

**SHANGHAI JIAO TONG UNIVERSITY**

**实验报告**

LAB REPORT



**操作系统**

实验三 可变分区存储管理

姓 名: 裘炜程

学 号: 516030910287

班 级: F1603602

# 基本算法——循环首次适应法

可变分区存储管理法是操作系统的基本存储管理方法之一，并不会预先将

内存划分成区，而是等到作业运行时向操作系统申请一块空闲的区域，其大小

等于作业所需内存大小，可以用链接法和连续线性表格法实现管理空闲内存区

的数据结构。

循环首次适应法是可变分区存储管理方法的一种，把空闲表设计成顺序结

构或链接结构的循环队列，各空闲区按从低到高登记在空闲区管理队列里，同

时需要设计一个起始查找指针，指向一个空闲区表项。在分配时从起始查找指

针对应的空闲区开始查找，第一次找到满足条件的空闲区时便分配，并调整查

找指针使其指向队列中被分配的后面的那块空闲区，其实质是利用查找指针所

指的空闲区和其后的空闲区群长时间未被利用的特点，它们合成较大的符合要

求的空闲区的概率较大，提高了分配时查找的速度。

本实验中，我选择用双向链表记录空闲区，首先用malloc得到1000字节的内存，然后分配和回收内存来进行该算法的模拟。需要模拟的有两部分：分配和回收。分配时需要循环遍历链表，找到第一个可分配的空闲区；回收时首先要找到新空闲区的位置，然后根据不同情况更新空闲区表——释放区域在所有空闲区之前、释放区域在所有空闲区之后、释放的空闲区和前后空闲区均相邻、释放的空闲区仅与前空闲区相邻、释放的空闲区仅与后空闲区相邻、释放的空闲区与前后空闲区均不相邻。因此整个程序就考虑这些情况，进行分类。

# 程序函数

1、 lmalloc 函数：lmalloc（）函数是用于分配空闲区的函数，该函数的输入是需要分配的 size 大小，无需返回任何值。在接受到 size 的值后，会从起始查找指针开始，遍历整个空闲区寻找第一个满足要求的空闲区并分配内存，并在运行过程中输出本次分配的起始地址信息。

