



**实验报告**

**学 院 计算机学院\_\_\_\_\_**

**专 业 软件工程\_\_\_\_\_\_**

**年级班别 16级（3）班\_\_\_\_\_**

**学 号 3116004823\_\_\_\_\_\_**

**学生姓名 林锦鹏\_\_\_\_\_\_\_\_**

**指导教师 丁国芳\_\_\_\_\_\_\_\_**

**2018年12月**

目录

[实验指导书](#_Toc15441_WPSOffice_Level1) [6](#_Toc15441_WPSOffice_Level1)

[—. 课程的性质、目的和任务](#_Toc8324_WPSOffice_Level1) [6](#_Toc8324_WPSOffice_Level1)

[二. 实验的意义和目的](#_Toc3857_WPSOffice_Level1) [6](#_Toc3857_WPSOffice_Level1)

[三.实验运行环境及上机前的准备](#_Toc13401_WPSOffice_Level1) [6](#_Toc13401_WPSOffice_Level1)

[四. 实验内容及安排](#_Toc2661_WPSOffice_Level1) [6](#_Toc2661_WPSOffice_Level1)

[五. 报告提交](#_Toc11396_WPSOffice_Level1) [7](#_Toc11396_WPSOffice_Level1)

[实验一 进程调度](#_Toc14226_WPSOffice_Level1) [8](#_Toc14226_WPSOffice_Level1)

[1.实验内容](#_Toc8324_WPSOffice_Level2) [8](#_Toc8324_WPSOffice_Level2)

[2.实验要求](#_Toc3857_WPSOffice_Level2) [8](#_Toc3857_WPSOffice_Level2)

[3.算法原理相关知识](#_Toc13401_WPSOffice_Level2) [8](#_Toc13401_WPSOffice_Level2)

[（1）短进程优先调度算法](#_Toc8324_WPSOffice_Level3) [8](#_Toc8324_WPSOffice_Level3)

[（2）时间片轮转算法](#_Toc3857_WPSOffice_Level3) [8](#_Toc3857_WPSOffice_Level3)

[（3）高响应比优先算法](#_Toc13401_WPSOffice_Level3) [8](#_Toc13401_WPSOffice_Level3)

[4.实验实现](#_Toc2661_WPSOffice_Level2) [9](#_Toc2661_WPSOffice_Level2)

[4.1短进程优先调度算法](#_Toc2661_WPSOffice_Level3) [9](#_Toc2661_WPSOffice_Level3)

[4.2时间片轮转算法](#_Toc11396_WPSOffice_Level3) [11](#_Toc11396_WPSOffice_Level3)

[4.3高响应比优先算法](#_Toc14226_WPSOffice_Level3) [14](#_Toc14226_WPSOffice_Level3)

[5.测试结果](#_Toc11396_WPSOffice_Level2) [17](#_Toc11396_WPSOffice_Level2)

[5.1短进程优先调度算法](#_Toc26519_WPSOffice_Level3) [17](#_Toc26519_WPSOffice_Level3)

[5.2时间片轮转算法](#_Toc13223_WPSOffice_Level3) [21](#_Toc13223_WPSOffice_Level3)

[5.3高响应比优先算法](#_Toc3081_WPSOffice_Level3) [26](#_Toc3081_WPSOffice_Level3)

[6.总结](#_Toc14226_WPSOffice_Level2) [32](#_Toc14226_WPSOffice_Level2)

[实验二 银行家算法](#_Toc26519_WPSOffice_Level1) [33](#_Toc26519_WPSOffice_Level1)

[1.实验内容](#_Toc26519_WPSOffice_Level2) [33](#_Toc26519_WPSOffice_Level2)

[2.实验要求](#_Toc13223_WPSOffice_Level2) [33](#_Toc13223_WPSOffice_Level2)

[3.算法原理相关知识](#_Toc3081_WPSOffice_Level2) [33](#_Toc3081_WPSOffice_Level2)

[4.实验实现](#_Toc6752_WPSOffice_Level2) [34](#_Toc6752_WPSOffice_Level2)

[4.1流程图](#_Toc6752_WPSOffice_Level3) [34](#_Toc6752_WPSOffice_Level3)

[4.2重要的数据结构](#_Toc345_WPSOffice_Level3) [36](#_Toc345_WPSOffice_Level3)

[4.3重要的算法](#_Toc19746_WPSOffice_Level3) [36](#_Toc19746_WPSOffice_Level3)

[5.测试结果](#_Toc345_WPSOffice_Level2) [43](#_Toc345_WPSOffice_Level2)

[初始申请状态](#_Toc918_WPSOffice_Level3) [43](#_Toc918_WPSOffice_Level3)

[注意，此时三个进程对A资源的请求数超过 可分配的，系统进入不安全状态](#_Toc32515_WPSOffice_Level3) [49](#_Toc32515_WPSOffice_Level3)

[接下来重开结束程序并重开程序列举【其他情况】](#_Toc11218_WPSOffice_Level3) [50](#_Toc11218_WPSOffice_Level3)

[P8 回到就绪队列，等待时机再申请资源运行](#_Toc5583_WPSOffice_Level3) [50](#_Toc5583_WPSOffice_Level3)

[（注意此时 p8 又分配到了其他线程归还资源）](#_Toc23949_WPSOffice_Level3) [52](#_Toc23949_WPSOffice_Level3)

[6.总结](#_Toc19746_WPSOffice_Level2) [54](#_Toc19746_WPSOffice_Level2)

[实验三 动态分区分配方式的模拟](#_Toc13223_WPSOffice_Level1) [55](#_Toc13223_WPSOffice_Level1)

[1.实验内容](#_Toc918_WPSOffice_Level2) [55](#_Toc918_WPSOffice_Level2)

[2.实验要求](#_Toc32515_WPSOffice_Level2) [55](#_Toc32515_WPSOffice_Level2)

[3.算法原理相关知识](#_Toc11218_WPSOffice_Level2) [55](#_Toc11218_WPSOffice_Level2)

[1.首次适应算法（First-fit）](#_Toc5583_WPSOffice_Level2) [55](#_Toc5583_WPSOffice_Level2)

[2.最佳适应算法（Best-fit）](#_Toc23949_WPSOffice_Level2) [55](#_Toc23949_WPSOffice_Level2)

[3.内存回收](#_Toc12387_WPSOffice_Level2) [55](#_Toc12387_WPSOffice_Level2)

[4.实验实现](#_Toc18462_WPSOffice_Level2) [56](#_Toc18462_WPSOffice_Level2)

[4.1重要的数据结构](#_Toc12387_WPSOffice_Level3) [56](#_Toc12387_WPSOffice_Level3)

[4.2首次适应算法](#_Toc18462_WPSOffice_Level3) [57](#_Toc18462_WPSOffice_Level3)

[4.3最佳适应算法](#_Toc25829_WPSOffice_Level3) [58](#_Toc25829_WPSOffice_Level3)

[4.4内存回收](#_Toc25752_WPSOffice_Level3) [60](#_Toc25752_WPSOffice_Level3)

[5.测试结果](#_Toc25829_WPSOffice_Level2) [61](#_Toc25829_WPSOffice_Level2)

[5.1首次适应算法](#_Toc7574_WPSOffice_Level3) [62](#_Toc7574_WPSOffice_Level3)

[5.1最佳适应算法](#_Toc23499_WPSOffice_Level3) [68](#_Toc23499_WPSOffice_Level3)

[6.总结](#_Toc25752_WPSOffice_Level2) [74](#_Toc25752_WPSOffice_Level2)

[实验四 仿真各种磁盘调度算法](#_Toc3081_WPSOffice_Level1) [75](#_Toc3081_WPSOffice_Level1)

[1.实验内容](#_Toc7574_WPSOffice_Level2) [75](#_Toc7574_WPSOffice_Level2)

[2.实验要求](#_Toc23499_WPSOffice_Level2) [75](#_Toc23499_WPSOffice_Level2)

[3.算法原理相关知识](#_Toc29008_WPSOffice_Level2) [75](#_Toc29008_WPSOffice_Level2)

[1、先来先服务算法（FCFS）](#_Toc29008_WPSOffice_Level3) [75](#_Toc29008_WPSOffice_Level3)

[2、最短寻道时间优先算法（SSTF）](#_Toc23880_WPSOffice_Level3) [75](#_Toc23880_WPSOffice_Level3)

[3、扫描算法（SCAN）](#_Toc31991_WPSOffice_Level3) [75](#_Toc31991_WPSOffice_Level3)

[4、循环扫描算法（CSCAN）](#_Toc30615_WPSOffice_Level3) [75](#_Toc30615_WPSOffice_Level3)

[4.实验实现](#_Toc23880_WPSOffice_Level2) [76](#_Toc23880_WPSOffice_Level2)

[4.1先来先服务算法](#_Toc11615_WPSOffice_Level3) [76](#_Toc11615_WPSOffice_Level3)

[4.2最短寻道时间优先算法](#_Toc17148_WPSOffice_Level3) [78](#_Toc17148_WPSOffice_Level3)

[4.3扫描算法](#_Toc9592_WPSOffice_Level3) [81](#_Toc9592_WPSOffice_Level3)

[4.4循环扫描算法](#_Toc31611_WPSOffice_Level3) [84](#_Toc31611_WPSOffice_Level3)

[5.测试结果](#_Toc31991_WPSOffice_Level2) [86](#_Toc31991_WPSOffice_Level2)

[6.总结](#_Toc30615_WPSOffice_Level2) [87](#_Toc30615_WPSOffice_Level2)

# 实验指导书

## —. 课程的性质、目的和任务

操作系统是计算机系统配置的基本软件之一。它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

　　 操作系统是计算机教学中最重要的环节之一，也是计算机专业学生的一门重要的专业课程。操作系统质量的好坏，直接影响整个计算机系统的性能和用户对计算机的使用。一个精心设计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；  
  准备好需要输入的测试数据；
* 对实验结果进行分析。

## 四. 实验内容及安排

|  |  |  |
| --- | --- | --- |
| **实验一** | **进程调度** |  |
| **实验二** | **银行家算法** |  |
| **实验三** | **内存管理** |  |
| **实验四** | **磁盘调度** |  |

## 五. 报告提交

1. **提交时间：第17周**
2. **打印实验报告**
3. **以班为单位刻一张光碟，每个同学一个目录（目录名：学号+姓名），内容包括报告及程序文件**

# 实验一 进程调度

## 1.实验内容

编写并调试一个模拟的进程调度程序，分别采用“短进程优先”、“时间片轮转”、“高响应比优先”调度算法对随机产生的五个进程进行调度，并比较算法的平均周转时间。以加深对进程的概念及进程调度算法的理解。

## 2.实验要求

1. 每个进程由一个进程控制块（ PCB）表示，进程控制块可以包含如下信息：进程名、优先数（响应比）、到达时间、需要运行时间（进程的长度）、已运行时间、进程状态等等（可以根据需要自己设定）。
2. 由程序自动生成进程（包括需要的数据，要注意数据的合理范围），第一个进程到达时间从0开始，其余进程到达时间随机产生。
3. 采用时间片轮转调度算法时，进程的运行时间以时间片为单位进行计算。
4. 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。
5. 每进行一次调度，程序都要输出一次运行结果：正在运行的进程、就绪队列中的进程、完成的进程以及各个进程的 PCB，以便进行检查。
6. 最后计算各调度算法的平均周转时间，并进行比较、分析。

## 3.算法原理相关知识

（1）短进程优先调度算法

“短进程优先”调度算法的基本思想是把CPU分配给就绪队列中需要时间最短的进程。

（2）时间片轮转算法

将系统中所有的就绪进程按照FCFS原则，排成一个队列。每次调度时将CPU分派给队首进程，让其执行一个时间片。时间片的长度从几个ms到几百ms。在一个时间片结束时，发生时钟中断。调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程。进程可以未使用完一个时间片，就出让CPU。

（3）高响应比优先算法

HRRN调度策略同时考虑每个作业的等待时间长短和估计需要的执行时间长短，从中选出响应比最高的作业投入执行。

每个作业完成后要打印该作业的开始运行时刻、完成时刻、周转时间和带权周转时间，这一组作业完成后要计算并打印这组作业的平均周转时间、带权平均周转时间。

## 4.实验实现

### 4.1短进程优先调度算法

#### 4.1.1流程图



#### 4.1.2重要的数据结构

// 进程控制块PCB

typedef struct pcb

{

char name[12];//进程名

int arrivalTime;//到达时间

int startTime;//开始运行时间

int finishTime;//结束运行时间

int wholeTime;//需要运行的时间（进程的长度）

struct pcb \*link;//下一个指针

}pcb;

int num;//进程数量

void input();

pcb \*ready=NULL,\*p=NULL,\*finish = NULL;//ready是未完成的进程的队列，finish是已经结束完成的进程的队列

#### 4.1.3重要的算法

/\* 短进程先响应sjf算法 \*/

//同样响应时间下，队列靠前的肯定是先到达的

//nowTime 是现在执行的时间

pcb\* sjf(int nowTime,int \*after)

{

int i = 0 ;

pcb \*nowProgress=NULL, \*p = ready;

int ProgressNum = 0; // 当前最短的是第几个线程

int minTime =0; // 最短运行时间

if(NULL != ready)

{

while(NULL != p) //遍历整个链表，查找出最短的进程，即运行时间最短

{

if(nowTime >= p->arrivalTime)

{

if(0 == minTime) { //首次赋值

nowProgress = p;

minTime = p->wholeTime;

} else {

if(p->wholeTime < minTime) { //同响应时间取最前（即先到达）

nowProgress = p;

minTime = p->wholeTime;

}

}

\*after = minTime+nowTime;

}

p = p->link;

}

}

return nowProgress;

}

### 4.2时间片轮转算法

#### 4.2.1流程图



#### 4.2.2重要的数据结构

#define TIME 2//时间片长度

int now\_time=0;//时间

int num;//进程数量

//进程控制块pcb

typedef struct pcb{

char name[12];//进程名

int queue;//进程所在的队列

int eTime;//进程在本队列可运行的时间片

int wholeTime;//需要运行的时间

int alreadyTime;//已运行的时间

int finishTime;//结束运行时间

int startTime;//开始时间，这里等同于到达时间

char state;//进程的状态,就绪 W（Wait）、运行R（Run）、或完成F（Finish）

struct pcb \*link;//下一个指针

}pcb;

pcb \*ready = NULL, \*pinsert = NULL, \*p =NULL,\*pFinish,\*pFinish1,\*finish;//pfend是逻一个辑队列的结尾，pinsert是就绪队列的下一个插入位置。

#### 4.2.3重要的算法

void insert()//插入进程

{

if(!ready ){//第一个，就绪

ready = p;

pinsert = p;

} else if (ready->queue == 1) {//第一队列存在

p->link = pinsert->link;

pinsert->link = p;

pinsert = p;

}

}

//第一种情况：正在进行的进程已经完成，立刻激活调度程序

void destroy()/\*建立进程撤销函数(进程findpos运行结束，撤销进程)\*/

{

ready -> state='F';//进程完成标志

ready->finishTime=now\_time+1;//下一个时刻完成，对应“eTime--”语句

printf("\n进程%s已完成!\n",ready->name);

//把已经完成的进程从就绪队列放到已完成队列中

p = ready;

ready = ready->link;

p->link=NULL;

//找到完成队列默认

pFinish=finish;

while(pFinish!=NULL){

pFinish1=pFinish;

pFinish=pFinish->link;

}

pFinish1->link=p;

}

void running()/\*建立进程就绪函数(进程运行时间到，置就绪状态)\*/

{

//更改运行线程的运行时间

(ready -> alreadyTime)++;

//如果没开始，记录开始时间

if(ready -> alreadyTime==1)

ready->startTime=now\_time;

//可用时间片时间减一,状态:W,进程所在逻辑队列:1,需要运行时间

ready ->eTime --;

//进程切换时机

if(ready->alreadyTime == ready->wholeTime){ //第一种情况：正在进行的进程已经完成，立刻激活调度程序

destroy();

return;

}else if(ready ->eTime == 0){ //第二种情况：时间片用完，计时器中断处理程序被激活

(ready -> queue)++;//放入下一逻辑队列

ready->eTime = TIME;//重新给予时间片

ready -> state='W';

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

if(!ready->link ||ready->queue < ready->link->queue) return;

p = ready->link;

ready->link = pinsert->link;

pinsert->link = ready;

pinsert = ready;

ready = p;

