操作系统试验指导

一．课程的性质、目的和任务

操作系统在整个计算机系统软件中占有中心地位。 其作用是对计算机系统进行统一的调度和管理， 提供各种强有力的系统服务， 为用户创造既灵活又方便的使用环境。 本课程是计算机及应用专业的一门专业主干课和必修课。通过本课程的学习 ,使学生掌握操作系统的基本概念、设计原理及实施技术 ,具有分析操作系统和设计、实现、开发实际操作系统的能力。

二．实验的意义和目的

操作系统是计算机专业学生的一门重要的专业课程。 操作系统质量对整个计算机系统的性能和用户对计算机的使用有重大的影响。 一个优良的操作系统能极大地扩充计算机系统的功能， 充分发挥系统中各种设备的使用效率， 提高系统工作的可靠性。 由于操作系统涉及计算机系统中各种软硬件资源的管理， 内容比较繁琐， 具有很强的实践性。 要学好这门课程， 必须把理论与实践紧密结合， 才能取得较好的学习效果。 培养计算机专业的学生的系统程序设计能力，是操作系统课程的一个非常重要的环节。通过操作系统上机实验，可以培养学生程序设计的方法和技巧， 提高学生编制清晰、 合理、 可读性好的系统程序的能力，加深对操作系统课程的理解。使学生更好地掌握操作系统的基本概念、基本原理、及基本功能 ,具有分析实际操作系统、设计、构造和开发现代操作系统的基本能力。

三.实验运行环境及上机前的准备

实验运行环境 ：C 语言编程环境

上机前的准备工作包括：

按实验指导书要求事先编好程序；

准备好需要输入的中间数据；

估计可能出现的问题；

预计可能得到的运行结果。

四．实验内容及安排

实验内容包括进程调度、银行家算法、页式地址重定位模拟， LRU 算法模拟和先来先服务算法五个实验。每个实验介绍了实习的目的要求、内容和方法。

实验一、进程调度试验

[目的要求 **]**

用高级语言编写和调试一个进程调度程序，以加深对进程的概念及进程调度算法的理解.

[准备知识 **]**

一、基本概念

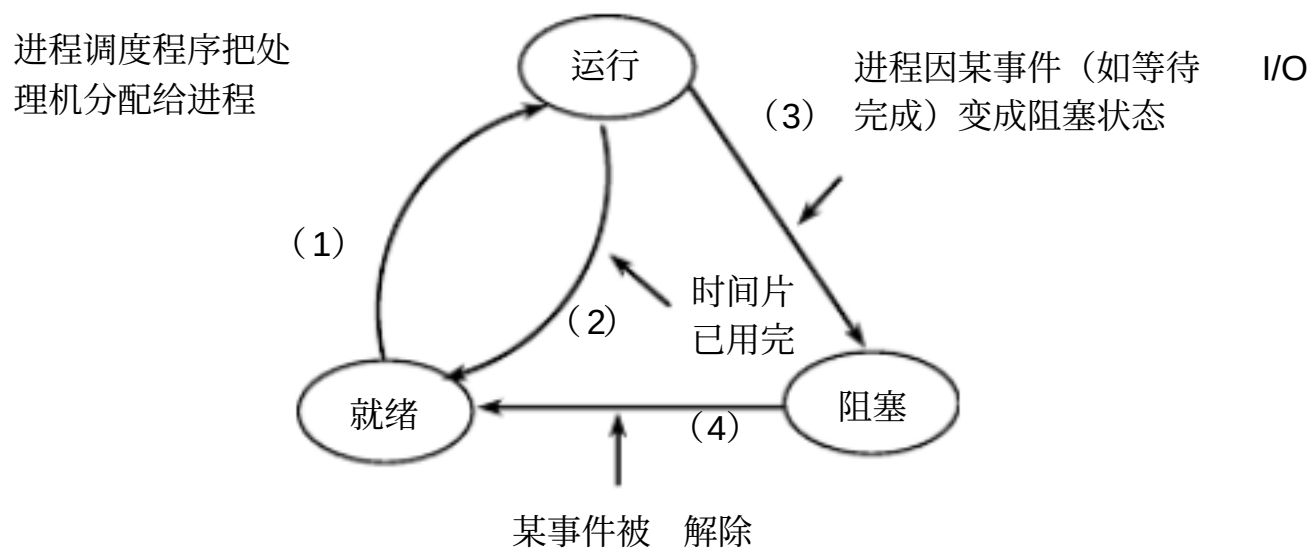
1、进程的概念；

2、进程的状态和进程控制块；

3、进程调度算法；

二、进程调度

1、进程的状态



2、进程的结构——PCB

进程都是由一系列操作 (动作)所组成，通过这些操作来完成其任务。因此，不同的进程，其内部操作也不相同。在操作系统中，描述一个进程除了需要程序和私有数据之外，最主要的是需要一个与动态过程相联系的数据结构，该数据结构用来描述进程的外部特性(名字、状态等)以及与其它进程的联系 (通信关系)等信息，该数据结构称为进程控制块(PCB， Process Control Block) 。

进程控制块 PCB 与进程一一对应， PCB 中记录了系统所需的全部信息、用于描述进程情况所需的全部信息和控制进程运行所需的全部信息。因此，系统可以通过进程的 PCB 来对进程进行管理。

