可用于内存的分配和释放，也可用于进程图像交换的辅存空间的分配和释放，其管理空闲区的数据结构也相同。

设计概要

在本实验中，程序会首先初始化分配一定的内存地址空间，接着通过终端接收用户的输入信息进行地址分配并显示处理结果，直到用户输入退出程序指令后程序会回收分配的所有空间并结束程序。下为程序的总体设计图，一些细节没有显示（如错误处理等）。

![avatar](./设计概要.jpg)

Figure 2：程序设计摘要

数据结构及重要变量

本实验采用**双向循环链接表**实现存储管理中空闲表的功能：

```c
typedef struct map
{
    unsigned m_size; // 空闲区大小
    char *m_addr; // 空闲区起始地址
    struct map *next, *prior; // 下一个&上一个空闲区
} map_m;
```

由于采用首次适应法的思想，设置了一个**指向当前空闲区的全局变量**cur_map：

```c
map_m * cur_map;
```

当然，**初始分配内存空间**也需要保存为一个变量 buf：

```c
char * buf;
```

除此之外，程序还设计了一个宏 SIZE 以保存**初始化空间的大小**

```c
#define SIZE 1000
```

模块设计说明

此部分将自顶向下逐一阐述实验中设计的不同模块的功能、接口以及实现方法。

程序初始化部分

空闲表初始化

该模块通过函数 init_map 实现，会初始分配大小为 SIZE 的内存空间，生成第一个空闲表块并返回其地址。该空闲表块的起始地址 m_addr 为分配内存空间的起始地址 buf，大小 size 为 SIZE，前序指针与后续指针均指向自己。

```c
// initialize the addr space and the current map
map_m * init_map() {
    buf = malloc(sizeof(char)*SIZE); // 分配空间
    map_m * object = (map_m *) malloc(sizeof(map_m));
// 初始化第一个空闲表块
    object->m_addr = buf;
    object->m_size = SIZE;
    object->next = object;
    object->prior = object;
    printf("map initialized, m_addr = %lu, size = %u\n",
(unsigned long)(object->m_addr - buf), object->m_size);
// 显示初始化地址、大小
    return object; // 返回空闲表指针给主函数，主函数会将其存于 cur_map 中
}
```

输入输出部分

一切输入输出的地址均当作偏移地址处理。

用户输入输出交互

该模块直接内置于主函数中。用户输入的指令会被 fgets 获取并将指令字符串保存在 cin 中，然后将其作为参数传递给判断函数 judge_str 进行指令类型的判断，分为帮助、分配、释放、退出和非法指令五种，然后根据指令类型作进一步处理。

```c
                int judge; // 用于判断用户输入的指令类型
    unsigned size; // 保存用户输入分配或释放内存空间的大小
    unsigned long addr; // 保存用户输入释放内存的起始地址

    char cin[100]; // 保存用户输入的指令字符串
    char command[6]; // 保存用户输入的命令（malloc、free），仅用于占位

    do
    {
                                // 提示用户开始输入
        printf("\n--------------------------------------------------------------------\
            \nPlease enter a commend(You can try type \"help\"): ");
        fgets( cin, 100, stdin );
        printf("\n");
        judge = judge_str(cin);

        // 根据不同输入类型进行处理
        switch (judge)
        {
        case 0: // help
            printf("malloc: m[alloc] size;\nfree: f[ree] size addr;\nexit: quit the program\n");
            break;
        case 1: // malloc
            sscanf(cin, "%s %u", command, &size);
            Lmalloc(size);
            break;
        case 2: // free
            sscanf(cin, "%s %u %lu", command, &size, &addr);
            if (IfFreeIllegal(size, (char *)(addr + (unsigned long)buf)) == 0) {
                printf("illegal free command!\n");
                ShowMapState();
            }
            else Lfree(size, (char *)(addr + (unsigned long)buf));
            break;
        case 3: // illegal command
            printf("illegal command\n");
            break;
        default:
            ShowMapState();
            break;
        }
    } while (judge != 4);
```