}

}

### 4.3高响应比优先算法

#### 4.3.1流程图



#### 4.3.2重要的数据结构

typedef struct pcb{

char name[12];//进程名

int arrivelTime;//到达时间

int wholeTime;//需要运行时间

int startTime;//开始时间

int finishTime;//完成时间

int turnoverTime;//周转时间

float useWeightTurnoverTime;//带权周转时间

char processStatus[10];//进程状态

};

static int currentTime = 0;//当前时间

static int finishNumber = 0;//进程完成数量

char pcbArray[PCBNUMBER][10];//存放数组名信息的二元数组

float priority[PCBNUMBER];//存放进程优先级的一元数组

int num;//进程数量

#### 4.3.3重要的算法

//比较各个进程之间的到达时间,按升序排列

void compare(struct pcb\* pcb){

int i,j,min,minIndex;

for(i = 0; i < num - 1; i++){

min = pcb[i].arrivelTime;

minIndex = i;

for(j = i + 1; j < num; j++){

if(pcb[j].arrivelTime < min){

min = pcb[j].arrivelTime;

minIndex = j;

}

}

struct pcb temp = pcb[i];

pcb[i] = pcb[minIndex];

pcb[minIndex] = temp;

}

}

//算法循环遍历部分

void loop(struct pcb\* pcb, int i){

pcb[i].startTime = currentTime;

pcb[i].finishTime = pcb[i].startTime + pcb[i].wholeTime;

pcb[i].turnoverTime = pcb[i].finishTime - pcb[i].arrivelTime;

pcb[i].useWeightTurnoverTime = pcb[i].turnoverTime \* 1.0 / pcb[i].wholeTime;

strcpy(pcb[i].processStatus, RUN);

while(1){

if(currentTime == pcb[i].finishTime){

strcpy(pcb[i].processStatus, FINISH);

finishNumber++;

if(finishNumber == num)

printJob(pcb);

currentTime--;

break;

}

else{

printJob(pcb);

currentTime++;

}

}

}

//高响应比优先调度算法

void highestResponseRatioNext(struct pcb\* pcb){

//创建线程组

createpcb(pcb);

//按到达时间排序

compare(pcb);

int i = 0, j = 0;

for(; finishNumber != num; currentTime++){

float maxPriority = 0.0;

int indexPriority = 0;

if(currentTime < pcb[0].arrivelTime)//当前时间小于第一个节点到来时间时，直接打印

printJob(pcb);

else{

int i;

for(i = 0; i < num; i++){

if(strcmp(pcb[i].processStatus, FINISH) != 0){

int waitTime = currentTime - pcb[i].arrivelTime;

priority[i] = (waitTime + pcb[i].wholeTime) \* 1.0 / pcb[i].wholeTime;

if(priority[i] > maxPriority){

maxPriority = priority[i];

indexPriority = i;

}

}

}

strcpy(pcbArray[j++], pcb[indexPriority].name);

loop(pcb, indexPriority);

}

}

}

//计算平均带权周转时间

float weightTurnoverTimeCount(struct pcb\* pcb){

float sum = 0.0;

int i;

for(i = 0; i < num; i++)

sum += pcb[i].useWeightTurnoverTime;

return sum / num;

}

//计算平均周转时间

float turnOverTimeCount(struct pcb\* pcb){

float sum = 0.0;

int i;

for(i = 0; i < num; i++)

sum += pcb[i].turnoverTime;

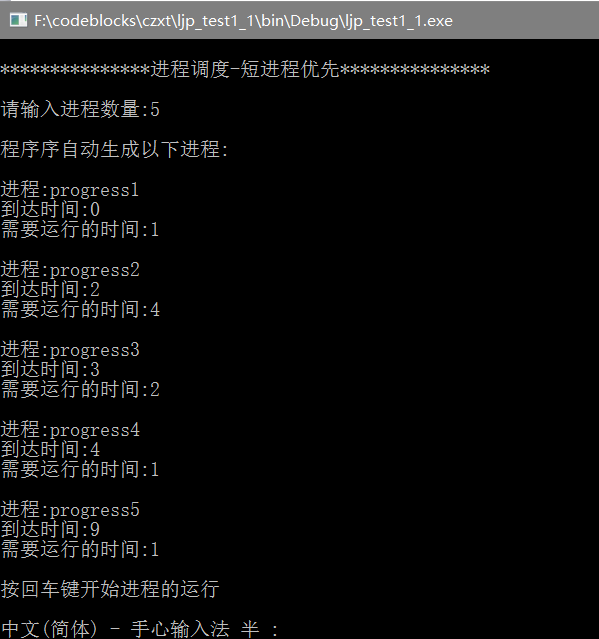
return sum / num;