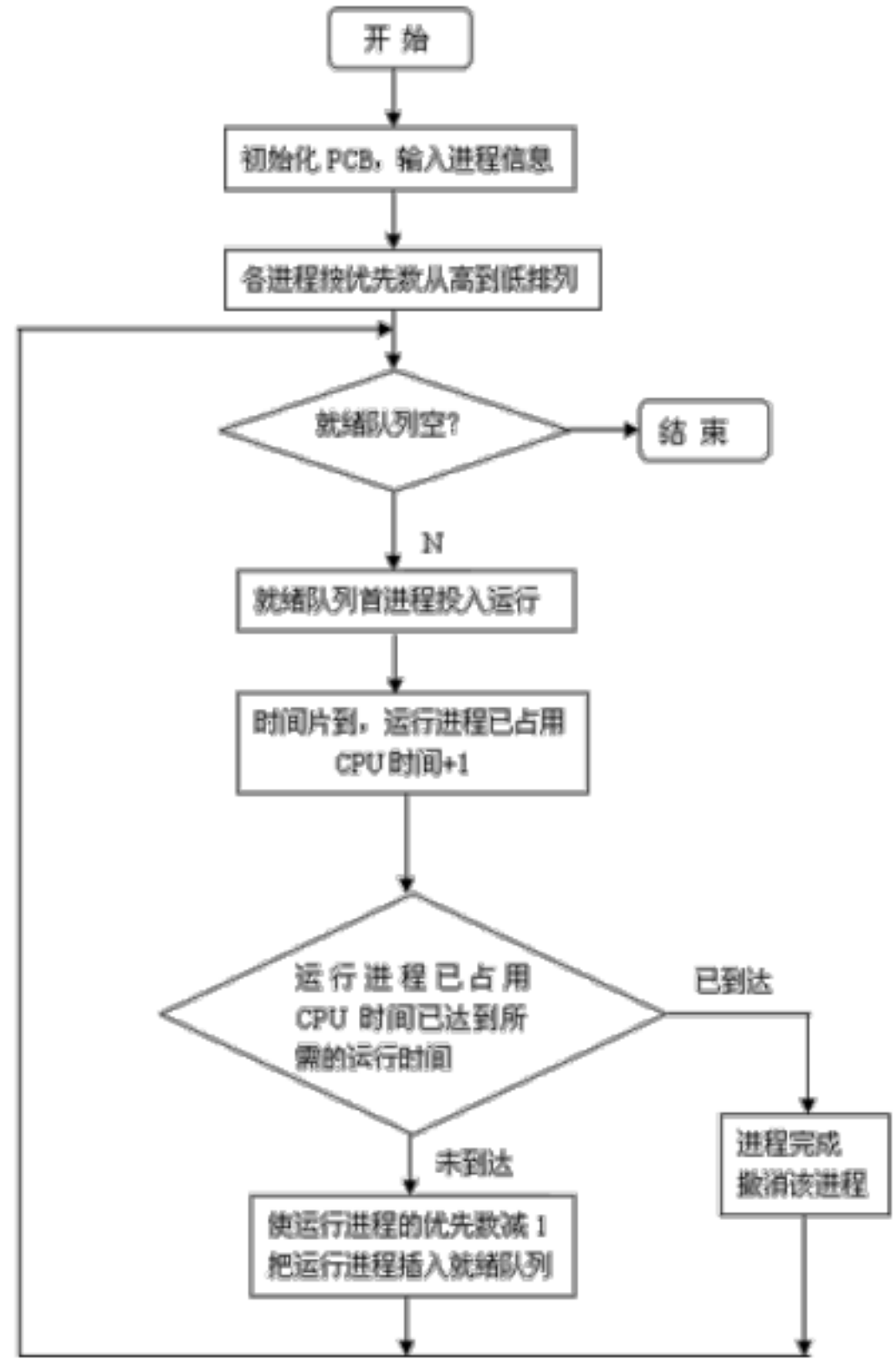
[试验内容 **]**

设计一个有 N 个进程共行的进程调度程序。

进程调度算法： 采用最高优先数优先的调度算法 （即把处理机分配给优先数最高的进程）和先来先服务算法。每个进程有一个进程控制块（ PCB）表示。进程控制块可以包含如下信息：进程名、优先数、到达时间、需要运行时间、已用 CPU 时间、进程状态等等。 进程的优先数及需要的运行时间可以事先人为地指定 （也可以由随机数产生） 。进程的到达时间为进程输入的时间。 进程的运行时间以时间片为单位进行计算。 每个进程的状态可以是就绪 W（Wait）、运行 R（Run）、或完成 F（Finish）三种状态之一。就绪进程获得 CPU 后都只能运行一个时间片。用已占用 CPU 时间加 1 来表示。如果运行一个时间片后，进程的已占用 CPU 时间已达到所需要的运行时间，则撤消该进程，如果运行一

个时间片后进程的已占用 CPU 时间还未达所需要的运行时间，也就是进程还需要继续运行，此时应将进程的优先数减 1（即降低一级），然后把它插入就绪队列等待 CPU。每进行一次调度程序都打印一次运行进程、就绪队列、以及各个进程的 PCB，以便进行检查。重复以上过程，直到所要进程都完成为止。

