《操作系统课程设计》

**实习报告**

**学 院： 信息科学与技术学院**

**专 业： 计算机科学与技术**

**班 级： 信1901-3**

**姓 名： 崔金泽**

**指导教师： 陈 娜**

**实习时间： 6月25日——7月2日**

填报日期 2022 年 7 月 2 日

**目 录**

[一、实训目的 3](#_Toc107644261)

[二、实训内容 3](#_Toc107644262)

[三、实训设备 3](#_Toc107644263)

[四、实训任务及要求 3](#_Toc107644264)

[五、实训基本操作方法 3](#_Toc107644265)

[六、实训项目 4](#_Toc107644266)

[任务一 分析操作系统所面临的操作需求 4](#_Toc107644267)

[任务二 进程管理 6](#_Toc107644268)

[多线程生产者消费者 7](#_Toc107644269)

[BufferPool.cpp 7](#_Toc107644270)

[Consume.cpp 9](#_Toc107644271)

[Main\_prg.cpp 10](#_Toc107644272)

[Product.cpp 12](#_Toc107644273)

[调度算法 16](#_Toc107644274)

[FCFS.cpp 16](#_Toc107644275)

[Job.cpp 17](#_Toc107644276)

[RR.cpp 17](#_Toc107644277)

[SJF.cpp 19](#_Toc107644278)

[HPF.cpp 20](#_Toc107644279)

[Main.cpp 21](#_Toc107644280)

[任务三 存储管理 25](#_Toc107644281)

[页面置换算法 28](#_Toc107644282)

[PageReplace.cpp 28](#_Toc107644283)

[任务四 设备管理 36](#_Toc107644284)

[银行家算法 37](#_Toc107644285)

[DeadLock.cpp 37](#_Toc107644286)

[任务五 文件管理 43](#_Toc107644287)

[磁盘调度算法 44](#_Toc107644288)

[main.cpp 44](#_Toc107644289)

[DiskTrack.h 45](#_Toc107644290)

[Array.h 48](#_Toc107644291)

[DiskSchedulingAlgorithms.h 55](#_Toc107644292)

[任务六 模块集成 62](#_Toc107644293)

[模拟操作系统 63](#_Toc107644294)

一、实训目的

通过操作系统实训，主要强化学生对本课程基础知识的掌握；使学生理论联 系实际，加强学生的动手能力；加深对操作系统的基本概念、工作原理和实现方 法等理论知识的理解；掌握操作系统各个部分之间的有机联系，从而了解操作系 统在整个计算机系统中的地位和作用，巩固和加强与本课程相关的其他计算机课 程的知识，提高对计算机专业知识理解的系统性和完整性，并加强合作完成系统 的团队精神和提高程序设计的能力。

二、实训内容

本实训的内容为实现一个模拟操作系统。要求使用实验室所提供的安装有 C 语言编程环境的计算机，模拟采用多道程序设计方法的单用户操作系统，该操作 系统包括进程管理、存储管理、设备管理和文件管理四部分。

三、实训设备

1、PC 计算机

2、VC++等软件系统

四、实训任务及要求

根据实训内容，认真完成模拟操作系统的实现，模拟操作系统需包括进程管 理、存储管理、设备管理和文件管理四部分。实训的基本原理主要包括操作系统 中的进程的同步与互斥；常用的进程调度算法；地址重定位；动态页式存储管理 技术的页面淘汰算法；设备管理中设备的分配和回收；用死锁避免方法来处理申 请独占设备可能造成的死锁；磁盘调度算法等。 本实训结束后，需要学生提交实训的源代码及可执行程序，并提交实训报告。

五、实训基本操作方法

1. 搜集与整理，设定操作系统所面临的操作需求；

2. 设计各部分的实现方案；

3. 程序开发；

4. 程序测试；

5. 系统集成；

6. 提交源程序，完成实训报告。

六、实训项目

任务一 分析操作系统所面临的操作需求

【实训目的】

使学生理解操作系统所面临的操作需求，掌握操作系统中的进程管理、存储管理、设备管理和文件管理等功能。

【实训内容】

分析操作系统所面临的操作需求；

熟悉实训环境；

资料搜集与整理，进行实训的前期准备。

【预习要求】

操作系统的功能及实现的基本原理。

【实训步骤】

分析操作系统所面临的操作需求：进程管理、存储管理、设备管理和文件管理，进一步熟悉各模块的工作原理；

根据操作需求，进行系统的整体设计，画出系统总体的功能模块图， 如下图所示；

 根据上一步得出的功能模块图，进行资料的搜集与整理，并熟悉实训环境，为之后实训任务的完成打下坚实的基础。

【注意事项】

操作系统中各模块之间的功能划分。

【思考题】

1.操作系统中各模块有怎样的功能？

操作系统的五大功能模块是：处理器管理、存储器管理、设备管理、文件管理和作业管理。操作系统是管理计算机硬件和软件资源的计算机程序，同时它也提供了一个让用户与系统交互的操作界面。

1、进程管理，其工作主要是进程调度，在单用户单任务的情况下，处理器仅为一个用户的一个任务所独占， 进程管理的工作十分简单。但在多道程序或多用户的情况 下，组织多个作业或任务时，就要解决处理器的调度、 分配和回收等问题 。

2、存储管理分为几种功能：存储分配、存储共享、存储保护 、存储扩张。

3、设备管理分为以下功能：设备分配、设备传输控制 、设备独立性。

4、文件管理：文件存储空间的管理、目录管理 、文件操作管理、文件保护。

5、作业管理是负责处理用户提交的任何要求。

2.它们之间有怎样的联系？

设备管理、文件管理和储存管理都需要进程的管理;文件需要文件管理进行存储，同时也需要储存管理来对文件存储分配空间等等。各模块之间能通过该接口实现交互。然后，再进一步将各模块细分为若干个具有一定功能的子模块之间的接口。若子模块较大时，可再进一步将它细分。在模块设计中如果模块划分得太小，虽然可以降低模块本身的复杂性，但会引起模块之间的联系过多，而会造成系统比较混乱；如果将模块划分得过大，又会增加模块内部的复杂性，使内部的联系增加。因此，在划分模块时，应在两者之间进行权衡。

模块之间是互相独立又互相有联系的，模块的独立性越高，各模块间的交互就越少，系统的结构也就越清晰。内聚性指模块内部各部分间联系的紧密程度。内聚性越高，模块的独立性越强。耦合度指模块间相互联系和相互影响的程度。显然，耦合度越低，模块的独立性越好。

3.针对某一特定的应用环境，如何完善操作系统的功能？

可以把操作系统做成很多种不同的类型，每种类型都有各自的取舍。安装在各种设备上的操作系统可以从简单到复杂，分为智能卡操作系统、实时操作系统、传感器节点操作系统、嵌入式操作系统、个人计算机操作系统、多处理器操作系统、网络操作系统和大型机操作系统。按应用领域分，主要有三种：桌面操作系统、服务器操作系统和嵌入式操作系统。可以通过各种算法合理的安排各个模块工作。

任务二 进程管理

【实训目的】

掌握临界区的概念及临界区的设计原则；掌握信号量的概念、PV 操作的含义以及应用 PV 操作实现进程的同步与互斥；分析进程争用资源的现象，学习解决进程互斥的方法；掌握进程的状态及状态转换；掌握常用的进程调度算法。

【实训内容】

1. 分析进程的同步与互斥现象，编程实现经典的进程同步问题——生产者消费者问题的模拟；
2. 编写允许进程并行执行的进程调度程序，在常用的进程（作业）调度算法： 先来先服务算法、短作业优先算法、最高响应比优先算法、高优先权优先算法等调度算法中至少选择三种调度算法进行模拟，并输出平均周转时间和平均带权周转时间。

【预习要求】

进程同步与互斥的概念及实现方法；进程调度的作用及常用的调度算法。

【实训步骤】

1.分析计算机系统中对资源的分配与释放过程：计算机系统中的每个进程都可以消费或生产某类资源。当系统中某一进程使用某一资源时，可以看作是消耗， 且该进程称为消费者。而当某个进程释放资源时，则它就相当一个生产者。

2.定义生产者消费者问题中的各数据结构，并初始化信号量；

1. 创建生产者与消费者进程，利用信号量实现生产者与消费者之间的同步与互斥

代码：

多线程生产者消费者

BufferPool.cpp

#ifndef BUFFERPOOL

#define BUFFERPOOL

#include<windows.h>

#include<vector>

#include<iostream>

#include<string>

#define SIZE\_OF\_BUFFER 4

using namespace std;

//控制参数类

class BufferPool

{

public:

    static BufferPool \* GetInstance();

    void Produce();

    void Consume();

    void Product\_Sum();

    //多线程互斥信号量

    HANDLE mutex;

    HANDLE mutex2;

    //多线程同步信号量

    //空缓冲区

    HANDLE m\_hSemaphoreBufferEmpty;

    //非空缓冲区

    HANDLE m\_hSemaphoreBufferFull;

    unsigned int produce\_sum;           //生产者在线数量

    unsigned int consume\_sum;           //消费者在线数量

private:

    unsigned short in;                   //线程的数量

    unsigned short out;                  //离开的数量

    unsigned int Product\_ID;

    unsigned int Consume\_ID;

    static BufferPool \*m\_Instance;

    int buffer[SIZE\_OF\_BUFFER];          //缓冲池

    BufferPool()

    {

        int i = 0;

        in = out = 0;

        produce\_sum = consume\_sum = 0;

        Product\_ID = Consume\_ID = 0;

        for(;i < SIZE\_OF\_BUFFER; i++)

            buffer[i] = 0;

    };

};

#endif

//

#include "BufferPool.h"

//--------------------------------------------------------------------------

//--------------------------------------------------------------------------

//unsigned short in, out;

//in = out = 0;

//获取单例实例

BufferPool \* BufferPool::m\_Instance = new BufferPool();

BufferPool \* BufferPool::GetInstance()

{

if(NULL == m\_Instance)

{

m\_Instance = new BufferPool();

}

return  m\_Instance;

}

void BufferPool::Produce()

{

 int i;

 std::cout<<"produce";

 //if(Product\_ID>=10)

 // Product\_ID=0;

 Product\_ID++;

 produce\_sum++;

 buffer[in]=Product\_ID;

 printf(" buffer[%d]=%d    ",in,Product\_ID);

 in=(in+1)%SIZE\_OF\_BUFFER;

    Product\_Sum();

}

void BufferPool::Consume()

{

 int i;

 std::cout<<"consume";

 consume\_sum++;

 Consume\_ID=buffer[out];

 printf(" buffer[%d]=%d    ",out,Consume\_ID);

 buffer[out]=0;

 out=(out+1)%SIZE\_OF\_BUFFER;

 Product\_Sum();

}

void BufferPool::Product\_Sum()

{

 int i,sum=0;

 for(i=0;i<SIZE\_OF\_BUFFER;i++)

 {

  if(buffer[i]!=0)

   sum++;

 }

// std::cout<<"  "<<sum<<"         ";

 //for(i=0;i<SIZE\_OF\_BUFFER;i++)

 //{

 // std::cout<<buffer[i]<<" ";

 //}

 printf("   %d\n", sum);

}

Consume.cpp

#include "Consume.h"

//--------------------------------------------------------------------------

DWORD WINAPI Consume::ConsumeProduct(LPVOID lpParam)

{

    while (true)

    {

        //等待非空的缓冲区出现，消费者进行消费，直到事件值为0

        WaitForSingleObject(BufferPool::GetInstance()->m\_hSemaphoreBufferFull, INFINITE);

        WaitForSingleObject(BufferPool::GetInstance()->mutex2,INFINITE);                //the mutex P operation

        BufferPool::GetInstance()->Consume();

        Sleep(1000);

        ReleaseMutex(BufferPool::GetInstance()->mutex2);                                //resource semaphore P operation

        ReleaseSemaphore(BufferPool::GetInstance()->m\_hSemaphoreBufferEmpty,1,NULL);            //the mutex P operation

    }

    return 0;

}

Main\_prg.cpp

#include<stdlib.h>

#include<windows.h>

#include "BufferPool.h"

#include "Product.h"

#include "Consume.h"

typedef DWORD (WINAPI \*pTheadFunc)(LPVOID lpParam);

const unsigned short p\_count=5;        //生产者在线数量

const unsigned short c\_count=5;        //消费者在线数量

const unsigned short s\_count=p\_count+c\_count;  //进程的数量

bool control = true;

HANDLE threads[s\_count];               //每一个线程handle

DWORD Producer\_threadID[p\_count];            //标记生产者线程

DWORD Consumer\_threadID[c\_count];            //标记消费者线程

void Create\_P\_Threads();

void Create\_C\_Threads();

void Create\_P\_Threads()                                  //创建消费者线程

{

    for(int i=0;i<p\_count;i++)

    {

         DWORD (WINAPI Product::\*pProductPoint)(LPVOID lpParam) = &Product::producer;

         int\* ptemp= (int\*)&pProductPoint;

         DWORD (WINAPI \*pProductThread)(LPVOID lpParam) = \*(pTheadFunc\*)ptemp;

         //生产者线程

         threads[i]=CreateThread(NULL,0,pProductThread,NULL,0,&Producer\_threadID[i]);

         if(threads[i]==NULL)

            control=false;

    }

}

void Create\_C\_Threads()

{

    for(int i=p\_count;i<s\_count;i++)

    {

        DWORD (WINAPI Consume::\*pConsumePoint)(LPVOID lpParam) = &Consume::ConsumeProduct;

        int\* ptemp= (int\*)&pConsumePoint;

        DWORD (WINAPI \*pConsumeThread)(LPVOID lpParam) = \*(pTheadFunc\*)ptemp;

        //消费者线程

        threads[i]=CreateThread(NULL,0,pConsumeThread,NULL,0,&Consumer\_threadID[i-p\_count]);

        if(threads[i]==NULL)

            control=false;

     }

}

void info()

{

    std::cout<<"\n"<<std::endl;

    std::cout<<"produce/consume    used\_total  buffer\_state(from 0 to 9)"<<std::endl;

}

int main()

{

 info();

 BufferPool::GetInstance()->mutex=CreateMutex(NULL,FALSE,NULL);

 BufferPool::GetInstance()->mutex2=CreateMutex(NULL,FALSE,NULL);

 BufferPool::GetInstance()->m\_hSemaphoreBufferFull=CreateSemaphore(NULL,0,SIZE\_OF\_BUFFER,NULL);

 BufferPool::GetInstance()->m\_hSemaphoreBufferEmpty=CreateSemaphore(NULL,SIZE\_OF\_BUFFER,SIZE\_OF\_BUFFER,NULL);

 Create\_C\_Threads();

 Create\_P\_Threads();

     while(control)

     {

          if(getchar())

          {

               std::cout<<std::endl;

               std::cout<<"the total produce product number is "<<BufferPool::GetInstance()->produce\_sum<<std::endl;

               std::cout<<"the total consume product number is "<<BufferPool::GetInstance()->consume\_sum<<std::endl;

               break;

          }

     }

     return 0;

}

Product.cpp

#include "Product.h"

//--------------------------------------------------------------------------

DWORD WINAPI  Product::producer(LPVOID lpParam)

{

    while (true)

    {

        //等待非空的缓冲区出现，消费者进行消费，直到信号量值为0

        WaitForSingleObject(BufferPool::GetInstance()->m\_hSemaphoreBufferEmpty, INFINITE);

        WaitForSingleObject(BufferPool::GetInstance()->mutex,INFINITE);                //the mutex P operation

        BufferPool::GetInstance()->Produce();

        Sleep(1000);

        ReleaseMutex(BufferPool::GetInstance()->mutex);                                //resource semaphore P operation