}

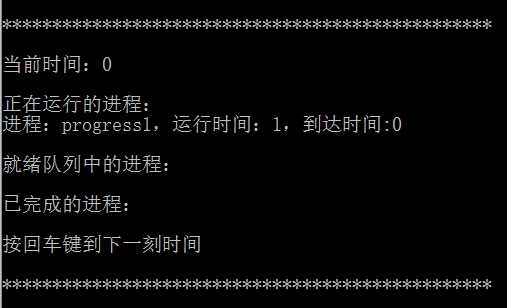
## 5.测试结果

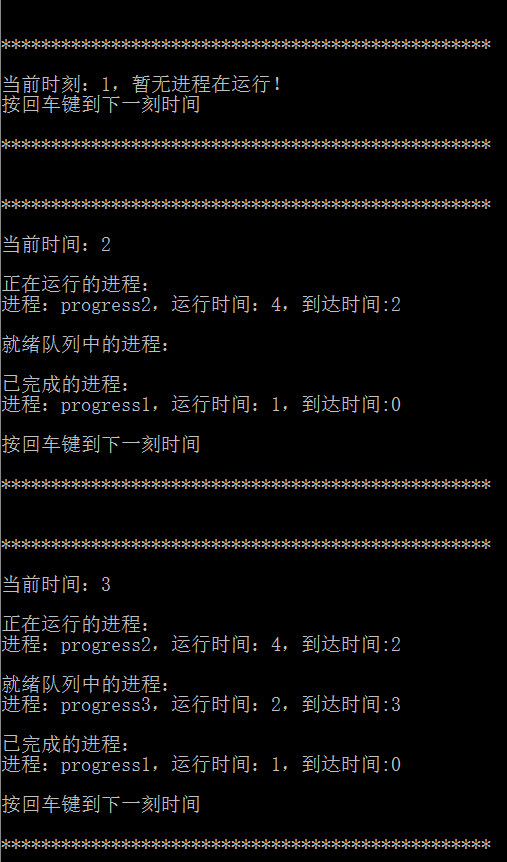
### 5.1短进程优先调度算法

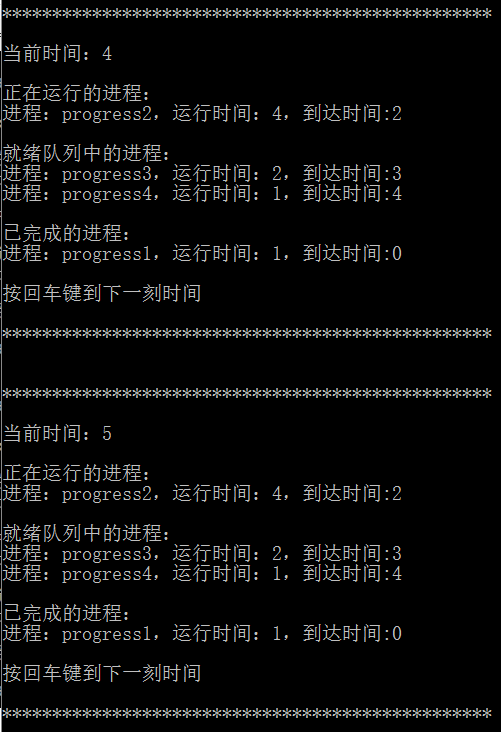
程序随机自动生成进程

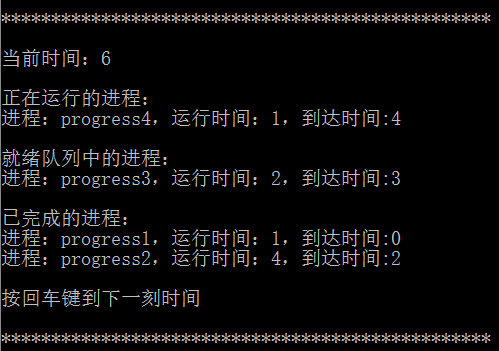


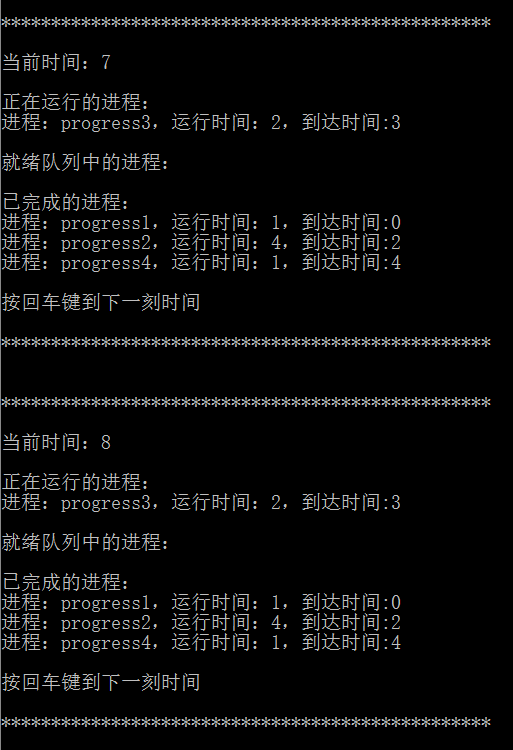
调度进程的情况

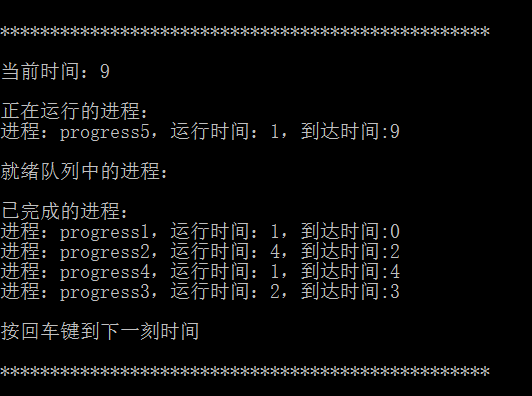


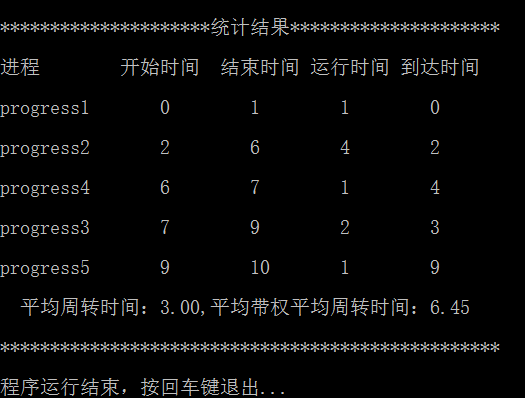






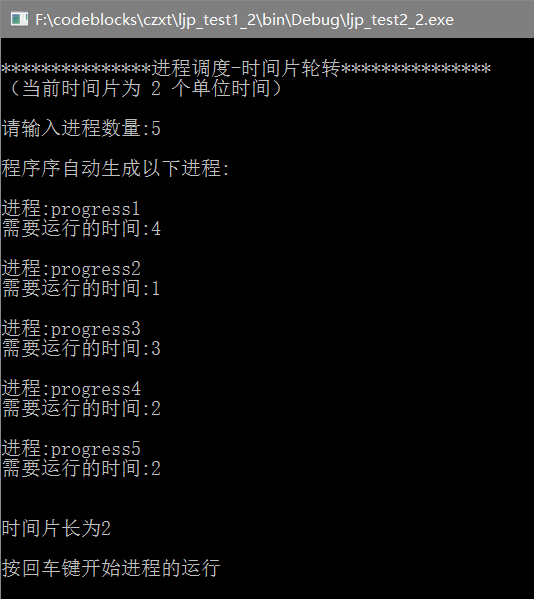




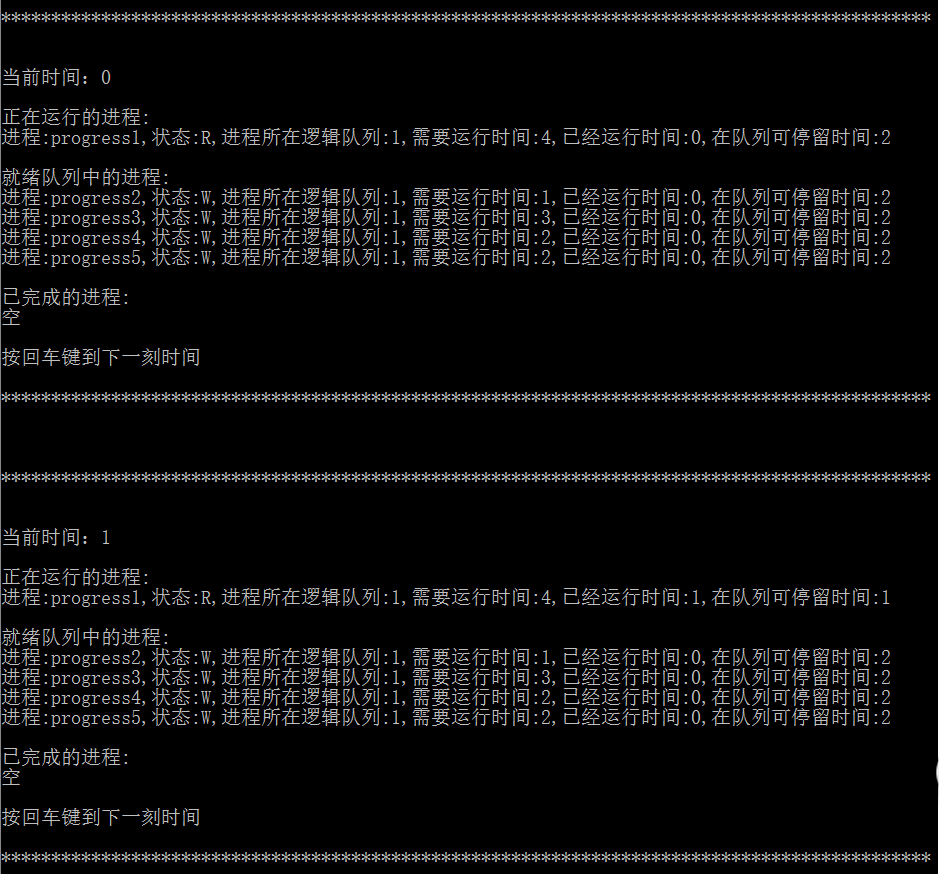


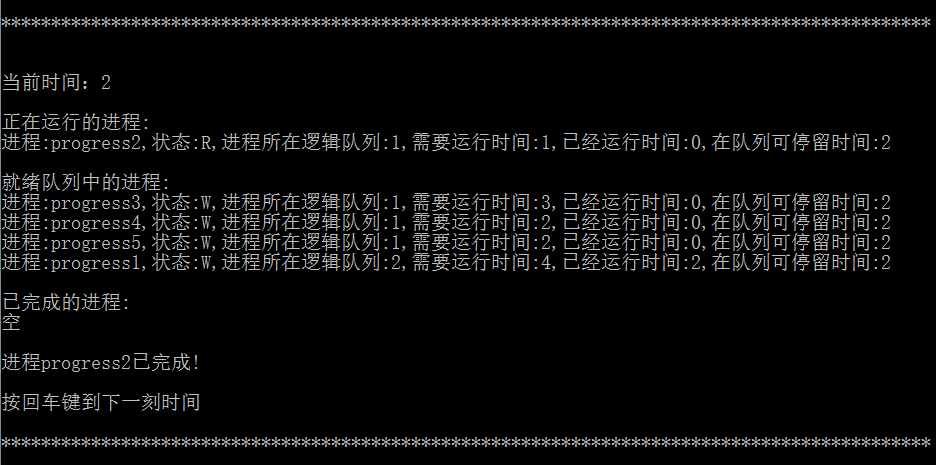
### 5.2时间片轮转算法

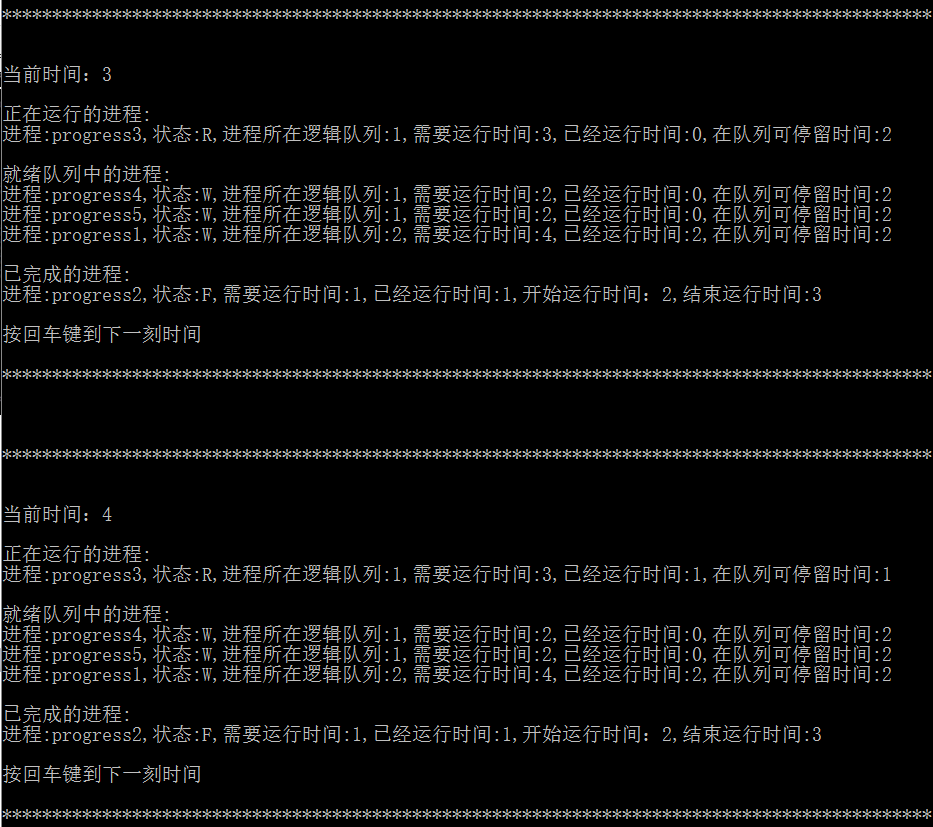
程序随机自动生成进程

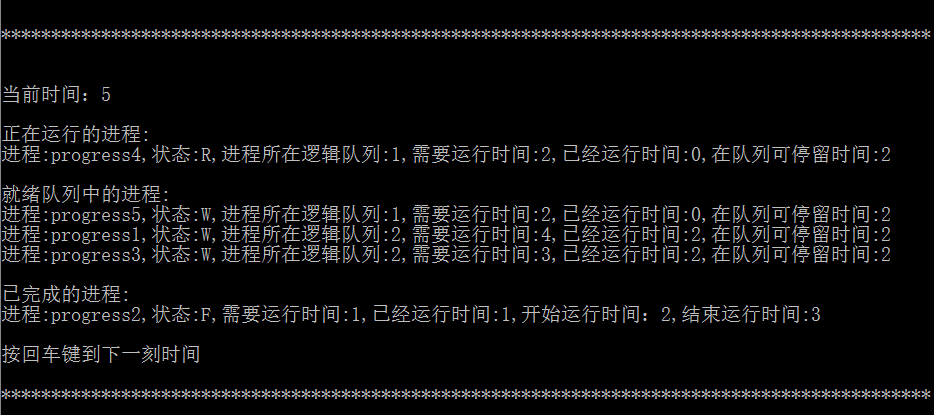


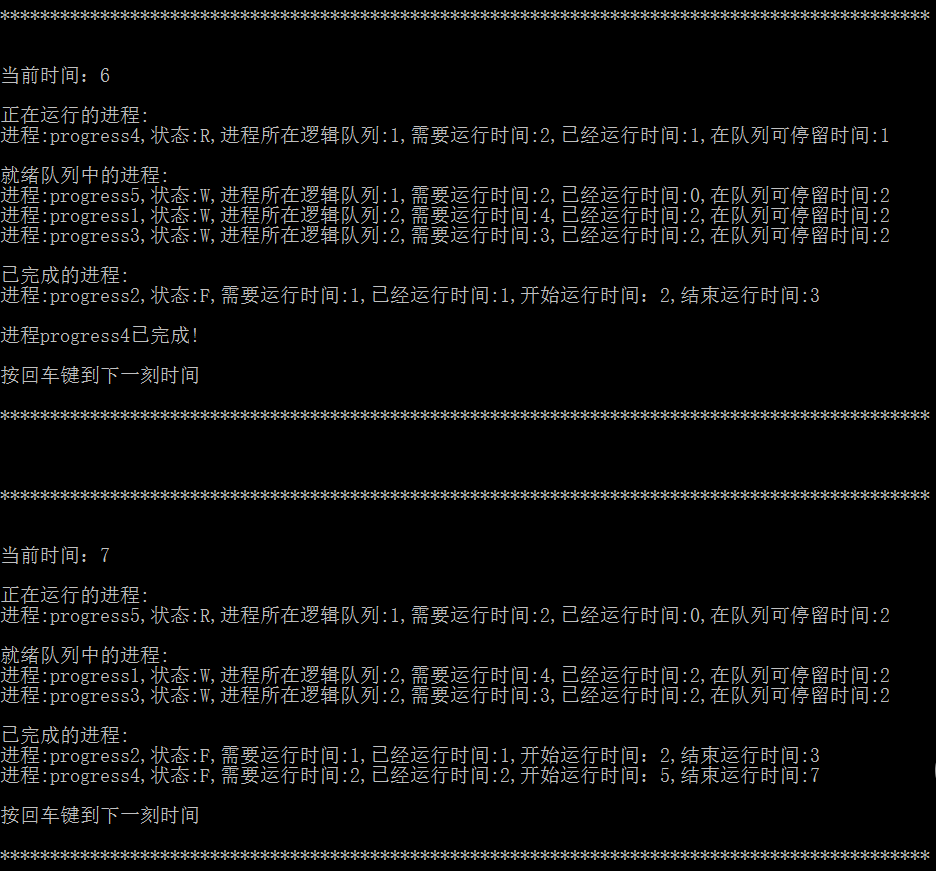
调度进程的情况

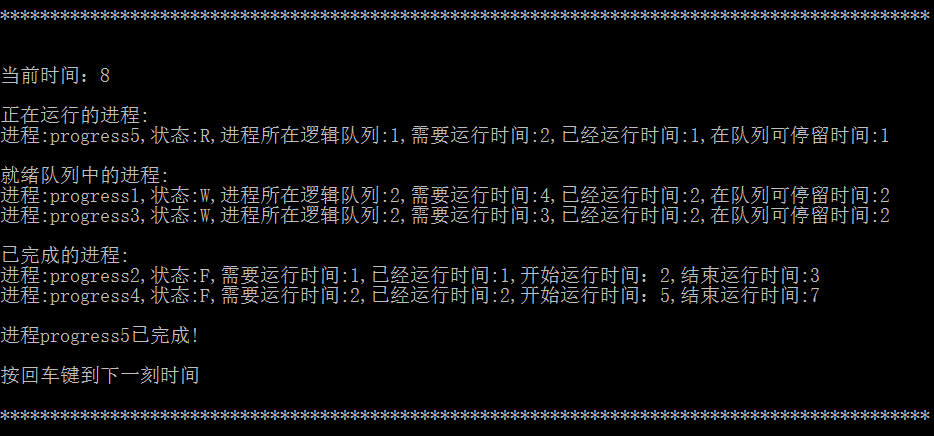


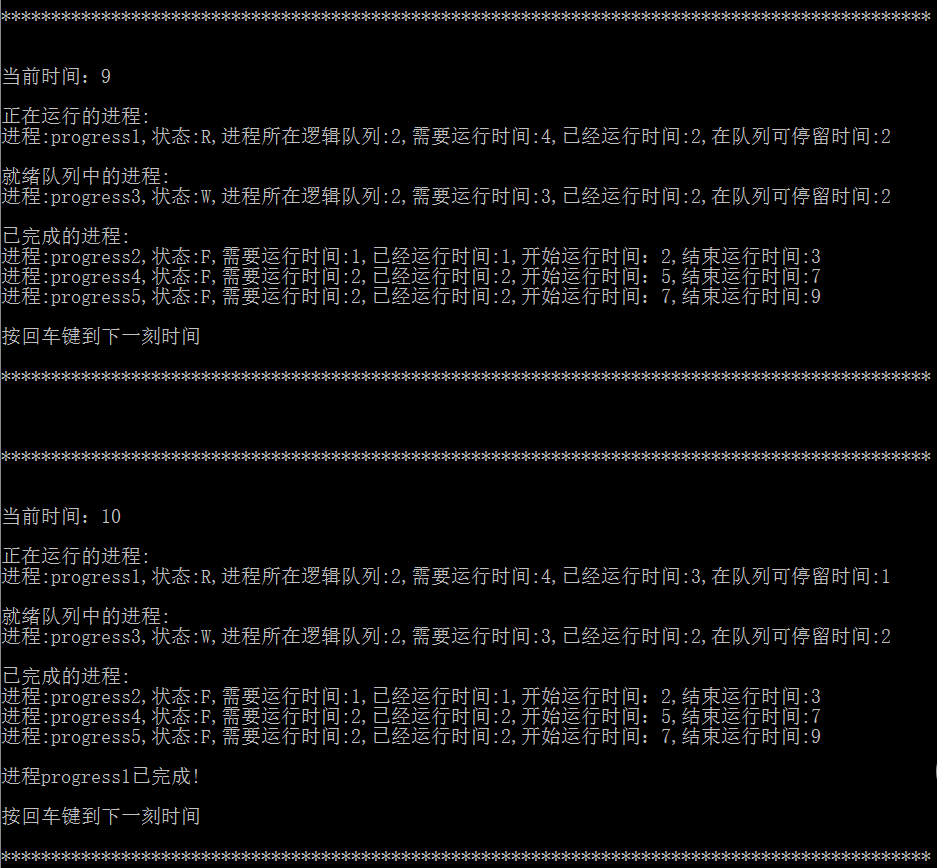


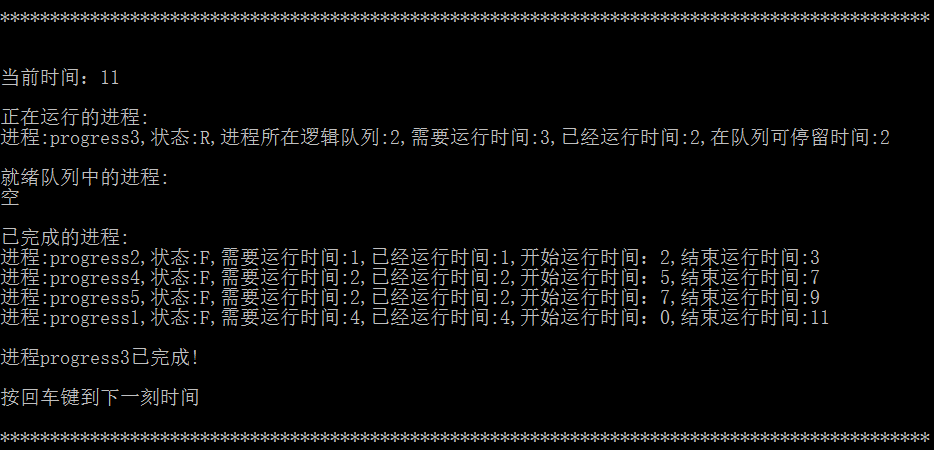








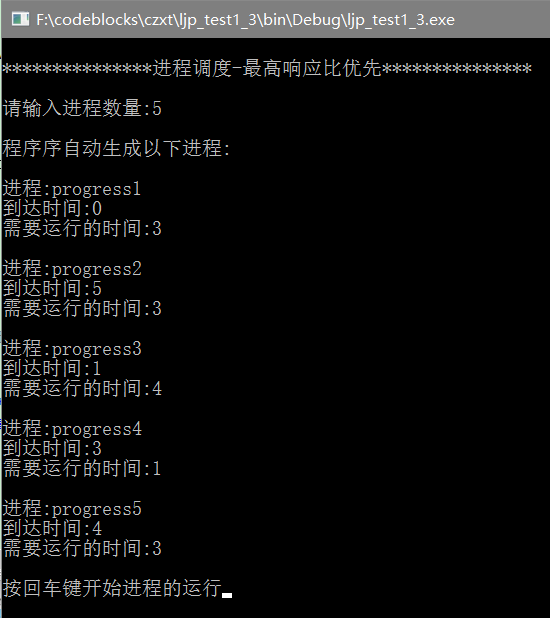




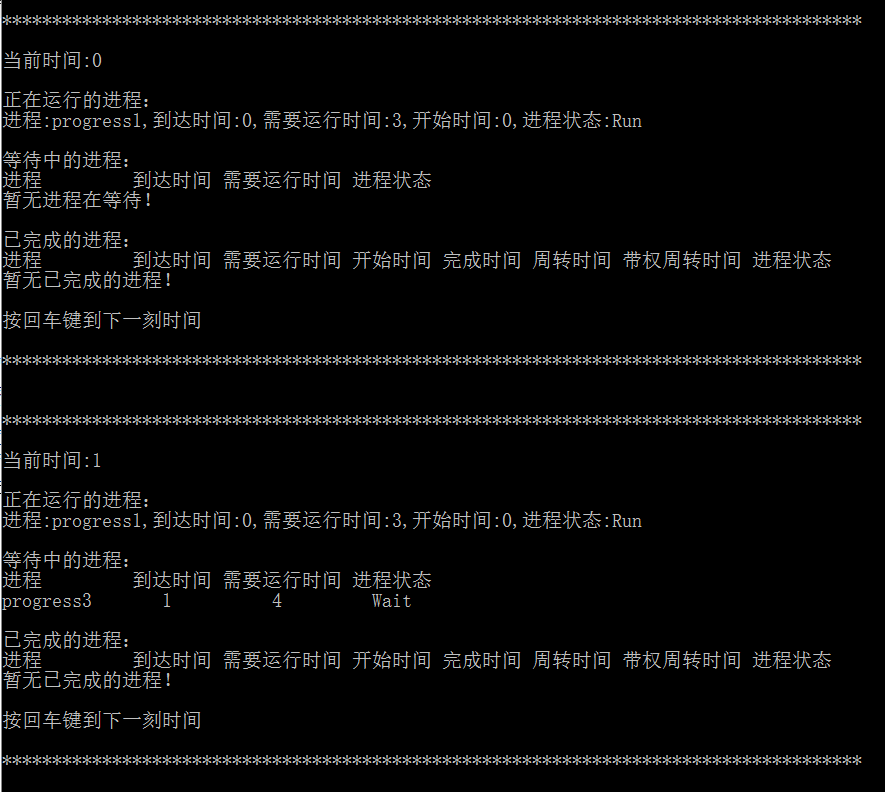


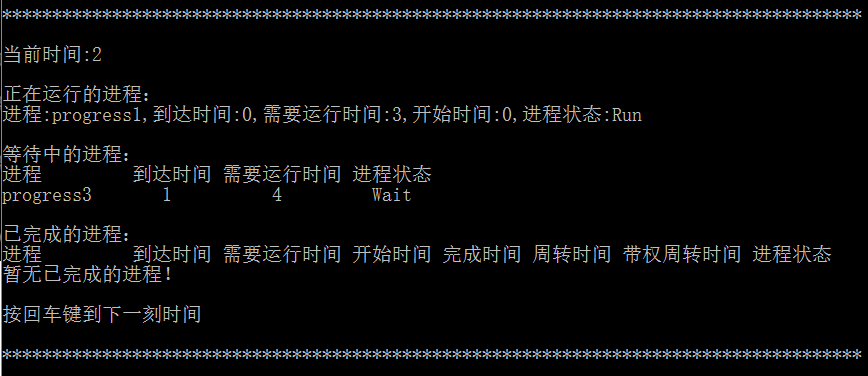
### 5.3高响应比优先算法

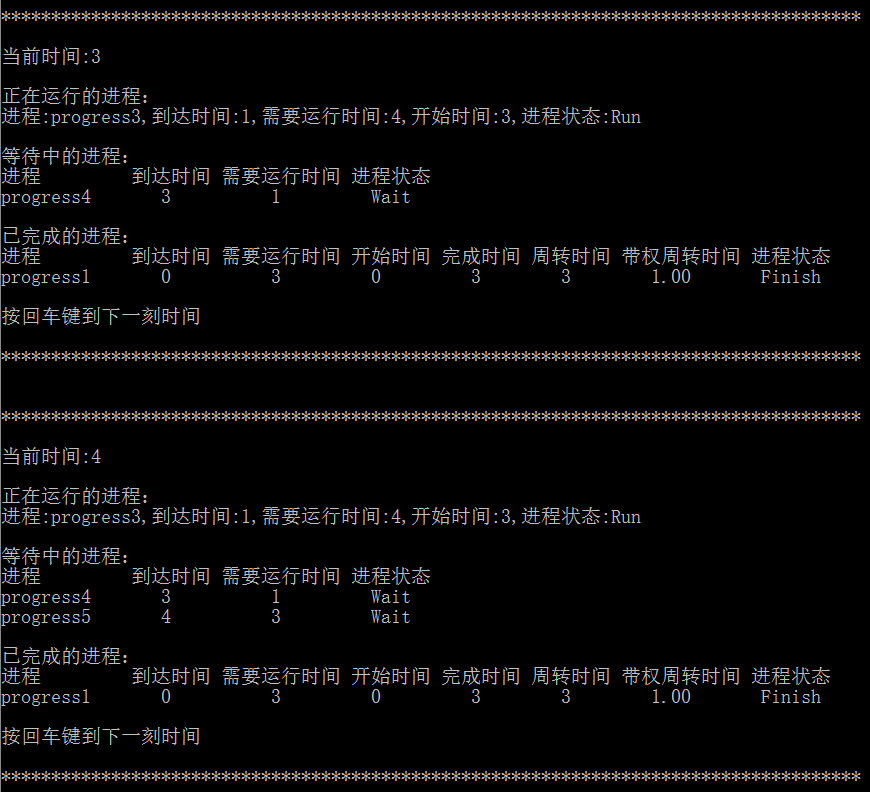
程序随机自动生成进程