指令类型判断

该模块通过函数 judge_str 完成，功能在于通过返回一个 int 类型判断用户输入指令的类型。

```
0:help          1:malloc        2:free
                3:illegal       4:exit
```

```c
/*
    0:help
    1:malloc
    2:free
    3:illegal
    4:exit
*/
int judge_str (char *str) {
    if ((strncmp(str, "help", 4) == 0) && (str[4]=='\n')) return 0;
    if (strncmp(str, "m ", 2) == 0) {
        int i = 2;
        if (str[i] == '\n') return 3;

        while (str[i] != '\n')
        {
            if (!isdigit(str[i])) return 3;
            i++;
        }
        return 1;
    }
    if (strncmp(str, "malloc ", 7) == 0){
        int i = 7;
        if (str[i] == '\n') return 3;

        while (str[i] != '\n')
        {
            if (!isdigit(str[i])) return 3;
            i++;
        }
        return 1;
    }
    if (strncmp(str, "f ", 2) == 0) {
        int i = 2;
        if (str[i] == '\n') return 3;

        while (str[i] != ' ')
        {
            if (!isdigit(str[i])) return 3;
            i++;
        }
        i++;
        if (str[i] == '\n') return 3;
        while (str[i] != '\n')
        {
            if (!isdigit(str[i])) return 3;
            i++;
        }
        return 2;
    }
    if (strncmp(str, "free ", 5) == 0) {
        int i = 5;
        if (str[i] == '\n') return 3;

        while (str[i] != ' ')
        {
            if (!isdigit(str[i])) return 3;
            i++;
        }
        i++;
        if (str[i] == '\n') return 3;
        while (str[i] != '\n')
        {
            if (!isdigit(str[i])) return 3;
            i++;
        }
        return 2;
    }

    if ((strncmp(str, "exit", 4) == 0) && (str[4]=='\n')) return 4;
    return 3;
}
```

显示空闲表状态

该模块用于从起始查找指针开始打印当前空闲表的信息。

```c
// show map
void ShowMapState() {
    printf("\ncurrent maps:\n");
    if (cur_map == NULL) {
        printf("All addrs have been malloced!\n");
        return;
    }

    map_m * ser_map = cur_map;
    do
    {
        printf("map: m_size = %u, m_addr = %lu\n",
                ser_map->m_size, (unsigned long)(ser_map->m_addr - buf));
        ser_map = ser_map->next;
    } while (ser_map != cur_map);
    return;
}
```

空闲表处理部分

地址空间分配模块

该模块用函数 Lmalloc 实现，参数为所需分配空间大小 size，函数从起始查找指针开始寻找大小大于等于 size 的空闲表块进行地址分配。若没有符合条件的地址块则输出分配失败信息。

```c
// malloc
void Lmalloc(unsigned size) {
    map_m * ser_map = cur_map;

    if (cur_map == NULL)
    {
        printf("malloc failed!\n");
        ShowMapState();
        return;
    }

    do
    {
        if (ser_map->m_size > size)
        {
            cur_map = ser_map->next;
            ser_map->m_addr += sizeof(char) * size;
            ser_map->m_size -= size;
            printf("applied for size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                    size, (unsigned long)(ser_map->m_addr - sizeof(char) * size - buf), (unsigned long)(ser_map->m_addr - 1 - buf));
            ShowMapState();
            return;
        }
        else if (ser_map->m_size == size)
        {
            printf("applied for size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                    size, (unsigned long)(ser_map->m_addr - buf), (unsigned long)(ser_map->m_addr + sizeof(char) * size -1 - buf));
            if (ser_map->next == ser_map) cur_map = NULL;
            else cur_map = ser_map->next;

            ser_map->prior->next = ser_map->next;
            ser_map->next->prior = ser_map->prior;
            free(ser_map);
            ShowMapState();
            return;
        }
        else {
            ser_map = ser_map->next;
        }

    } while (ser_map != cur_map);
    printf("malloc failed!\n");
    ShowMapState();
    return;
}
```