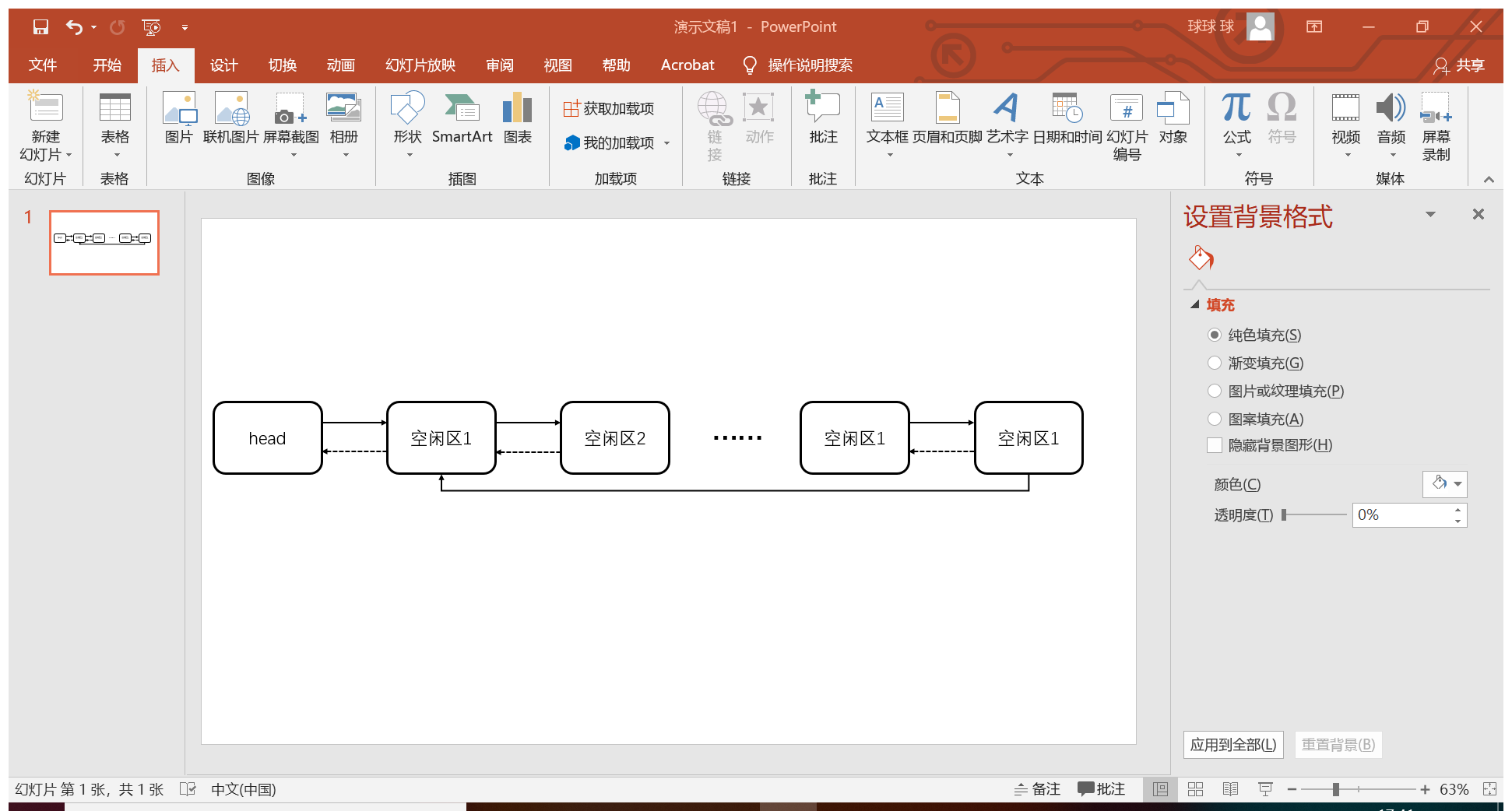
2、 fmalloc 函数：fmalloc（）函数是用于回收空闲区的函数，该函数需要获取需要释放的空间的起始地址和释放空间大小信息，无返回值。函数会释放相应区域回到空闲区，并根据情更新空闲区链表。

3、 main 函数：main（）函数是控制函数，该函数初始化空闲区链表，并根据用户的输入调用 lmalloc 函数、fmalloc 函数或者退出程序。

4、 display 函数：display（）函数是打印空闲区状态的函数，该函数会从第一个空闲区开始，逐个打印空闲区的开始位置和长度，方便测试进程。

# 数据结构

1. 程序主体结构：程序开始后首先调用 malloc 函数向操作系统申请一块1000 字节的空间，然后对头指针所对应的节点和空闲区链表队列进行初始化操作，接下来开始接受键盘输入并执行相应的分配或回收操作，在执行过程中输出信息，在接收到退出指令后退出。
2. 结构体 map：结构体 map 维护的是单个空闲区节点，其中包含指向char 类型的指针 address、表示空闲区大小的无符号整数 size、指向后面的空闲区的指针 next、指向前一个空闲区的指针 prior。
3. 全局变量 head：head 是一个结构体 map 的实例，主要是为了处理释放区位于所有空闲区之前的情况以及显示空闲区状况而设定，head 的address 设为最初的 malloc 函数返回的地址，size 恒为零，prior 置为NULL，next 经过处理永远指向最靠前的空闲区，当没有空闲区时next 会置为空。
4. 全局变量 pt：pt 是初始查找指针，当分配空闲区时会从此指针所在空闲区开始向后查找，直到找到新的空闲区并分配空间，并将初始查找指针指向分配区的后一块空闲区（并不是原来的空闲区）， 另外在回收空闲区的时候，如果更改了初始查找指针的空闲区，要对初始查找指针进行更新。
5. 为实现循环，双向链表需要做一些改变，数据结构如下：



其中实线箭头表示next，虚线剪头表示previous，head前一个节点为NULL，对于查询指针pt，做一个循环的链表。

# 源程序及运行结果

1. 源代码

#include <iostream>

#include <malloc.h>

#include <stdio.h>

using namespace std;

struct map //结构体定义

{

unsigned int size;

char\* address;

struct map \*next,\*prior;

};

struct map\* pt=new map; //记录起始查找指针

struct map\* head=new map; //记录链表头的指针

//分配算法，传入需要分配的内存大小，输出分配的地址

void lmalloc(unsigned size)