调度算法的流程图如下：



进程调度源程序如下：

jingchendiaodu.cpp

```
#include "stdio.h"
#include <stdlib.h>
#include <conio.h>
#define getpch(type) (type*)malloc(sizeof(type))
#define NULL 0
struct pcb { /*  定义进程控制块  PCB */
char name[10];
char state; /* 状态 */
int super; /* 优先级 */
int ntime;
int rtime;
struct pcb* link; /*  结构体  pcb 的指针类型  */
```

```
}*ready=NULL,*p;/*    指向当前用行的    pcb*/
typedef struct pcb PCB;


sort() /*   建立对进程进行优先级排列函数      */
{
PCB *first, *second;
int insert=0;
if((ready==NULL)||((p->super)>(ready->super))) /*     优先级最大者   ,插入队首  */
{
p->link=ready;
ready=p;
}
else /*   进程比较优先级   ,插入适当的位置中    */
{
first=ready;
second=first->link;
while(second!=NULL)
{
if((p->super)>(second->super)) /*   若插入进程比当前进程优先数大      ,*/
{ /*  插入到当前进程前面    */
p->link=second;
first->link=p;
second=NULL;
insert=1;
}
else /*   插入进程优先数最低    ,则插入到队尾  */
{
first=first->link;
second=second->link;
}
}
if(insert==0) first->link=p;
}
}


input() /*    建立进程控制块函数    */
{
int i,num;
clrscr(); /*  清屏 */
printf("\n    请输入进程号   ?");
scanf("%d",&num);
for(i=0;i<num;i++)
```

```
{
printf("\n    进程号   No.%d:\n",i);
p=getpch(PCB);
printf("\n    输入进程名  :");
scanf("%s",p->name);
printf("\n    输入进程优先数    :");
scanf("%d",&p->super);
printf("\n    输入进程运行时间    :");
scanf("%d",&p->ntime);
printf("\n");
p->rtime=0;p->state='w';
p->link=NULL;
sort(); /*    调用   sort 函数 */
}
}
int space()
{
int l=0; PCB* pr=ready;
while(pr!=NULL)
{
l++;
pr=pr->link;
}
return(l);
}
disp(PCB * pr) /*   建立进程显示函数    ,用于显示当前进程    */
{
printf("\n qname \t state \t super \t ndtime \t runtime \n");
printf("|%s\t",pr->name);
printf("|%c\t",pr->state);
printf("|%d\t",pr->super);
printf("|%d\t",pr->ntime);
printf("|%d\t",pr->rtime);
printf("\n");
}

check() /*    建立进程查看函数      */
{
PCB* pr;
printf("\n ****       当前正在运行的进程是     :%s",p->name); /*  显示当前运行进程    */
disp(p);
pr=ready;
printf("\n ****     当前就绪队列状态为     :\n"); /*  显示就绪队列状态    */
while(pr!=NULL)
```

```
{
disp(pr);
pr=pr->link;
}
}
destroy() /* 建立进程撤消函数　(进程运行结束　,撤消进程　)*/
{
printf("\n　进程　[%s]　已完成 .\n",p->name);
free(p);
}
running() /*　建立进程就绪函数　(进程运行时间到　,置就绪状态　*/
{
(p->rtime)++;
if(p->rtime==p->ntime)
destroy(); /*　调用 destroy 函数 */
else
{
(p->super)--;
p->state='w';
sort(); /* 调用 sort 函数 */
}
}
main() /* 主函数 */
{
int len,h=0;
char ch;
input();
len=space();
while((len!=0)&&(ready!=NULL))
{
ch=getchar();
h++;
printf("\n The execute number:%d \n",h);
p=ready;
ready=p->link;
p->link=NULL;
p->state='R';
check();
running();
printf("\n　按任一键继续　.....");
ch=getchar();
}
printf("\n\n　进程已经完成　.\n");
ch=getchar(); }
```