地址空间释放模块

释放模块由函数 Lfree 实现，参数为释放空间的大小、释放空间的起始地址。由于采用了双向链表，除了教材的四种分配情况，这里实现还增加了释放地址小于空闲表最小地址与大于空闲表最大地址、空闲表为空共五种情况讨论。

```c
// free
void Lfree(unsigned size, char * addr) {
    map_m * ser_map = cur_map;
    map_m * tmp_map;
    map_m * new_map;
    map_m * to_be_freed;

    // 空闲表为空
    if (cur_map == NULL)
    {
        cur_map = (map_m*) malloc(sizeof(map_m));
        cur_map->m_addr = addr;
        cur_map->m_size = size;
        cur_map->next = cur_map;
        cur_map->prior = cur_map;
        printf("freed size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                size, (unsigned long)(addr - buf), (unsigned long)(addr + size -1 - buf));
        ShowMapState();
        return;
    }

                // 分配地址小于最小地址
    if (addr < (tmp_map = min_addr())->m_addr)
    {
        if (addr + size -1 == tmp_map->m_addr -1)
        {
            tmp_map->m_addr = addr;
            tmp_map->m_size += size;
            printf("freed size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                    size, (unsigned long)(addr - buf), (unsigned long)(addr + size -1 - buf));
            ShowMapState();
            return;
        }
        else
        {
            new_map = (map_m*) malloc(sizeof(map_m));
            new_map->m_addr = addr;
            new_map->m_size = size;
            new_map->next = tmp_map;
            new_map->prior = tmp_map->prior;
            new_map->prior->next = new_map;
            tmp_map->prior = new_map;
            printf("freed size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                    size, (unsigned long)(addr - buf), (unsigned long)(addr + size -1 - buf));
            ShowMapState();
            return;
        }
    }

                // 分配地址大于最大地址
    if (addr > (tmp_map = max_addr())->m_addr)
    {
        if (tmp_map->m_addr + tmp_map->m_size -1 == addr -1)
        {
            tmp_map->m_size += size;
            printf("freed size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                    size, (unsigned long)(addr - buf), (unsigned long)(addr + size -1 - buf));
            ShowMapState();
            return;
        }
        else
        {
            new_map = (map_m*) malloc(sizeof(map_m));
            new_map->m_addr = addr;
            new_map->m_size = size;
            new_map->next = tmp_map->next;
            new_map->next->prior = new_map;
            tmp_map->next = new_map;
            new_map->prior = tmp_map;
            printf("freed size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                    size, (unsigned long)(addr - buf), (unsigned long)(addr + size -1 - buf));
            ShowMapState();
            return;
        }
    }

                // 寻找释放地址处于哪两个空闲表块之间
    for (tmp_map = cur_map; !(tmp_map->m_addr < addr && addr < tmp_map->next->m_addr); tmp_map = tmp_map->next);
    if (tmp_map->m_addr + tmp_map->m_size -1 == addr -1) // case a, b
    {
        if (addr + size -1 < tmp_map->next->m_addr -1) // case a
        {
            tmp_map->m_size += size;
            printf("freed size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                    size, (unsigned long)(addr - buf), (unsigned long)(addr + size -1 - buf));
            ShowMapState();
            return;
        }
        else // case b
        {
            tmp_map->m_size = tmp_map->m_size + size + tmp_map->next->m_size;
            if (cur_map = tmp_map->next) cur_map = tmp_map; // in case cur_map freed unexpectly
            to_be_freed = tmp_map->next;
            tmp_map->next = tmp_map->next->next;
            tmp_map->next->next->prior = tmp_map;
            free(to_be_freed);
            printf("freed size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                    size, (unsigned long)(addr - buf), (unsigned long)(addr + size -1 - buf));
            ShowMapState();
            return;
        }
    }
    else // case c, d
    {
        if (addr + size -1 == tmp_map->next->m_addr -1) // case c
        {
            tmp_map->next->m_addr = addr;
            tmp_map->next->m_size += size;
            printf("freed size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                    size, (unsigned long)(addr - buf), (unsigned long)(addr + size -1 - buf));
            ShowMapState();
            return;
        }
        else // case d
        {
            new_map = (map_m*) malloc(sizeof(map_m));
            new_map->m_addr = addr;
            new_map->m_size = size;
            new_map->prior = tmp_map;
            new_map->next = tmp_map->next;
            new_map->next->prior = new_map;
            tmp_map->next = new_map;
            printf("freed size: %u,\nfrom addr: %lu,\nto   addr: %lu\n",
                    size, (unsigned long)(addr - buf), (unsigned long)(addr + size -1 - buf));
            ShowMapState();
            return;
        }
    }

    printf("free address failed");
    ShowMapState();
    return;
}
```