{

struct map\* record=pt;

if(pt==NULL) //处理起始查找指针为空的情况！！！！！

{

cout<<"没有符合分配要求的空闲区,情况1"<<endl;

return;

}

if (pt->size < size) //找到满足条件的空闲区

{

do{

pt=pt->next;

if(pt==record) //防止进入死循环

{

cout<<"没有符合分配要求的空闲区，情况2"<<endl;

return;

}

}while(pt->size < size);

}

cout<<"分配的地址为："<<(int)pt->address<<"-"<<(int)(pt->address+size-1)<<" 长度为："<<size<<endl;

if (pt->size > size) //如果空闲区大于需要的长度，直接截取,变化地址和空闲区长度

{

pt->size-=size;

pt->address+=size;

pt=pt->next;

}

else if(pt->size==size) //如果空闲区等于需要的长度，将此链表节点删除并连接前后节点

{

struct map\* rec=pt;

pt->prior->next=pt->next;

pt->next->prior=pt->prior;

if(pt==pt->next) //如果只有这一块了却要删除，要把pt置为空，并把头指针置为空

{

pt=NULL;

head->next=NULL;

}

else if(pt==head->next) //如果正好删掉了第一块，要更新头指针信息

{

head->next=pt->next;

pt=pt->next;

}

else

{

pt=pt->next;

}

delete rec; //删去pt指向的链表节点

}

}

//释放算法，传入要回收的内存大小和起始地址，释放对应空间

void lfree(unsigned size,char\* address)

{

struct map \*tmp\_map=head;

char \*a;

a=address;

if(head->next==NULL) //如果此时一点空闲区也不剩的话 要重新初始化

{

struct map\* newnode=new map;

head->next=newnode;

pt=newnode;

newnode->address=a;

newnode->size=size;

newnode->prior=head;

newnode->next=newnode;

cout<<"已释放"<<(int)a<<"位置处"<<size<<"大小的内存空间，情况0"<<endl;

return;

}

do{

tmp\_map=tmp\_map->next;

if((tmp\_map->next==head->next) && (tmp\_map->address<a) )//如果是链表最后的节点，需要在尾部加上一个节点

{

if(tmp\_map->address+tmp\_map->size==a)

{

tmp\_map->size+=size;

cout<<"已释放"<<(int)a<<"位置处"<<size<<"大小的内存空间，情况1"<<endl;

return;

}

else if(tmp\_map->address+tmp\_map->size<a)

{

struct map\* newnode=new map;

newnode->prior=tmp\_map;

tmp\_map->next->prior=newnode;

newnode->next=tmp\_map->next;

tmp\_map->next=newnode;

newnode->size=size;

newnode->address=a;

cout<<"已释放"<<(int)a<<"位置处"<<size<<"大小的内存空间，情况2"<<endl;

return;

}

}

}while(!(tmp\_map->prior->address<=a && tmp\_map->address>a)); //确定释放区的位置

if(tmp\_map->prior->address+tmp\_map->prior->size==a) //如果正好和前面的空闲区相邻

{

if(tmp\_map->prior->size==0) //如果正好是申请的地址开头，也就是成为第一块空闲区

{

struct map\* newnode=new map; //创建新节点

newnode->prior=tmp\_map->prior;

newnode->next=tmp\_map;

newnode->size=size;

newnode->address=a;

tmp\_map->prior->next=newnode;

tmp\_map->prior=newnode;

head->next=newnode; //更新头指针

if(tmp\_map->next==tmp\_map)

{

tmp\_map->next=newnode;

}

if(a+size==tmp\_map->address) //如果与后面的空闲区相邻

{

tmp\_map->prior->size+=tmp\_map->size;

tmp\_map->prior->next=tmp\_map->next;

if(tmp\_map->next!=newnode)

{

tmp\_map->next->prior=tmp\_map->prior;

struct map\* men=tmp\_map;

do{

men=men->next;

}

while(men->next!=tmp\_map);

men->next=newnode;

}

if(pt==tmp\_map) //更新查找指针

{

pt=tmp\_map->prior;

}

delete tmp\_map;

}

cout<<"已释放"<<(int)a<<"位置处"<<size<<"大小的内存空间，情况3"<<endl;