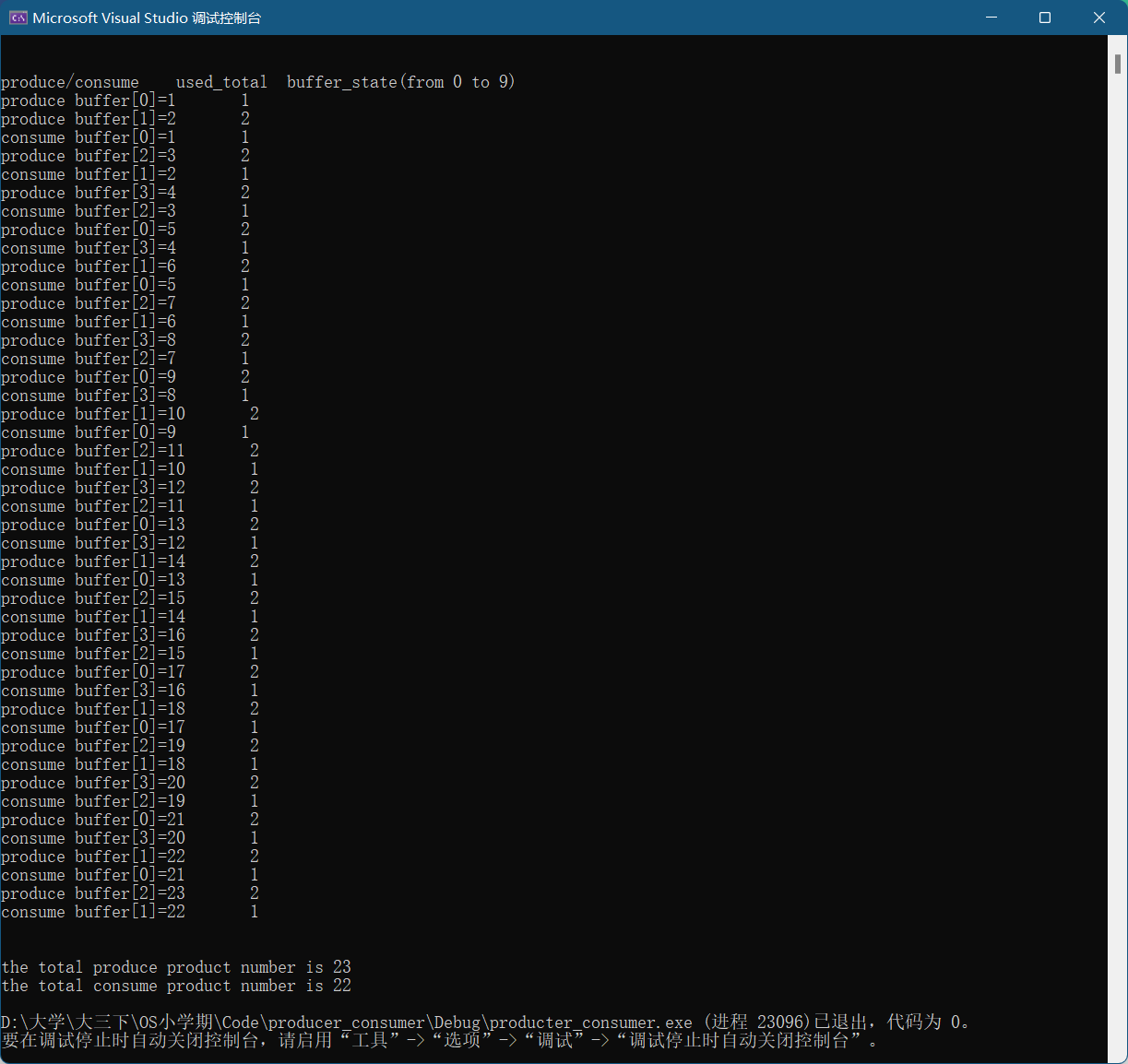
        ReleaseSemaphore(BufferPool::GetInstance()->m\_hSemaphoreBufferFull,1,NULL);            //the mutex P operation

    }

    return 0;

}

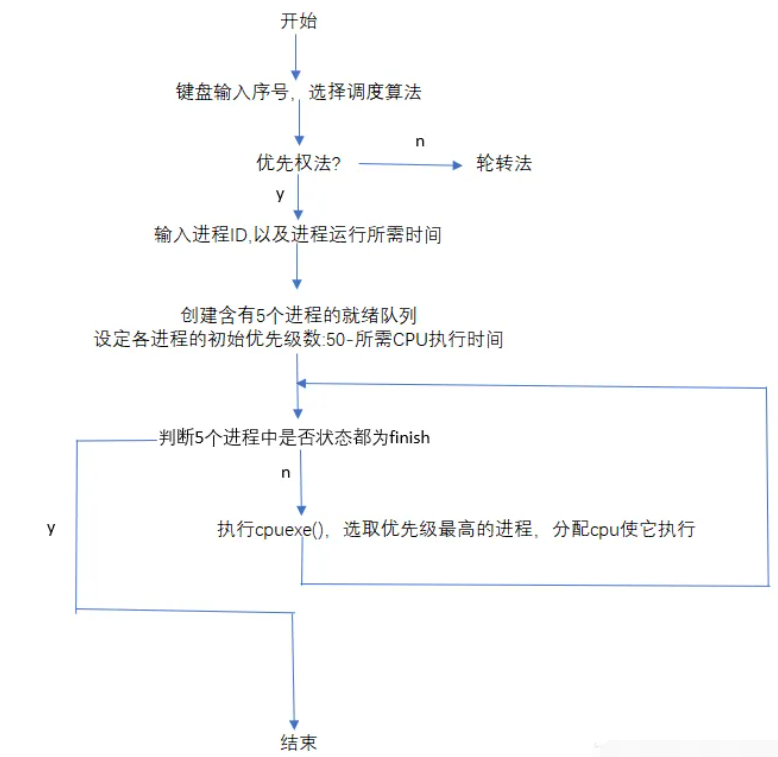
1. 运行并测试程序，运行界面如下图 2 所示：

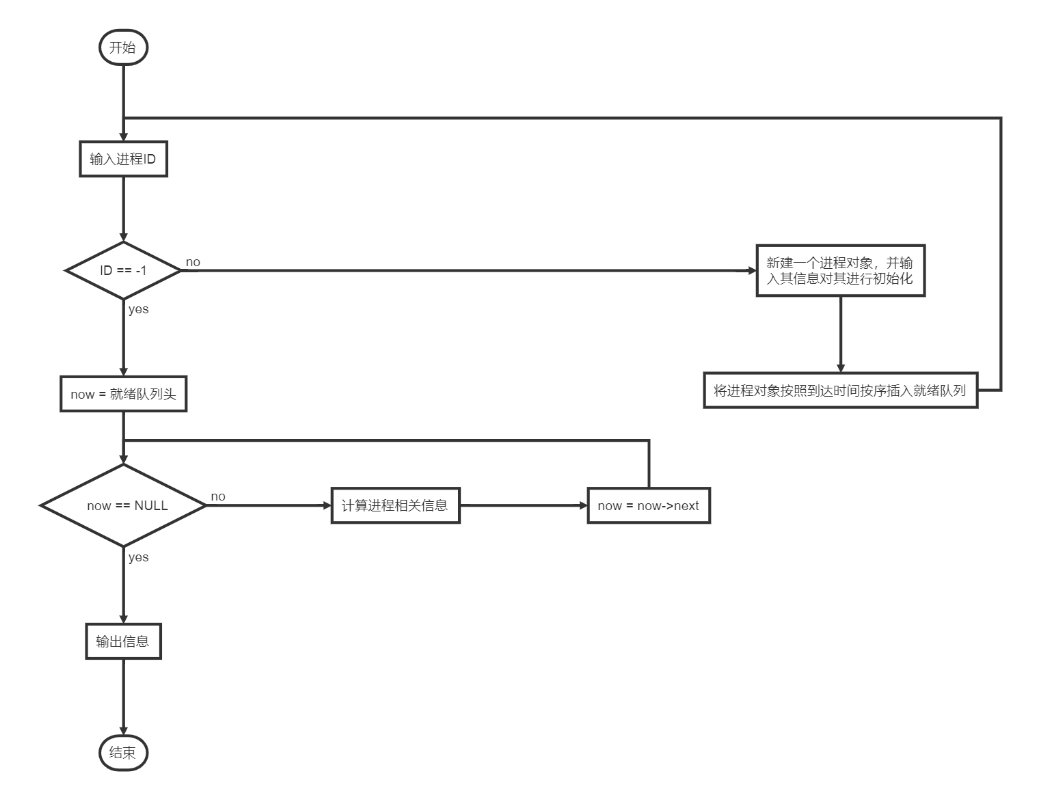


1. 分析常用的进程调度算法的工作原理，优先级法可被用作业或进程的调度策略。以下步骤以优先级法为例说明实训步骤。图 3 高优先权算法的工作流程：

首先，系统或用户按某种原则为作业或进程指定一个优先级来表示该作业或进程所享有的调度优先级权。该算法的核心是确定进程或作业的优先级。确定优先级的方法分为两类：静态法和动态法。静态法根据作业或进程的静态特性，在作业或进程开始执行之前就确定它们的优先级，一旦开始执行之后就不能改变。动态法则不然，它把作业或进程的静态特性和动态特性结合起来确定作业或进程的优先级，随着作业或进程的执行过程，其优先级不断变化。本实训指导书以动态法为例进行说明。静态优先级的调度算法实现虽然简单，系统开销小，但由于静态优先级一旦确定之后，直到执行结束为止始终保持不变，从而系统效率较低， 调度性能不高，而动态优先级的系统效率较高，调度性能也高。

1. 高优先权算法的设计，下图 4 所示：





先读入进程，比较进程的优先级，排列出分配 CPU 的队列，按时间片分配 C PU，一个时间片后，优先级减 1，再一次比较优先级，再排分配 CPU 的队列，按时间片分配 CPU，直到进程全部执行完毕。

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）。进程的到达时间为输入进程的时间。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（Finis h）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。

1. 根据以上算法编写程序，最后计算平均周转时间与平均带权周转时间。

调度算法

FCFS.cpp

#include "FCFS.h"

//执行

void FCFS::execute(std::ostream & output) {

    if(list.GetLength() == 0) {

        output << "任务表为空！\n";

        return;

    }

    Job job;

    //当前时间，总数

    unsigned int now = 0, total = 0;

    //总等待时间，总周转时间，总带权周转时间

    double totalWait = 0, totalTurnover = 0, totalRightTurnover = 0;

    while(list.GetLength() > 0) {

        list.Find(0, job);

        if(now < job.arrivalTime) {

            now = job.arrivalTime;

        }

        now += job.executionTime;

        output << "执行完的作业：" << job

            << "\n等待时间：" << (now - job.executionTime - job.arrivalTime)

            << "\t周转时间：" << (now - job.arrivalTime)

            << "\t带权周转时间：" << (double(now - job.arrivalTime) / job.executionTime)

            << "\n";

        total++;

        totalWait += now - job.executionTime - job.arrivalTime;

        totalTurnover += now - job.arrivalTime;

        totalRightTurnover += double(now - job.arrivalTime) / job.executionTime;

        list.Delete(0);

    }

    output << "\n平均等待时间：" << (totalWait / total)

        << "\t平均周转时间：" << (totalTurnover / total)

        << "\t平均带权周转时间：" << (totalRightTurnover / total)

        << "\n";

}

Job.cpp

#include "Job.h"

std::ostream & operator<<(std::ostream &output, Job &job) {

    output << "JobID:" << job.jobId

        << "\tArrivalTime:" << job.arrivalTime

        << "\t ExecutionTime:" << job.executionTime

        << "\tPriority:" << job.priority;

    return output;

}

//按到达时间排序函数

int jobSortArrivalTime(Job& left, Job& right) {

    int x = left.arrivalTime - right.arrivalTime;

    return x < 0 ? -1 : x > 0 ? 1 : 0;

}

//按执行时间长短排序函数

int jobSortExecutionTime(Job& left, Job& right) {

    int x = left.executionTime - right.executionTime;

    return x < 0 ? -1 : x > 0 ? 1 : 0;

}

//按优先级排序函数

int jobSortPriority(Job& left, Job& right) {

    int x = left.priority - right.priority;

    return x < 0 ? -1 : x > 0 ? 1 : 0;

}

RR.cpp

#include "RR.h"

//执行

void RR::execute(std::ostream & output) {

    if(list.GetLength() == 0) {

        output << "任务表为空！\n";

        return;

    }

    Job job;

    NodeList<JobPack> arrived;

    JobPack jp;

    //当前时间，总数

    unsigned int now = 0, total = 0;

    //总等待时间，总周转时间，总带权周转时间

    double totalWait = 0, totalTurnover = 0, totalRightTurnover = 0;

    while(list.GetLength() > 0 || arrived.GetLength() > 0) {

        while(list.GetLength() > 0) {

            list.Find(0, job);

            if(job.arrivalTime <= now) {

                //已到达的列表，不排序

                arrived.Insert(arrived.GetLength(), { job, 0 });

                list.Delete(0);

                continue;

            }

            break;

        }

        if(now != 0 && jp.job.executionTime > jp.operated) {

            arrived.Insert(arrived.GetLength(), jp);

        }

        if(now < job.arrivalTime && arrived.GetLength() == 0) {

            now = job.arrivalTime;

            arrived.Insert(arrived.GetLength(), { job, 0 });

            list.Delete(0);

        }

        arrived.Find(0, jp);

        arrived.Delete(0);

        job = jp.job;

        if(job.executionTime - jp.operated < slice) {

            now += job.executionTime - jp.operated;

            output << job.jobId << " 号作业执行了 " << (job.executionTime - jp.operated) << "\n";

            jp.operated = job.executionTime;

        } else {

            now += slice;

            output << job.jobId << " 号作业执行了 " << slice << "\n";

            jp.operated += slice;

        }

        if(job.executionTime == jp.operated) {

            output << "执行完的作业：" << job

                << "\n等待时间：" << (now - job.executionTime - job.arrivalTime)

                << "\t周转时间：" << (now - job.arrivalTime)

                << "\t带权周转时间：" << (double(now - job.arrivalTime) / job.executionTime)

                << "\n";

            total++;

            totalWait += now - job.executionTime - job.arrivalTime;

            totalTurnover += now - job.arrivalTime;

            totalRightTurnover += double(now - job.arrivalTime) / job.executionTime;

        }

    }

    output << "\n平均等待时间：" << (totalWait / total)

        << "\t平均周转时间：" << (totalTurnover / total)

        << "\t平均带权周转时间：" << (totalRightTurnover / total)

        << "\n";

}

SJF.cpp

#include "SJF.h"

//执行

void SJF::execute(std::ostream & output) {

    if(list.GetLength() == 0) {

        output << "任务表为空！\n";

        return;

    }

    Job job;

    NodeList<Job> arrived;

    arrived.SetCompareFunction(jobSortExecutionTime);

    //当前时间，总数

    unsigned int now = 0, total = 0;

    //总等待时间，总周转时间，总带权周转时间

    double totalWait = 0, totalTurnover = 0, totalRightTurnover = 0;

    while(list.GetLength() > 0 || arrived.GetLength() > 0) {

        while(list.GetLength() > 0) {

            list.Find(0, job);

            if(job.arrivalTime <= now) {

                //已到达的列表，按执行时间长短排序

                arrived.Insert(arrived.GetLength(), job, true);

                list.Delete(0);

                continue;

            }

            break;

        }

        if(now < job.arrivalTime && arrived.GetLength() == 0) {

            now = job.arrivalTime;

            list.Delete(0);

        } else {

            arrived.Find(0, job);

            arrived.Delete(0);

        }

        now += job.executionTime;

        output << "执行完的作业：" << job

            << "\n等待时间：" << (now - job.executionTime - job.arrivalTime)

            << "\t周转时间：" << (now - job.arrivalTime)

            << "\t带权周转时间：" << (double(now - job.arrivalTime) / job.executionTime)

            << "\n";

        total++;

        totalWait += now - job.executionTime - job.arrivalTime;

        totalTurnover += now - job.arrivalTime;

        totalRightTurnover += double(now - job.arrivalTime) / job.executionTime;

    }

    output << "\n平均等待时间：" << (totalWait / total)

        << "\t平均周转时间：" << (totalTurnover / total)

        << "\t平均带权周转时间：" << (totalRightTurnover / total)

        << "\n";

}

HPF.cpp

#include "HPF.h"

//执行

void HPF::execute(std::ostream & output) {

    if(list.GetLength() == 0) {

        output << "任务表为空！\n";

        return;

    }

    Job job;

    NodeList<Job> arrived;

    arrived.SetCompareFunction(jobSortPriority);

    //当前时间，总数

    unsigned int now = 0, total = 0;

    //总等待时间，总周转时间，总带权周转时间

    double totalWait = 0, totalTurnover = 0, totalRightTurnover = 0;

    while(list.GetLength() > 0 || arrived.GetLength() > 0) {

        while(list.GetLength() > 0) {

            list.Find(0, job);

            if(job.arrivalTime <= now) {

                //已到达的列表，按优先级排序

                arrived.Insert(arrived.GetLength(), job, true);

                list.Delete(0);

                continue;

            }

            break;

        }

        if(now < job.arrivalTime && arrived.GetLength() == 0) {

            now = job.arrivalTime;

            list.Delete(0);

        } else {

            arrived.Find(0, job);

            arrived.Delete(0);

        }

        now += job.executionTime;

        output << "执行完的作业：" << job

            << "\n等待时间：" << (now - job.executionTime - job.arrivalTime)

            << "\t周转时间：" << (now - job.arrivalTime)

            << "\t带权周转时间：" << (double(now - job.arrivalTime) / job.executionTime)

            << "\n";

        total++;

        totalWait += now - job.executionTime - job.arrivalTime;

        totalTurnover += now - job.arrivalTime;

        totalRightTurnover += double(now - job.arrivalTime) / job.executionTime;

    }

    output << "\n平均等待时间：" << (totalWait / total)

        << "\t平均周转时间：" << (totalTurnover / total)