修改之后：

```
#include "stdio.h"
#include <stdlib.h>
#include <conio.h>
#define getpch(type) (type*)malloc(sizeof(type))
#define NULL 0
struct pcb { /*    定义进程控制块     PCB */
    char name[10];
    char state; /*状态 */
    int super; /* 优先级  */
    int ntime;
    int rtime;
    struct pcb* link; /*  结构体   pcb 的指针类型  */
}*ready=NULL,*p;/*     指向当前用行的     pcb*/
typedef struct pcb PCB;


void sort() /*    建立对进程进行优先级排列函数       */
{
    PCB *first, *second;
    int insert=0;
    if((ready==NULL)||((p->super)>(ready->super))) /*     优先级最大者   ,插入队首  */
    {
        p->link=ready;
        ready=p;
    }
    else /*  进程比较优先级    ,插入适当的位置中     */
    {
        first=ready;
        second=first->link;
        while(second!=NULL)
        {
            if((p->super)>(second->super)) /*  若插入进程比当前进程优先数大       ,*/
            { /*  插入到当前进程前面    */
                p->link=second;
                first->link=p;
                second=NULL;
                insert=1;
            }
            else /*  插入进程优先数最低     ,则插入到队尾   */
            {
                first=first->link;
                second=second->link;
            }
```

```
            }
            if(insert==0) first->link=p;
        }
    }


void input() /*    建立进程控制块函数    */
{
    int i,num;
    //clrscr(); /*  清屏 */
    printf("\n    请输入进程个数    :");
    scanf("%d",&num);
    for(i=0;i<num;i++)
    {
        printf("\n    进程号    No.%d:\n",i);
        p=getpch(PCB); /*  为进程分配一块空间    */
        printf("\n    输入进程名  :");
        scanf("%s",p->name);
        printf("\n    输入进程优先数    :");
        scanf("%d",&p->super);
        printf("\n    输入进程运行时间    :");
        scanf("%d",&p->ntime);
        printf("\n");
        p->rtime=0;p->state='w';
        p->link=NULL;
        sort(); /*    调用  sort 函数 */
    }
}
int space()
{
    int l=0; PCB* pr=ready;
    while(pr!=NULL)
    {
        l++;
        pr=pr->link;
    }
    return(l);
}
disp(PCB * pr) /*   建立进程显示函数    ,用于显示当前进程    */
{
    printf("\n qname \t state \t super \t ndtime \t runtime \n");
    printf("|%s\t",pr->name);
    printf("|%c\t",pr->state);
    printf("|%d\t",pr->super);
```

```
    printf("|%d\t",pr->ntime);
    printf("|%d\t",pr->rtime);
    printf("\n");
}

check() /*  建立进程查看函数    */
{
    PCB* pr;
    printf("\n ****     当前正在运行的进程是    :%s",p->name); /* 显示当前运行进程   */
    disp(p);
    pr=ready;
    printf("\n ****    当前就绪队列状态为    :\n"); /* 显示就绪队列状态   */
    while(pr!=NULL)
    {
        disp(pr);
        pr=pr->link;
    }
}
destroy() /* 建立进程撤消函数   (进程运行结束   ,撤消进程  )*/
{
    printf("\n   进程   [%s]   已完成 .\n",p->name);
    free(p);
}
running() /*   建立进程就绪函数   (进程运行时间到   ,置就绪状态   */
{
    (p->rtime)++;
    if(p->rtime==p->ntime)
        destroy(); /*   调用  destroy 函数 */
    else
    {
        (p->super)--;
        p->state='w';
        sort(); /* 调用  sort 函数 */
    }
}
main() /* 主函数 */
{
    int len,h=0;
    char ch;
    input();
    len=space();
    while((len!=0)&&(ready!=NULL))
    {
        ch=getchar();
```

```
h++;
printf("\n The execute number:%d \n",h);
p=ready;
ready=p->link;
p->link=NULL;
p->state='R';
check();
running();
printf("\n    按任一键继续    ......");
      ch=getchar();
}
printf("\n\n    进程已经完成    .\n");
ch=getchar(); }
```

实验二、银行家算法

（一）　目的和要求

银行家算法是由　Dijkstra　设计的最具有代表性的避免死锁的算法。本实验要求用高级语言编写一个银行家的模拟算法。通过本实验可以对预防死锁和银行家算法有更深刻的认识。

