

hzau-1

**实验3**

**存储管理**

姓 名：马家乐

学 号: 2019310220321

专业班级：信息1903班

课程名：操作系统实验

指导教师：倪福川

完成日期：2019.6.1

**一、实验目的**

**1. 目的**

提高内存管理的效率始终是操作系统研究的重要课题之一，虚拟存储技术是用来提高存储容量的一种重要方法，所以，本项实验的目的是让学生独立地设计几个常用的存储分配算法，并用高级语言编写程序对各种算法进行分析比较，评测其性能的优劣，从而加深对这些算法的了解。

**2. 要求**

本实验要求学生用C语言独立编写分区分配算法、回收算法、请求式分页分配算法。在请求式分页分配算法中，通过程序的执行结果来分析计算不同页面淘汰算法情况下的访问命中率，并以此来比较各种算法的优劣，同时，还要求分析改变页面大小和实际存储容量对计算结果的影响，为选择好的算法，合适的页面尺寸和实存容量提供依据。

本次实验的上机时间为2～4学时。

**二、实验内容**

**1. 分区分配算法**

本实验要求采用首次适应算法和最佳适应算法两种分区分配的内存管理算法。

(1) 建立分区描述器：分区描述器可根据自己编写程序的需要来建立，描述器本身所包含的内容以描述清楚内存分区情况为准。

(2) 建立自由主存队列：针对两种不同的放置策略（首次适应和最佳适应）来建立相应的队列结构。

(3) 用C语言编写实现首次适应算法和最佳适应算法的程序。

(4) 用C语言编写回收算法。

**2. 请求式分页存储管理算法**

本实验要求采用请求式分页存储算法，淘汰算法采用先进先出算法FIFO和最近最少使用页面淘汰算法（LRU）。

**三、实验设计**

**1. 分区分配算法**

**任务分析：**

首次适应算法从空闲[分区表](https://baike.baidu.com/item/%E5%88%86%E5%8C%BA%E8%A1%A8/215102" \t "_blank)的第一个表目起查找该表，把最先能够满足要求的空闲区分配给作业，这种方法目的在于减少查找时间。为适应这种算法，空闲分区表(空闲区链)中的空闲分区要按地址由低到高进行排序。该算法优先使用低址部分空闲区，在低址空间造成许多小的空闲区，在高[地址空间](https://baike.baidu.com/item/%E5%9C%B0%E5%9D%80%E7%A9%BA%E9%97%B4/1423980" \t "_blank)保留大的空闲区。

最佳适应算法是指从全部空闲区中找出能满足作业要求且大小最小的空闲分区的一种计算方法，这种方法能使碎片尽量小。

**概要设计：**

定义内存的大小为640

#define MAX\_length 640

定义空闲分区结构

typedef struct freeArea//定义空闲分区结构

{

int flag;

int size;

int ID;

int address;

}Elemtype;

定义空闲分区的链表结点

typedef struct Free\_Node//定义空闲分区的链表结点

{

Elemtype date;

struct Free\_Node \*front;

struct Free\_Node \*next;

}Free\_Node,\*FNodeList;

定义空闲分区链表的头指针和尾指针

FNodeList block\_first;

FNodeList block\_last;

定义功能函数模板：

int alloc(int tag);//内存分配

int free(int ID);//内存回收

int first\_fit(int ID,int size);//首次适应算法

int best\_fit(int ID,int size);//最佳适应算法

void show();//查看分配

void init();//初始化分区链表

void menu();//菜单

初始化分区链表；

输入要进行的操作：1-首次适应算法，2-最佳适应算法，3-内存回收，4-显示内存状况，5-退出，根据输入的操作分别调用不同的函数。

主函数流程：

调用菜单函数

**源代码：**

#include <iostream>

#include <stdlib.h>

using namespace std;

#define FREE 0

#define BUSY 1

#define MAX\_length 640

typedef struct freeArea//定义空闲分区结构

{

int flag;

int size;

int ID;

int address;

}Elemtype;

typedef struct Free\_Node//定义空闲分区的链表结点

{

Elemtype date;

struct Free\_Node \*front;

struct Free\_Node \*next;

}Free\_Node,\*FNodeList;

FNodeList block\_first;