        << "\t平均带权周转时间：" << (totalRightTurnover / total)

        << "\n";

}

Main.cpp

#include <iostream>

#include <fstream>

#include "Job.h"

#include "LinkList.h"

#include "FCFS.h"

#include "SJF.h"

#include "HPF.h"

#include "RR.h"

using namespace std;

int main() {

    ifstream file("data.txt", ios::in);

    if(!file.is\_open()) {

        cout << "data.txt can't open for read!" << endl;

        system("pause");

        exit(1);

    }

    char str[45];

    //将首行读出

    file.getline(str, 45);

    //作业表

    NodeList<Job> jobs;

    jobs.SetCompareFunction(jobSortArrivalTime);

    Job job;

    while(file >> job.jobId) {

        file >> job.arrivalTime;

        file >> job.executionTime;

        file >> job.priority;

        jobs.Insert(jobs.GetLength(), job, true);

    }

    file.close();

    //先来先服务

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 开始模拟 FCFS 先来先服务 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

    FCFS fcfs(jobs);

    fcfs.execute(cout);

    cout << '\n';

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

    //短作业优先

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 开始模拟 SJF 短作业优先 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

    SJF sjf(jobs);

    sjf.execute(cout);

    cout << '\n';

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

    //优先权高者优先

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 开始模拟 HPF 优先权高者优先 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

    HPF hpf(jobs);

    hpf.execute(cout);

    cout << '\n';

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

    //时间片轮转

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 准备模拟 RR 时间片轮转 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

        << endl << "请输入时间片大小（slice > 0）：";

    unsigned int slice = 0;

    while(slice <= 0) {

        cin >> slice;

    }

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 开始模拟 RR 时间片轮转 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

    RR rr(jobs, slice);

    rr.execute(cout);

    cout << '\n';

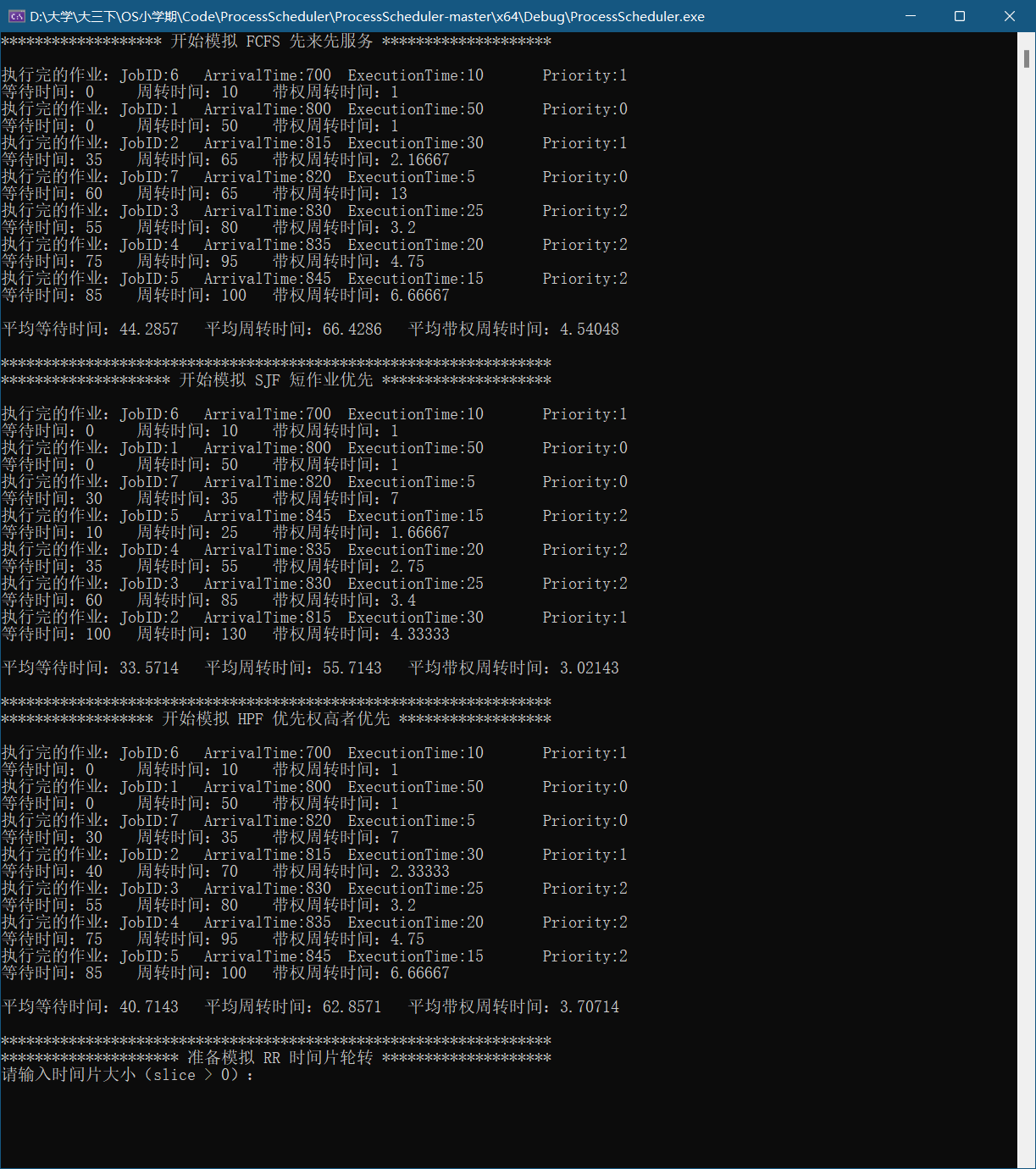
    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

    system("pause");

    return 0;

}

实验结果



【注意事项】

动态优先权与静态优先权的区别：

静态优先级：在创建进程时确定的，且在进程的整个运行期间保持不变。  
动态优先级：在创建进程时所赋予的优先权，是可以随进程的推进或随其等待时间的增加而改变的，以便获得更好的调度性能。  
确定静态优先权的依据有如下三个方面：  
①进程类型：通常，系统进程（如接收进程、对换进程、磁盘I/O进程）的优先权高于一般用户进程的优先权  
②进程对资源的要求：如进程的估计执行时间及内存需要量的多少，对这些要求少的进程应赋予较高的优先权  
③用户要求：这是用户进程的紧迫程度及用户所付费用的多少来确定优先权的

【思考题】

针对某一具体应用环境，如何选择合适的调度算法？

Windows 系统在运行多个应用程序时，比如同时运行10个App，这个时候其调度方式比较复杂，它的处理器调度的调度单位是线程而不是进程，是基于优先级的抢占式多处理器调度，依据优先级和分配时间片来调度。Windows操作系统的调度系统总是运行优先级最高的就绪线程。在同一优先级的各线程按时间片轮转算法进行调度。如果一个高优先级的线程进入就绪状态，当前运行的线程可能在用完它的时间片之前就被抢占处理机。临界资源只有一个进程可以访问。多任务、有线程优先级、多种中断级别这是现代操作系统的共同特点。

任务三 存储管理

【实训目的】

掌握物理内存和虚拟内存的基本概念；掌握重定位的基本概念及其要点，理解逻辑地址与绝对地址；掌握各种存储管理的实现方法，包括基本原理、地址变换和缺页中断、主存空间的分配及分配算法；掌握常用淘汰算法。

【实训内容】

编写一个模拟的动态页式存储管理程序，实现对动态页式存储的淘汰算法的模拟（在先进先出淘汰算法、最近最少使用淘汰算法、最不经常使用淘汰算法三种算法中至少选择两种进行模拟）并计算各个算法的缺页率；并且页面淘汰算法在淘汰一页时，只将该页在页表中抹去，而不再判断它是否被改写过，也不将它写回到辅存。

【预习要求】

常用的存储管理方法及其基本原理；物理内存与虚拟内存、逻辑地址与绝对地址的概念；常用的淘汰算法。

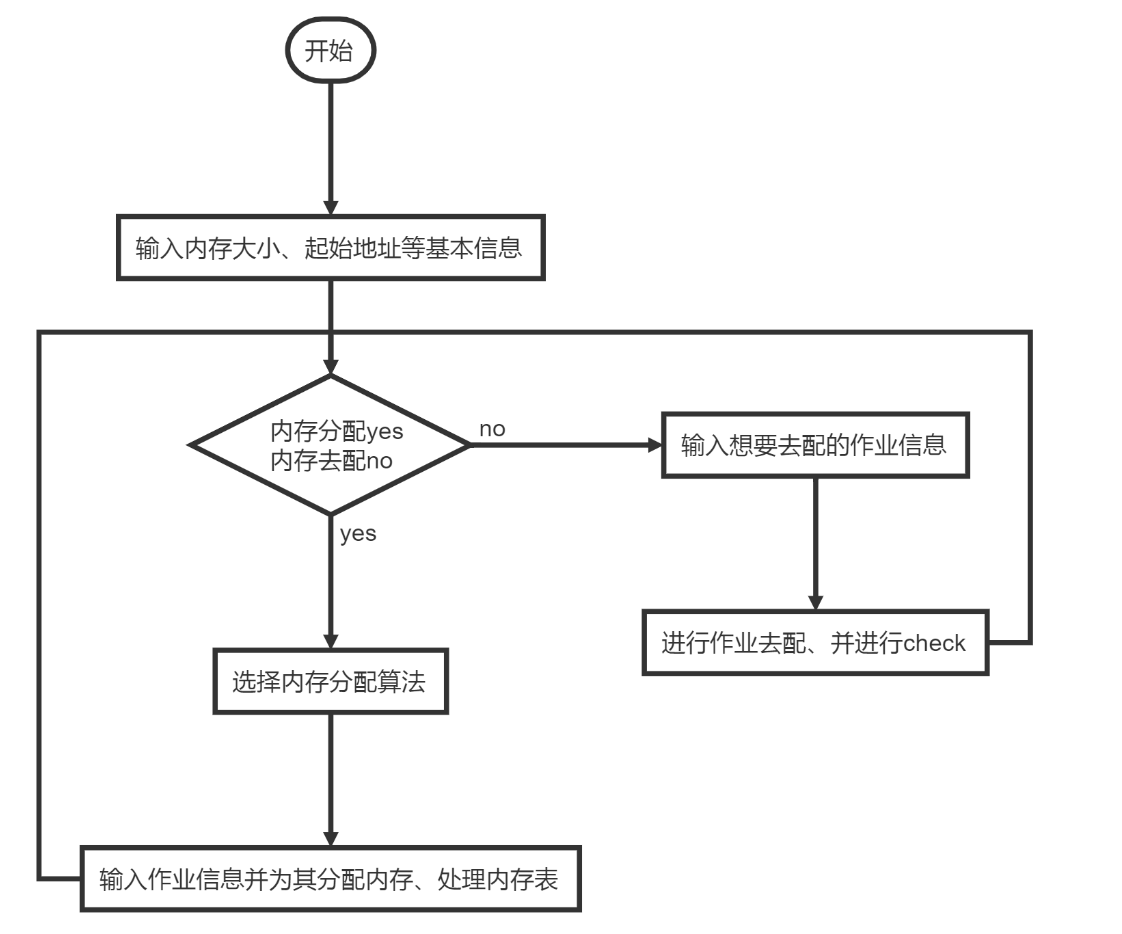
【实训步骤】

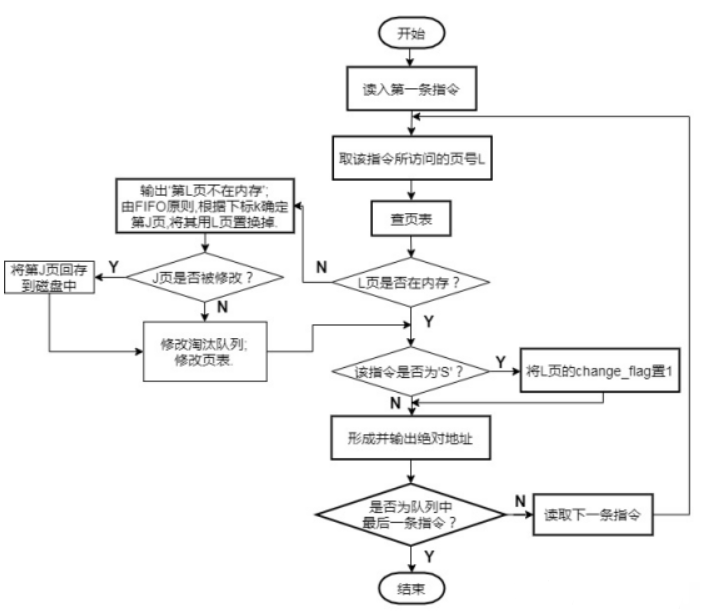
以先进先出淘汰算法为例说明动态页式存储管理的实现过程：

1. 产生一个需要访问的指令地址流，它是一系列需要访问的指令的地址。为不失一般性，你可以适当地（用人工指定地方法或用随机数产生器）生成这个序列，使得 50％的指令是顺序执行的。25％的指令均匀地散布在前地址部分， 25％的地址是均匀地散布在后地址部分；
2. 指定合适的页面尺寸（例如以 1K 或 2K 为 1 页）；
3. 指定内存页表的最大长度，并对页表进行初始化；

每访问一个地址时，首先要计算该地址所在的页的页号，然后查页表，判断该页是否在主存——如果该页已在主存，则打印页表情况；如果该页不在主存且页表未满，则调入一页并打印页表情况；如果该页不足主存且页表已满， 则按 FIFO 页面淘汰算法淘汰一页后调入所需的页，打印页表情况； 逐个地址访问，直到所有地址访问完毕。

1. 存储管理算法的流程图





1. 根据图 5 编写并运行程序，程序运行界面

在地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。当发生缺页中断时，如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。 （1）最佳置换算法（OPT） 这是一种理想情况下的页面置换算法，但实际上是不可能实现的。该算法的基本思想是：发生缺页时，有些页面在内存中，其中有一页将很快被访问（也包含紧接着的下一条指令的那页），而其他页面则可能要到10、100或者1000条指令后才会被访问，每个页面都可以用在该页面首次被访问前所要执行的指令数进行标记。 （2）先进先出置换算法（FIFO） 最简单的页面置换算法是先入先出（FIFO）法。这种算法的实质是，总是选择在主存中停留时间最长（即最老）的一页置换，即先进入内存的页，先退出内存。理由是：最早调入内存的页，其不再被使用的可能性比刚调入内存的可能性大。建立一个FIFO队列，收容所有在内存中的页。被置换页面总是在队列头上进行。当一个页面被放入内存时，就把它插在队尾上。 （3）最近最久未使用（LRU）算法 FIFO算法和OPT算法之间的主要差别是，FIFO算法利用页面进入内存后的时间长短作为置换依据，而OPT算法的依据是将来使用页面的时间。如果以最近的过去作为不久将来的近似，那么就可以把过去最长一段时间里不曾被使用的页面置换掉。它的实质是，当需要置换一页时，选择在之前一段时间里最久没有使用过的页面予以置换。这种算法就称为最久未使用算法。

页面置换算法

PageReplace.cpp

//cjz 2022-6-29

#include "stdafx.h"

void init();

void Work();

void Welcome();

void Opt();     //OPT算法

void Fifo();    //FIFO算法

void Lru();     //LRU算法

void Input(unsigned &, unsigned &); //输入设置的内存大小和页面数量

void Output(vector<int>::iterator); //输出当前页面

void Outputmem(vector<int>&);       //输出当前时刻内存使用情况

void Inputpages(vector<int>&, const unsigned &);    //输入页面

vector<int>::iterator Found(vector<int>&, vector<int>::iterator, unsigned); //从向量中查找一个数并返回它的迭代器

vector<int>pages, mem;  //pages:输入的页面顺序表， mem:当前时刻内存占用情况

vector<int>::iterator pagesIter, memIter;   //迭代器

unsigned size, pagesNum;    //size:内存大小， pagesNum:页面数量

int main()

{

    Work();

    return 0;

}

void init()

{

    pages.clear();

    mem.clear();

    size = -1;

    pagesNum = -1;

}

void Work()

{

    int comd;

    while (true)

    {

        Welcome();

        cin >> comd;

        system("cls");

        switch (comd)

        {

        case 1: {

            init();

            Input(size, pagesNum);

            Inputpages(pages, pagesNum);

            break;

        }

        case 2: {

            Opt();

            break;

        }

        case 3: {

            Fifo();

            break;

        }

        case 4: {

            Lru();

            break;

        }

        case 0: {

            return;

        }

        default: {

            cout << endl << "请输入正确选项" << endl << endl;

            break;

        }

        }

        system("pause");

        system("cls");

    }

}

void Welcome()

{

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*  请选择选项  \*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

    cout << "\* 1. 输入数据                          \*" << endl;

    cout << "\* 2. OPT算法(最佳置换算法)             \*" << endl;

    cout << "\* 3. FIFO算法(先进先出置换算法)        \*" << endl;

    cout << "\* 4. LRU算法(最近最久未使用)           \*" << endl;

    cout << "\* 0. 退出                              \*" << endl;