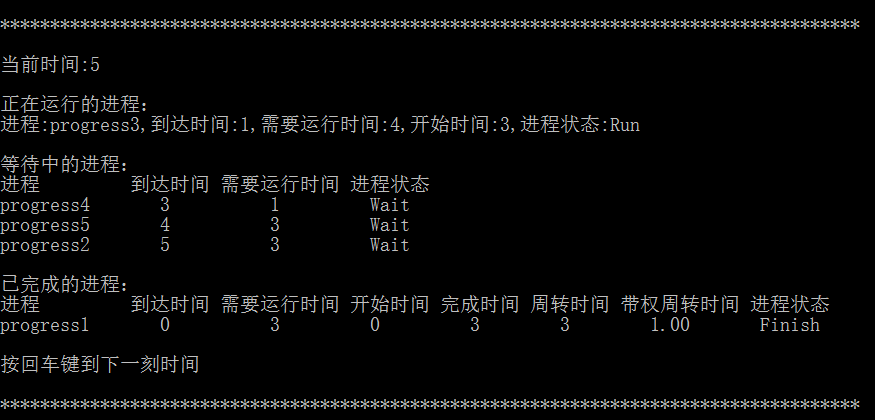


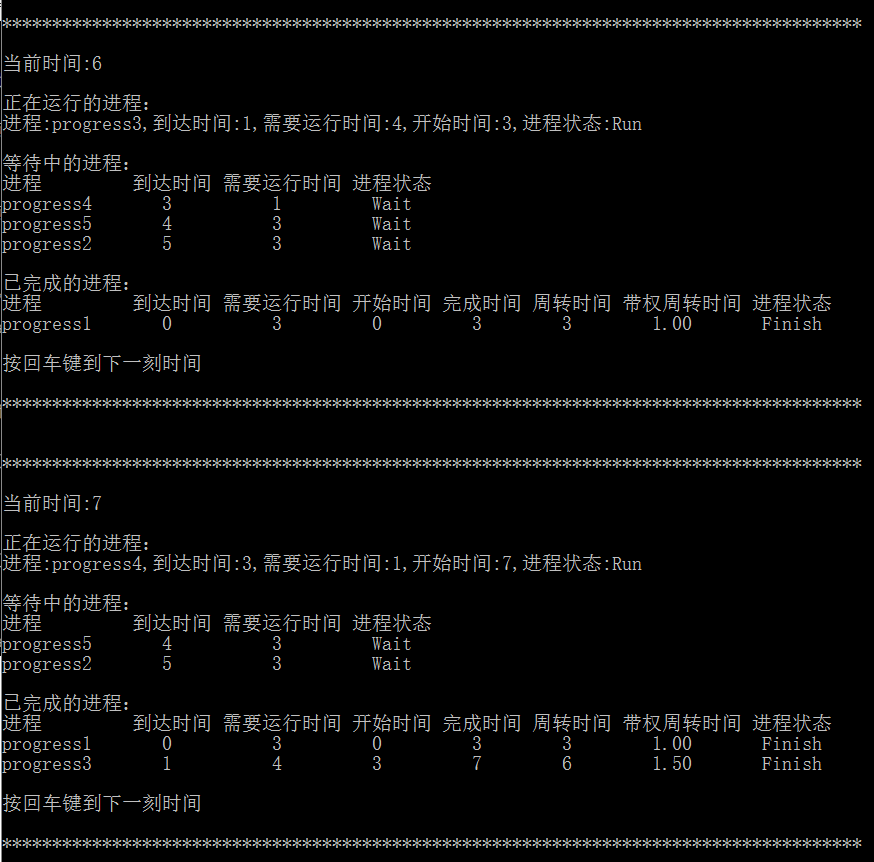
调度进程的情况

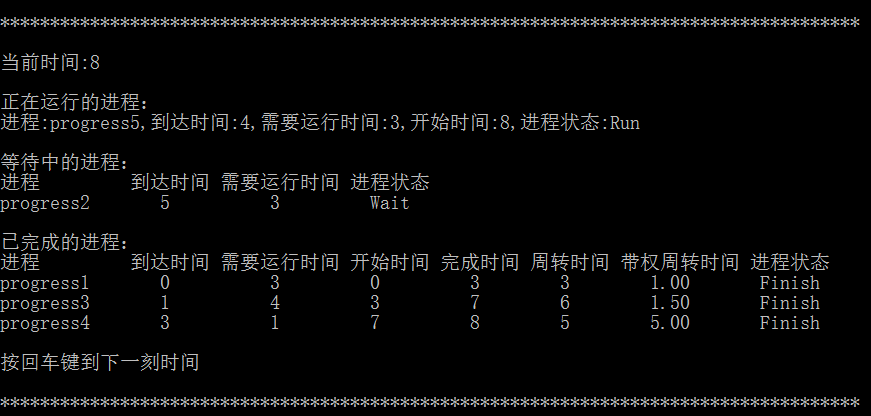


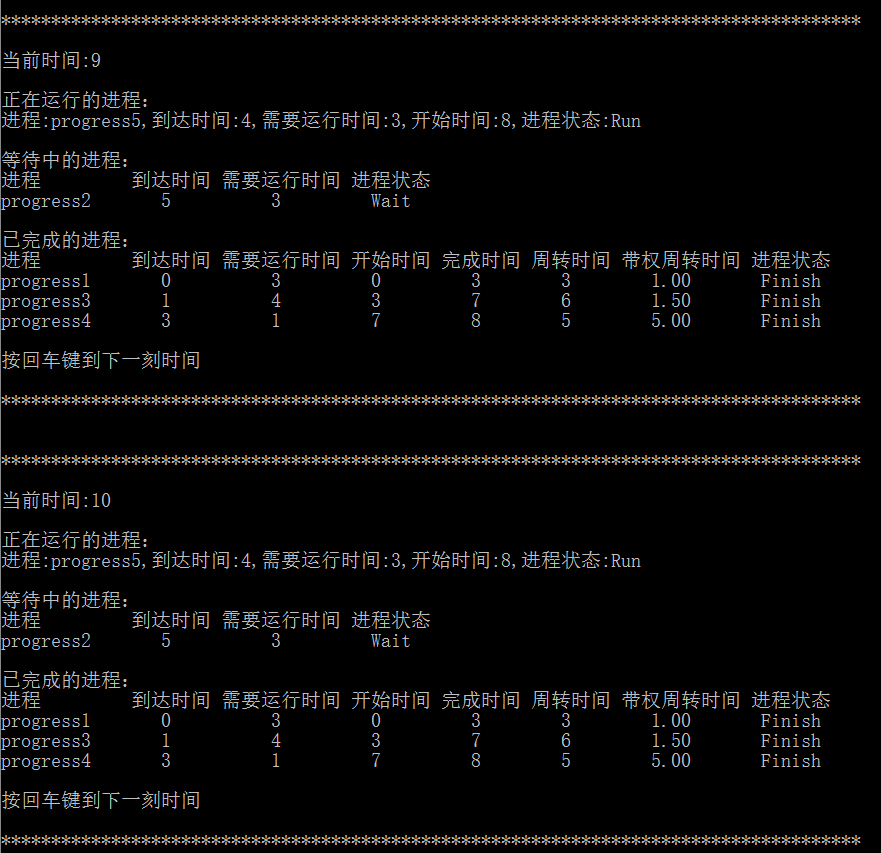


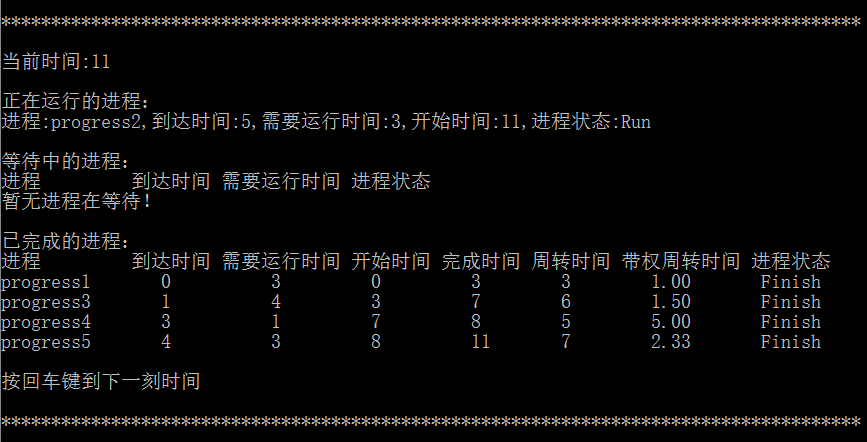


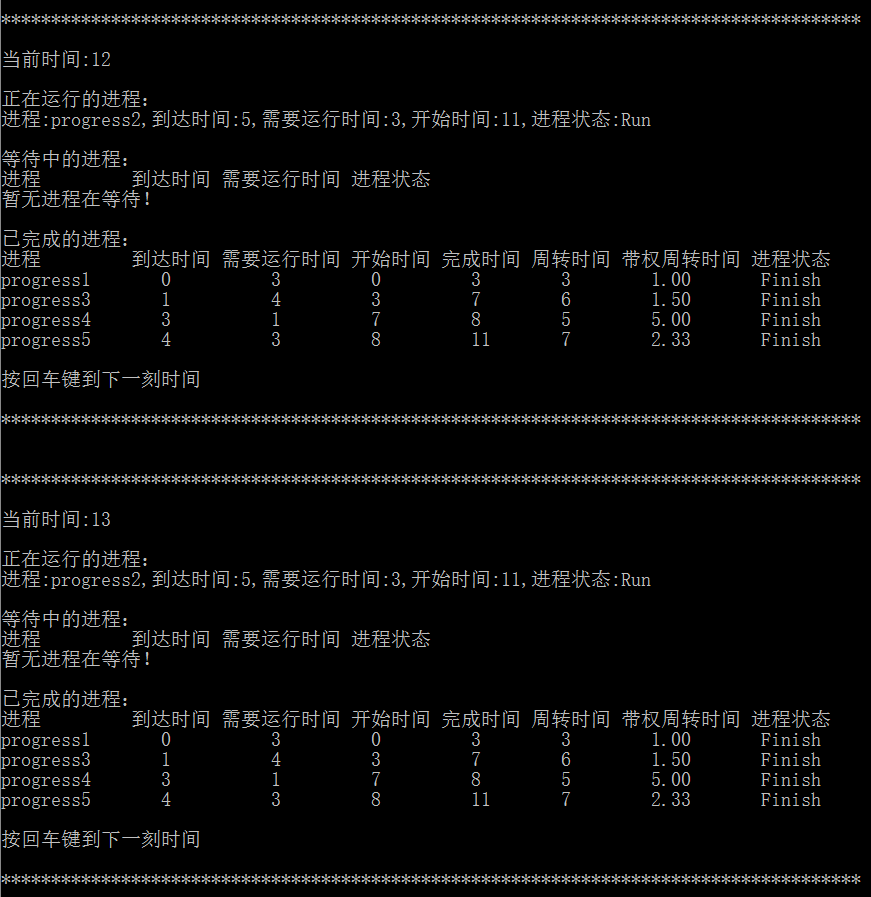


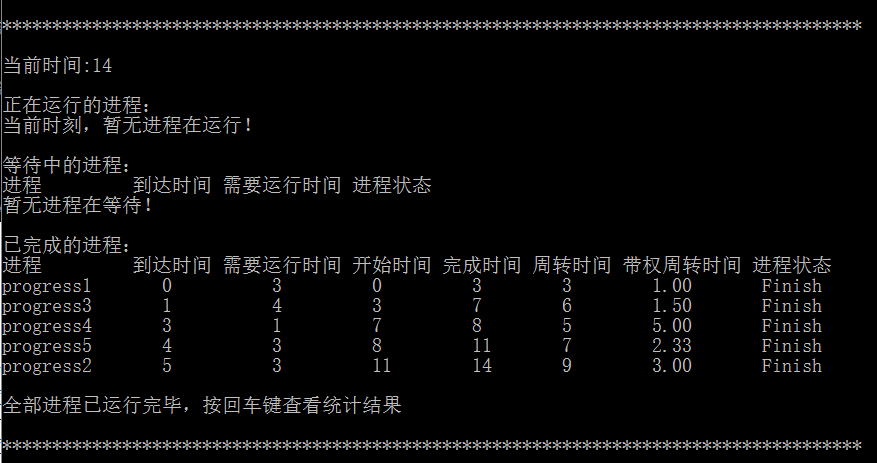














## 6.总结

经过测试结果的比较算法的平均周转时间与分析，可以得出：

短进程优先调度优先算法能降低进程的平均等待时间，提高系统吞吐量。但对长进程不利；未考虑进程的紧迫程度；对进程估计执行时间难以预测。

时间片轮转调度算法的时间片的长度选择比较困难，因为时间片长度的选择直接关系到系统开销和进程的响应时间。

最高优先权优先调度算法的性能介于时间片轮转算法和先来先服务算法之间。

# 实验二 银行家算法

## 1.实验内容

用银行家算法避免死锁，实现系统合理分配资源，加深对进程同步及死锁理解。

## 2.实验要求

1. 假定系统有3类资源A（10个）、B（15个）、C（12个），系有5个进程并发执行，进程调度采用时间片轮转调度算法。
2. 每个进程由一个进程控制块（ PCB）表示，进程控制块可以包含如下信息：进程名、需要的资源总数、已分配的资源数、进程状态。
3. 由程序自动生成进程（包括需要的数据，要注意数据的合理范围）。
4. 进程在运行过程中会随机申请资源（随机生成请求的资源数），如果达到最大需求，表示该进程可以完成；如果没有达到最大需求，则运行一个时间片后，调度其它进程运行。资源分配采用银行家算法来避免死锁。
5. 每个进程的状态可以是就绪 W（Wait）、运行R（Run）、阻塞B（Block）或完成F（Finish）状态之一。
6. 每进行一次调度，程序都要输出一次运行结果：正在运行的进程、就绪队列中的进程、阻塞队列中的进程、完成的进程以及各个进程的 PCB，以便进行检查。