FNodeList block\_last;

int alloc(int tag);//内存分配

int free(int ID);//内存回收

int first\_fit(int ID,int size);//首次适应算法

int best\_fit(int ID,int size);//最佳适应算法

void show();//查看分配

void init();//初始化

void menu();//菜单

void init()//

{

block\_first=new Free\_Node;

block\_last =new Free\_Node;

block\_first->front=NULL;

block\_first->next=block\_last;

block\_last->front=block\_first;

block\_last->next=NULL;

block\_last->date.address=0;

block\_last->date.flag=FREE;

block\_last->date.ID=0;

block\_last->date.size=MAX\_length;

}

int alloc(int tag) //实现内存分配

{

int ID,size1;

cout<<"请输入作业号：";

cin>>ID;

cout<<"请输入所需内存大小：";

cin>>size1;

if (ID<=0 || size1<=0)

{

cout<<"ERROR,请输入正确的ID和请求大小"<<endl;

return 0;

}

if (tag==1)//采用首次适应算法

{

if(first\_fit(ID,size1))

{

cout<<"分配成功！"<<endl;

}

else cout<<"分配失败！"<<endl;

return 1;

}

else

{

if (best\_fit(ID,size1))

{

cout<<"分配成功！"<<endl;

}

else cout<<"分配失败！"<<endl;

return 1;

}

}

int first\_fit(int ID,int size)//首次适应算法

{

FNodeList temp=(FNodeList)malloc(sizeof(Free\_Node));

Free\_Node \*p=block\_first->next;

temp->date.ID=ID;

temp->date.size=size;

temp->date.flag=BUSY;

while(p)

{

if (p->date.flag==FREE && p->date.size==size)//请求大小刚好满足

{

p->date.flag=BUSY;

p->date.ID=ID;

return 1;

break;

}

if (p->date.flag==FREE && p->date.size>size)//说明还有其他的空闲区间

{

temp->next=p;

temp->front=p->front;

temp->date.address=p->date.address;

p->front->next=temp;

p->front=temp;

p->date.address=temp->date.address+temp->date.size;

p->date.size-=size;

return 1;

break;

}

p=p->next;

}

return 0;

}

int best\_fit(int ID,int size)//最佳适应算法

{

int surplus;//记录可用内存与需求内存的差值

FNodeList temp=(FNodeList)malloc(sizeof(Free\_Node));

Free\_Node \*p=block\_first->next;

temp->date.ID=ID;

temp->date.size=size;

temp->date.flag=BUSY;

Free\_Node \*q=NULL;//记录最佳位置

while(p)//遍历链表，找到第一个可用的空闲区间将他给q

{

if (p->date.flag==FREE&&p->date.size>=size)

{

q=p;

surplus=p->date.size-size;

break;

}

p=p->next;

}

while(p)//继续遍历，找到更加合适的位置

{

if (p->date.flag==FREE&&p->date.size==size)

{

p->date.flag=BUSY;

p->date.ID=ID;

return 1;

break;

}

if (p->date.flag==FREE&&p->date.size>size)

{

if (surplus>p->date.size-size)

{

surplus=p->date.size-size;

q=p;

}

}

p=p->next;

}

if (q==NULL)//如果没有找到位置

{

return 0;

}

else//找到了最佳位置

{

temp->next=q;

temp->front=q->front;

temp->date.address=q->date.address;

q->front->next=temp;

q->front=temp;

q->date.size=surplus;

q->date.address+=size;

return 1;

}

}

int free(int ID)//内存回收

{

Free\_Node \*p=block\_first->next;

while(p)

{

if (p->date.ID==ID)

{

p->date.flag=FREE;

p->date.ID=FREE;

//判断P与前后区域关系

if (p->front->date.flag==FREE&&p->next->date.flag!=FREE)

{

p->front->date.size+=p->date.size;

p->front->next=p->next;

p->next->front=p->front;

break;

}

if (p->front->date.flag!=FREE&&p->next->date.flag==FREE)

{

p->date.size+=p->next->date.size;

if(p->next->next)

{

p->next->next->front=p;

p->next = p->next->next;

}

else p->next=p->next->next;

break;

}

if(p->front->date.flag==FREE&&p->next->date.flag==FREE)

{

p->front->date.size+=p->date.size+p->next->date.size;

if(p->next->next)

{

p->next->next->front=p->front;

p->front->next=p->next->next;

}

else p->front->next=p->next->next;

break;

}

if(p->front->date.flag!=FREE&&p->next->date.flag!=FREE)

{

break;

}

}

p=p->next;

}

cout<<"回收成功！"<<endl;

return 1;