}

else

{

tmp\_map->prior->size+=size; //仅与前空闲区相邻

if(a+size==tmp\_map->address) //与前后都相邻

{

tmp\_map->prior->size+=tmp\_map->size;

tmp\_map->prior->next=tmp\_map->next;

tmp\_map->next->prior=tmp\_map->prior;

if(pt==tmp\_map) //更新查找指针

{

pt=tmp\_map->prior;

}

delete tmp\_map;

}

cout<<"已释放"<<(int)a<<"位置处"<<size<<"大小的内存空间，情况4"<<endl;

}

}

else

{

if(a+size==tmp\_map->address) //仅与后空闲区相邻

{

tmp\_map->address-=size;

tmp\_map->size+=size;

cout<<"已释放"<<(int)a<<"位置处"<<size<<"大小的内存空间，情况5"<<endl;

}

else{ //都不相邻的话，新增一个节点

struct map\* newnode=new map;

newnode->prior=tmp\_map->prior;

newnode->next=tmp\_map;

newnode->size=size;

newnode->address=a;

tmp\_map->prior->next=newnode;

tmp\_map->prior=newnode;

if(tmp\_map->next==tmp\_map)

{

tmp\_map->next=newnode;

}

cout<<"已释放"<<(int)a<<"位置处"<<size<<"大小的内存空间，情况6"<<endl;

}

}

}

void display(int iaddr)

{

struct map\* pointer=head;

if(head->next==NULL)

{

cout<<"现在没有任何可用的空闲区！"<<endl;

}

else

{

do{

pointer=pointer->next;

cout<<"空闲区范围:"<<(int)pointer->address-iaddr<<'-'<<(int)pointer->address+pointer->size-iaddr<<endl;

}while(pointer->next!=head->next);

}

cout<<endl;

}

int main()

{

char\* iniaddr;

iniaddr= (char\*) malloc(1000); //记录初始地址

cout<<"初始地址为："<<(int)iniaddr<<"共1000字节"<<endl;

head->address=iniaddr; //记录分配到的初始地址

head->size=0;

head->next=pt;

head->prior=NULL;

pt->address=iniaddr; //初始化分配查找指针

pt->size=1000;

pt->prior=head;

pt->next=pt;

char ctr;

unsigned int size;

int addr;

do{

scanf ("%c", &ctr);

if(ctr=='m')

{

scanf("%u",&size);

lmalloc(size);

display(int(iniaddr));

}

else if(ctr=='f')

{

scanf("%u%u",&size,&addr);

char\* ADDR=(char\*) addr;

lfree(size,ADDR);

display(int(iniaddr));

}

else if(ctr=='q')

{

return 0;