释放地址合法性判断

在执行地址释放之前，需要判断地址释放的合法性，即释放的空间必须处于初始分配的空间内，并且不能涵盖空闲区。参数为释放空间大小以及释放的起始地址。

```c
// 0: illegal   1: legal
int IfFreeIllegal(unsigned size, char * addr) {
    if (size <= 0) return 0;

    if (addr < buf || addr + size -1 > buf + SIZE - 1) return 0;

    // map is empty
    if (cur_map == NULL) return 1;

    map_m * ser_map = cur_map;

    // addr cannot be in a map's addr area
    do
    {
        if (addr >= ser_map->m_addr && addr <= ser_map->m_addr + ser_map->m_size - 1) return 0;
        ser_map = ser_map->next;
    } while (ser_map != cur_map);

    // a map's m_addr cannot be in addr and addr + size
    do
    {
        if (ser_map->m_addr >= addr && ser_map->m_addr <= addr + size -1) return 0;
        ser_map = ser_map->next;
    } while (ser_map != cur_map);


    return 1;
}
```

退出程序部分

当检测到用户输入 exit 指令时，程序将进入退出部分。

释放空间

主函数中会输出程序关闭提示，接着释放地址空间以及全部空闲表。

```c
    printf("program closed\n");
    free(buf);
                free_all();
    return 0;
```

释放全部空闲表实现：

```c
void free_all(){
    map_m * ser_map = cur_map;
    map_m * to_be_freed;
    int num = getmapnum();
    for (int i = 0; i < num; i++)
    {
        to_be_freed = ser_map;
        ser_map = ser_map->next;
        free(to_be_freed);
    }
    return;
}
```

测试

测试方法

由于用户输入输出均采用了**偏移地址**（实验中**分配了 1000 单位地址，偏移范围为 0~999**），测试相对简单。

测试会使用“./a.out <./test_in/xxx.txt >./test_out/xxx.txt”命令直接生成输出文件，再对比是否符合预期。

测试将按照“循环首次”特性释放的 9 种情况以及部分错误处理分别进行。

为了便于管理输入和输出的测试文件，使用一个**linux shell 程序**通过读取所有测试输入自动生成输出文件，shell 程序如下：

```shell
#!/bin/bash
cd test_in
for filename in *.txt; do
		echo "--------Test using" $filename "--------"
		echo "------------------testing command------------------" >../test_out/$filename.out
		while read line
		do
				echo $line
		done <$filename >>../test_out/$filename.out
		echo "------------------testing result------------------" >>../test_out/$filename.out
		../a.out <$filename >>../test_out/$filename.out
done
cd ..
```

测试过程截图：

`![avatar](./test.png)`

<center>Figure3：测试过程<center/>

测试结果

测试采用了白盒法遍历了程序的全部路径，要把这些结果全部展示出来极占篇幅，因此以下均只显示部分测试生成文件中重要的测试结果

- **循环首次**

```
// loop.txt.out

current maps:
map: m_size = 100, m_addr = 0
map: m_size = 200, m_addr = 200
map: m_size = 300, m_addr = 600

------------------------------------------------------------------
---------
Please enter a commend(You can try type "help"):
applied for size: 50,
from addr: 0,
to   addr: 49

current maps:
map: m_size = 200, m_addr = 200
map: m_size = 300, m_addr = 600
map: m_size = 50, m_addr = 50

------------------------------------------------------------------
---------
Please enter a commend(You can try type "help"):
applied for size: 250,
from addr: 600,
to   addr: 849

current maps:
map: m_size = 50, m_addr = 50
map: m_size = 200, m_addr = 200
map: m_size = 50, m_addr = 850
```

初始空闲表分为100，200，300三块，申请50空间的话会分配100的那一块，同时指针指向后面那一块即200块的表；再次请求申请250空间，则会分配300的那一块，指针指向再后面那一块即又回到了50的空闲表块。这一过程符合“循环首次”特性。