## 3.算法原理相关知识

操作系统按照银行家制定的规则为进程分配资源，当进程首次申请资源时，要测试该进程对资源的最大需求量，如果系统现存的资源可以满足它的最大需求量则按当前的申请量分配资源，否则就推迟分配。当进程在执行中继续申请资源时，先测试该进程本次申请的资源数是否超过了该资源所剩余的总量。若超过则拒绝分配资源，若能满足则按当前的申请量分配资源，否则也要推迟分配。

## 4.实验实现

### 4.1流程图



### 4.2重要的数据结构

#define MAX 20 //最大作业数

#define RESS 100//最大资源数

int Avaiable[RESS];//可利用资源向量 Avaiable

char name[RESS];//可利用资源名

//n x m 矩阵 即 MAX x RESS

int Max[MAX][RESS];//最大需求矩阵 Max

int Allocation[MAX][RESS];//分配矩阵 Allocation

int Need[MAX][RESS];//需求矩阵 Need

int request[MAX][RESS];//请求矩阵 request

int Wnumber[MAX];//记录正在就绪队列的进程的索引的数组

int Rnumber[MAX];//记录正在运行队列的进程的索引的数组，即并发轮转的数组

int Fnumber[MAX]= {RESS,RESS,RESS,RESS,RESS,RESS,RESS,RESS,RESS,RESS,RESS}; //记录已经完成队列进程的索引的数组

int n=5,W=5,F=0;//n为Rnumber的size,W为Wnumber的size,F同理

int m=3;//m是程序资源数

int pan;//是否为安全状态，安全=1

int haveSet=0;//申请并成功分配了资源标志=1

typedef struct job {

int Finish;//进程是否完成，完成=1

char State;//进程状态，出了R，W，F，增加了B（block阻塞状态）

char Name[10];//进程名

int Max[10];//进程最大需求向量

int Allocation[10];//进程已经分配向量

}job;

int size = 0;//当前进程数

job \*Job;

### 4.3重要的算法

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*安全性算法函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void safe()

{

int i,j;

int f;//可分配资源是否足够标志

int w[RESS]= {0};//工作向量work

for(i=0; i<m; i++)

w[i]=Avaiable[i];//初始化为系统资源

int k=0;//完成数

int u=0;//资源不够次数

int end = 0;//while结束标志=1

while(1)

{

u=0;

for(i=0; i<n; i++)//n为运行队列size

{

f=1;

for(j=0; j<m; j++)//m为系统资源数

{

if(Need[i][j]>w[j])//需要的大于可分配的，不足

{

f=0;

u++;

if (u == n - k) {//未完成的资源都不足，不安全

end = 1;

}

break;

}

}

if(Job[i].Finish==0&&f==1)//进程未完成并且未不足

{

for(j=0; j<m; j++)

w[j]=w[j]+Allocation[i][j];//完成后归还系统资源

Job[i].Finish=1;//标志完成

k=k+1;

//u=0;//资源不足重新计数

break;

}

}

if(k==n)//所有进程的Finish=1都满足，系统处于安全状态

{

printf("\n系统状态：系统安全\n");

pan=1;

break;

}

if(1 == end)

{

printf("\n系统状态：不安全状态\n");

pan=0;

break;

}

}//while : end

//安全算法完成，把完成标志归零

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

Job[i].Finish=0;

}

//\*\*\*\*\*\*\*\*\*申请资源函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void TT()

{

int i;

int c=1;//是否请求成功。是=1

pan=1;

a=0;

haveSet=0;

printf("\n");

//随机请求各资源数（在所需之内）

for(i=0; i<m; i++)

{

if(Need[a][i]!=0)

request[a][i]=random(Need[a][i]+1);//随机

else request[a][i]=0;

printf("申请资源%c数量:%d\n", name[i],request[a][i]);

}

//如果任意一个请求的资源大于可分配的，不安全

for(i=0; i<m; i++)

{

if(request[a][i]>Avaiable[i])

{

c=0;//请求不成功

pan=0;

break;

}

}

if(c==1)//请求成功

{

for(i=0; i<m; i++)

{

Avaiable[i]=Avaiable[i]-request[a][i];//剩余可分配资源-

Need[a][i]=Need[a][i]-request[a][i];//需要的资源矩阵-

Allocation[a][i]=Allocation[a][i]+request[a][i];//分配矩阵+

Job[Rnumber[a]].Allocation[i]= Allocation[a][i];//进程分配到的资源=分配矩阵的

}

haveSet=1;

safe();//分配后要执行安全算法

}

}

//\*\*\*\*\*\*\*\*\*\*申请失败归还数据函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void TD()

{

int i;

for(i=0; i<m; i++)

{

Avaiable[i]=Avaiable[i]+request[a][i];

Need[a][i]=Need[a][i]+request[a][i];

Allocation[a][i]=Allocation[a][i]-request[a][i];

Job[Rnumber[a]].Allocation[i]= Allocation[a][i];

}

haveSet=0;

}

void pan0()//请求资源失败，把请求的线程阻塞

{

int tmp,i;

tmp= Rnumber[0]; //当前运行的第一个进程调去最后位置，改成阻塞状态

Job[tmp].State='B';

for(i=0; i<n-1; i++)

{

Rnumber[i]=Rnumber[i+1];

}

Rnumber[n-1]=tmp;

}

void pan1()//同意申请，把运行队列移到完成队列或者运行队列末尾

{

int i,tmp,k=1;

for(i=0; i<3; i++)

{

if(Job[Rnumber[0]].Max[i]!=Job[Rnumber[0]].Allocation[i])

{

k=0;//未完成，不用移到完成队列

break;

}

}

if(k==0)//未完成，当前运行的第一个进程调去最后位置，

{

tmp= Rnumber[0]; //当前运行的第一个进程调去最后位置，

Job[tmp].State='W';

for(i=0; i<n-1; i++)

{

Rnumber[i]=Rnumber[i+1];

}

Rnumber[n-1]=tmp;

}

else //已经完成，将当前运行的第一个进程调到Fnumber数组,如果Wnumber还有就调过来

{

Fnumber[F]=Rnumber[0] ;

Job[Rnumber[0]].State='F';

for(i=0; i<m; i++) //释放资源

{

Avaiable[i]=Avaiable[i]+ Job[Rnumber[0]].Allocation[i];

}

F++;

if(W!=0) //如果Wnumber还有就调过来

{

for(i=0; i<n-1; i++)

{

Rnumber[i]=Rnumber[i+1];

}

Rnumber[i]=Wnumber[W-1];

W--;

}

else //如果没有就减少n

{

for(i=0; i<n-1; i++)

{

Rnumber[i]=Rnumber[i+1];

}

n--;

}

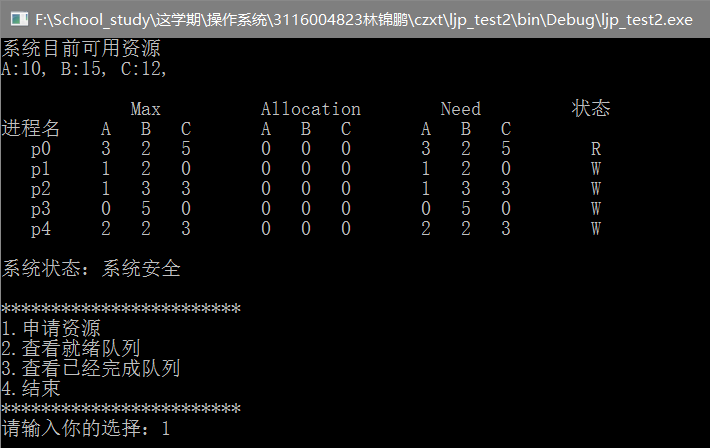
}

}

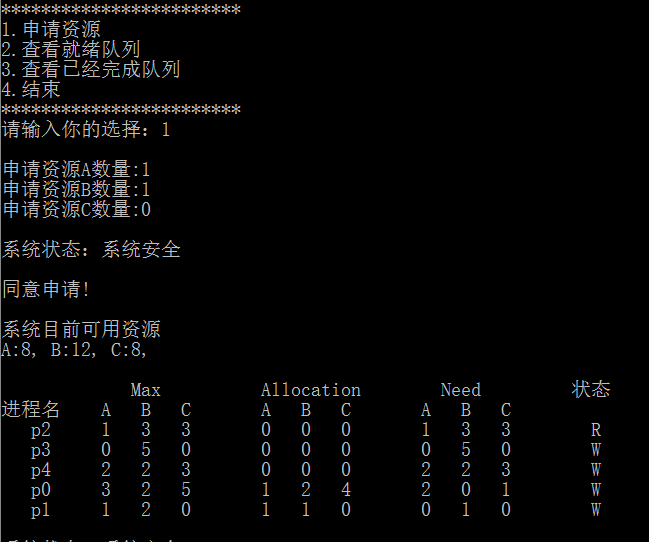
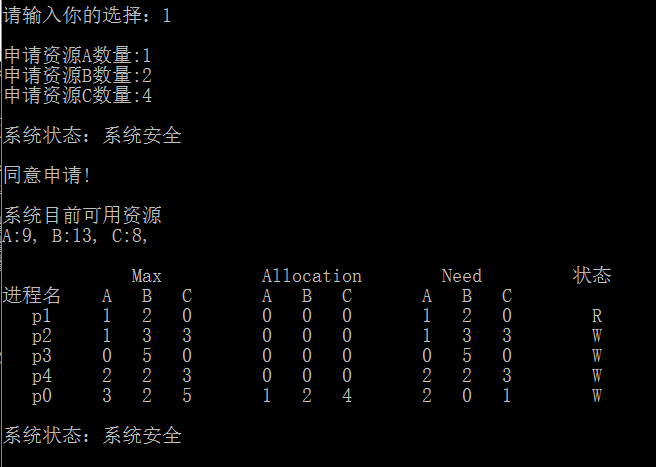
## 5.测试结果