}

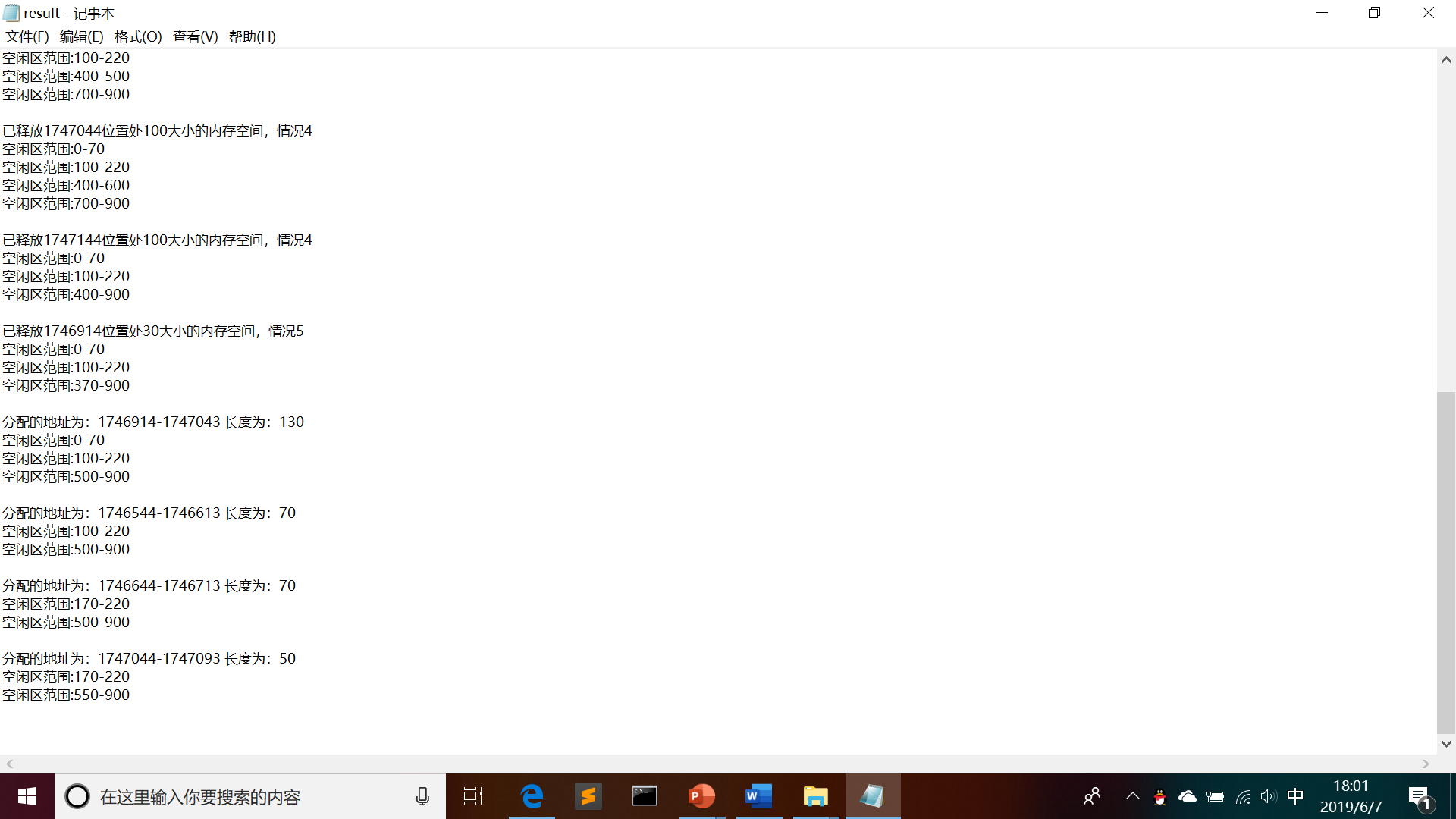
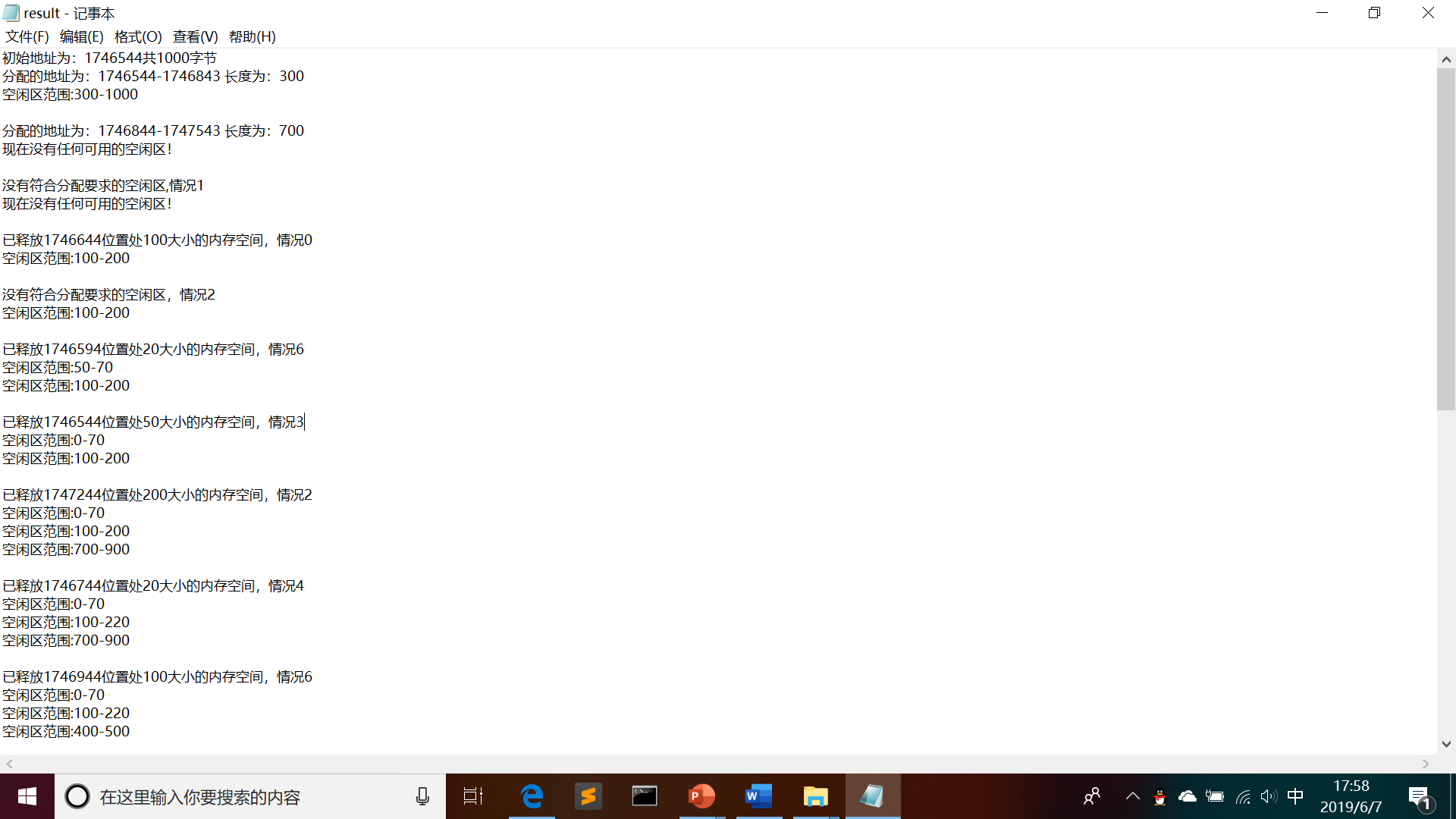
}while (true);