- **释放情况**

```
// free_caseA.txt.out

current maps:
map: m_size = 50, m_addr = 50
map: m_size = 100, m_addr = 200
map: m_size = 300, m_addr = 600

------------------------------------------------------------------
---------
Please enter a commend(You can try type "help"):
freed size: 25,
from addr: 300,
to   addr: 324

current maps:
map: m_size = 50, m_addr = 50
map: m_size = 125, m_addr = 200
map: m_size = 300, m_addr = 600
```

```
// free_caseB.txt.out

current maps:
map: m_size = 50, m_addr = 50
map: m_size = 100, m_addr = 200
map: m_size = 300, m_addr = 600

------------------------------------------------------------------
---------
Please enter a commend(You can try type "help"):
freed size: 300,
from addr: 300,
to   addr: 599

current maps:
map: m_size = 700, m_addr = 200
map: m_size = 50, m_addr = 50
```

```
// free_caseD.txt.out

current maps:
map: m_size = 50, m_addr = 50
map: m_size = 100, m_addr = 200
map: m_size = 300, m_addr = 600

------------------------------------------------------------------
---------
Please enter a commend(You can try type "help"):
freed size: 50,
from addr: 125,
to   addr: 174

current maps:
map: m_size = 50, m_addr = 50
map: m_size = 50, m_addr = 125
map: m_size = 100, m_addr = 200
map: m_size = 300, m_addr = 600
```

可以从测试的输出文件看到程序正确完成了释放空间时空闲表的合并、增加等工作。

- **错误处理**

```
// illegal_m.txt.out

------------------testing command------------------
m 1000
f 100 100
f 300 500
m 150
m 170
exit

------------------testing result------------------
map initialized, m_addr = 0, size = 1000

// 中间代码省略

current maps:
map: m_size = 100, m_addr = 100
map: m_size = 150, m_addr = 650

------------------------------------------------------------------
---------
Please enter a commend(You can try type "help"):
malloc failed!

current maps:
map: m_size = 100, m_addr = 100
map: m_size = 150, m_addr = 650
```

```
// illegal_f3.txt.out

------------------testing command------------------
m 1000
f 100 100
f 300 500
f 500 400
exit

------------------testing result------------------
map initialized, m_addr = 0, size = 1000

// 中间代码省略

current maps:
map: m_size = 100, m_addr = 100
map: m_size = 300, m_addr = 500

------------------------------------------------------------------
---------
Please enter a commend(You can try type "help"):
illegal free command!

current maps:
map: m_size = 100, m_addr = 100
map: m_size = 300, m_addr = 500

------------------------------------------------------------------
---------
Please enter a commend(You can try type "help"):
program closed
```

可见程序对不合法的分配、释放指令均会正确处理。在上述第一个测试案例中，分配的空间大于空闲表任何一块的大小，因此分配失败；第二个测试案例中需要释放的空间已经包含了一个空闲表的地址范围，因此会提示释放失败。

从所有测试结果来看，**程序暂未发现有bug**。

附录

程序改进

目前来看，程序改进主要有代码优化、输出优化等。例如可以进一步把输出分配、释放成功的语句打包成一个函数；检测用户输入的do-while语句可以打包成一个函数；把不同模块相关的功能函数按照文件分类等。

实验心得

通过本次实验，我理解了操作系统动态分配内存空间的基本思想，熟悉了“循环首次适应法”的设计思路，学会了对空闲表的使用。除此之外，通过一步步的实现，我学会了完整的c语言程序编写方法，linux环境对c成语运行、调试、测试的方法。

实验过程并不是一帆风顺的，测试过程经常会发现有错误。比如有一次调试时发现相邻的空闲表块没有自动合并成一块，排查发现是表示空闲表地址空间采用的是m_size ~ m_size + size，但这并不正确，因为m_size + size已经不属于这块空闲表了，因此正确的地址空间应该表示为m_size ~ m_size + size - 1。找到出错原因后，我不得不把先前写的所有分配、释放空间的判断语句以及相应的输出函数全部改写了一遍。

最后感谢老师和助教的指导和答疑使我顺利完成了本次实验。

可见程序对不合法的分配、释放指令均会正确处理。在上述第一个测试案例中，分配的空间大于空闲表任何一块的大小，因此分配失败；第二个测试案例中需要释放的空间已经包含了一个空闲表的地址范围，因此会提示释放失败。

从所有测试结果来看，**程序暂未发现有bug**。

附录

程序改进

目前来看，程序改进主要有代码优化、输出优化等。例如可以进一步把输出分配、释放成功的语句打包成一个函数；检测用户输入的do-while语句可以打包成一个函数；把不同模块相关的功能函数按照文件分类等。