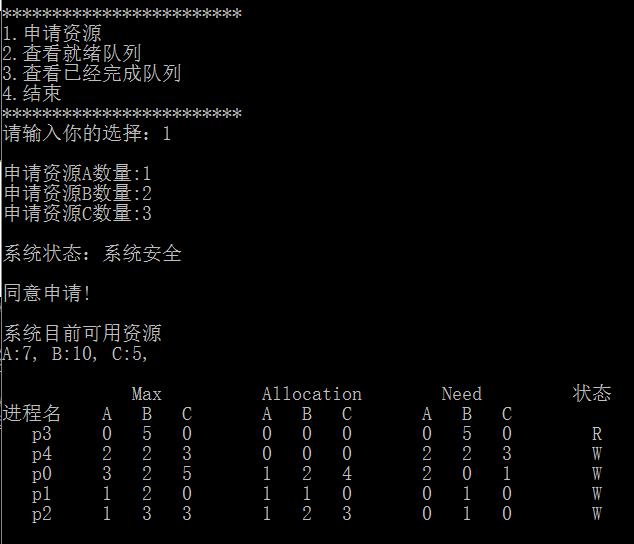
（经过多次测试，终于找出一个又不安全状态的测试用例了）

初始申请状态

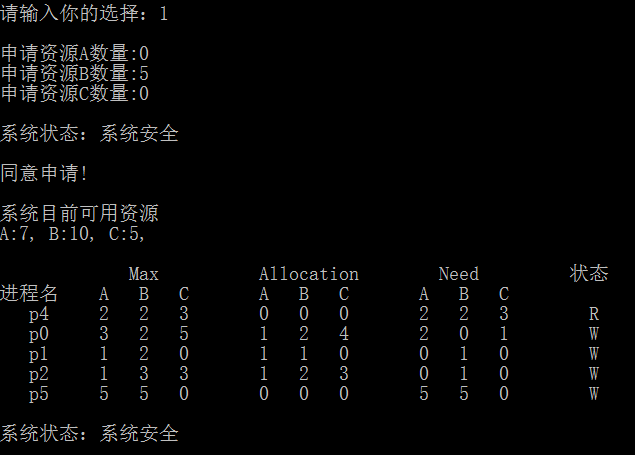


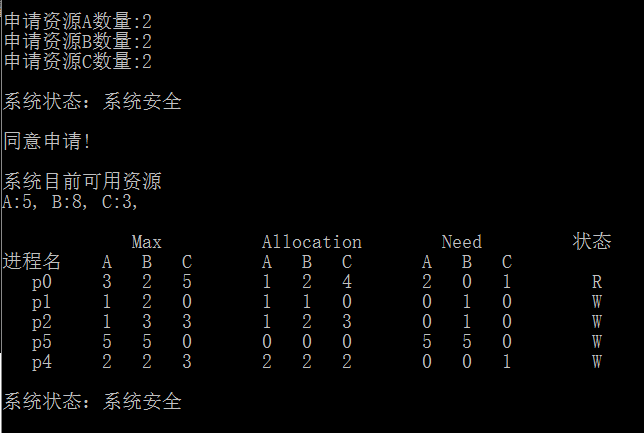
系统随机申请资源

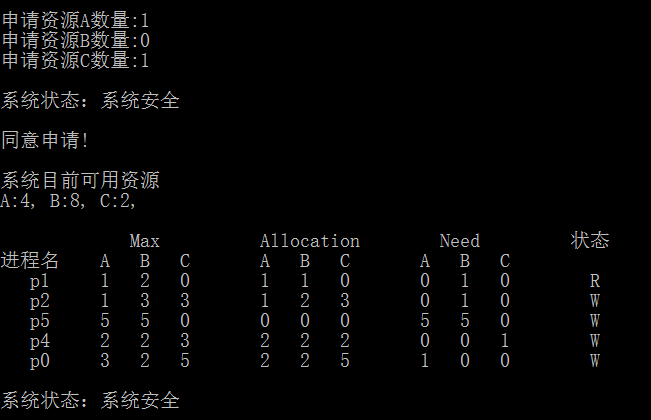


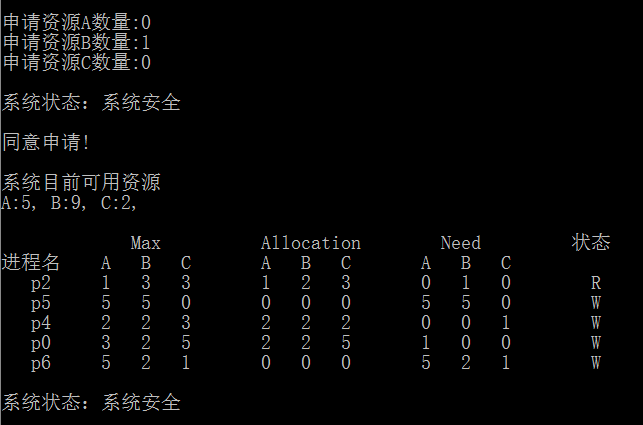


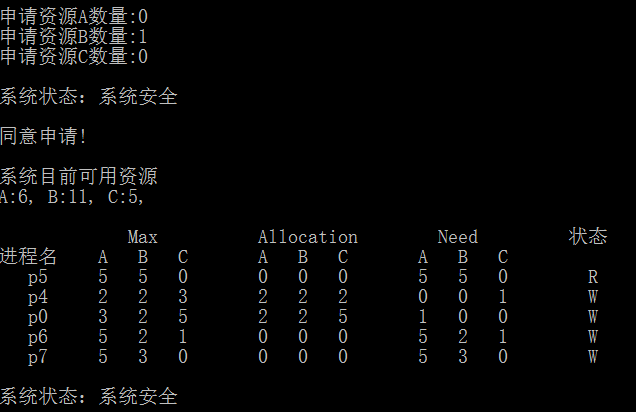
注意到接下来系统可用资源不变，是因为申请的进程p3完成了

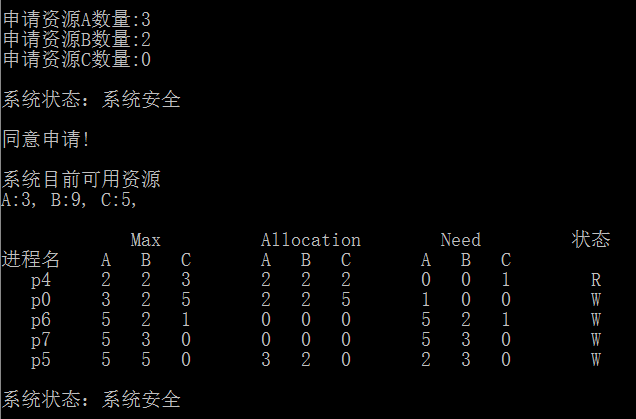


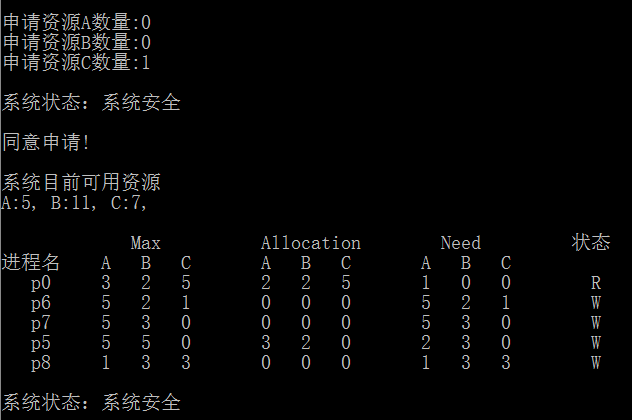


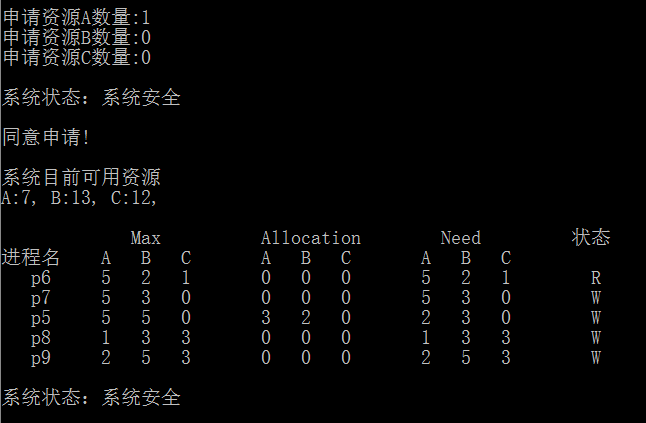


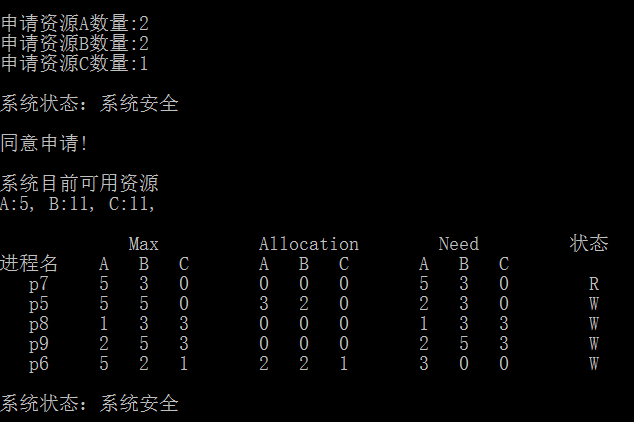


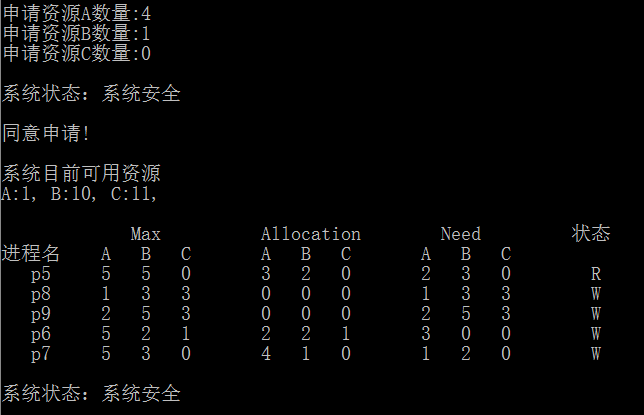




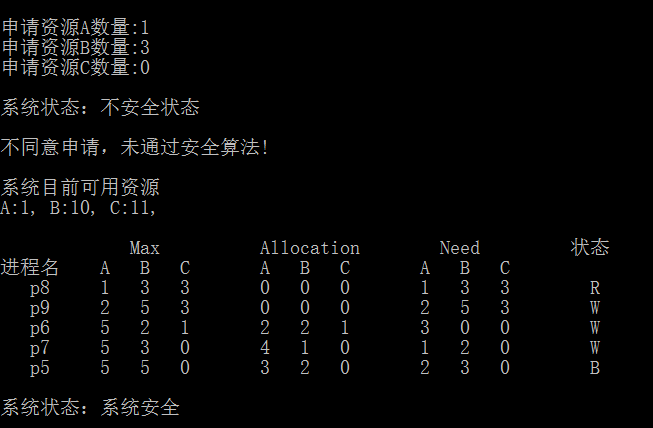




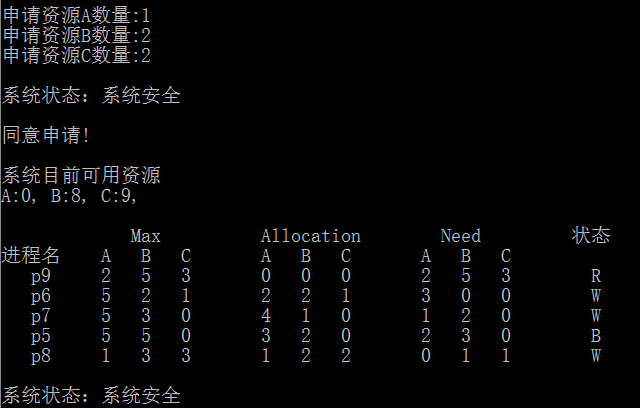




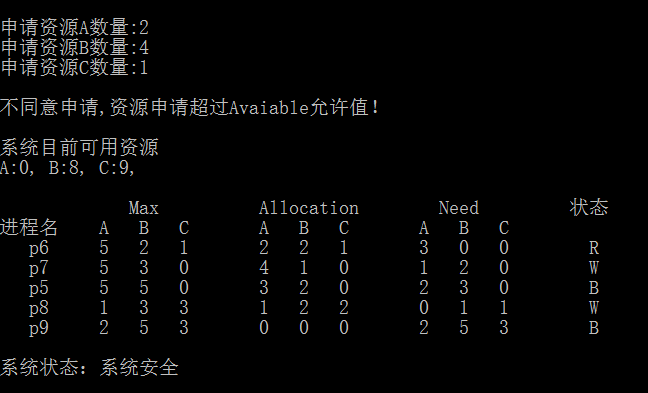
不安全状态，p5申请资源失败，归还资源后安全，到队列末尾等待时机



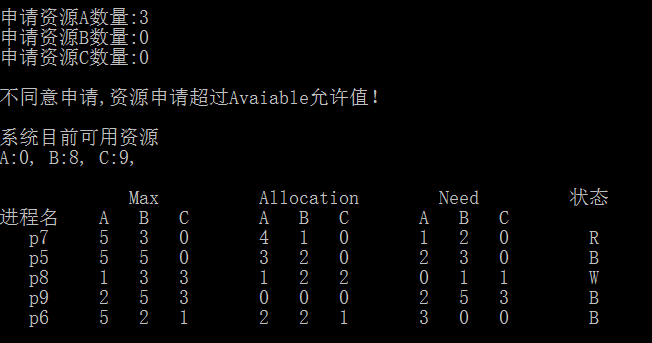
系统继续随机申请资源



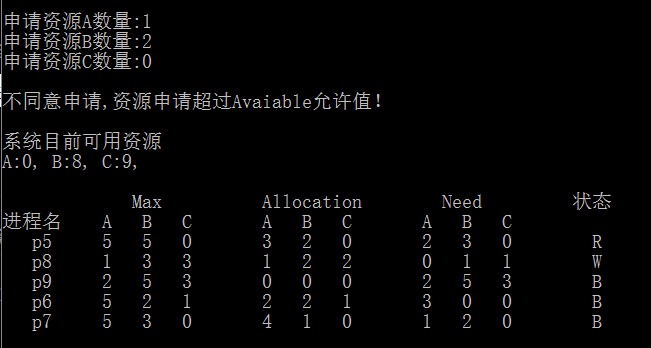
资源不足，进程阻塞及处理 p9，不同意申请后放到队列末尾

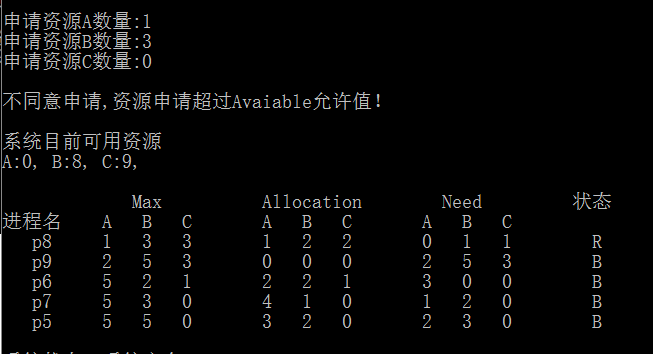


资源不足，进程阻塞及处理 p6，不同意申请后放到队列末尾

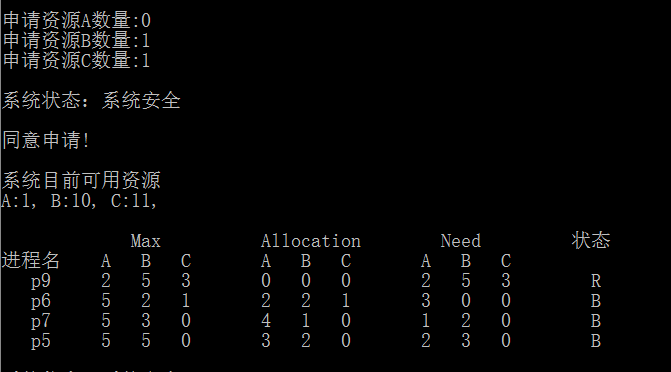


其他申请资源不足的进程也阻塞，回到队列末尾等待时机申请资源运行后

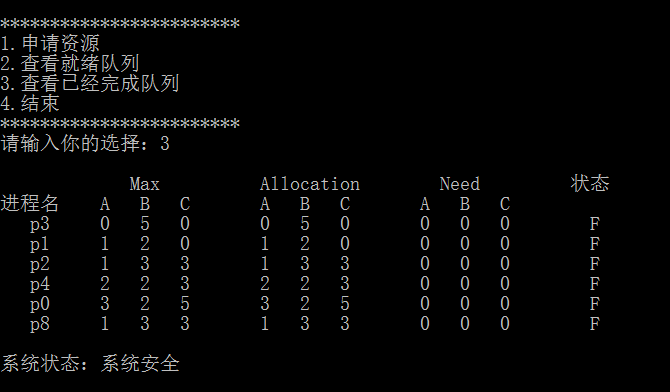




此时p8等来时机，申请资源成功！完成

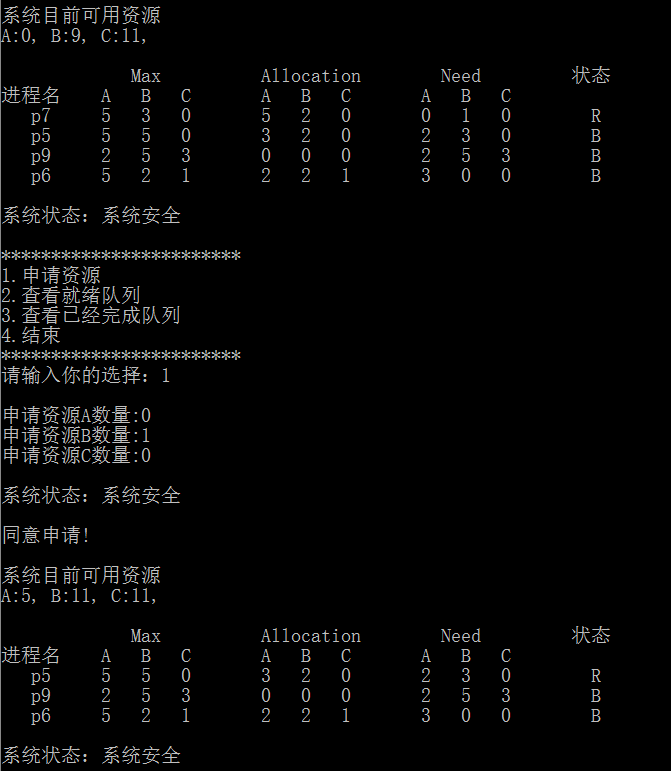


查看已完成队列，可以看到p8在最末尾，刚完成

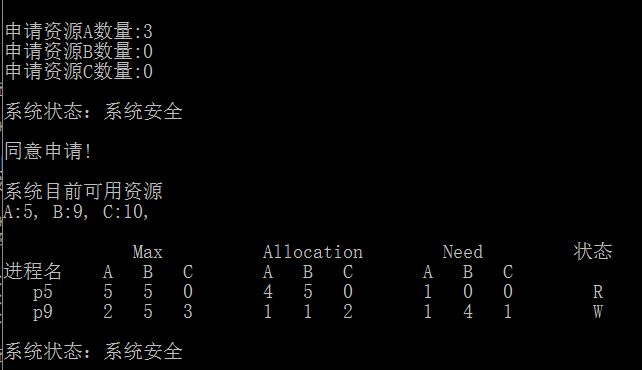


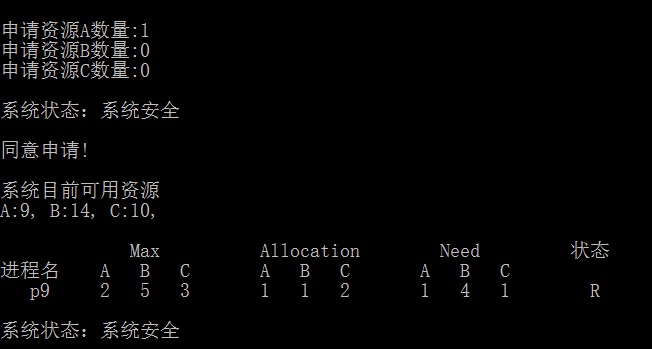
继续申请资源（篇幅所限，这里省略一些过程）

在某个时刻p7申请到了前面p8完成释放的资源，也完成了

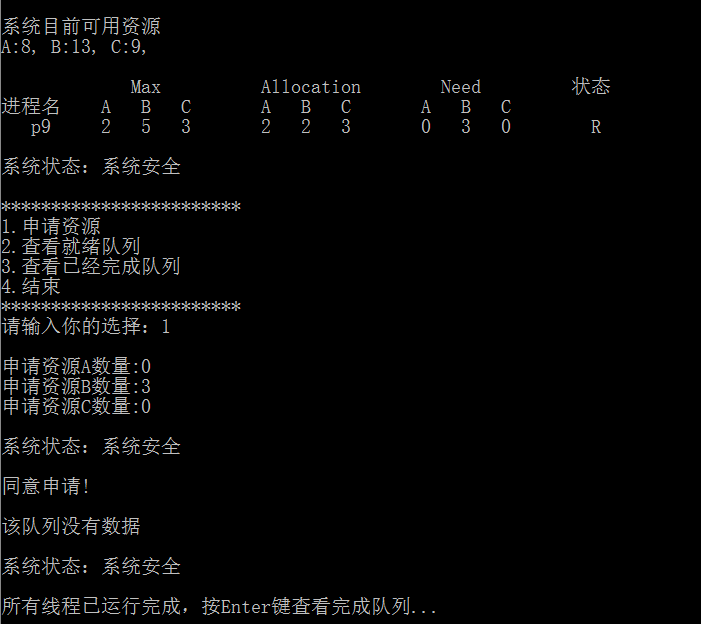


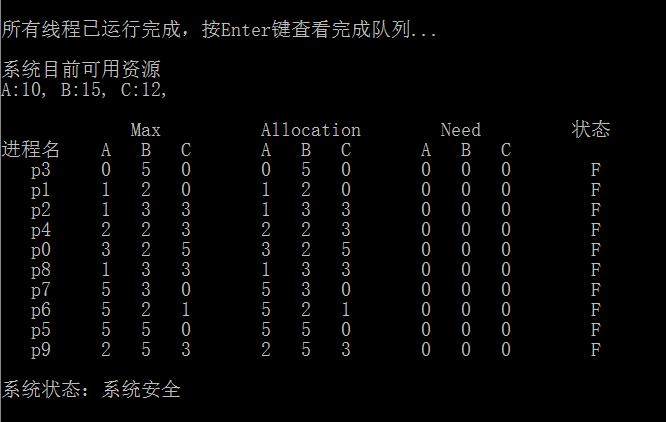
其他资源也在多次申请资源以及等待之后相继完成





继续执行，最后全部进程执行完成



查看完成队列

## 6.总结

多个进程同时运行时，系统根据各类系统资源的最大需求和各类系统的剩余资源为进程安排安全序列，使得系统能快速且安全地运行进程，不至发生死锁，银行家算法是避免死锁的主要方法。