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

}

void Opt()

{

    vector<int>tmp;

    vector<int>::iterator tmpIter;

    map<unsigned, int>mp;

    tmp.clear();

    mp.clear();

    mem.clear();

    unsigned cnt = 0;

    for (pagesIter = pages.begin(); pagesIter != pages.end(); pagesIter++)

    {

        Output(pagesIter);

        if (mp[\*pagesIter] == 0)

        {

            cnt++;

            if (mem.size() < size)

            {

                mem.push\_back(\*pagesIter);

                mp[\*pagesIter] = 1;

                tmp = mem;

            }

            else

            {

                for (auto ixIter = pagesIter + 1; ixIter != pages.end(); ixIter++)

                {

                    if (tmp.size() == 1)

                        break;

                    tmpIter = Found(tmp, tmpIter, \*ixIter);

                    if (tmpIter != tmp.end())

                        tmp.erase(tmpIter);

                }

                tmpIter = tmp.begin();

                memIter = Found(mem, memIter, \*tmpIter);

                mem.insert(memIter, \*pagesIter);

                memIter = Found(mem, memIter, \*tmpIter);

                mp[\*memIter] = 0;

                mem.erase(memIter);

                mp[\*pagesIter] = 1;

                tmp = mem;

            }

            Outputmem(mem);

        }

    }

    cout << endl;

    cout << "缺页中断次数: " << cnt << endl;

    cout << "缺页率: "<< cnt / (1.0\*pagesNum) << endl << endl;

}

void Fifo()

{

    queue<unsigned>que;

    map<unsigned, int>mp;

    while (!que.empty())

        que.pop();

    mp.clear();

    mem.clear();

    unsigned cnt = 0;

    for (pagesIter = pages.begin(); pagesIter != pages.end(); pagesIter++)

    {

        Output(pagesIter);

        if (mp[\*pagesIter] == 0)

        {

            cnt++;

            if (mem.size() < size)

            {

                mem.push\_back(\*pagesIter);

                que.push(\*pagesIter);

                mp[\*pagesIter] = 1;

            }

            else

            {

                unsigned x = que.front();

                que.pop();

                memIter = Found(mem, memIter, x);

                mem.insert(memIter, \*pagesIter);

                que.push(\*pagesIter);

                mp[\*pagesIter] = 1;

                memIter = Found(mem, memIter, x);

                mem.erase(memIter);

                mp[x] = 0;

            }

            Outputmem(mem);

        }

    }

    cout << endl;

    cout << "缺页中断次数: " << cnt << endl;

    cout << "缺页率: "<< cnt / (1.0\*pagesNum) << endl << endl;

}

void Lru()

{

    vector<int>tmp;

    vector<int>::iterator tmpIter;

    map<unsigned, int>mp;

    mem.clear();

    tmp.clear();

    mp.clear();

    unsigned cnt = 0;

    for (pagesIter = pages.begin(); pagesIter != pages.end(); pagesIter++)

    {

        Output(pagesIter);

        if (mp[\*pagesIter] == 0)

        {

            cnt++;

            if (mem.size() < size)

            {

                mem.push\_back(\*pagesIter);

                mp[\*pagesIter] = 1;

                tmp.push\_back(\*pagesIter);

            }

            else

            {

                tmpIter = tmp.begin();

                memIter = Found(mem, memIter, \*tmpIter);

                mem.insert(memIter, \*pagesIter);

                memIter = Found(mem, memIter, \*tmpIter);

                mem.erase(memIter);

                mp[\*tmpIter] = 0;

                mp[\*pagesIter] = 1;

                tmp.erase(tmpIter);

                tmp.push\_back(\*pagesIter);

            }

            Outputmem(mem);

        }

        else

        {

            tmpIter = Found(tmp, tmpIter, \*pagesIter);

            tmp.erase(tmpIter);

            tmp.push\_back(\*pagesIter);

        }

    }

    cout << endl;

    cout << "缺页中断次数: " << cnt << endl;

    cout << "缺页率: "<< cnt / (1.0\*pagesNum) << endl << endl;

}

void Input(unsigned &size, unsigned &pagesNum)

{

    cout << "请输入内存大小和页面数量" << endl;

    cin >> size >> pagesNum;

}

void Inputpages(vector<int>&ivec, const unsigned &pagesNum)

{

    cout << "请输入" << pagesNum << "个页面" << endl;

    for (unsigned ix = 0; ix < pagesNum; ix++)

    {

        unsigned page;

        cin >> page;

        ivec.push\_back(page);

    }

}

void Output(vector<int>::iterator iter)

{

    cout << \*iter << endl;

}

void Outputmem(vector<int>&ivec)

{

    for (auto ix : ivec)

    {

        cout << " " << ix;

    }

    cout << endl;

}

vector<int>::iterator Found(vector<int>&ivec, vector<int>::iterator iter, unsigned x)

{

    for (iter = ivec.begin(); iter != ivec.end(); iter++)

    {

        if (\*iter == x)

        {

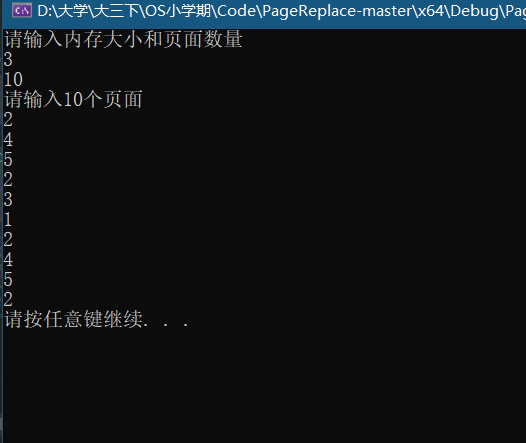
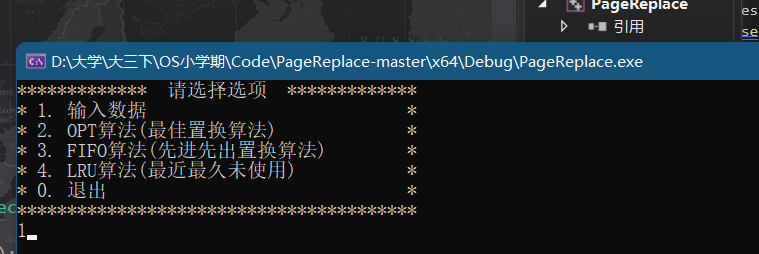
            return iter;

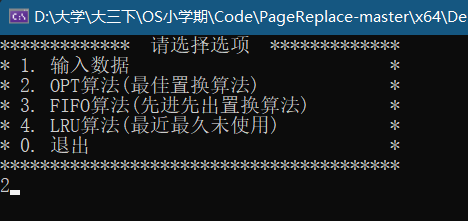
        }

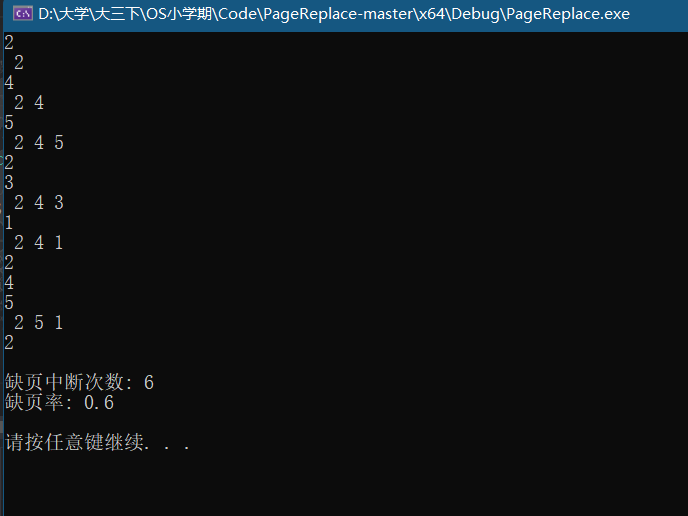
    }

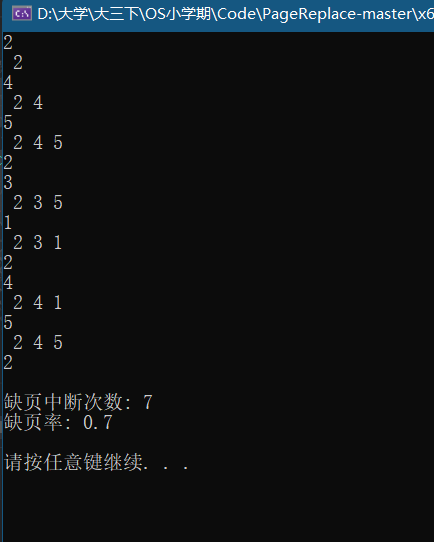
    return ivec.end();

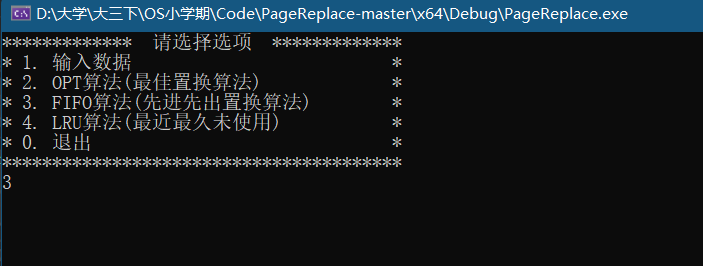
}

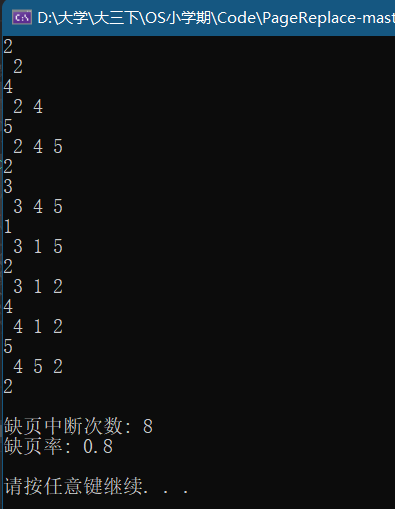


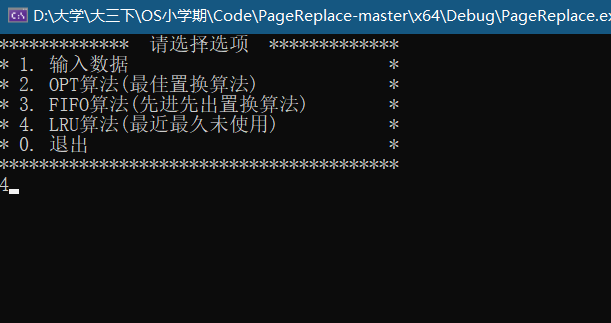












【注意事项】

页面尺寸的指定；如何选择合适的页面淘汰算法以保证较低的缺页率。

【思考题】

各种不同的页面淘汰算法有哪些优缺点？

先进先出置换算法(FIFO）是最直观的算法，由于它可能是性能最差的算法，故实际应用极少。FIFO 页面置换算法易于理解和编程。

发现 Belady 异常的一个结果是寻找最优页面置换算法，这个算法具有所有算法的最低的缺页错误率，并且不会遭受 Belady 异常。这种算法被称为 OPT 或 MIN。该算法的思想是：置换最长时间不会使用的页面。采用最佳值换算法通常可保证获得最低的缺页率。但是由于人们目前还无法预知一个进程在内存的若干个页面中，哪一个页面是未来最长时间内不再被访问的，因此该算法是无法实现的，但可以利用该算法去评价其它算法。

如果我们使用最近的过去作为不远将来的近似，那么可以置换最长时间没有使用的页。这种方法称为最近最少使用算法。LRU 置换将每个页面与它的上次使用的时间关联起来。当需要置换页面时，LRU选择最长时间没有使用的页面。这种策略可当作在时间上向后看而不是向前看的最优页面置换算法。最近最久未使用置换算法(LRU)虽然是一种比较好的算法，但要求系统有较多的支持硬件。

为什么会产生页面抖动？

因为刚被淘汰出去的页，过后不久又要访问它，需要再次调入，而调入后不久又再次被淘汰，然后又要访问，如此反复，使得系统的把大部分时间用在页面的调进调出上，这种形象称为“抖动”。

什么是 b elady 现象？这种现象该如何避免？

在分页式虚拟存储器管理中，发生缺页时的置换算法采用FIFO（先进先出）算法时，如果对一个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多但缺页率反而提高的异常现象。这种现象该如何避免？尽量避免物理块数不断增多缺页率最大。

任务四 设备管理

【实训目的】

掌握独占设备的使用方式，以及设备的分配和回收；掌握用死锁避免方法来处理申请独占设备可能造成的死锁。

【实训内容】

用死锁避免方法来处理申请独占设备可能造成的死锁，程序实现对银行家算法的模拟。

【预习要求】

设备的分类；独占设备的分配与回收；处理死锁的方法。

【实训步骤】

1. 设计银行家算法的数据结构：

假设有 M 个进程 N 类资源，则有如下数据结构：

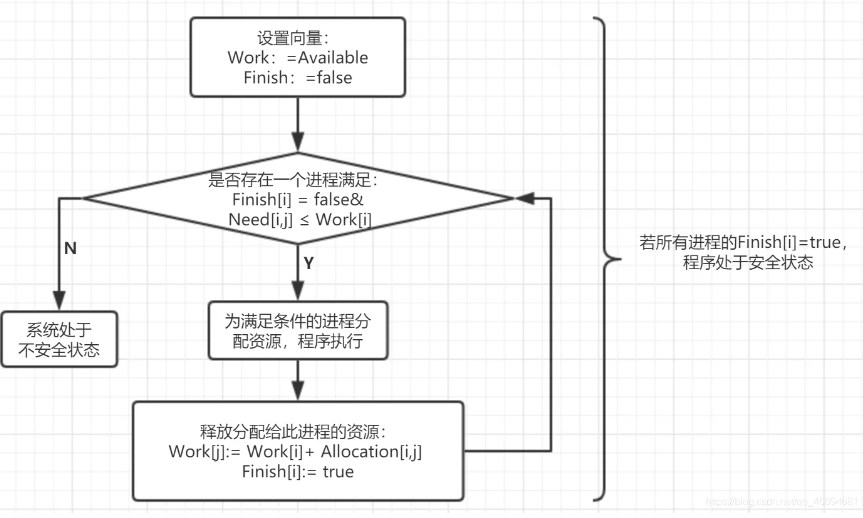
MAX[M\*N] M 个进程对 N 类资源的最大需求量

AVAILABLE[N] 系统可用资源数

ALLOCATION[M\*N] M 个进程已经得到 N 类资源的资源量

NEED[M\*N] M 个进程还需要 N 类资源的资源量

1. 分析银行家算法的实现过程，流程图如下图 7 所示：



银行家算法

DeadLock.cpp

根据流程图编写程序，部分源代码如下所示： #include <iostream>

#define sourceNum 3//源数量

#define processNum 4//处理进程数量

typedef unsigned int NUM;//把无符号整形变量设置为NUM

using namespace std;

class bankAlgorithm{

    //封装后的类

private:

    NUM ava[sourceNum]={1,1,2};//availble数组

    NUM work[sourceNum]={0};//work数组

    NUM max[processNum][sourceNum]={{3,2,2},{6,1,3},{3,1,4},{4,2,2}};//max数组

    NUM allocation[processNum][sourceNum]={{1,0,0},{5,1,1},{2,1,1},{0,0,2}};//allocation数组

    NUM need[processNum][sourceNum]={{2,2,2},{1,0,2},{1,0,3},{4,2,0}};//need数组

    NUM request[4]={0};//request数组

    bool finish[processNum]={0};//标志是否完成数组

public:

    bool sendRequest();//发送request请求

    bool askForSafetyCheck();//请求安全性检查

    void display();//展示函数

};

bool bankAlgorithm::sendRequest(){

    cout<<"Please insert the request vector, 1 for process,2-4 for request num\n";

    //输入数据

    cin>>request[0]>>request[1]>>request[2]>>request[3];

    cout<<"Now checking the request vector...\n";

    //检查need数组

    if (need[request[0]][0]>=request[1]&&need[request[0]][1]>=request[2]&&need[request[0]][2]>=request[3]) {

        //检查availble数组

        if (ava[0]>=request[1]&&ava[1]>=request[2]&&ava[2]>=request[3]) {

            cout<<"Now checking the safety...\n";

            NUM oldAvail\_0 = ava[0];

            NUM oldAvail\_1 = ava[1];

            NUM oldAvail\_2 = ava[2];

            //检查allocation数组

            NUM oldAllocation\_0=allocation[request[0]][0];

            NUM oldAllocation\_1=allocation[request[0]][1];