}

void show()

{

cout<<"------------------"<<endl;

cout<<"内存分配情况"<<endl;

cout<<"------------------"<<endl;

Free\_Node \*p=block\_first->next;

while(p)

{

cout<<"分区号：";

if (p->date.ID==FREE)

cout<<"FREE"<<endl;

else cout<<p->date.ID;

cout<<"起始地址："<<p->date.address<<endl;

cout<<"内存大小："<<p->date.size<<endl;

cout<<"分区状态：";

if (p->date.flag==FREE)

cout<<"空闲"<<endl;

else

cout<<"已分配"<<endl;

cout<<"------------------"<<endl;

p=p->next;

}

}

void menu()//菜单

{

int tag=0;

int ID;

init();

cout<<"分区模拟："<<endl;

while(tag!=5)

{

cout<<"请输入要进行的操作："<<endl;

cout<<"1-首次适应算法，2-最佳适应算法，3-内存回收，4-显示内存状况，5-退出"<<endl;

cin>>tag;

switch(tag)

{

case 1:

alloc(tag);

break;

case 2:

alloc(tag);

break;

case 3:

cout<<"请输入需要回收的ID号："<<endl;

cin>>ID;

free(ID);

break;

case 4:

show();

break;

}

}

}

int main()

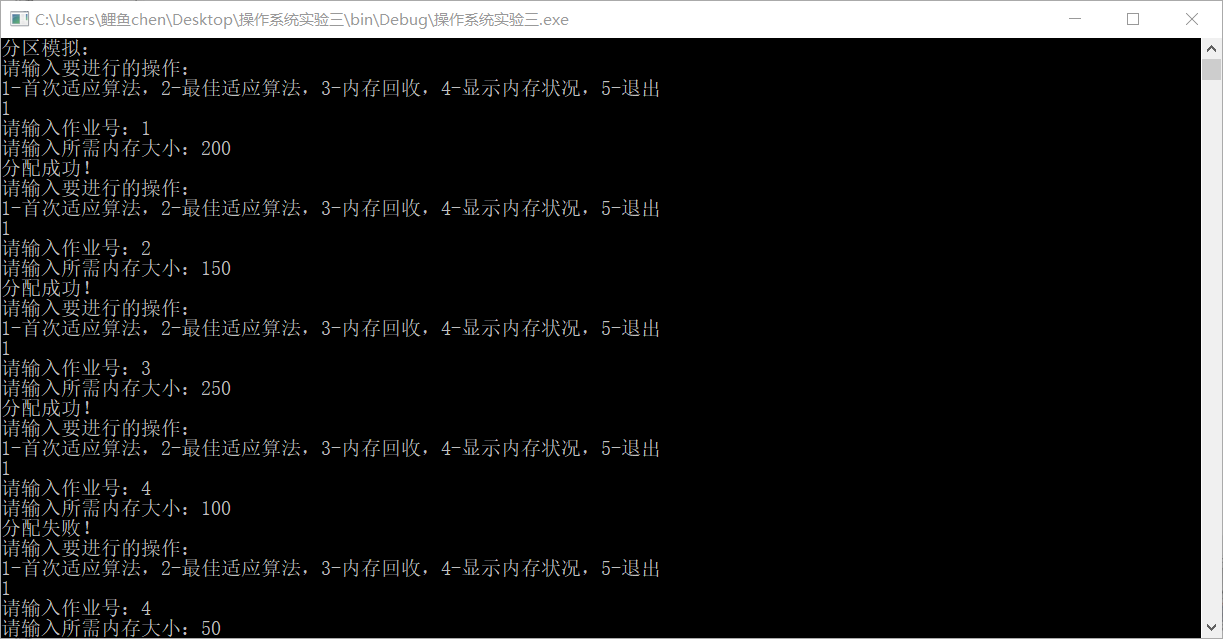
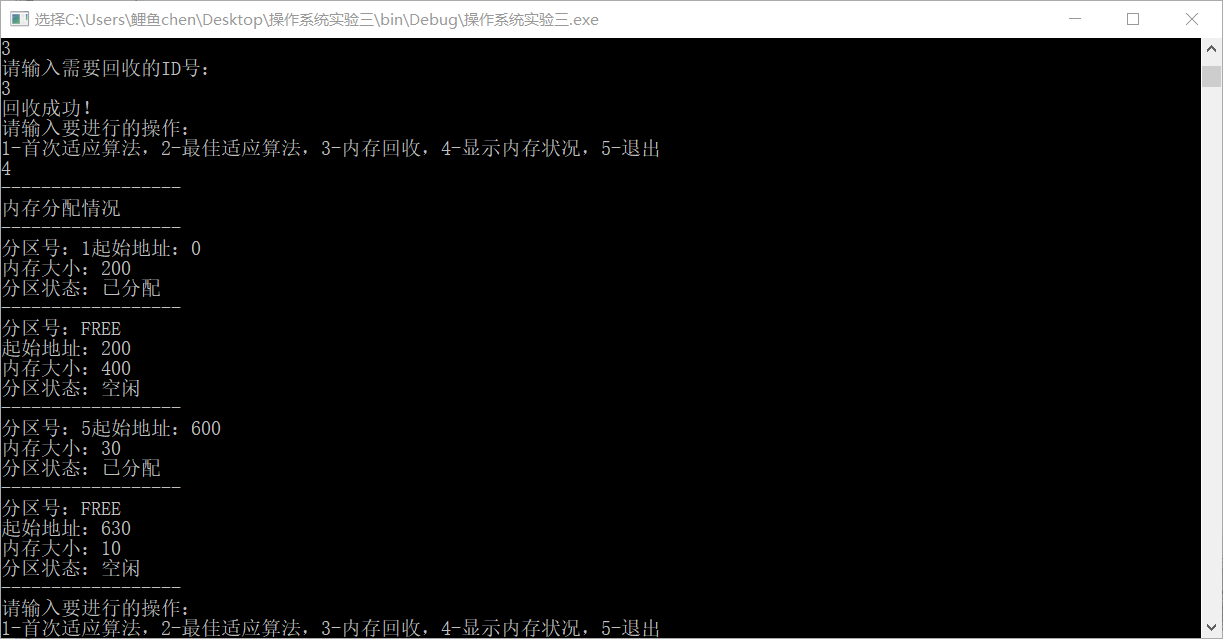
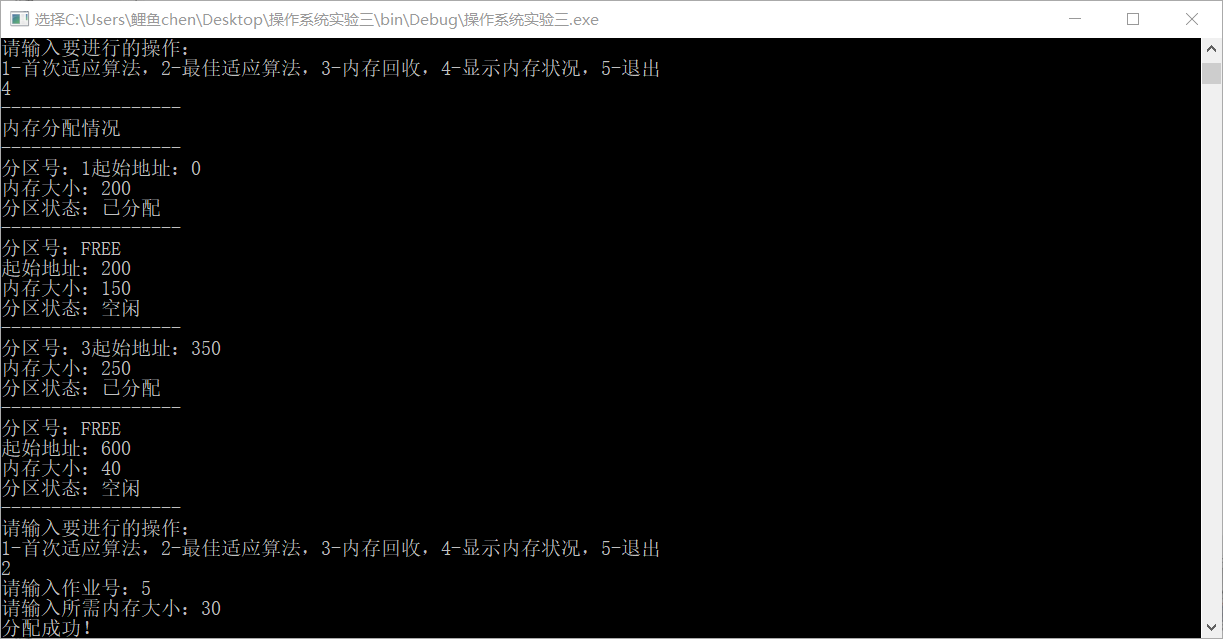
{

menu();

return 0;