（二）　实验内容

1、 设置数据结构

包括可利用资源向量　（Availiable）,最大需求矩阵　（Max）,分配矩阵（Allocation　）,需求矩阵（Need）

2、 设计安全性算法

设置工作向量　Work　表示系统可提供进程继续运行可利用资源数目，　Finish　表示系统是否有足够的资源分配给进程

（三）　实验环境

1、 pc

2、 vc++

（四）、程序源代码：

```
/* 子函数声明  */
int Isprocessallover();                    //判断系统中的进程是否全部运行完毕
void Systemstatus();                       //显示当前系统中的资源及进程情况
int Banker(int ,int *);                    //银行家算法
void Allow(int ,int *);                    //若进程申请不导致死锁，用此函数分配资源
void Forbidenseason(int );                 //若发生死锁，则显示原因

/* 全局变量  */
int Availiable[3]={3,3,2};                 //初始状态，系统可用资源量
int Max[5][3]={{7,5,3},{3,2,2},{9,0,2},{2,2,2},{4,3,3}};
                                           //各进程对各资源的最大需求量
int Allocation[5][3]={{0,1,0},{2,0,0},{3,0,2},{2,1,1},{0,0,2}};
                                           //初始状态，各进程占有资源量
int Need[5][3]={{7,4,3},{1,2,2},{6,0,0},{0,1,1},{4,3,1}};
```

```
                                        //初始状态时，各进程运行完毕，还需要的资源量
int over[5]={0,0,0,0,0};                //标记对应进程是否得到所有资源并运行完毕

#include <iostream.h>

/* 主函数 */
void main()
{
    int process=0;                              //发出请求的进程
    int decide=0;                               //银行家算法的返回值
    int Request[3]={0,0,0};                     //申请的资源量数组
    int sourcenum=0;                            // 申请的各资源量
    /* 判断系统中进程是否全部运行完毕      */
step1:    if(Isprocessallover()==1)
    {
        cout<<" 系统中全部进程运行完毕！    ";
        return;
    }
    /* 显示系统当前状态   */
    Systemstatus();
    /* 人机交互界面   */
step2: cout<<"\n  输入发出请求的进程（输入    “ 0 ”退出系统）  : ";
    cin>>process;
if(process==0)
{
    cout<<" 放弃申请 ,退出系统！  ";
    return;
}
if(process<1||process>5||over[process-1]==1)
{
    cout<<" 系统无此进程！   \n";
    goto step2;
}
cout<<" 此进程申请各资源（   A，  B，  C）数目：  \n";
for(int h=0;h<3;h++)
{
    cout<<char(65+h)<<"  资源 :";
    cin>>sourcenum;
    Request[h]=sourcenum;
}
/* 用银行家算法判断是否能够进行分配      */
  decide=Banker(process,Request);
  if (decide==0)
  {
```

```
        /* 将此进程申请资源分配给它      */
        Allow(process,Request);
        goto step1;
}
else
{
        /* 不能分配，显示原因    */
        Forbidenseason(decide);
        goto step2;
            }
}

/* 子函数   Isprocessallover( ) 的实现 */
int Isprocessallover()
{
        int processnum=0;
        for(int i=0;i<5;i++)
        {
                /* 判断每个进程是否运行完毕      */
                if(over[i]==1)
                        processnum++;
        }
        if(processnum==5)
                /* 系统中全部进程运行完毕     */
                return 1;
        else
                return 0;
}

/* 子函数   Systemstatus( )的实现 */
void Systemstatus()
{
        cout<<" 此刻系统中存在的进程：      \n";
        for(int i=0;i<5;i++)
        {
                if(over[i]!=1)
                        cout<<"P"<<i+1<<"       ";
}
cout<<endl;
cout<<" 此刻系统可利用资源（单位：个）       ：\n";
cout<<"A      B     C\n";
for(int a=0;a<3;a++)
{
        cout<<Availiable[a]<<"       ";
```

```
}
cout<<endl;
cout<<" 此刻各进程已占有资源如下（单位：个）          : \n"
        <<"         A     B     C\n";
for(int b=0;b<5;b++)
{
        if(over[b]==1)
                continue;
        cout<<"P"<<b+1<<"       ";
        for(int c=0;c<3;c++)
                cout<<Allocation[b][c]<<"         ";
        cout<<endl;
}
cout<<" 各进程运行完毕还需各资源如下（单位：个）          :\n"
        <<"         A     B     C\n";
for(int f=0;f<5;f++)
{
        if(over[f]==1)
                continue;
        cout<<"P"<<f+1<<"       ";
        for(int g=0;g<3;g++)
                cout<<Need[f][g]<<"       ";
                cout<<endl;
        }
}

/* 子函数   Banker(int ,int &)  的实现  */
int Banker(int p,int *R)
{
        int num=0;                                      //标记各资源是否能满足各进程需要
        int Finish[5]={0,0,0,0,0};                      //标记各进程是否安全运行完毕
        int work[5]={0,0,0,0,0};                        //用于安全检查
        int AvailiableTest[3];                          //用于试分配
        int AllocationTest[5][3];                       //同上
        int NeedTest[5][3];                             //同上
        /* 判断申请的资源是否大于系统可提供的资源总量          */
        for(int j=0;j<3;j++)
        {
                if(*(R+j)>Availiable[j])
                        /* 返回拒绝分配原因    */
                return 1;
        }
        /* 判断该进程申请资源量是否大于初始时其申明的需求量          */
        for(int i=0;i<3;i++)
```

```
{
        if(*(R+i)>Need[p-1][i])
        /* 返回拒绝原因   */
            return 2;
}
   /* 为检查分配的各数据结构赋初值       */
for(int t=0;t<3;t++)
{
   AvailiableTest[t]=Availiable[t];
}
for(int u=0;u<5;u++)
{
   for(int v=0;v<3;v++)
   {
        AllocationTest[u][v]=Allocation[u][v];
   }
}
for(int w=0;w<5;w++)
{
   for(int x=0;x<3;x++)
   {
        NeedTest[w][x]=Need[w][x];
   }
}
  /* 进行试分配   */
for(int k=0;k<3;k++)
//修改  NeedTest[]
{
   AvailiableTest[k]-=*(R+k);
   AllocationTest[p-1][k]+=*(R+k);
   NeedTest[p-1][k]-=*(R+k);
}
  /* 检测进程申请得到满足后，系统是否处于安全状态          */
for(int l=0;l<3;l++)
{
   work[l]=AvailiableTest[l];
}
for(int m=1;m<=5;m++)
{
   for(int n=0;n<5;n++)
   {
        num=0;
        /* 寻找用此刻系统中没有运行完的进程       */
        if(Finish[n]==0&&over[n]!=1)
```

```
{
        for(int p=0;p<3;p++)
        {
                if(NeedTest[n][p]<=work[p])
                        num++;
        }
        if(num==3)
        {
              for(int q=0;q<3;q++)
                        {
                                work[q]=work[q]+AllocationTest[n][q];
                        }
                        Finish[n]=1;
                    }
                }
    }
}
for(int r=0;r<5;r++)
{
    if(Finish[r]==0&&over[r]!=1)
            /* 返回拒绝分配原因    */
            return 3;
        }
        return 0;
}

/* 子函数   Allow(int ,int &)    的实现 */
void Allow(int p,int *R)
{
        cout<<" 可以满足申请！    ";
        static int overnum;
        /* 对进程所需的资源进行分配     */
        for(int t=0;t<3;t++)
        {
                Availiable[t]=Availiable[t]-*(R+t);
                Allocation[p-1][t]=Allocation[p-1][t]+*(R+t);
        Need[p-1][t]=Need[p-1][t]-*(R+t);
}
/* 分配后判断其是否运行完毕      */
overnum=0;
for(int v=0;v<3;v++)
{
        if(Need[p-1][v]==0)
                overnum++;
```

```
}
if(overnum==3)
{
        /* 此进程运行完毕，释放其占有的全部资源        */
        for(int q=0;q<3;q++)
                Availiable[q]=Availiable[q]+Allocation[p-1][q];
        /* 标记该进程运行完毕    */
        over[p-1]=1;
        cout<<" 进程 P"<<p<<" 所需资源全部满足，此进程运行完毕！        \n";
        }
}

/* 子函数  Forbidenseason(int )的实现 */
void Forbidenseason(int d)
{
        cout<<" 不能满足申请，此进程挂起，原因为：        \n";
        switch (d)
        {
        case 1:cout<<" 申请的资源量大于系统可提供的资源量！        ";break;
        case 2:cout<<" 申请的资源中有某种资源大于其声明的需求量！        ";break;
        case 3:cout<<" 若满足申请，系统将进入不安全状态，可能导致死锁！        ";
        }
}
```