            NUM oldAllocation\_2=allocation[request[0]][2];

            //以下为改变老数组的值

            NUM oldNeed\_0=need[request[0]][0];

            NUM oldNeed\_1=need[request[0]][1];

            NUM oldNeed\_2=need[request[0]][2];

            ava[0]=ava[0]-request[1];

            ava[1]=ava[1]-request[2];

            ava[2]=ava[2]-request[3];

            allocation[request[0]][0]=allocation[request[0]][0]+request[1];

            allocation[request[0]][1]=allocation[request[0]][1]+request[2];

            allocation[request[0]][2]=allocation[request[0]][2]+request[3];

            need[request[0]][0]=need[request[0]][0]-request[1];

            need[request[0]][1]=need[request[0]][1]-request[2];

            need[request[0]][2]=need[request[0]][2]-request[3];

            //要求安全性检查

            if(!askForSafetyCheck()){

                ava[0]=oldAvail\_0;

                ava[1]=oldAvail\_1;

                ava[2]=oldAvail\_2;

                allocation[request[0]][0]=oldAllocation\_0;

                allocation[request[0]][1]=oldAllocation\_1;

                allocation[request[0]][2]=oldAllocation\_2;

                need[request[0]][0]=oldNeed\_0;

                need[request[0]][1]=oldNeed\_1;

                need[request[0]][2]=oldNeed\_2;

                cout<<"False! Failing the safety test\n";

                return false;

            }else{

                display();

            }

        }else{

            cout<<"False! The request num is bigger than availble num\n";

            return false;

        }

    }else{

        cout<<"False! The request num is bigger than need num\n";

        return false;

    }

    return true;

}

bool bankAlgorithm::askForSafetyCheck(){

    //安全性检查函数

    for(int i=0;i<processNum;i++){finish[i]=false;}//初始化完成数组

    int k=processNum-1;

    bool key=true;

    //初始化work向量

    for (int i=0; i< sourceNum; i++) {

            work[i]=ava[i];

    }

    //找到一个能满足finish==false，need《=work的进程，该进程获得资源后可顺利执行，直到释放自己的所有资源

    while(k>=0){

        if (finish[k]==false) {

            for (int i=0; i<sourceNum; i++) {

                if (need[k][i]>work[i]) {

                    key=false;

                }

            }

            if (key) {

                for (int i=0; i<sourceNum; i++) {

                    work[i]=work[i]+allocation[k][i];

                }

                finish[k]=true;

                k=processNum;

            }

            key=true;

        }

        k--;

    }

    //只有当所有进程的finish都为true，系统才处于安全状态

    if((finish[0])&&(finish[1])&&(finish[2])&&(finish[3])){

        return true;

    }else{

        cout<<"No. ";

        if (!finish[0]) {

            cout<<"1 ";

        }if (!finish[1]) {

            cout<<"2 ";

        }if (!finish[2]) {

            cout<<"3 ";

        }if (!finish[3]) {

            cout<<"4 ";

        }

        cout<<"process can not pass the vector security test\n";

    }

    return false;

}

void bankAlgorithm::display(){

    //展示函数

    cout<<"==================================\n";

    cout<<"max\n";

    for (int i=0; i<processNum; i++) {

        for (int j=0; j<sourceNum; j++) {

            cout<<max[i][j]<<" ";

        }

        cout<<"\n";

    }

    cout<<"allocation\n";

    for (int i=0; i<processNum; i++) {

        for (int j=0; j<sourceNum; j++) {

            cout<<allocation[i][j]<<" ";

        }

        cout<<"\n";

    }

    cout<<"need\n";

    for (int i=0; i<processNum; i++) {

        for (int j=0; j<sourceNum; j++) {

            cout<<need[i][j]<<" ";

        }

        cout<<"\n";

    }

    cout<<"availble\n";

    for (int i=0; i<sourceNum; i++) {

        cout<<ava[i]<<" ";

    }

    cout<<"\nwork\n";

    for (int i=0; i<sourceNum; i++) {

        cout<<work[i]<<" ";

    }

    cout<<"\n";

    cout<<"==================================\n";

}

int main(int argc, const char \* argv[]) {

    bankAlgorithm bank;

    cout<<"=================Welcome to the bank algorithm=================\n";

    while (true) {

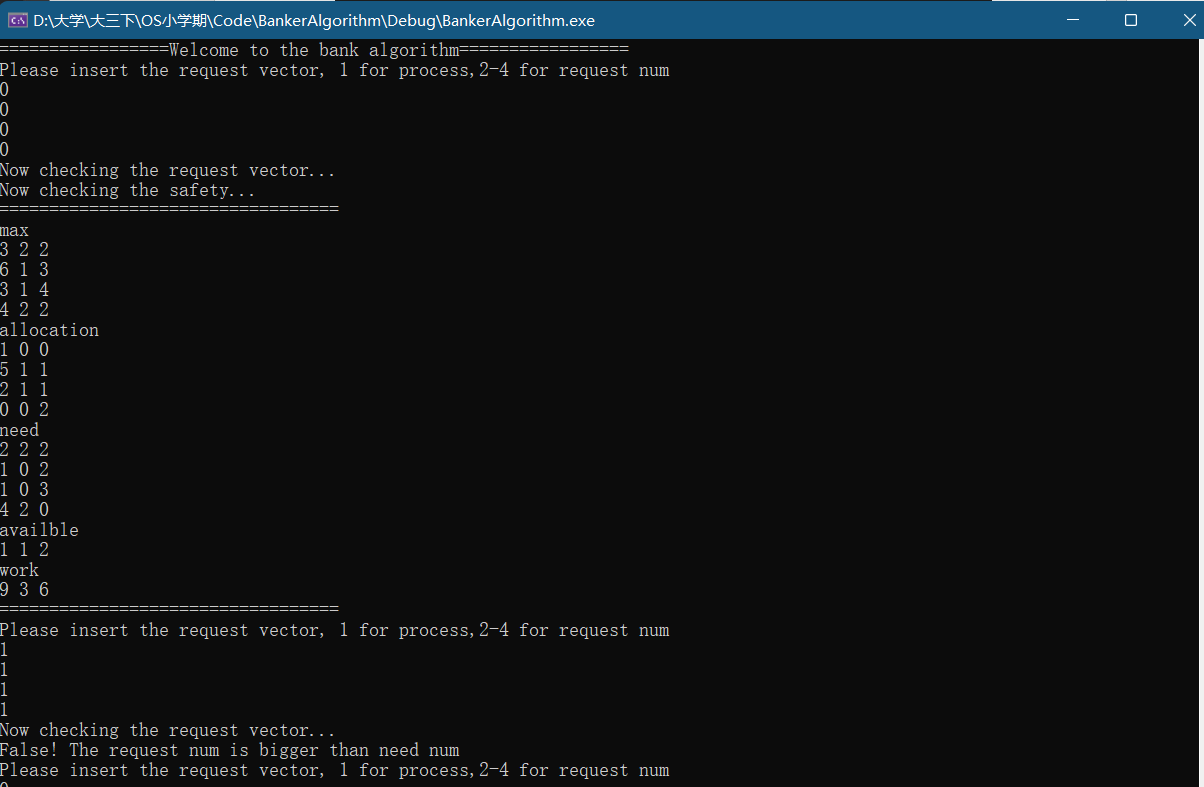
        bank.sendRequest();

    }

    return 0;

}

1. 测试并运行程序，运行界面如图所示：



【注意事项】

独占设备的分配方式。

【思考题】

如果产生了死锁，应如何解除？

1.合理分配

首先，第一种方法就是预防死锁的产生，一次性合理地分配所有的资源，只要有一个资源得不到分配，也不给这个进程分配其他资源。这一方面主要就是预防死锁条件的产生。可以使用银行家算法，合理分配资源。

2.允许抢占

第二种就是发现系统中有进程死锁时，我们可以强制性地剥夺抢占某些进程的资源，然后分配给死锁进程，以解除死锁状态。用一部分死锁进程的资源来解决另外一部分进程的资源。

3.进程挂起

在第二种中，我们是直接销毁某些死锁的进程来解除另外一部分进程的死锁。但是现在我们可以不用销毁这部分的死锁进程。我们可以将其挂起到CPU外，将资源空出来让给死锁进程，这样也可以解决死锁进程。

任务五 文件管理

【实训目的】

掌握文件的存取方法；掌握文件的逻辑结构和物理结构；掌握存储空间的分配和回收；掌握磁盘管理与调度。

【实训内容】

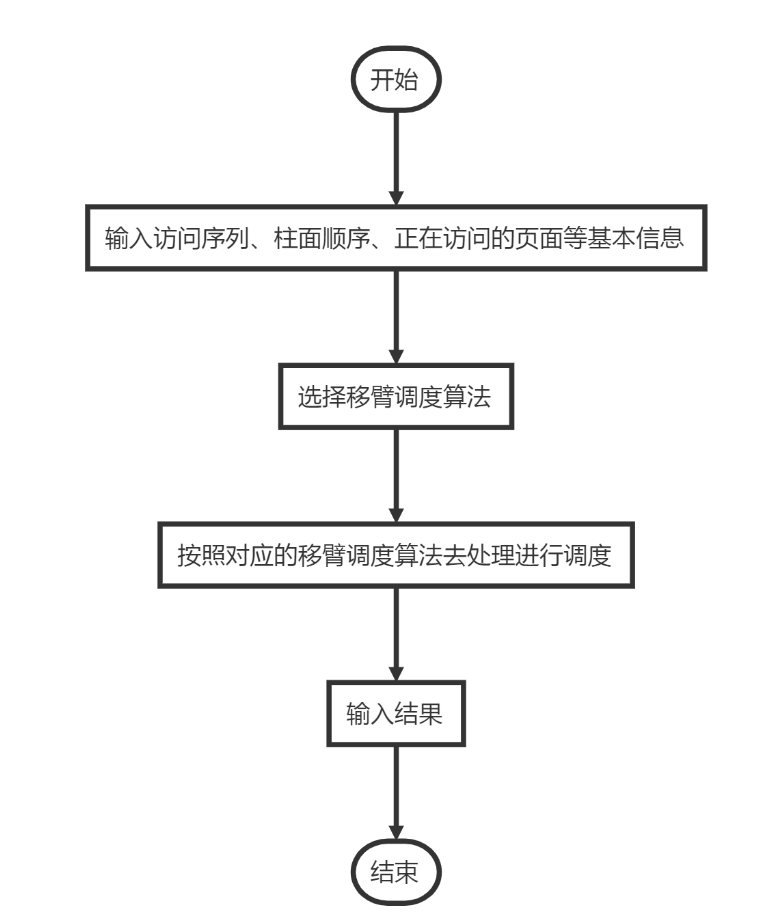
用程序模拟磁盘的调度过程，并计算各磁盘调度算法包括先来先服务算法、最短寻道时间优先算法、扫描算法和循环扫描算法的平均寻道长度。

【预习要求】

文件的逻辑结构和物理结构；常用的磁盘调度算法。

【实训步骤】

1. 分析常用的磁盘调度算法，熟悉其基本原理。



1. 自行设定起始扫描磁道号及最大磁道号数，程序根据设定值随机产生 n 个磁道号进行模拟（n 也可自行设定）；
2. 编写程序实现磁盘调度算法，并显示该算法寻道顺序并计算出寻道总数和平均寻道数；对各种算法的优劣进行比较，得出比较结果。部分源代码如下所示：

磁盘调度算法

main.cpp

//

//  main.cpp

//  DiskSchedulingAlgorithms

//

//  Created by cjz on 2022/6/30.

//

#include <iostream>

#include "DiskTrack.h"

#include "DiskSchedulingAlgorithms.h"

using namespace std;

int main() {

    startTrack = 100;

    diskTrack.append(78);

    diskTrack.append(30);

    diskTrack.append(9);

    diskTrack.append(15);

    diskTrack.append(102);

    diskTrack.append(140);

    diskTrack.append(156);

    diskTrack.append(54);

    diskTrack.append(45);

    diskTrack.append(127);

    // 注释以下代码以从键盘输入数据(使用时需要注释上方内容)

    /\*

    cout << "输入初始位置: ";

    cin >> startTrack;

    cout << "输入磁道数: ";

    int n;

    cin >> n;

    cout << "依次输入磁道序号: ";

    int m;

    for (int i = 0; i < n; i++) {

        cin >> m;

        diskTrack.append(m);

    }

    \*/

    SSTF();

    FCFS();

    SCAN(bigToSmall);

    SCAN(smallToBig);

    CSCAN(bigToSmall);

    CSCAN(smallToBig);

    return 0;

}

DiskTrack.h

//

//  DiskTrack.h

//  DiskSchedulingAlgorithms

//

//  Created by cjz on 2022/6/30.

//

#ifndef DiskTrack\_h

#define DiskTrack\_h

#include <iostream>

#include "Array.h"

using namespace std;

// 磁道访问序列数组

Array<int> diskTrack;

// 被访问的下一个磁道数组

Array<int> nextTrack;

// 寻道距离数组

Array<int> moveDistance;

// 开始磁道号

int startTrack;

// 磁头移动方向

enum HeadMovementDirection {

    smallToBig,

    bigToSmall

};

// 平均寻道长度

float averageSeekLength() {

    int total = 0;

    for (int i = 0; i < moveDistance.length(); i++) {

        total += moveDistance[i];

    }

    return float(total) / float(diskTrack.length());

}

// 输出结果表格

void printResultTable(string algorithmName) {

    // 输出表头

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

    if (algorithmName.length() == 4) {

        cout << "|            " << algorithmName << "            |" << endl;

    }

    else if (algorithmName.length() == 5) {

        cout << "|            " << algorithmName << "           |" << endl;

    }

    else if (algorithmName.length() == 20) {

        cout << "|    " << algorithmName << "    |" << endl;

    }

    else if (algorithmName.length() == 19) {

        cout << "|     " << algorithmName << "    |" << endl;

    }

    cout << "|--------------+-------------|" << endl;

    cout << "|next disktrack|move distance|" << endl;

    cout << "|--------------+-------------|" << endl;

    // 输出表

    for (int i = 0; i < nextTrack.length(); i++) {

        // 输出被访问的下一磁道号

        if (0 <= nextTrack[i] && nextTrack[i] <= 9) {

            printf("|      %d       |", nextTrack[i]);

        }

        else if (10 <= nextTrack[i] && nextTrack[i] <= 99) {

            printf("|      %d      |", nextTrack[i]);

        }

        else if (100 <= nextTrack[i] && nextTrack[i] <= 999) {

            printf("|      %d     |", nextTrack[i]);

        }

        else {

            cout << "|最大支持3位数   ";

        }

        // 输出移动距离

        if (0 <= moveDistance[i] && moveDistance[i] <= 9) {

            printf("      %d      |\n", moveDistance[i]);

        }

        else if (10 <= moveDistance[i] && moveDistance[i] <= 99) {

            printf("      %d     |\n", moveDistance[i]);

        }

        else if (100 <= moveDistance[i] && moveDistance[i] <= 999) {

            printf("      %d    |\n", moveDistance[i]);

        }

        else {

            cout << "  最大支持3位数|" << endl;

        }

        cout << "|--------------+-------------|" << endl;

    }

    // 输出平均寻道长度

    printf("|average seek distance: %.2f|\n", averageSeekLength());

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

}

#endif /\* DiskTrack\_h \*/

Array.h

#include<bits/stdc++.h>

#ifndef Array\_h

#define Array\_h

#include <iostream>

using namespace std;

#define DEFAULTCAP 10

template <typename T>

class Array {

private:

    int size;//数组真实长度

    int capacity;//申请的空间大小

    void expand() {

        while (size >= capacity) {

            T\* oldElem = elem;

            elem = new T[capacity \*= 2];

            for (int i = 0; i < size; i++) {

                elem[i] = oldElem[i];

            }

            delete[] oldElem;

        }

    }//扩容

public:

    T\* elem;//数据

    Array();//构造函数，创建顺序表

    Array(int s);//创建顺序表, 申请初始空间为s

    Array(int a, int b, T\* e);//基于复制的构造函数, 将e数组的[a, b)复制为新数组

    int length() { return size; }

    void append(T e);

    void insert(T e, int p);//插入指定序号的元素，将元素e插入在下标为p的位置

    T remove(int p);//删除指定序号元素，返回其数据备份

    T operator [] (int i) { return elem[i]; }//通过下标读取表元

    void clear();//清除所有数据并收回内存空间重新分配

    void sort();//排序接口，随机使用下列排序方法中的一种

    void mergingSort(int low, int high);//归并排序

    void straightInsertionSort();//直接插入排序

    void binaryInsertionSort();//折半插入排序

    void bubbleSort();//起泡排序

    void simpleSelectionSort();//简单选择排序

    void heapSort();//堆排序

    T getMin() { return getMin(0, size); }//获取整个数组中的最小值

    T getMin(int low, int high);//在区间[i, j)中获取最小值

    T getMax() { return getMax(0, size); }//获取整个数组中的最大值

    T getMax(int low, int high);//在区间[i, j)中获取最大值

    int search(T e);//查找接口，随机使用下列查找方法中的一种

    int sequentialSearch(T e);//顺序查找是否有数据为e的元素并返回其下标

    int reverseSearch(T e);//倒序查找是否有数据为e的元素并返回其下标

    int binarySearch(T e);//折半查找是否有数据为e的元素并返回其下标

    void reverse();//表元素逆置

    void input();//输入

    void output();//输出

    void swap(T& e1, T& e2);//交换两个元素的值

};

template <typename T>

Array<T> merge(Array<T> A1, Array<T> A2) {//归并两个有序顺序表

    Array<T> A3;

    int i = 0, j = 0, k = 0;

    while (i < A1.length() && j < A2.length()) {//停止循环的条件是其中一个顺序表已经比较结束，另一个顺序表就可以直接把剩余部分接在新顺序表后

        if (A1[i] <= A2[j]) {

            A3.insert(A1[i], k++);

            i++;

        }

        else {

            A3.insert(A2[j], k++);

            j++;

        }

    }

    if (i == A1.length()) {

        for (; j < A2.length(); j++) {//将剩余部分直接连接到新表后

            A3.insert(A2[j], k++);

        }

    }

    else {

        for (; i < A1.length(); i++) {

            A3.insert(A1[i], k++);

        }

    }

    return A3;

}

template <typename T>

Array<T>::Array() {

    size = 0;

    capacity = DEFAULTCAP;

    elem = new T[capacity];

}

template <typename T>

Array<T>::Array(int s) {

    size = s;

    capacity = size;

    elem = new T[capacity];

}

template <typename T>

Array<T>::Array(int a, int b, T\* e) {

    size = b - a;

    capacity = size;

    elem = new T[capacity];

    for (int i = 0; i < size; i++) {

        elem[i] = e[a + i];

    }

}

template <typename T>

void Array<T>::append(T e) {

    expand();

    elem[size] = e;

    size++;

}

template <typename T>

void Array<T>::insert(T e, int p) {

    expand();

    size++;

    if (p >= size) p = size;//如果p大于等于长度，则插入在列表末尾

    for (int i = size - 1; i > p; i--) {//将当前在p及之后的元素依次后移一位

        elem[i] = elem[i - 1];

    }

    elem[p] = e;

}

template <typename T>

T Array<T>::remove(int p) {

    T backup = elem[p];

    for (int i = p; i < size; i++) {

        elem[i] = elem[i + 1];

    }

    size--;

    return backup;

}

template <typename T>

void Array<T>::clear() {

    size = 0;

    capacity = DEFAULTCAP;

    elem = new T[capacity];

}

// MARK: - 排序算法

template <typename T>

void Array<T>::sort() {

    switch (size % 4) {

    case 0: simpleSelectionSort();

    case 1: straightInsertionSort();

    case 2: binaryInsertionSort();

    case 3: bubbleSort();

    }

}

// 直接插入排序

template <typename T>

void Array<T>::straightInsertionSort() {

    for (int i = 1; i < size; i++) {

        int j;

        if (elem[i] < elem[i - 1]) {

            elem[size] = elem[i];//elem[size]作临时存储的哨兵

            elem[i] = elem[i - 1];

            for (j = i - 2; elem[size] < elem[j] && j >= 0; j--) {

                elem[j + 1] = elem[j];

            }

            elem[j + 1] = elem[size];

        }

    }

}

// 折半插入排序

template <typename T>

void Array<T>::binaryInsertionSort() {

    for (int i = 1; i < size; i++) {

        int j;

        if (elem[i] < elem[i - 1]) {

            elem[size] = elem[i];

            int low = 0, high = i - 1;

            while (low <= high) {

                int mid = (low + high) / 2;

                if (elem[size] < elem[mid])

                    high = mid - 1;

                else

                    low = mid + 1;

            }

            for (j = i - 1; j >= high + 1; j--)

                elem[j + 1] = elem[j];

            elem[high + 1] = elem[size];

        }

    }

}

// 起泡排序

template <typename T>

void Array<T>::bubbleSort() {

    bool sorted = false;//判别起泡排序结束的条件时在一趟排序中没有进行过交换记录的操作

    while (!sorted) {

        sorted = true;

        for (int j = 1; j < size; j++) {

            if (elem[j] < elem[j - 1]) {

                sorted = false;

                swap(elem[j], elem[j - 1]);

            }

        }

    }

}

// 简单选择排序

template <typename T>

void Array<T>::simpleSelectionSort() {

    for (int i = 0; i < size; i++) {

        if (elem[i] != getMin(i, size)) {

            swap(elem[i], elem[binarySearch(getMin(i, size))]);

        }

    }

}

template <typename T>

T Array<T>::getMin(int low, int high) {

    T min = elem[low];

    for (int i = low + 1; i < high; i++)

        if (elem[i] < min)

            min = elem[i];

    return min;

    /\*

     sort();

     return elem[i];

     \*/

}

template <typename T>

T Array<T>::getMax(int low, int high) {

    T max = elem[low];

    for (int i = low + 1; i < high; i++)

        if (elem[i] > max)

            max = elem[i];

    return max;

    /\*

     sort();

     return elem[j - 1];

     \*/

}

// MARK: - 查找算法

template <typename T>

int Array<T>::search(T e) {

    switch (size % 3) {

    case 0: return sequentialSearch(e);

    case 1: return reverseSearch(e);

    default: return binarySearch(e);

    }

}

// 顺序查找

template <typename T>

int Array<T>::sequentialSearch(T e) {

    for (int i = 0; i < size; i++) {

        if (elem[i] == e)

            return i;

    }

    return -1;

}

// 倒序查找

template <typename T>

int Array<T>::reverseSearch(T e) {

    for (int i = size - 1; i >= 0; i--) {

        if (elem[i] == e)

            return i;

    }

    return -1;

}

// 折半查找

template <typename T>

int Array<T>::binarySearch(T e) {

    sort();

    int low = 0;

    int high = size - 1;

    int mid = (low + high) / 2;

    while (low <= high && elem[mid] != e) {

        if (elem[mid] > e) {

            high = mid - 1;

        }

        else {

            low = mid + 1;

        }

        mid = (low + high) / 2;

    }

    if (elem[mid] == e)

        return mid;

    else

        return -1;

}

// 转置

template <typename T>

void Array<T>::reverse() {

    for (int i = 0; i < size / 2; i++) {

        swap(elem[i], elem[size - i - 1]);

    }

}

template <typename T>

void Array<T>::input() {

    cout << "输入元素个数: ";

    cin >> size;

    expand();//检查是否需要扩容

    cout << "输入元素: ";

    for (int i = 0; i < size; i++)

        cin >> elem[i];

}

template <typename T>

void Array<T>::output() {

    for (int i = 0; i < size; i++)

        cout << elem[i] << " ";

    cout << endl;

}

template <typename T>

void Array<T>::swap(T& e1, T& e2) {

    T t;

    t = e1;

    e1 = e2;

    e2 = t;

}

#endif /\* Array\_h \*/

DiskSchedulingAlgorithms.h

#ifndef DiskSchedulingAlgorithms\_h

#define DiskSchedulingAlgorithms\_h

#include "DiskTrack.h"

#include <iostream>

#include <math.h>

using namespace std;

// MARK: - 先来先服务(FCFS)

void FCFS() {

    // 下一磁道号即原始数组顺序

    nextTrack = diskTrack;

    moveDistance.append(abs(diskTrack[0] - startTrack));

    for (int i = 1; i < diskTrack.length(); i++) {

        moveDistance.append(abs(diskTrack[i] - diskTrack[i - 1]));

    }

    // 输出表格

    printResultTable("FCFS");

    // 重置nextTrack数组

    nextTrack.clear();

    // 重置moveDistance数组

    moveDistance.clear();

}

// MARK: - 最短寻道时间优先(SSTF)

void SSTF() {

    // 复制磁道序列函数用于修改数组的操作, 避免修改原始数组

    Array<int> tempArray(0, diskTrack.length(), diskTrack.elem);

    // 当前磁道

    int currentTrack = startTrack;

    // 最短移动距离

    int shortestMoveDistance;

    // 最近磁道的下标 (这个初值0永远不会用到, 只是为了消除一个编译warning)

    int nearestTrack = 0;

    // 遍历整个序列

    for (int i = 0; i < diskTrack.length(); i++) {

        shortestMoveDistance = INT\_MAX;

        for (int i = 0; i < tempArray.length(); i++) {

            // 当有待访问磁道号等于当前磁道号时

            if (tempArray[i] == currentTrack) {

                shortestMoveDistance = 0;

                nearestTrack = i;

                break;

            }

            if (abs(currentTrack - tempArray[i]) < shortestMoveDistance) {

                shortestMoveDistance = abs(currentTrack - tempArray[i]);

                nearestTrack = i;

            }

        }

        // 将当前移动距离加入移动距离数组

        moveDistance.append(shortestMoveDistance);

        // 更新当前磁道

        currentTrack = tempArray[nearestTrack];

        // 将当前磁道加入下一个磁道号数组

        nextTrack.append(currentTrack);

        // 移除当前磁道号, 便于下一次在数组中寻找最近磁道

        tempArray.remove(nearestTrack);

    }

    // 输出表格

    printResultTable("SSTF");

    // 重置nextTrack数组

    nextTrack.clear();

    // 重置moveDistance数组

    moveDistance.clear();

}

// MARK: - 扫描(SCAN)算法

void SCAN(HeadMovementDirection hmd) {

    // 复制磁道序列函数用于修改数组的操作, 避免修改原始数组

    Array<int> tempArray(0, diskTrack.length(), diskTrack.elem);

    // 将磁头当前磁道号加入进行升序排序

    tempArray.append(startTrack);

    tempArray.sort();

    // 如果磁头移动方向为从大到小, 则把升序数组逆置为降序数组

    if (hmd == bigToSmall) {

        tempArray.reverse();

    }

    // 从起始位置向后

    for (int i = tempArray.sequentialSearch(startTrack) + 1; i < tempArray.length(); i++) {

        nextTrack.append(tempArray[i]);

        moveDistance.append(abs(tempArray[i] - tempArray[i - 1]));

    }

    // 从最后磁道移动到起始点前一个磁道

    nextTrack.append(tempArray[tempArray.sequentialSearch(startTrack) - 1]);

    moveDistance.append(abs(tempArray[tempArray.length() - 1] - tempArray[tempArray.sequentialSearch(startTrack) - 1]));

    // 从起始点前一个磁道向磁道号减小方向移动

    for (int i = tempArray.sequentialSearch(startTrack) - 1; i > 0; i--) {

        nextTrack.append(tempArray[i - 1]);

        moveDistance.append(abs(tempArray[i] - tempArray[i - 1]));

    }

    if (hmd == smallToBig) {

        printResultTable("SCAN (small to big)");

    }

    else {

        printResultTable("SCAN (big to small)");

    }

    nextTrack.clear();

    moveDistance.clear();

}

// MARK: - 循环扫描(CSCAN)算法

void CSCAN(HeadMovementDirection hmd) {

    // 复制磁道序列函数用于修改数组的操作, 避免修改原始数组

    Array<int> tempArray(0, diskTrack.length(), diskTrack.elem);

    // 将磁头当前磁道号加入进行升序排序

    tempArray.append(startTrack);

    tempArray.sort();

    // 如果磁头移动方向为从大到小, 则把升序数组逆置为降序数组

    if (hmd == bigToSmall) {

        tempArray.reverse();

    }

    // 从起始位置延数组下标序号向后

    for (int i = tempArray.sequentialSearch(startTrack) + 1; i < tempArray.length(); i++) {

        nextTrack.append(tempArray[i]);

        moveDistance.append(abs(tempArray[i] - tempArray[i - 1]));

    }

    // 从最后磁道移动到第一个磁道

    nextTrack.append(tempArray[0]);

    moveDistance.append(abs(tempArray[tempArray.length() - 1] - tempArray[0]));

    // 从第一个磁道向磁道数组序号增加方向移动

    for (int i = 1; i < tempArray.sequentialSearch(startTrack); i++) {

        nextTrack.append(tempArray[i]);

        moveDistance.append(abs(tempArray[i] - tempArray[i - 1]));

    }

    if (hmd == smallToBig) {

        printResultTable("CSCAN (small to big)");

    }

    else {

        printResultTable("CSCAN (big to small)");

    }

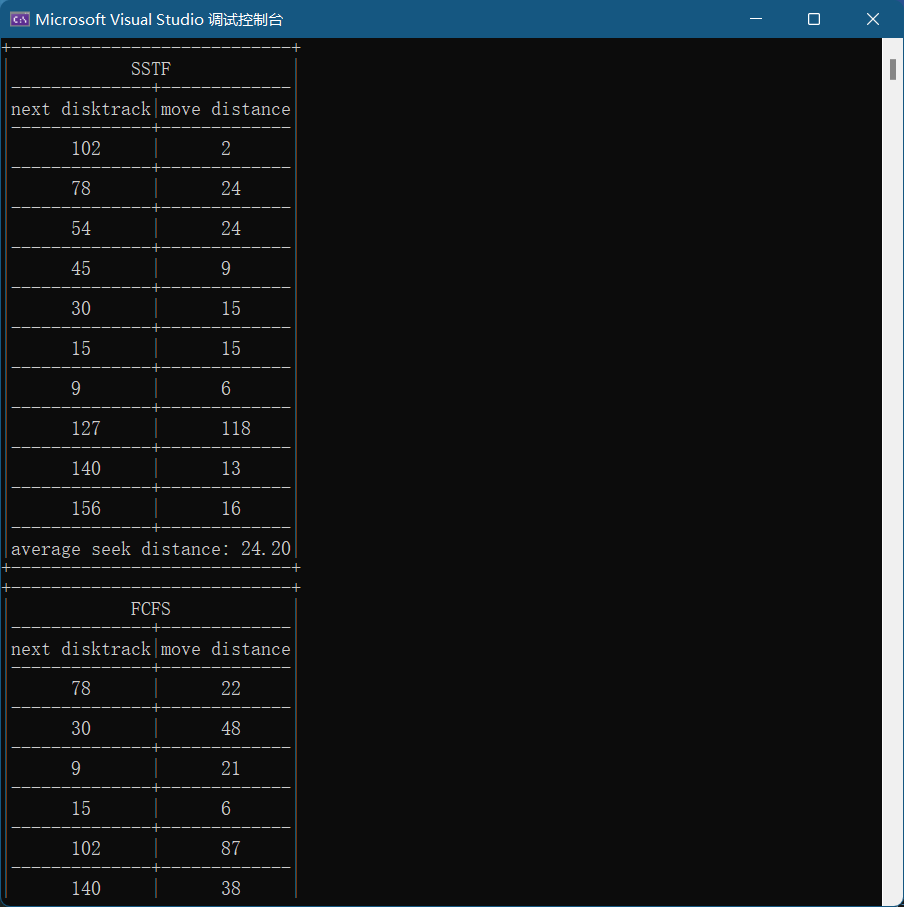
    nextTrack.clear();