实验心得

通过本次实验，我理解了操作系统动态分配内存空间的基本思想，熟悉了“循环首次适应法”的设计思路，学会了对空闲表的使用。除此之外，通过一步步的实现，我学会了完整的c语言程序编写方法，linux环境对c成语运行、调试、测试的方法。

实验过程并不是一帆风顺的，测试过程经常会发现有错误。比如有一次调试时发现相邻的空闲表块没有自动合并成一块，排查发现是表示空闲表地址空间采用的是m_size ~ m_size + size，但这并不正确，因为m_size + size已经不属于这块空闲表了，因此正确的地址空间应该表示为m_size ~ m_size + size - 1。找到出错原因后，我不得不把先前写的所有分配、释放空间的判断语句以及相应的输出函数全部改写了一遍。

最后感谢老师和助教的指导和答疑使我顺利完成了本次实验。

可见程序对不合法的分配、释放指令均会正确处理。在上述第一个测试案例中，分配的空间大于空闲表任何一块的大小，因此分配失败；第二个测试案例中需要释放的空间已经包含了一个空闲表的地址范围，因此会提示释放失败。

从所有测试结果来看，**程序暂未发现有bug**。

附录

程序改进

目前来看，程序改进主要有代码优化、输出优化等。例如可以进一步把输出分配、释放成功的语句打包成一个函数；检测用户输入的do-while语句可以打包成一个函数；把不同模块相关的功能函数按照文件分类等。

实验心得

通过本次实验，我理解了操作系统动态分配内存空间的基本思想，熟悉了“循环首次适应法”的设计思路，学会了对空闲表的使用。除此之外，通过一步步的实现，我学会了完整的c语言程序编写方法，linux环境对c成语运行、调试、测试的方法。

实验过程并不是一帆风顺的，测试过程经常会发现有错误。比如有一次调试时发现相邻的空闲表块没有自动合并成一块，排查发现是表示空闲表地址空间采用的是m_size ~ m_size + size，但这并不正确，因为m_size + size已经不属于这块空闲表了，因此正确的地址空间应该表示为m_size ~ m_size + size - 1。找到出错原因后，我不得不把先前写的所有分配、释放空间的判断语句以及相应的输出函数全部改写了一遍。

最后感谢老师和助教的指导和答疑使我顺利完成了本次实验。

可见程序对不合法的分配、释放指令均会正确处理。在上述第一个测试案例中，分配的空间大于空闲表任何一块的大小，因此分配失败；第二个测试案例中需要释放的空间已经包含了一个空闲表的地址范围，因此会提示释放失败。

从所有测试结果来看，**程序暂未发现有bug**。

附录

程序改进

目前来看，程序改进主要有代码优化、输出优化等。例如可以进一步把输出分配、释放成功的语句打包成一个函数；检测用户输入的do-while语句可以打包成一个函数；把不同模块相关的功能函数按照文件分类等。

实验心得

通过本次实验，我理解了操作系统动态分配内存空间的基本思想，熟悉了“循环首次适应法”的设计思路，学会了对空闲表的使用。除此之外，通过一步步的实现，我学会了完整的c语言程序编写方法，linux环境对c成语运行、调试、测试的方法。

实验过程并不是一帆风顺的，测试过程经常会发现有错误。比如有一次调试时发现相邻的空闲表块没有自动合并成一块，排查发现是表示空闲表地址空间采用的是m_size ~ m_size + size，但这并不正确，因为m_size + size已经不属于这块空闲表了，因此正确的地址空间应该表示为m_size ~ m_size + size - 1。找到出错原因后，我不得不把先前写的所有分配、释放空间的判断语句以及相应的输出函数全部改写了一遍。

最后感谢老师和助教的指导和答疑使我顺利完成了本次实验。

可见程序对不合法的分配、释放指令均会正确处理。在上述第一个测试案例中，分配的空间大于空闲表任何一块的大小，因此分配失败；第二个测试案例中需要释放的空间已经包含了一个空闲表的地址范围，因此会提示释放失败。

从所有测试结果来看，**程序暂未发现有bug**。

附录

程序改进

目前来看，程序改进主要有代码优化、输出优化等。例如可以进一步把输出分配、释放成功的语句打包成一个函数；检测用户输入的do-while语句可以打包成一个函数；把不同模块相关的功能函数按照文件分类等。