实验三、页式地址重定位模拟

一、实验目的：

1、 用高级语言编写和调试模拟实现页式地址重定位。

2、 加深理解页式地址重定位技术在多道程序设计中的作用和意义。

二、实验原理：

当进程在 CPU 上运行时，如指令中涉及逻辑地址时，操作系统自动根据页长得到页号和页内偏移，把页内偏移拷贝到物理地址寄存器，再根据页号，查页表，得到该页在内存中的块号，把块号左移页长的位数，写到物理地址寄存器。

三、实验内容：

1、 设计页表结构

2、 设计地址重定位算法

3、 有良好的人机对话界面

四、程序源代码：

```
#define pagesize    1024
#define pagetablelength    64

/* 系统页表  */
const int pagetable[pagetablelength]={0,42,29,15,45,31,44,43,
                                      41,28,1,30,12,24,6,32,
                                      14,27,13,46,7,33,10,22,
                                      40,2,51,11,39,23,49,50,
                                      26,16,25,4,47,17,3,48,
                                      52,36,58,35,57,34,21,63,
                                      5,37,18,8,62,56,20,54,
                                      60,19,38,9,61,55,59,53};
#include<iostream.h>
#include<iomanip.h>

void main()
```

```
{
        int logicaladdress=0;
        int pagenum=0;
        int w=0;
        cout<<" 系统页号对应块号情况（页号——       >块号）：\n";
        for(int i=0;i<64;i++)
        {
                cout<<setw(2)<<i<<"-->"<<setw(2)<<pagetable[i]<<"          ";
                if(i%8==7)
                        cout<<endl;
        }
        cout<<endl<<" 请输入逻辑地址（十进制）     ：\n";
        cin>>logicaladdress;
        pagenum=logicaladdress/pagesize;        //求页号
        w=logicaladdress%pagesize;                //求页内偏移地址
        if(pagenum>pagetablelength)                //判断是否跃界
        {
                cout<<" 本次访问的地址已超出进程的地址空间，系统将产生越界中断！          \n";
                return;
        }
        cout<<" 对应的物理地址为（十进制）     ：\n"<<pagetable[pagenum]*pagesize+w<<endl;
}
```

二、程序调试：

调试数据一：

系统页号对应块号情况（页号——   >块号）：

```
 0--> 0      1-->42      2-->29      3-->15      4-->45      5-->31      6-->44      7-->43
 8-->41      9-->28     10--> 1     11-->30     12-->12     13-->24     14--> 6     15-->32
16-->14     17-->27     18-->13     19-->46     20--> 7     21-->33     22-->10     23-->22
24-->40     25--> 2     26-->51     27-->11     28-->39     29-->23     30-->49     31-->50
32-->26     33-->16     34-->25     35--> 4     36-->47     37-->17     38--> 3     39-->48
40-->52     41-->36     42-->58     43-->35     44-->57     45-->34     46-->21     47-->63
48--> 5     49-->37     50-->18     51--> 8     52-->62     53-->56     54-->20     55-->54
56-->60     57-->19     58-->38     59--> 9     60-->61     61-->55     62-->59     63-->53

请输入逻辑地址（十进制）     ：
2500
对应的物理地址为（十进制）     ：
30148
Press any key to continue
```

调试数据二：

系统页号对应块号情况（页号——　　>块号）：

0--> 0　1-->42　2-->29　3-->15　4-->45　5-->31　6-->44　7-->43
8-->41　9-->28　10--> 1　11-->30　12-->12　13-->24　14--> 6　15-->32
16-->14　17-->27　18-->13　19-->46　20--> 7　21-->33　22-->10　23-->22
24-->40　25--> 2　26-->51　27-->11　28-->39　29-->23　30-->49　31-->50
32-->26　33-->16　34-->25　35--> 4　36-->47　37-->17　38--> 3　39-->48
40-->52　41-->36　42-->58　43-->35　44-->57　45-->34　46-->21　47-->63
48--> 5　49-->37　50-->18　51--> 8　52-->62　53-->56　54-->20　55-->54
56-->60　57-->19　58-->38　59--> 9　60-->61　61-->55　62-->59　63-->53