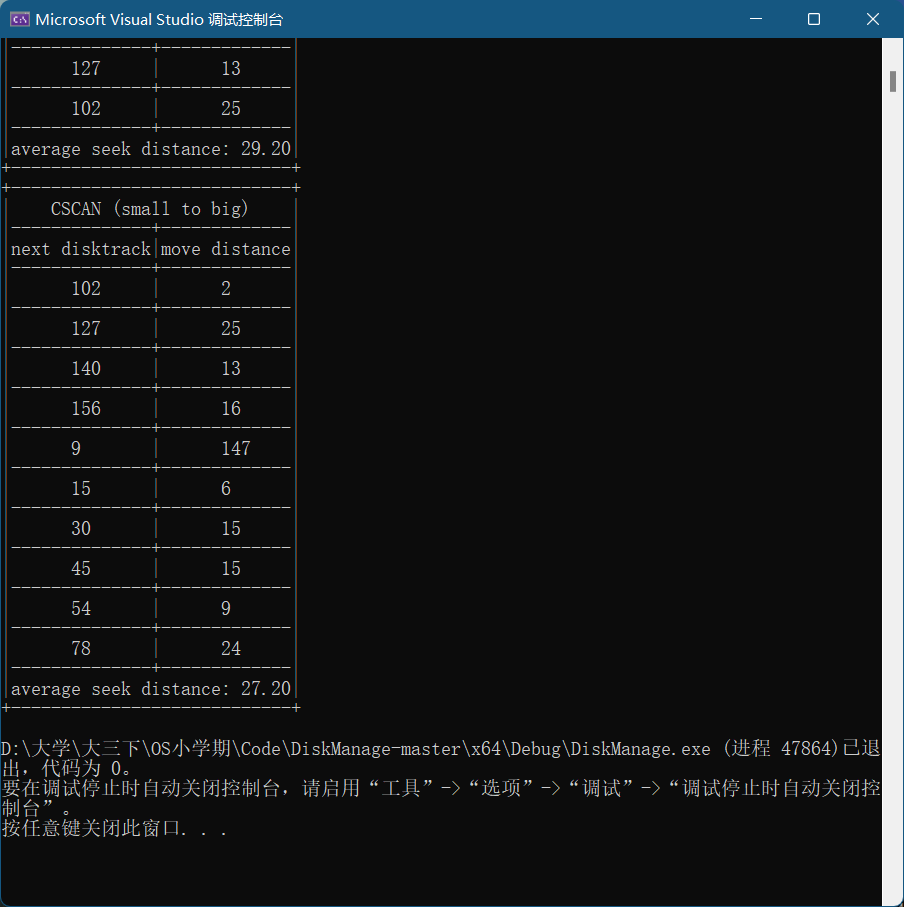
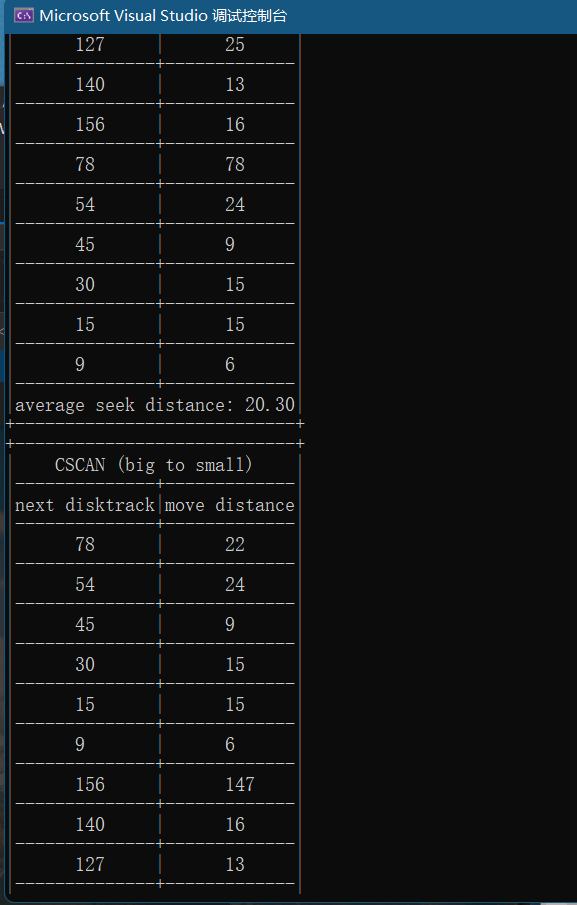
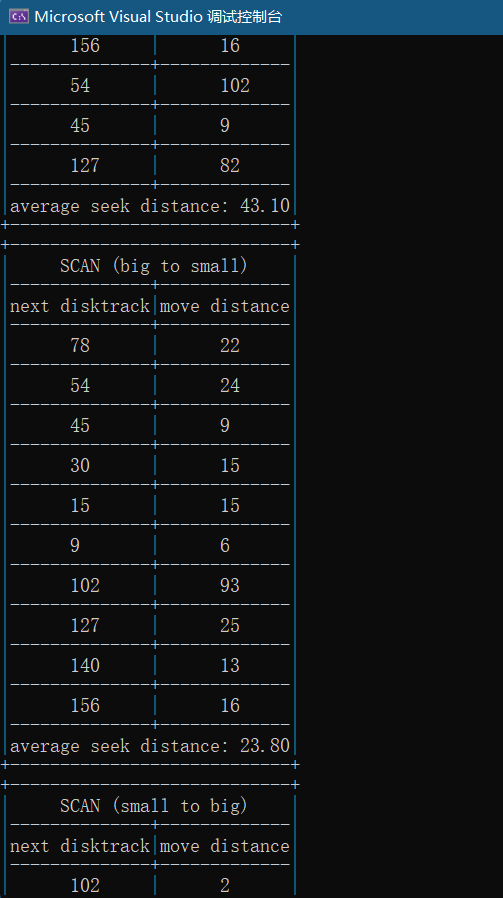
    moveDistance.clear();

}

#endif /\* DiskSchedulingAlgorithms\_h \*/

1. 测试并运行程序，运行界面如图所示：





磁盘调度算法运行界面

【注意事项】

避免扫描算法中的饥饿现象。

【思考题】

如何在文件管理模块增加如下功能:

改变目录：格式：cd <目录名>

显示目录：格式：dir [<目录名>]

创建目录：格式：md <目录名>

除目录：格式：rd <目录名>

新建文件：格式：edit <文件名>

删除文件：格式：del <文件名>

退出文件系统：exit

答，通过输入函数进行命令读取，在程序内部进行命令的执行，以实现到各命令功能。可以通过算法来实现class Command ：该类用来实现对文件各种指令操作 ; 实现程序的核心代码 ;

class MainFrame ：该类用来实现界面显示，给与用户进行人机交互的操作界面

class ExpandListener ：该类用来监听用户对树的操作 ; 并查找每一个文件根目录下的子文件路径

class FileMouseListener ：该类用来监听用户的鼠标操作 ; 实现鼠标选定根目录文件

class FileTreeCellRenderer ：该类用来设置文件在界面中的图标

任务六 模块集成

【实训目的】

增强学生的编程能力及团队精神。

【实训内容】

1.将以上四个模块集成为一个较完整的模拟操作系统。

2.进行系统中整体代码的调试和优化

3.归纳并解决实训过程中遇到的问题，书写实训报告

4.提交源程序和实训报告。

【预习要求】

操作系统中各模块之间的有机联系。

【实训步骤】

1.四个模块集成为一个较完整的模拟操作系统。

2.进行系统中整体代码的调试和优化

3.归纳并解决实训过程中遇到的问题，书写实训报告

4.提交源程序和实训报告。【注意事项】

1.源代码格式规范，注释不少于三分之一；

2.对程序的每一部分要有详细的设计分析说明；

3.程序执行的每个步骤要有具体的提示内容或输出；

4.画出程序的基本结构框图和流程图；

5.提交完整程序代码、可执行程序、实训设计报告等相关文档。

模拟操作系统

#include <windows.h>

#include <iostream>

#include<string>

#include<queue>

#include <iomanip>

#include <random>

#include <algorithm>

using namespace std;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

const unsigned short SIZE\_OF\_BUFFER = 10;   //缓冲区长度

unsigned short ProductID = 0;               //产品号

unsigned short ConsumeID = 0;               //将被消耗的产品号

unsigned short in = 0;                      //产品进缓冲区时的下标

unsigned short out = 0;                     //产品出缓冲区时的下标

int pv\_buffer[SIZE\_OF\_BUFFER];               //缓冲区是个循环队列

bool pv\_continue = true;                     //控制程序结束

HANDLE pv\_hMutex;                            //用于线程间的互斥，主要用于控制台打印

HANDLE fullSemaphore;                        //消费者进程的私用信号量

HANDLE emptySemaphore;                       //生产者进程的私用信号量

int sum = 0;

DWORD WINAPI Producer(LPVOID);              //生产者线程

DWORD WINAPI Consumer(LPVOID);              //消费者线程

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int couttime = 0;           //计数时间

float sumtime = 0;      //总周转时间

float sumrighttime = 0; //总带权周转时间

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

const int Max\_process = 50;//最大进程数

const int Max\_source = 50;//最大资源数

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//生产一个产品。简单模，仅输出新产品的ID号

void Produce()

{

    cout << "生产 " << ++ProductID << "号产品   ";

    Sleep(1500);

    cout << "  " << endl;

    cout<<"##############################################"<<endl;

}

//把新生产的产品放入缓冲区

void Put()

{

    cout << "向仓库中放入产品     ";

    pv\_buffer[in] = ProductID;

    in = (in + 1) % SIZE\_OF\_BUFFER;

    Sleep(1500);

    sum = sum + 1;

    cout << "产品总数为："<<sum << endl;

    cout<<"##############################################"<<endl;

}

//从缓冲区中取出一个产品

void Take()

{

    ConsumeID = pv\_buffer[out];

    cout << "取走 " << ConsumeID << " 号产品    ";

    pv\_buffer[out] = -1;

    out = (out + 1) % SIZE\_OF\_BUFFER;

    Sleep(1500);

    cout << " " << endl;

    cout<<"##############################################"<<endl;

}

//消耗一个产品

void Consume()

{

    cout << "消费 " << ConsumeID << " 号产品        ";

    Sleep(1500);

    sum = sum - 1;

    cout << "产品总数为："<<sum << endl;

    cout<<"##############################################"<<endl;

}

//生产者

DWORD  WINAPI Producer(LPVOID lpPara)

{

    while (pv\_continue) {

        WaitForSingleObject(emptySemaphore, INFINITE);  // P(emptySemaphore) 生产者信号量 -1

        WaitForSingleObject(pv\_hMutex, INFINITE);      // P(Mutex) 获取线程间互斥信号

        Produce();

        Put();

        Sleep(1500);

        ReleaseSemaphore(fullSemaphore, 1, NULL);        // V(fullSemaphore) 消费者信号量 +1

        ReleaseMutex(pv\_hMutex);                      // V(Mutex) 释放线程间互斥信号

    }

    return 0;

}

//消费者

DWORD  WINAPI Consumer(LPVOID lpPara)

{

    while (pv\_continue) {

        WaitForSingleObject(fullSemaphore, INFINITE);   //P(fullSemaphore) 消费者信号量-1

        WaitForSingleObject(pv\_hMutex, INFINITE);       //P(Mutex) 获得线程间互斥信号

        Take();

        Consume();

        Sleep(1500);

        ReleaseSemaphore(emptySemaphore, 1, NULL);        //V(emptySemaphore) 生产者信号量+1

        ReleaseMutex(pv\_hMutex);                       //V(Mutex) 释放线程间互斥信号

    }

    return 0;

}

//高优先权算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

typedef struct pcb {

    string pName;//进程名

    int priorityNumber;//优先数

    float serviceTime;//服务时间

    float estimatedRunningtime;//估计运行时间

    float startTime;

    float endTime;

    char state;//状态

    bool operator<(const struct pcb &a)const {

        return priorityNumber < a.priorityNumber || priorityNumber == a.priorityNumber&&estimatedRunningtime < a.estimatedRunningtime;

    }

}PCB;

void createProcess(priority\_queue<PCB> &p, int n) {//创建n个进程，带头结点

    cout << "开始创建进程" << endl;

    PCB r;//工作结点

    for (int i = 0; i<n; i++) {

        cout << "请输入第" << i + 1 << "个进程的名字、优先数、服务时间(例如:A 1 2 ):";

        cin >> r.pName;

        cin >> r.priorityNumber;

        cin >> r.serviceTime;

        r.estimatedRunningtime = r.serviceTime;

        r.state = 'R';

        p.push(r);

    }

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

}

void printProcess(priority\_queue<PCB> p) {

    PCB s;

    cout << "进程名\t优先数 服务时间 已运行时间 还剩运行时间" << endl;

    while (p.size() != 0) {

        s = p.top();

        cout << s.pName << "\t" << s.priorityNumber << "\t " << s.serviceTime << "\t  ";

        cout << s.serviceTime - s.estimatedRunningtime << "\t     " << s.estimatedRunningtime << endl;

        p.pop();

    }

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

}

void runProcess(priority\_queue<PCB> &p) {//运行进程

    PCB s;

    while (p.size() != 0) {

        s = p.top();

        p.pop();

        cout << "正在运行的进程" << endl;

        cout << "进程名\t优先数 服务时间 已运行时间 还剩运行时间" << endl;//输出当前进程

        cout << s.pName << "\t" << s.priorityNumber << "\t " << s.serviceTime << "\t  ";

        cout << s.serviceTime - s.estimatedRunningtime << "\t     " << s.estimatedRunningtime << endl;

        if (s.serviceTime == s.estimatedRunningtime)

        {

            s.startTime = couttime;

        }

        couttime = couttime + 1;

        s.priorityNumber--;//优先数减1

        s.estimatedRunningtime--;//估计运行时间减1

        if (s.estimatedRunningtime == 0) {

            s.state = 'C';

            s.endTime = couttime;

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

            cout << "进程"<<s.pName <<"执行完成"<<endl;

            cout << s.pName << "周转时间为" <<s.endTime-s.startTime<<endl;

            cout << s.pName << "带权周转时间为" << (s.endTime - s.startTime)/s.serviceTime<<endl;

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

            sumtime = sumtime + (s.endTime - s.startTime);

            sumrighttime = sumrighttime + ((s.endTime - s.startTime) / s.serviceTime);

        }

        else

            p.push(s);

        cout << "进程" << s.pName << "执行一次之后就绪队列中的进程" << endl;

        printProcess(p);

    }

    cout << endl;

}

//存储管理\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void FIFO(int b, string a)

{

    char m[10] = { '#','#','#','#','#','#','#','#','#','#' };//将物理块数内容初始化

    int i = 0, j, x, y = 0, count1 = 0;//y为置换标记,count1为缺页标记

    do {

        if (i<b)

        {

            m[y] = a[i];//缺页后将页面写入

            i++;

            count1++;//缺页标记++

            y = (y + 1) % b;//置换标记++

            cout << a[i - 1] << " ";

            for (j = 0;j<b;j++)//打印

                cout << m[j];

            cout << endl;

            continue;

        }

        for (j = 0;j<b;j++)//判断是否缺页

        {

            if (a[i] == m[j])//未缺页则将x赋值1

            {

                x = 1;

                break;

            }

        }

        if (x == 1)//未缺页处理

        {

            i++;

            x = 0;

            cout << a[i - 1] << " " << endl;

            continue;

        }

        else//缺页处理

        {

            m[y] = a[i];

            i++;

            y = (y + 1) % b;

            cout << a[i - 1] << " ";

            for (j = 0;j<b;j++)//打印

            cout << m[j];

            cout << endl;

            count1++;

        }

    } while (a[i] != '#');

    cout << "页面缺页次数" << "  " << "页面置换次数" << "  "<<"缺页率"<< endl;

    cout << "     " << count1 << "           " << count1 - b << "           "<<1.0\*count1/(a.size()-1)<< endl;;

    /\*打印置换次数和缺页次数\*/

}

void LRU(int b, string a)

{

    char m[10] = { '#','#','#','#','#','#','#','#','#','#' };

    int i = 0, j, x, y = 0, count1 = 0, min = 10000;//min为最久未使用的初始值

    cout << "置换过程如下表" << endl;

    do {

        if (i<b)

        {

            m[y] = a[i];//缺页后将页面写入

            i++;

            count1++;//缺页标记++

            y = y + 1;//置换标记++

            cout << a[i - 1] << " ";

            for (j = 0;j<b;j++)//打印

                cout << m[j];

            cout << endl;

            continue;

        }

        for (j = 0;j<b;j++)//判断是否缺页

        {

            if (a[i] == m[j])

            {

                x = 1;

                break;

            }

        }

        if (x == 1)//未缺页才处理

        {

            i++;

            x = 0;

            cout << a[i - 1] << " " << endl;

            continue;

        }

        else//缺页处理

        {

            for (j = 0;j < b;j++)//查找最久未使用的页面

            {

                for (int t = i - 1;t >= 0;t--)

                {

                    if (m[j] == a[t])

                        break;

                if (t < min)

                    min = t;

                    y = j;

                }

            }

            m[y] = a[i];//置换最久未使用的页面

            min = 10000;//将min初始化

            i++;

            cout << a[i - 1] << " ";

            for (j = 0;j<b;j++)//打印

                cout << m[j];

            cout << endl;

            count1++;

        }

    } while (a[i] != '#');

    cout << "页面缺页次数" << "  " << "页面置换次数" << "  "<<"缺页率"<< endl;

    cout << "     " << count1 << "           " << count1 - b << "           "<<1.0\*count1/(a.size()-1)<< endl;;

}

//银行家算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class bank

{

private:

    int available[Max\_source];//可用资源数

    int max[Max\_process][Max\_source];//最大需求

    int allocation[Max\_process][Max\_source];//已分配资源数

    int need[Max\_process][Max\_source];//还需资源数

    int request[Max\_process][Max\_source];//进程需要资源数

    bool finish[Max\_process];//判断系统是否有足够的资源分配

    int p[Max\_process];//记录序列

    int m;//用来表示进程

    int n;//表示资源

public:

    void Init();//完成对变量的初始化

    bool Safe();//安全检测算法

    void Banker();//银行家算法

    void Display(int, int);//显示进程与资源状态

};

void bank::Init()

{

    cout << "请输入进程数：";

    cin >> m;

    cout << "请输入资源种类数：";

    cin >> n;

    cout << "请输入每个进程最多所需的各资源数" << endl;

    for (int i = 0; i < m; i++)

        for (int j = 0; j < n; j++)

            cin >> max[i][j];

    cout << "请输入每个进程已分配的各资源数"  << endl;

    for (int i = 0; i < m; i++)

        for (int j = 0; j < n; j++)