# 实验三 动态分区分配方式的模拟

## 1.实验内容

了解动态分区分配方式中的数据结构和分配算法，并进一步加深对动态分区存储管理方式及其实现过程的理解

## 2.实验要求

1. 用C语言分别实现采用首次适应算法和最佳适应算法的动态分区分配过程和回收过程。其中，空闲分区通过空闲分区链（表）来管理；在进行内存分配时，系统优先使用空闲区低端的空间。
2. 假设初始状态下，可用的内存空间为640KB，并有下列的请求序列：

|  |  |  |
| --- | --- | --- |
| 请求顺序 | 操作 | 请求内存大小（KB） |
| 作业1 | 申请 | 130 |
| 作业2 | 申请 | 60 |
| 作业3 | 申请 | 100 |
| 作业2 | 释放 | 60 |
| 作业4 | 申请 | 200 |
| 作业3 | 释放 | 100 |
| 作业1 | 释放 | 130 |
| 作业5 | 申请 | 140 |
| 作业6 | 申请 | 60 |
| 作业7 | 申请 | 50 |
| 作业8 | 申请 | 60 |

1. 要求每次分配和回收后显示出作业分配情况及空闲内存分区链的情况。

## 3.算法原理相关知识

1.首次适应算法（First-fit）

当要分配内存空间时，就查表，在各空闲区中查找满足大小要求的可用块。只要找到第一个足以满足要球的空闲块就停止查找，并把它分配出去；如果该空闲空间与所需空间大小一样，则从空闲表中取消该项；如果还有剩余，则余下的部分仍留在空闲表中，但应修改分区大小和分区始址。

2.最佳适应算法（Best-fit）

当要分配内存空间时，就查找空闲表中满足要求的空闲块，并使得剩余块是最小的。然后把它分配出去，若大小恰好合适，则直按分配；若有剩余块，则仍保留该余下的空闲分区，并修改分区大小的起始地址。

3.内存回收

将释放作业所在内存块的状态改为空闲状态，删除其作业名，设置为空。并判断该空闲块是否与其他空闲块相连，若释放的内存空间与空闲块相连时，则合并为同一个空闲块，同时修改分区大小及起始地址。

## 4.实验实现

### 4.1重要的数据结构

#define Free 0 //空闲状态

#define Busy 1 //已用状态

#define MAX\_length 640 //最大内存空间为640KB

#define HEAD -1 //头结点ID、size、address、state等，方便判断

typedef int Status;

typedef struct freearea//定义一个空闲区说明表结构

{

int ID; //分区号

int size; //分区大小

int address; //分区地址

int state; //状态

} ElemType;

//---------- 线性表的双向链表存储结构 ------------

typedef struct DuLNode //double linked list

{

ElemType data;

struct DuLNode \*prior; //前趋指针

struct DuLNode \*next; //后继指针

} DuLNode,\*DuLinkList;

DuLinkList block\_first; //头结点

DuLinkList block\_last; //尾结点

Status Initblock()//开创带头结点的内存空间链表

{

block\_first=(DuLinkList)malloc(sizeof(DuLNode));

block\_last=(DuLinkList)malloc(sizeof(DuLNode));

block\_first->prior=NULL;

block\_first->next=block\_last;

block\_last->prior=block\_first;

block\_last->next=NULL;

block\_last->data.address=0;

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

block\_last->data.ID=0;

block\_last->data.state=Free;

//+++++++++++++++++++++++++++++++++++

block\_first->data.ID = -1;

block\_first->data.address = -1;

block\_first->data.size = -1;

block\_first->data.state -1;

//或者给很多个加上前向指针不为0

return OK;

}

### 4.2首次适应算法

#### 4.2.1流程图



#### 4.2.2重要算法

//------------------ 首次适应算法 -----------------------

Status First\_fit(int ID,int request)//传入作业名及申请量

{

//为申请作业开辟新空间且初始化

DuLinkList temp=(DuLinkList)malloc(sizeof(DuLNode));

temp->data.ID=ID;

temp->data.size=request;

temp->data.state=Busy;

DuLNode \*p=block\_first->next;

while(p)

{

if(p->data.state==Free && p->data.size==request)

{

//有大小恰好合适的空闲块

p->data.state=Busy;

p->data.ID=ID;

//+++++++++++++++++++++++++++++++++++++++++

free(temp);

return OK;

break;

}

if(p->data.state==Free && p->data.size>request)

{

//有空闲块能满足需求且有剩余"

temp->prior=p->prior;

temp->next=p;

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

p->prior->next=temp;

p->prior=temp;

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

p->data.size -= request;

return OK;

break;

}

p=p->next;

}

return ERROR;

}

### 4.3最佳适应算法

#### 4.3.1流程图



#### 4.3.2重要算法

//-------------------- 最佳适应算法 ------------------------

Status Best\_fit(int ID,int request)

{

int ch = MAX\_length; //记录最小剩余空间，初始化为最大内存

DuLinkList temp=(DuLinkList)malloc(sizeof(DuLNode));

temp->data.ID=ID;

temp->data.size=request;

temp->data.state=Busy;

DuLNode \*p=block\_first->next;

DuLNode \*q=NULL; //记录最佳插入位置

while(p)

{

if(p->data.state==Free && p->data.size==request)

{

//空闲块大小恰好合适

p->data.ID=ID;

p->data.state=Busy;

//+++++++++++++++++++++++++++++++++++++++++

free(temp);

return OK;

break;

}

if(p->data.state==Free && p->data.size>request)

{

//空闲块大于分配需求

int rest = p->data.size-request;

if (rest < ch) {

ch = rest;//更新剩余最小值

q = p;//更新最佳位置指向

}

}

p=p->next;

}

if(q==NULL) return ERROR;//没有找到空闲块

else

{

//找到了最佳位置并实现分配

temp->prior=q->prior;

temp->next=q;

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

q->prior->next=temp;

q->prior=temp;

q->data.address+=request;

q->data.size=ch;

return OK;

}

}

### 4.4内存回收

#### 4.4.1流程图



#### 4.4.2重要算法

//----------------------- 主 存 回 收 --------------------

Status freeBlock(int ID)

{

DuLNode \*p=block\_first->next;

while(p)

{

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

{

p->data.state=Free;

p->data.ID=Free;

if(p->prior->data.state==Free)//与前面的空闲块相连，合并到前面

{

p->prior->data.size+=p->data.size;

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

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

//+++++++++++++++++++++++记得释放剩下的p

free(p);

p = NULL;

break;

}

if(p->next != NULL && p->next->data.state==Free)//与后面的空闲块相连，合并到后面

{

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

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

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

//+++++++++++++++++++++++记得释放剩下的p

free(p);

p = NULL;

break;

}

}

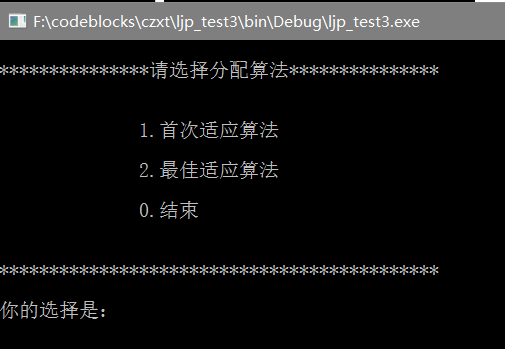
p=p->next;

}

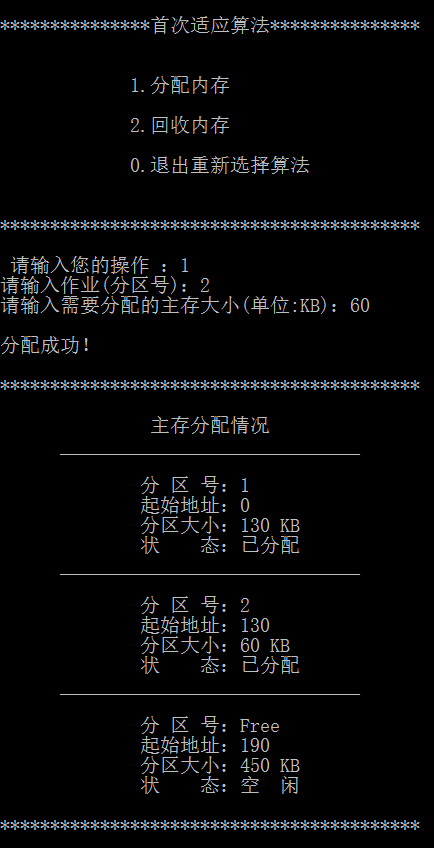
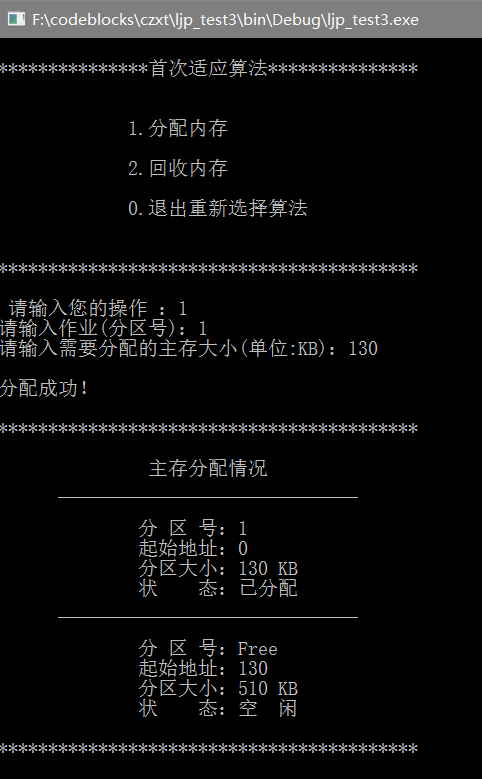
return OK;

}

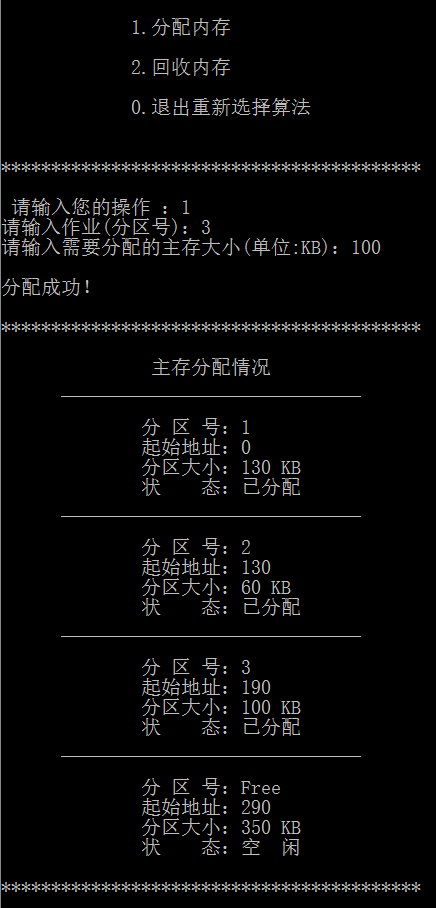
## 5.测试结果



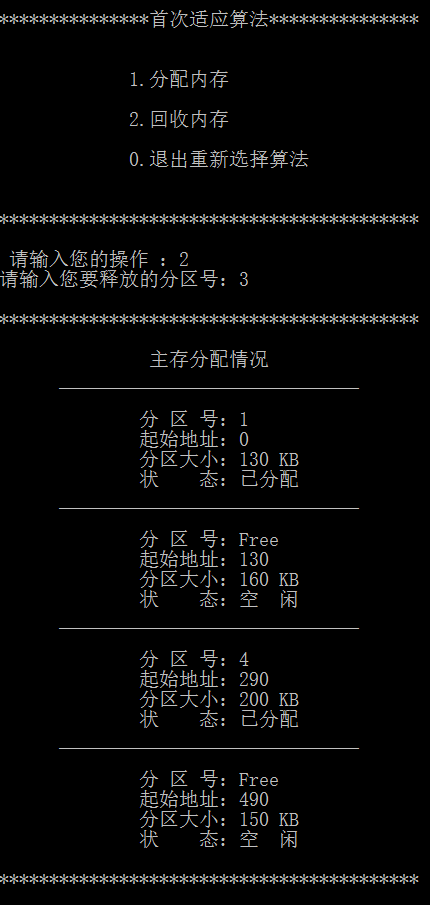
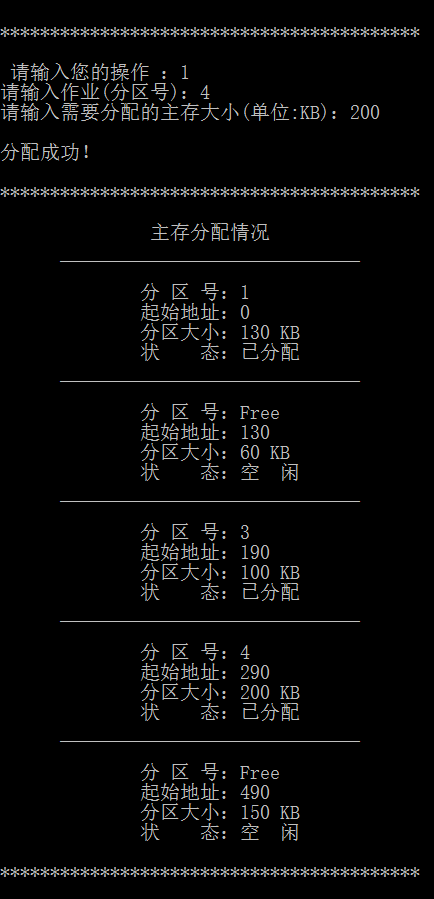
### 5.1首次适应算法



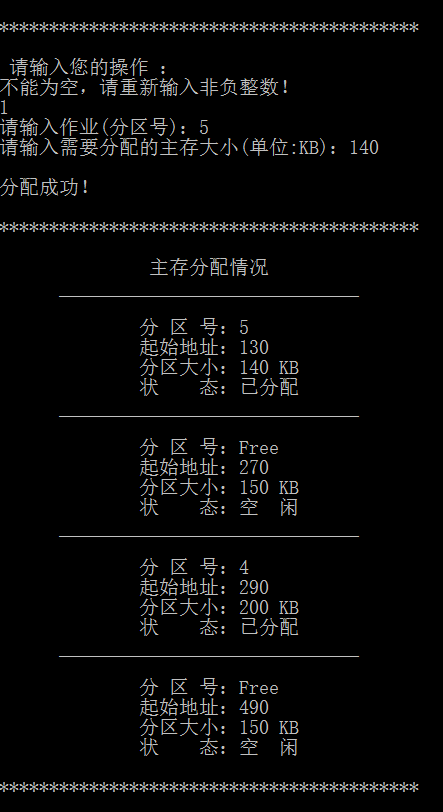
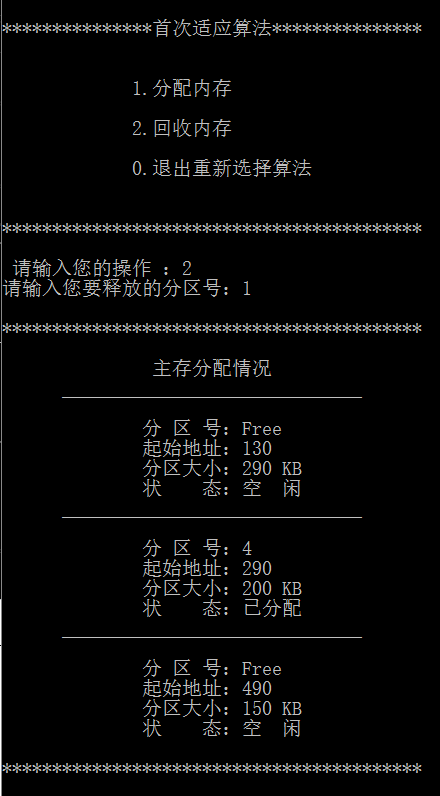
作业1申请主存130KB 作业2申请主存60KB