实验心得

通过本次实验，我理解了操作系统动态分配内存空间的基本思想，熟悉了“循环首次适应法”的设计思路，学会了对空闲表的使用。除此之外，通过一步步的实现，我学会了完整的c语言程序编写方法，linux环境对c成语运行、调试、测试的方法。

实验过程并不是一帆风顺的，测试过程经常会发现有错误。比如有一次调试时发现相邻的空闲表块没有自动合并成一块，排查发现是表示空闲表地址空间采用的是m_size ~ m_size + size，但这并不正确，因为m_size + size已经不属于这块空闲表了，因此正确的地址空间应该表示为m_size ~ m_size + size - 1。找到出错原因后，我不得不把先前写的所有分配、释放空间的判断语句以及相应的输出函数全部改写了一遍。

最后感谢老师和助教的指导和答疑使我顺利完成了本次实验。

可见程序对不合法的分配、释放指令均会正确处理。在上述第一个测试案例中，分配的空间大于空闲表任何一块的大小，因此分配失败；第二个测试案例中需要释放的空间已经包含了一个空闲表的地址范围，因此会提示释放失败。

从所有测试结果来看，**程序暂未发现有bug**。

附录

程序改进

目前来看，程序改进主要有代码优化、输出优化等。例如可以进一步把输出分配、释放成功的语句打包成一个函数；检测用户输入的do-while语句可以打包成一个函数；把不同模块相关的功能函数按照文件分类等。

实验心得

通过本次实验，我理解了操作系统动态分配内存空间的基本思想，熟悉了“循环首次适应法”的设计思路，学会了对空闲表的使用。除此之外，通过一步步的实现，我学会了完整的c语言程序编写方法，linux环境对c成语运行、调试、测试的方法。

实验过程并不是一帆风顺的，测试过程经常会发现有错误。比如有一次调试时发现相邻的空闲表块没有自动合并成一块，排查发现是表示空闲表地址空间采用的是m_size ~ m_size + size，但这并不正确，因为m_size + size已经不属于这块空闲表了，因此正确的地址空间应该表示为m_size ~ m_size + size - 1。找到出错原因后，我不得不把先前写的所有分配、释放空间的判断语句以及相应的输出函数全部改写了一遍。

最后感谢老师和助教的指导和答疑使我顺利完成了本次实验。

可见程序对不合法的分配、释放指令均会正确处理。在上述第一个测试案例中，分配的空间大于空闲表任何一块的大小，因此分配失败；第二个测试案例中需要释放的空间已经包含了一个空闲表的地址范围，因此会提示释放失败。

从所有测试结果来看，**程序暂未发现有bug**。

附录

程序改进

目前来看，程序改进主要有代码优化、输出优化等。例如可以进一步把输出分配、释放成功的语句打包成一个函数；检测用户输入的do-while语句可以打包成一个函数；把不同模块相关的功能函数按照文件分类等。

实验心得

通过本次实验，我理解了操作系统动态分配内存空间的基本思想，熟悉了“循环首次适应法”的设计思路，学会了对空闲表的使用。除此之外，通过一步步的实现，我学会了完整的c语言程序编写方法，linux环境对c成语运行、调试、测试的方法。

实验过程并不是一帆风顺的，测试过程经常会发现有错误。比如有一次调试时发现相邻的空闲表块没有自动合并成一块，排查发现是表示空闲表地址空间采用的是m_size ~ m_size + size，但这并不正确，因为m_size + size已经不属于这块空闲表了，因此正确的地址空间应该表示为m_size ~ m_size + size - 1。找到出错原因后，我不得不把先前写的所有分配、释放空间的判断语句以及相应的输出函数全部改写了一遍。

最后感谢老师和助教的指导和答疑使我顺利完成了本次实验。

可见程序对不合法的分配、释放指令均会正确处理。在上述第一个测试案例中，分配的空间大于空闲表任何一块的大小，因此分配失败；第二个测试案例中需要释放的空间已经包含了一个空闲表的地址范围，因此会提示释放失败。

从所有测试结果来看，**程序暂未发现有bug**。

附录

程序改进

目前来看，程序改进主要有代码优化、输出优化等。例如可以进一步把输出分配、释放成功的语句打包成一个函数；检测用户输入的do-while语句可以打包成一个函数；把不同模块相关的功能函数按照文件分类等。

实验心得