        {

            cin >> allocation[i][j];

            need[i][j] = max[i][j] - allocation[i][j];//注意这里的need可能小于0；要进行报错并重新输入，可以用continue来跳出当前循环

            if (need[i][j] < 0)

            {

                cout << "你输入的第" << i + 1 << "个进程的第" << j + 1 << "个资源数有问题！\n请重新输入！";

                j--;//忽略这个资源数

                continue;//跳出本次循环

            }

        }

    cout << "请输入各个资源现有的数目" << endl;

    for (int i = 0; i < n; i++)

    {

        cin >> available[i];

    }

}

//m表示进程，n表示资源

void bank::Display(int n, int m)

{

    cout << endl << "+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++" << endl;

    cout << "系统可用的资源数为： ";

    for (int i = 0; i < n; i++)

    {

        cout << available[i] << "   ";

    }

    cout << endl << "各进程还需要的资源量：" << endl;

    for (int i = 0; i < m; i++)

    {

        cout << "   进程" << i << ":";

        for (int j = 0; j < n; j++)

            cout << "       " << need[i][j];

        cout << endl;

    }

    cout << endl << "各进程已经得到的资源:        " << endl;

    for (int i = 0; i < m; i++)

    {

        cout << "   进程" << i << ":";

        for (int j = 0; j < n; j++)

        {

            cout << "       " << allocation[i][j];

        }

        cout << endl;

    }

    cout << endl << endl;

}

void bank::Banker()

{

    int i, cusneed, flag = 0;//cusneed表示资源进程号

    char again;//键盘录入一个字符用于判断是否继续请求资源

    while (1)

    {

        Display(n, m);

        cout << endl;

        /\*请求资源\*/

        while (true)

        {

            cout << "请输入要申请的进程号" << endl;

            cin >> cusneed;

            if (cusneed > m)

            {

                cout << "没有该进程，请重新输入" << endl;

                continue;

            }

            cout << "请输入进程所请求的各资源数" << endl;

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

                cin >> request[cusneed][i];

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

            {

                if (request[cusneed][i] > need[cusneed][i])

                {

                    cout << "你输入的资源请求数超过进程数需求量！请重新输入" << endl;

                    continue;

                }

                if (request[cusneed][i] > available[i])

                {

                    cout << "你输入的资源请求数超过系统当前资源拥有数！" << endl;

                    break;

                }

            }

            break;

        }

        /\*上述是资源请求不合理的情况，下面是资源请求合理时则执行银行家算法\*/

        for (int i = 0; i < n; i++)

        {

            available[i] -= request[cusneed][i];//可用资源减去成功申请的

            allocation[cusneed][i] += request[cusneed][i];//已分配资源加上成功申请的

            need[cusneed][i] -= request[cusneed][i];//进程还需要的减去成功申请的

        }

        /\*判断分配申请资源后系统是否安全，如果不安全则将申请过的资源还给系统\*/

        if (Safe())

            cout << "分配成功！";

        else

        {

            cout << "分配失败！" << endl;

            /\*进行向系统还回资源操作\*/

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

            {

                available[i] += request[cusneed][i];

                allocation[cusneed][i] -= request[cusneed][i];

                need[cusneed][i] += request[cusneed][i];

            }

        }

        /\*对进程的需求资源进行判断，是否还需要资源，简言之就是判断need数组是否为0\*/

        for (int i = 0; i < n; i++)

            if (need[cusneed][i] <= 0)

                flag++;

        if (flag == n)

        {

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

            {

                available[i] += allocation[cusneed][i];

                allocation[cusneed][i] = 0;

                need[cusneed][i] = 0;

            }

            cout << "进程" << cusneed << "占有的资源已释放！！" << endl;

            flag = 0;

        }

        for (int i = 0; i < m; i++)

            finish[i] = false;

        /\*判断是否继续申请\*/

        cout << "继续分配: y   停止分配：n" << endl;

        cin >> again;

        if (again == 'y' )

            continue;

        break;

    }

}

bool bank::Safe()

{

    int l = 0, j, i;

    int work[Max\_source];

    /\*对work数组进行初始化，初始化时和avilable数组相同\*/

    for (int i = 0; i < n; i++)

        work[i] = available[i];

    /\*对finish数组初始化全为false\*/

    for (int i = 0; i < m; i++)

        finish[i] = false;

    for (int i = 0; i < m; i++)

    {

        if (finish[i] == true)

            continue;

        else

        {

            for (j = 0; j < n; j++)

            {

                if (need[i][j] > work[j])

                    break;

            }

            if (j == n)

            {

                finish[i] = true;

                for (int k = 0; k < n; k++)

                    work[k] += allocation[i][k];

                p[l++] = i;//记录进程号

                i = -1;

            }

            else

                continue;

        }

    }

    if (l == m)

    {

        cout << "系统是安全的" << endl;

        cout << "安全序列：" << endl;

        for (int i = 0; i < l; i++)

        {

            cout << p[i];

            if (i != l - 1)

                cout << "-->";

        }

        cout << endl << endl;

        return true;

    }

    return false;

}

//磁盘调度算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class DiskScheduling {

public:

    DiskScheduling() :DiskScheduling(10, 0, 200) {}

    DiskScheduling(int diskNum, int startPos, int maxPos) {

        uniform\_int\_distribution<unsigned> u(0, maxPos);

        default\_random\_engine e(10);

        while (diskNum--) {

            diskInputSeq.push\_back(u(e));

        }

        diskStartPos = startPos;

        diskMaxPos = maxPos;

    }

    DiskScheduling(vector<int>& diskInput) {

        diskMaxPos = \*max\_element(diskInput.begin(), diskInput.end());

        diskStartPos = 0;

        diskInputSeq = diskInput;

    }

    DiskScheduling(vector<int>& diskInput, int startPos, int maxPos) {

        diskMaxPos = maxPos;

        diskStartPos = startPos;

        diskInputSeq = diskInput;

    }

    void FCFS() {

        diskOutputSeq = diskInputSeq;

        for (auto i = diskInputSeq.begin(); i != diskInputSeq.end() - 1; ++i) {

            sumStep += abs(\*i - \*(i + 1));

        }

        sumStep += abs(diskStartPos - diskInputSeq[0]);

        cout << "FCFS算法\n";

        Print();

        Reset();

    }

    void SSTF() {

        vector<int> diskTemp(diskInputSeq);

        int diskStartTemp = diskStartPos;

        int count = diskTemp.size();

        while (count--) {

            auto minNow = min\_element(diskTemp.begin(), diskTemp.end(), [diskStartTemp](const int& first, const int& second) {

                return abs(diskStartTemp - first) < abs(diskStartTemp - second);

            });

            diskOutputSeq.push\_back(\*minNow);

            sumStep += abs(diskStartTemp - \*minNow);

            diskStartTemp = \*minNow;

            \*minNow = INT\_MAX;

        }

        cout << "SSTF算法\n";

        Print();

        Reset();

    }

    // dir = true 移动方向为磁盘号小到大的方向

    void SCAN(bool dir = true) {

        ScanCscanHelper(dir, 0);

        cout << "SCAN算法\n";

        Print();

        Reset();

    }

    void CSCAN(bool dir = true) {

        ScanCscanHelper(dir, 1);

        cout << "CSCAN算法\n";

        Print();

        Reset();

    }

    void NstepSCAN(int nGroup, bool dir = true) {

        int diskStartTemp = diskStartPos;

        vector<int> diskTemp;

        int groupSize = (int)ceil(diskInputSeq.size() / (double)nGroup);

        for (auto i = diskInputSeq.begin(); i != diskInputSeq.end(); i += groupSize) {

            if (diskInputSeq.end() - i < groupSize)

                diskTemp.assign(i, diskInputSeq.end()), i = diskInputSeq.end() - groupSize;

            else

                diskTemp.assign(i, i + groupSize);

            DiskScheduling diskT(diskTemp, diskStartTemp, diskMaxPos);

            diskT.ScanCscanHelper(dir, 0);

            sumStep += diskT.getSumStep();

            vector<int> diskOutputGroup = diskT.getDiskOutputSeq();

            diskStartTemp = \*diskOutputGroup.rbegin();

            diskOutputSeq.insert(diskOutputSeq.end(), diskOutputGroup.begin(), diskOutputGroup.end());

        }

        cout << "NstepSCAN\n";

        Print();

        Reset();

    }

    void ScanCscanHelper(int dir, int opt) {

        vector<int> diskTemp(diskInputSeq);

        sort(diskTemp.begin(), diskTemp.end(), [dir](const int& first, const int& second) {

return dir ? first > second: first < second;

});

        auto i = diskTemp.begin();

        for (; i != diskTemp.end() - 1; i++) {

            if ((\*i - diskStartPos) \* (\*(i + 1) - diskStartPos) <= 0) {

                if ((\*(i + 1) - diskStartPos) == 0) i++;

                break;

            }

        }

        reverse(diskTemp.begin(), i + 1);

        if (opt == 1) reverse(i + 1, diskTemp.end());

        diskOutputSeq = diskTemp;

        for (auto j = diskOutputSeq.begin(); j != diskOutputSeq.end() - 1; ++j) {

            sumStep += abs(\*j - \*(j + 1));

        }

        sumStep += abs(diskStartPos - diskOutputSeq[0]);

    }

    int getSumStep() { return sumStep; }

    vector<int> getDiskOutputSeq() { return diskOutputSeq; }

private:

    void Print() {

        cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

        cout << "起始磁道号：" << diskStartPos << endl;

        cout << "最大磁道号：" << diskMaxPos << endl;

        cout << "待访问顺序\t\n";

        for (int diskNow : diskInputSeq)

            cout << diskNow << " ";

        cout << "\n实际访问顺序\t\n";

        for (int diskActual : diskOutputSeq)

            cout << diskActual << " ";

        cout << "\n移动的磁道总数\t" << sumStep << endl;

        cout << "平均寻道数\t" << sumStep / diskOutputSeq.size() << "\n";

        cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

    }

    void Reset() {

        sumStep = 0;

        diskOutputSeq.clear();

    }

private:

    int diskStartPos;           // 起始的磁道号 or 磁盘机械臂当前位置

    int diskMaxPos;             // 最大的磁道号

    double sumStep = 0;         // 访问的总步数

    vector<int> diskInputSeq;   // 磁道待访问顺序

    vector<int> diskOutputSeq;  // 磁道实际的访问顺序

};

//主程序测试\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int main()

{

    int NumberChoice;

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

    cout << "1：生产者消费者问题模拟" << endl;

    cout << "2：抢占式高优先级算法模拟" << endl;

    cout << "3：FIFO算法和LRU算法模拟" << endl;

    cout << "4：银行家算法模拟" << endl;

    cout << "5：5种磁盘调度算法模拟" << endl;

    cout << "\*\*\*\*\*\*\*\*\*\*请选择某一功能进行模拟\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

    cin >> NumberChoice;

    cout<<  "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

    if (NumberChoice == 1) {

        //创建Mutex和Semaphore

        pv\_hMutex = CreateMutex(NULL, FALSE, NULL);

        fullSemaphore = CreateSemaphore(NULL, 0, SIZE\_OF\_BUFFER, NULL);

        emptySemaphore = CreateSemaphore(NULL, SIZE\_OF\_BUFFER,

            SIZE\_OF\_BUFFER, NULL);

        //缓冲区初始化

        for (int i = 0; i< SIZE\_OF\_BUFFER;++i) {

            pv\_buffer[i] = -1;   //当值为-1时该项为空

        }

        const unsigned short PRODUCERS\_COUNT = 3;       //生产者的个数

        const unsigned short CONSUMERS\_COUNT = 2;       //消费者的个数                        //总的线程数

        const unsigned short THREADS\_COUNT = PRODUCERS\_COUNT + CONSUMERS\_COUNT;

        HANDLE pvThreads[THREADS\_COUNT];     //各线程的handle

        DWORD producerID[PRODUCERS\_COUNT];  //生产者线程的标识符

        DWORD consumerID[CONSUMERS\_COUNT];  //消费者线程的标识符

        cout<<"##############################################"<<endl;

        cout<<"开始，缓冲区大小为10 "<<endl;

        cout<<"##############################################"<<endl;

        for (int i = 0;i<PRODUCERS\_COUNT;i++) {

            pvThreads[i] = CreateThread(NULL, 0, Producer, NULL, 0, &producerID[i]);

            if (pvThreads[i] == NULL) break;

        }

        //创建消费者线程

        for (int i = 0;i<CONSUMERS\_COUNT;i++) {

            pvThreads[PRODUCERS\_COUNT + i]

                = CreateThread(NULL, 0, Consumer, NULL, 0, &consumerID[i]);

            if (pvThreads[i] == NULL) break;

        }

        while (pv\_continue) {

            if (getchar()) {  //按回车后终止程序运行

                pv\_continue = false;

            }

        }

        system("pause");

        return 0;

    }

    else if (NumberChoice == 2)

    {

        priority\_queue<PCB> p;

        int n;

        cout << "请输入进程的个数：";

        cin >> n;

        createProcess(p, n);

        runProcess(p);

        cout << "所有进程执行完毕" << endl;

        cout << "平均周转时间为" << fixed << setprecision(2) << sumtime / n << endl;

        cout << "平均带权周转时间为" <<fixed<< setprecision(2)<<sumrighttime / n << endl;

        getchar();

        getchar();

        return 0;

    }

    else if (NumberChoice == 3)

    {

        string a;

        int b;

        cout << "请输入物理块数:";

        cin >> b;

        cout << "请输入页号序列:（#结尾）";

        cin >> a;

        cout << "##############################################" << endl;

        cout << "FIFO算法如下:" << endl;

        FIFO(b, a);

        cout << "##############################################" << endl;

        cout << "LRU算法如下:" << endl;

        LRU(b, a);

        cout << "##############################################" << endl;

        return 0;

    }

    else if (NumberChoice == 4)

    {

        bank peter;

        peter.Init();

        peter.Safe();

        peter.Banker();

        return 0;

    }

    else if (NumberChoice == 5)

    {

        DiskScheduling disk1(20,100, 200);

        int choice;

        cout << "请选择要使用的算法" << endl;

        cout << "0：结束" << endl;

        cout << "1：FCFS" << endl;

        cout << "2：SSTF" << endl;

        cout << "3：SCAN" << endl;

        cout << "4：CSCAN"<<endl;

        cout << "5：NsteptSCAN" << endl;

    while(cin >> choice){

        if (choice == 0)  {cout << "结束"; break;};//结束

        if (choice == 1)  disk1.FCFS();//调用FCFS算法

        if (choice == 2)  disk1.SSTF();//调用SSTF算法

        if (choice == 3)  disk1.SCAN(false);//调用SCAN算法

        if (choice == 4)  disk1.CSCAN(false);//调用CSCAN算法

        if (choice == 5)  disk1.NstepSCAN(3);//调用NstepSCAN算法

        cout << "请选择要使用的算法" << endl;

        cout << "0：结束" << endl;

        cout << "1：FCFS" << endl;

        cout << "2：SSTF" << endl;

        cout << "3：SCAN" << endl;

        cout << "4：CSCAN"<<endl;

        cout << "5：NsteptSCAN" << endl;

    }

        return 0;

    }

    else{

        cout << "输入有误，请重新输入" << endl;

    }

}

【思考题】

在实训过程中，你是如何解决遇到的问题的？实训成果有哪些有待完善的地方？应如何完善？

**七、实验总结**

这一次的实训内容牵涉到了进程管理、文件管理、设备管理和存储管理还有磁盘管理五个大的功能模块，每当我要完成一个功能模块，都要相应的熟悉了解知识点，让我无论是在操作系统方面，还是编程语言方面都有所进步，我发现我的c++语言水平相比我的其他编程语言水平有一定的优势，但是和其他同学比起来还是有一定差距，不能做到应用自如，其他语言有的时候甚至连基础的语法都会出问题。

通过本次操作系统的程序设计使我认识到要将操作系统这门计算机专业的课学好不仅仅是要把书上的基本知识学好而且还要不断进行实践，将所学的跟实践操作结合起来才能更好地巩固所学，才能提高自己实践能力.通过这次的设计使我认识到只停留在表面理解问题是很难使问题得到很好的解决的，实践能力与理论知识同样重要。可以说此程序设计的理论难度并不大，但是若要深入发掘其中的东西，并且实际去编程实现，就遇到了相当大的难度。因为与之涉及的很多方面并没有学过，需要自己去自学和实践检验。

我非常高兴有这样的一次实训机会，同时也感谢老师为我们准备的实训题。这次实训让我学到了很多在书本上学不到的东西，同时也让我了解到了自己还有很多的缺点和不足。在最后我也要感谢干冲同学，在我遇到困难的时候为我提供了无私的帮助，在他的帮助下，本次实验课设我才能够顺利的完成。