}

**输入输出示例：**

![C:\Users\鲤鱼chen\AppData\Roaming\Tencent\Users\252697194\QQ\WinTemp\RichOle\`Y{S}HNHKFQYC8SF0G{0](W.png](data:image/png;base64,)

**2. 请求式分页存储管理算法**

**任务分析：**

先进先出算法FIFO：该算法的实质是选择作业中在主存驻留时间最长的一页淘汰，这种算法容易实现，例如分配一个作业的存储块数为m，则只需建立一张m个元素的队列表Q(0)、Q(1)、…、Q(m-1)和一个替换指针。这个队列是按页调入主存的一页。如图4-1所示，某时刻调入主存四个块，（即m=4），它们按页进入主存的先后顺序为4、5、1、2，当需要置换时，总是淘汰替换指针所指向的那一页，见图3.1.

（指向最老的一页）

页 号

2

4

5

1

替换指针

图3.1 队列表的实现示意图

新调进的页面装入主存后，修改相应的队列元素，然后将替换指针往前移动，使其指向当前最老的一页。

最近最少使用页面淘汰算法（LRU）：这是一种经常使用的方法，有多种不同的实施方案。这里采用不断调整页表链的方法，即总是淘汰页表链链首的页面，而把新访问的页面插入链尾。如果当前调用页面已在页表内，则把它再次调整到链尾。这样就能保证最近使用的页面总处于靠近链尾部分，而不常使用的页面被移到链首，逐个被淘汰。

**概要设计：**

定义页面结构体

typedef struct page{

int id;

struct page \*next;

}Page;

定义页面链表

typedef struct pages{

int count;

Page \*head;Page \*tail;

}Pages;

Pages \*pages=(Pages\*)malloc(sizeof(Pages)) ;

定义功能函数模板：

void Init( )；//初始化页面链表

void Insert(int i)；//插入页面

void Out( )；//页面调出

int Check(int i)；//在链表中查找,若找到则返回1，否则返回0

int Move(int i)；//在链表中查找,若找到则返回1并且移动到链尾，否则返回0

void Print( )；//打印页面链表

void FIFO(int A[])；//先进先出算法

void LRU(int A[])；//最近最少使用页面淘汰算法

主函数流程：

初始化页面链表->输入作业的页面调入串->按先进先出算法进行页面置换，并计算命中率->初始化页面链表->按最近最少使用页面淘汰算法进行页面置换，并计算命中率。

程序框图：

开 始

产生给定长度符合某种假定的指令地址流

Alg=FIFO/LRU?

为每一指令地址形

成对应的访问页号

置初值：size=1

assigned=4

输入淘汰算法Alg

用FIFO淘汰算法计算命中率

最近最少使用页面淘汰算法计算命中率

输出完整结果

结 束

请求式分页存储算法流程图

**源代码：**

#include <iostream>

#include <stdlib.h>

using namespace std;

#define M 12//作业的页面数

//定义页面

typedef struct page{

int id;

struct page \*next;

}Page;

//定义页面链表

typedef struct pages{

int count;

Page \*head;Page \*tail;

}Pages;

Pages \*pages=(Pages\*)malloc(sizeof(Pages)) ;

//初始化页面链表

void Init( ){

pages->count=0;

pages->head=NULL;

pages->tail=NULL;

}

//插入页面

void Insert(int i){

page \*p=(Page\*)malloc(sizeof(Page));

p->id=i;

p->next=NULL;

if(pages->head==NULL)

{

pages->head=pages->tail=p;

}

else{

pages->tail->next=p;

pages->tail=p;

}

pages->count++;

}

//页面调出

void Out( ){

pages->head=pages->head->next;

pages->count--;

}

//在链表中查找,若找到则返回1，否则返回0

int Check(int i){

for(page \*p=pages->head;p!=NULL;p=p->next)

{

if(p->id==i){return 1;}

}

return 0;

}

//在链表中查找,若找到则返回1并且移动到链尾，否则返回0

int Move(int i){

if(pages->head==NULL){return 0;}

else if(pages->head->id==i){pages->head=pages->head->next;pages->count--;Insert(i);return 1;}

else{

for(page \*p=pages->head;p->next!=NULL;p=p->next)