请输入逻辑地址（十进制）　：

765497

本次访问的地址已超出进程的地址空间，系统将产生越界中断！

Press any key to continue

实验四、 LRU 算法模拟

一、实验目的和要求

用高级语言模拟页面置换算法 LRU，加深对 LRU 算法的认识。

二、实验原理

其基本原理为： 如果某一个页面被访问了，它很可能还要被访问； 相反，如果它长时间不被访问，再最近未来是不大可能被访问的。

三、实验环境

1、 pc

2、 vc++

四、程序源代码：

```
#define MAXSIZE          20
#include <iostream.h>

void main()
{
        int input=0;                                          //用于输入作业号
        int worknum=0;                                        //输入的作业个数
        int storesize=0;                                      //系统分配的存储区块数
        int interrupt=0;                                      //缺页中断次数
        int stack[MAXSIZE];                                   //栈， LRU 算法的主要数据结构
        int workstep[MAXSIZE];                                //记录作业走向
        /* 初始化 */
        for(int i=0;i<MAXSIZE;i++)
        {
                stack[i]=0;
                workstep[i]=0;
        }
        cout<<" 请输入存储区块数：     ";
        cin>>storesize;
```

```
cout<<" 请输入作业的页面走向（输入       0 结束）：\n";
for(int j=0;j<MAXSIZE;j++)
{
        cout<<" 页面号    "<<j+1;
        cin>>input;
        workstep[j]=input;
        if(input==0)
        {
                cout<<" 输入结束！   \n";
                break;
        }
        worknum++;
}
if(workstep[0]==0)
{
        cout<<" 未输入任何作业，系统将退出！       \n";
        return;
}
cout<<" 置换情况如下：    \n";
for(int k=0;k<worknum;k++)
{
        /* 在栈中找相等的页号或空位置      */
                for(int l=0;l<storesize;l++)
                {
                        /* 是否有相等的页号    */
                        if(stack[l]==workstep[k])
                        {
                                cout<<" 内存中有  "<<workstep[k]<<"   号页面，无须中断！    \n";
                                goto step1;
                        }
                        /* 找栈中是否有空位置    */
                        if(stack[l]==0)
                        {
                                stack[l]=workstep[k];
                                cout<<" 发生中断，但内存中有空闲区，        "<<workstep[k]<<"   号页面直接调
入！ \n";
                                interrupt++;
                                goto step1;
                        }
                }
        /* 上述情况都不成立则调出栈顶，将调入页面插入栈顶          */
        cout<<" 发生中断，将   "<<stack[0]<<"  号页面调出，    "<<workstep[k]<<"   号装入！  \n";
        interrupt++;
        /* 新掉入的页面放栈顶    */
```

```
step1:      for(int m=0;m<storesize;m++)
            {
                stack[m]=stack[m+1];
            }
            stack[storesize-1]=workstep[k];

        }
        cout<<" 作 业 "<<worknum<<"  个 ,  "<<" 中 断 "<<interrupt<<"  次 ,  "<<" 缺 页 率 ： "<<float(interrupt)/float(worknum)*100<<"%\n";
}
```

二、程序调试：

调试一：

```
请输入存储区块数：    3
请输入作业走向（输入     0 结束）：
页面号  1：4
页面号  2：3
页面号  3：2
页面号  4：1
页面号  5：4
页面号  6：3
页面号  7：5
页面号  8：4
页面号  9：3
页面号  10：2
页面号  11：1
页面号  12：5
页面号  13：0
输入结束！
置换情况如下：
发生中断，但内存中有空闲区，       4 号页面直接调入！
发生中断，但内存中有空闲区，       3 号页面直接调入！
发生中断，但内存中有空闲区，       2 号页面直接调入！
发生中断，将    4 号页面调出，    1 号装入！
发生中断，将    3 号页面调出，    4 号装入！
发生中断，将    2 号页面调出，    3 号装入！
发生中断，将    1 号页面调出，    5 号装入！
内存中有   4 号页面，无须中断！
内存中有   3 号页面，无须中断！
```

发生中断，将　5 号页面调出，　2 号装入！
发生中断，将　4 号页面调出，　1 号装入！
发生中断，将　3 号页面调出，　5 号装入！
作业 12 个，中断　10 次，缺页率：　83.3333%
Press any key to continue

调试二：

请输入存储区块数：　4
请输入作业走向（输入　0 结束）：
页面号 1：4
页面号 2：3
页面号 3：2
页面号 4：1
页面号 5：4
页面号 6：3
页面号 7：5
页面号 8：4
页面号 9：3
页面号 10：2
页面号 11：1
页面号 12：5
页面号 13：0
输入结束！
置换情况如下：
发生中断，但内存中有空闲区，　4 号页面直接调入！
发生中断，但内存中有空闲区，　3 号页面直接调入！
发生中断，但内存中有空闲区，　2 号页面直接调入！
发生中断，但内存中有空闲区，　1 号页面直接调入！
内存中有　4 号页面，无须中断！
内存中有　3 号页面，无须中断！
发生中断，将　2 号页面调出，　5 号装入！
内存中有　4 号页面，无须中断！
内存中有　3 号页面，无须中断！
发生中断，将　3 号页面调出，　2 号装入！
发生中断，将　5 号页面调出，　1 号装入！
发生中断，将　4 号页面调出，　5 号装入！
作业 12 个，中断　8 次，缺页率：　66.6667%
Press any key to continue

实验五、 FIFO 算法模拟

一、实验目的

一个作业有多少个进程， 处理机只分配固定的主存页面供该作业执行。 往往页面数小于进程数， 当请求调页程序调进一个页面时， 可能碰到主存中并没有空闲块的情况， 此时就产生了在主存中淘汰哪个页面的情况。本实验要求模拟 FIFO 算法 /