}

1. 输入



1. 输出



# 测试过程分析

我采用了白盒法的路径测试方法，设计了测试流程对程序中的每一条路径都进行了测试，其中包括：

（1） 正常分配时分配区长度大于需要的长度；

（2） 正常分配时分配区长度等于所需的长度；

（3） 正常回收时回收区与前后空闲区均相邻、与前空闲区相邻却与后

空闲区不相邻、与前空闲区不相邻但是与后空闲区相邻、与前后

空闲区均不相邻；

（4） 正常回收时回收区位于所有空闲区之前时，与后面的空闲区相

邻、与后面的空闲区不相邻的情况；

（5） 正常回收时回收区位于所有空闲区之后时，与前面的空闲区相

邻、与前面的空闲区不相邻的情况。

此外，对于各种没有空闲区、有空闲区但是都不能满足分配需求的情况也做了提示。最重要的就是在测试中需要体现“循环”和“首次”，在最后的几次m操作中，就测试了这个部分。

在测试时，配合程序中的各种情况，得到了result.txt中的结果，以下对释放内存时出现的6种情况简单分析：

（1）情况0：之前没有空闲区；

（2）情况1：遍历到最后一个空闲区，并能合并；

（3）情况2：遍历到最后一个空闲区，不能合并，加一个节点；

（4）情况3：在内存区头释放，能和后一节点合并；

（5）情况4：不在内存区头，与前区相邻；

（6）情况5：不在内存区头，与后区相邻；

（7）情况6：不在内存区头，不与前后相邻；

（8）都相邻的情况被包括在4和5中。

# 体会和收获

在本次实验中，主要学习了循环首次适应法的具体实现。由于对c语言的指针、分配内存等不太熟悉，还是选择了C++来实现。写代码的过程中复习了指针的用法、结构体的用法以及全局变量和静态变量的用法，对语言的细节有了更深的了解。其次我对操作系统的可变内存存储管理的方法也有了更深入的理解，循环首次适应法在首次适应法的基础上，利用长时间未分配的空闲区容易变大的特点增加了查找效率。在整个写代码的过程中，有很多情况一开始没有考虑到，在与同学的讨论中逐渐完善分类，最后才得到比较完整的算法。