{

if(p->next->id==i){

p->next=p->next->next;

pages->count--;

Insert(i);

return 1;

}

}

return 0;

}

}

//打印页面链表

void Print( ){

cout<<"内存中的页面：";

for(page \*p=pages->head;p!=NULL;p=p->next)

{

cout<<p->id<<" ";

}

cout<<endl;

}

//先进先出算法

void FIFO(int A[]){

int n=0;

for(int i=0;i<M;i++)

{

if(Check(A[i]))

{ Print();

n++;

}

else

{

if(pages->count<4){Insert(A[i]);Print();}

else{Out();Insert(A[i]);Print( );}

}

}

cout<<"FIFO的命中率为: "<<(float)n/M;

}

//最近最少使用页面淘汰算法

void LRU(int A[]){

int n=0;

for(int i=0;i<M;i++)

{

if(Move(A[i]))

{ Print();

n++;

}

else

{

if(pages->count<4){Insert(A[i]);Print();}

else{Out();Insert(A[i]);Print( );}

}

}

cout<<"LRU的命中率为: "<<(float)n/M;

}

int main(){

Init( );

int A[100];

cout<<"请输入作业的页面调入串：";

for(int i=0;i<M;i++){int j;cin>>j;A[i]=j;}

cout<<"------------先进先出算法------------"<<endl;

FIFO(A);

cout<<endl;

cout<<"------------最近最少使用页面淘汰算法------------"<<endl;

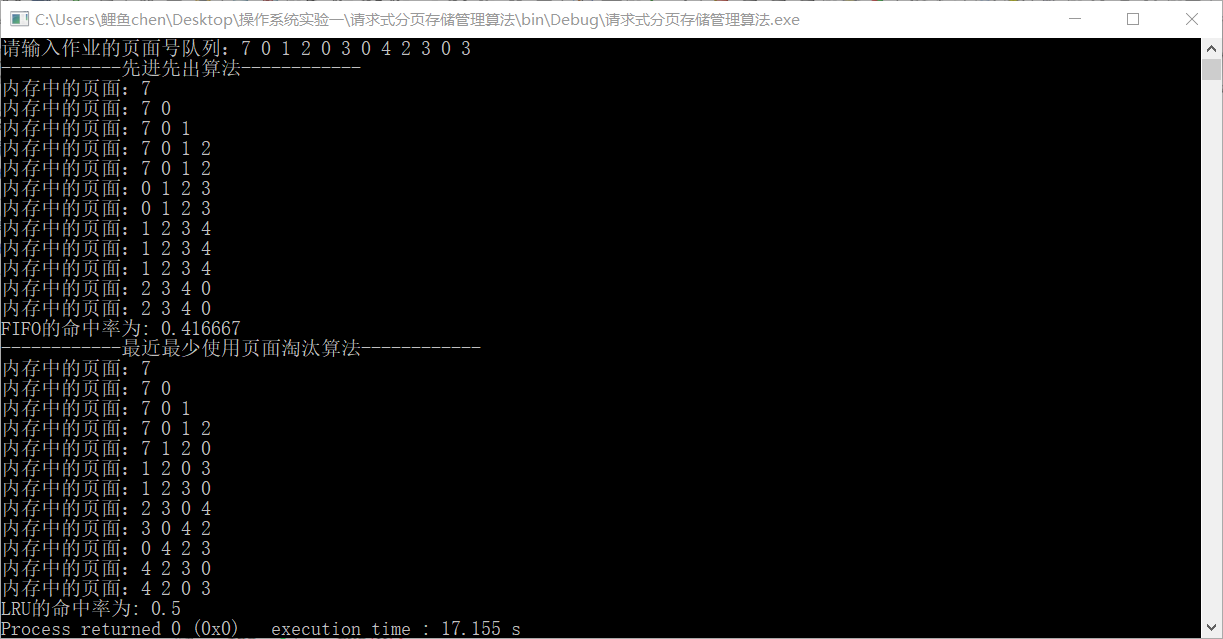
Init( );

LRU(A);

return 0;

}

**输入输出示例：**



**四、实验心得**

通过这次实验， 我了解了内存存储管理的两种方式——分区存储管理和请求式分页存储管理，掌握了首次适应算法和最佳适应算法两种分区分配的内存管理算法，同时对请求式分页存储管理算法中的先进先出算法和最近最少使用页淘汰算法有了更深入的了解，为之后的学习奠定了基础。