二、实验原理

此算法的实质是， 总是选择在主存中居留最长时间的页面淘汰。 理由是： 最早调入主存的页，其不再被访问的可能性最大。

三、实验环境

1、 pc

2、 vc++

四、程序源代码：

```
#define MAXSIZE          20
#include <iostream.h>

void main()
{
    int label=0;                                    //标记此页是否已经装入内存
    int input=0;                                    //用于输入作业号
    int worknum=0;                                  //记录作业个数
    int storesize=0;                                //系统分配的存储块数
    int interrupt=0;                                //中断次数
    int quence[MAXSIZE];                            //队列， FIFO 算法的主要数据结构
    int workstep[MAXSIZE];                          //用于记录作业走向
    /* 初始化 */
    for(int i=0;i<MAXSIZE;i++)
    {
        quence[i]=0;
```

```
        workstep[i]=0;
}
cout<<" 请输入存储区块数：    ";
cin>>storesize;
cout<<" 请输入作业走向（输入     0 结束）：\n";
for(int j=0;j<MAXSIZE;j++)
{
        cout<<" 页面号：  "<<j+1;
        cin>>input;
        workstep[j]=input;
        if(input==0)
        {
                cout<<" 输入结束！   \n";
                break;
        }

        worknum++;
}
if(workstep[0]==0)
{
        cout<<" 未输入任何作业，系统将退出！       \n";
        return;
        }
        cout<<" 置换情况如下：    \n";
        for(int k=0;k<worknum;k++)
        {
                label=0;
                /* 看队列中是否有相等的页号或空位置       */
                for(int l=0;l<storesize;l++)
                {
                        /* 是否有相等的页号    */
                        if(quence[l]==workstep[k])
                        {
                                cout<<" 内存中有  "<<workstep[k]<<"   号页面，无须中断！     \n";
                                label=1;                                                        //标记此
页面已装入内存
                                break;
                        }
                        /* 是否有空位置   */
                        if(quence[l]==0)
                        {
                                quence[l]=workstep[k];
                                cout<<" 发生中断  ,但内存中有空闲区，     "<<workstep[k]<<"   号页面直接调入！
\n";
```

```
                interrupt++;
                label=1;
                break;
        }
}
/* 上述情况都不成立则调出对首，将调入页面插入对尾          */
if(label==0)
{
        cout<<" 发生中断， 将"<<quence[0]<<" 号页面调出， "<<workstep[k]<<" 号装入！
\n";
                interrupt++;
                for(int m=0;m<storesize;m++)
                {
                    quence[m]=quence[m+1];
                }
                quence[storesize-1]=workstep[k];
            }
        }
        cout<<" 作 业 "<<worknum<<" 个， "<<" 中 断 "<<interrupt<<" 次， "<<" 缺 页 率 ：
"<<float(interrupt)/float(worknum)*100<<"%\n";
}
```

二、程序调试：

调试一：

```
请输入存储区块数：    3
请输入作业走向（输入     0 结束）：
页面号  1：4
页面号  2：3
页面号  3：2
页面号  4：1
页面号  5：4
页面号  6：3
页面号  7：5
页面号  8：4
页面号  9：3
页面号  10： 2
页面号  11： 1
页面号  12： 5
页面号  13： 0
```

输入结束！
置换情况如下：
发生中断 ,但内存中有空闲区, 4 号页面直接调入！
发生中断 ,但内存中有空闲区, 3 号页面直接调入！
发生中断 ,但内存中有空闲区, 2 号页面直接调入！
发生中断, 将 4 号页面调出, 1 号装入！
发生中断, 将 3 号页面调出, 4 号装入！
发生中断, 将 2 号页面调出, 3 号装入！
发生中断, 将 1 号页面调出, 5 号装入！
内存中有 4 号页面, 无须中断！
内存中有 3 号页面, 无须中断！
发生中断, 将 4 号页面调出, 2 号装入！
发生中断, 将 3 号页面调出, 1 号装入！
内存中有 5 号页面, 无须中断！
作业 12 个, 中断 9 次, 缺页率： 75%
Press any key to continue

调试二：

请输入存储区块数： 4
请输入作业走向（输入 0 结束）：
页面号 1：4
页面号 2：3
页面号 3：2
页面号 4：1
页面号 5：4
页面号 6：3
页面号 7：5
页面号 8：4
页面号 9：3
页面号 10：2
页面号 11：1
页面号 12：5
页面号 13：0
输入结束！
置换情况如下：
发生中断 ,但内存中有空闲区, 4 号页面直接调入！
发生中断 ,但内存中有空闲区, 3 号页面直接调入！
发生中断 ,但内存中有空闲区, 2 号页面直接调入！
发生中断 ,但内存中有空闲区, 1 号页面直接调入！
内存中有 4 号页面, 无须中断！
内存中有 3 号页面, 无须中断！
发生中断, 将 4 号页面调出, 5 号装入！
发生中断, 将 3 号页面调出, 4 号装入！

发生中断，将　2 号页面调出，　3 号装入！
发生中断，将　1 号页面调出，　2 号装入！
发生中断，将　5 号页面调出，　1 号装入！
发生中断，将　4 号页面调出，　5 号装入！
作业 12 个，中断　10 次，缺页率：　83.3333%
Press any key to continue