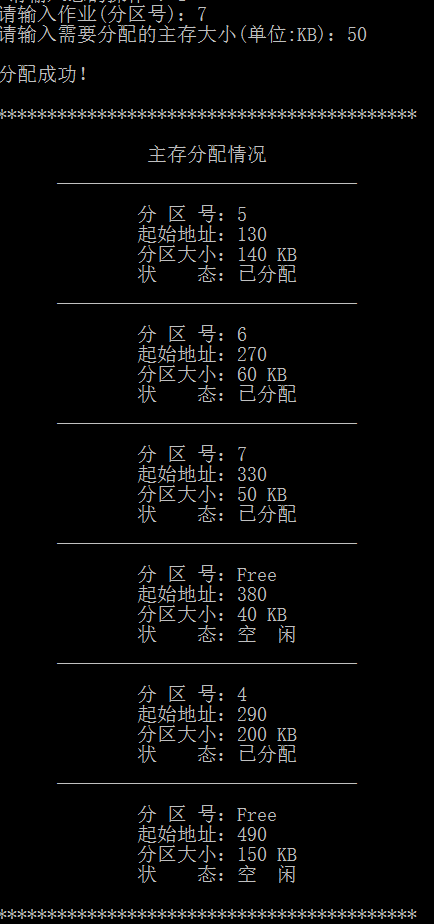
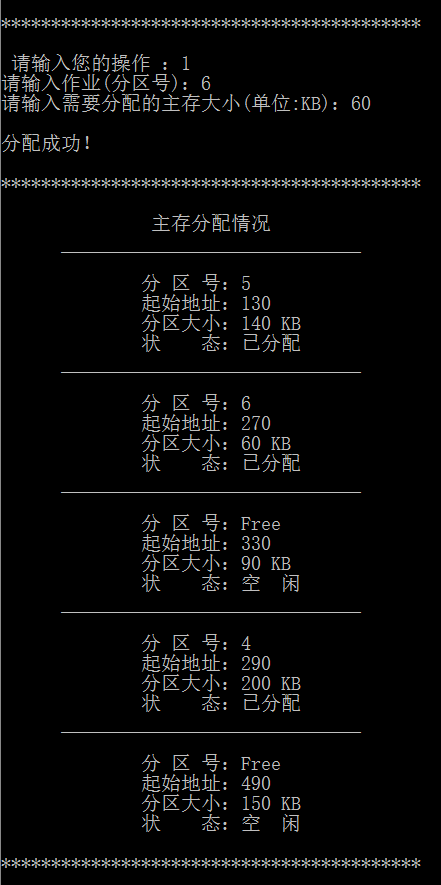
作业3申请主存100KB 作业2释放主存60KB



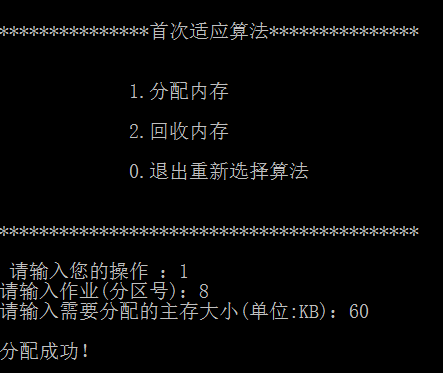
作业4申请主存200KB 作业3释放主存100KB

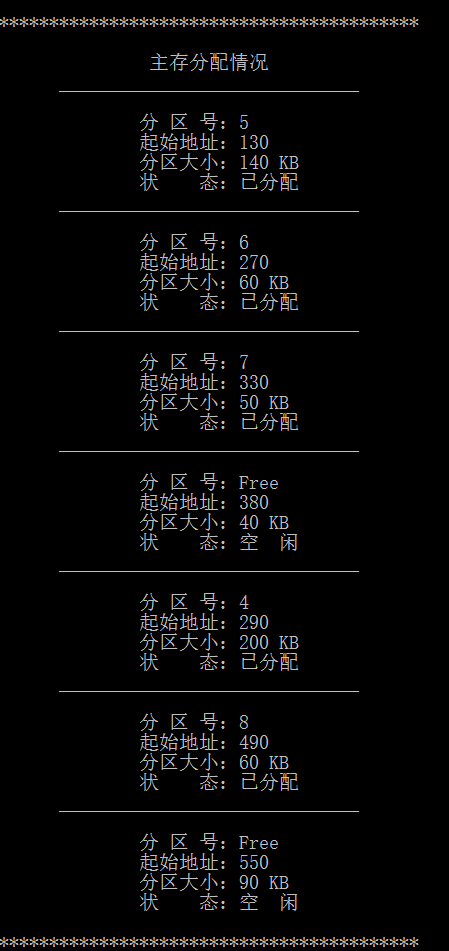


作业1释放主存130KB 作业5申请主存140KB



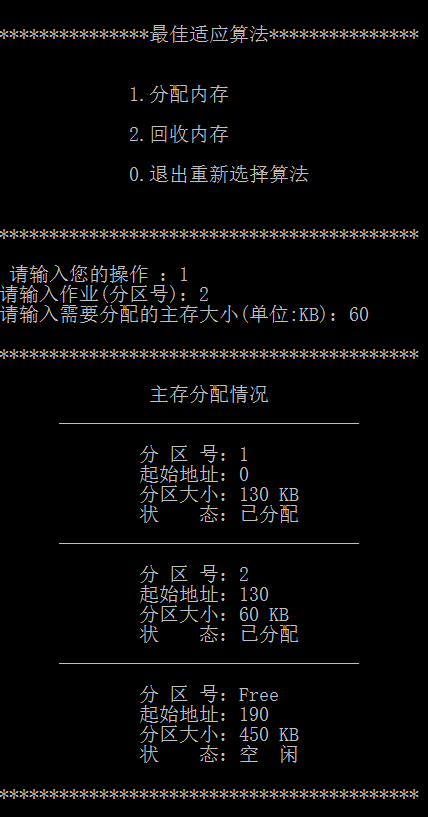
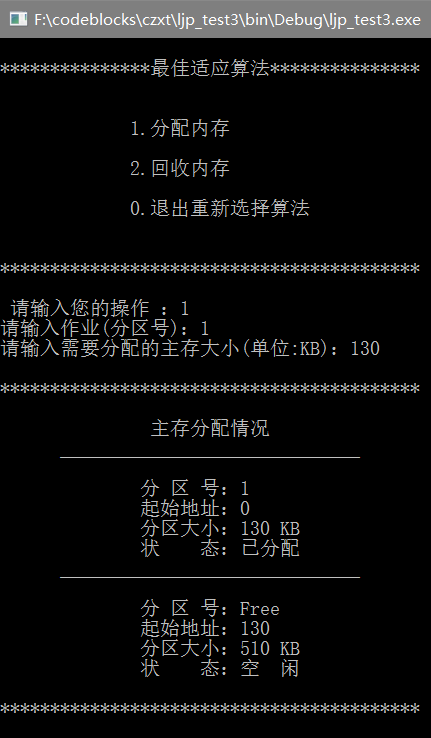
作业6申请主存60KB 作业7申请主存50KB



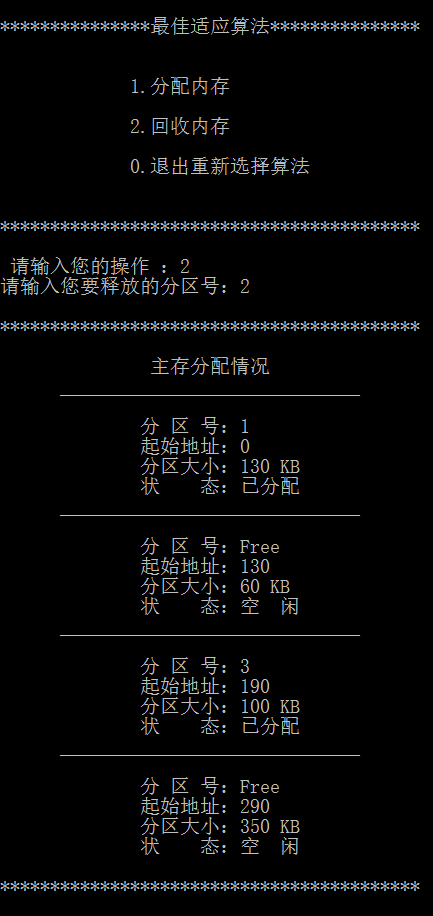
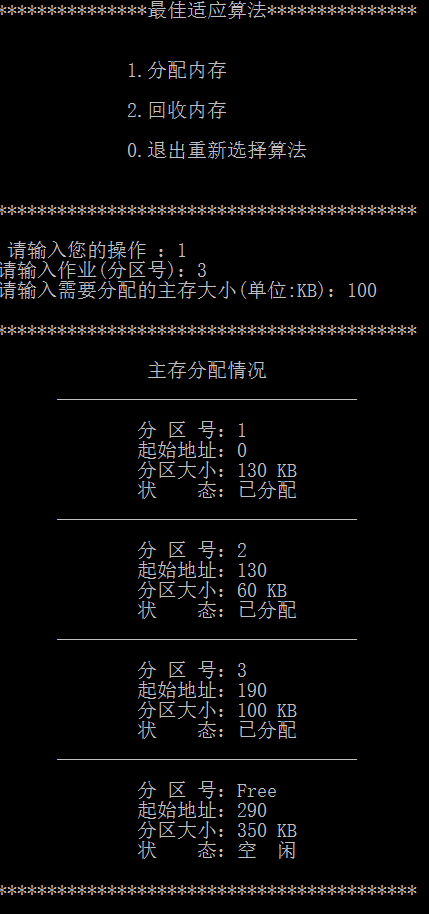


作业8申请主存60KB

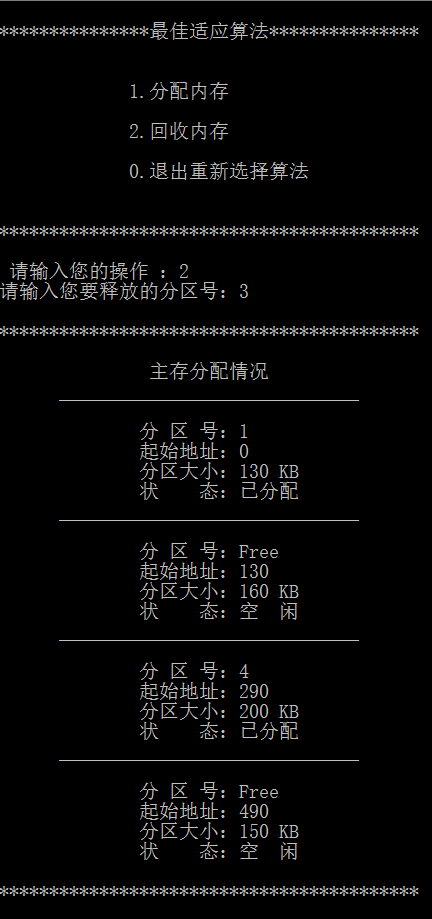
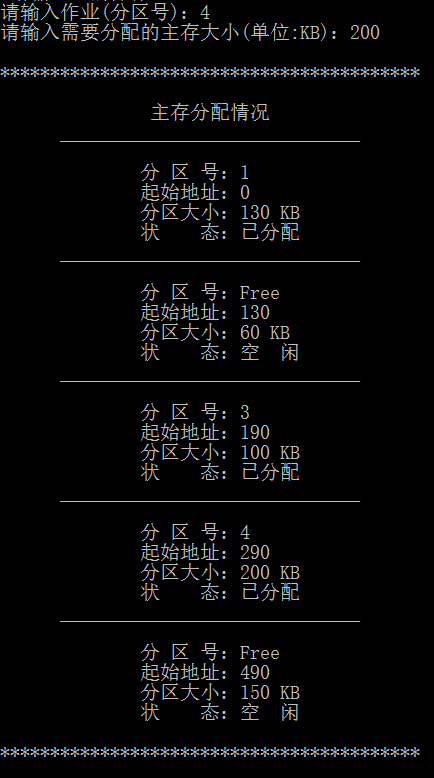
### 5.1最佳适应算法



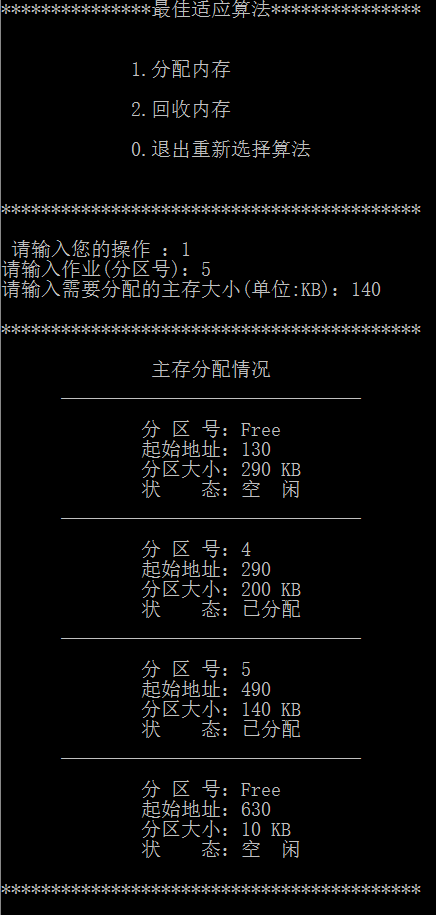
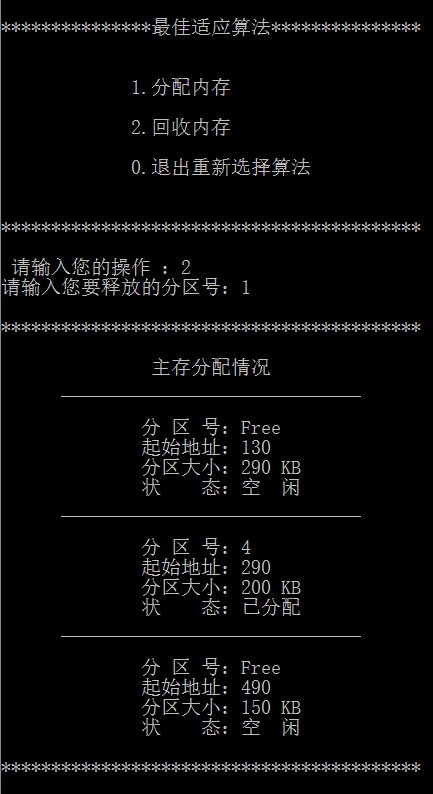
作业1申请主存130KB 作业2申请主存60KB



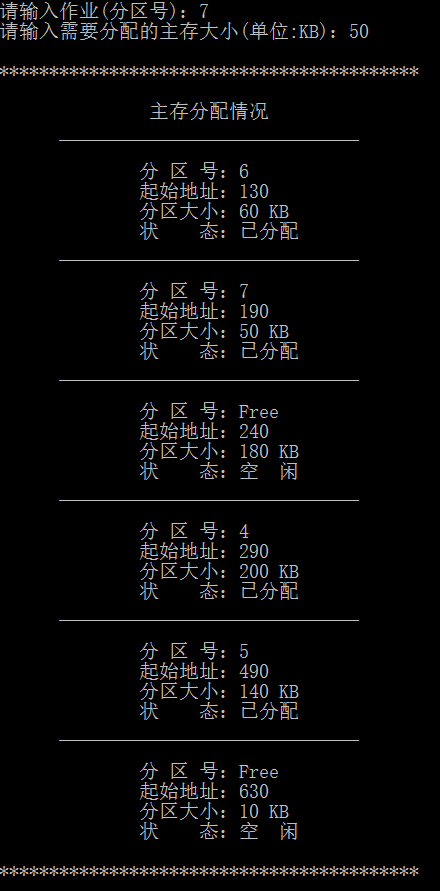
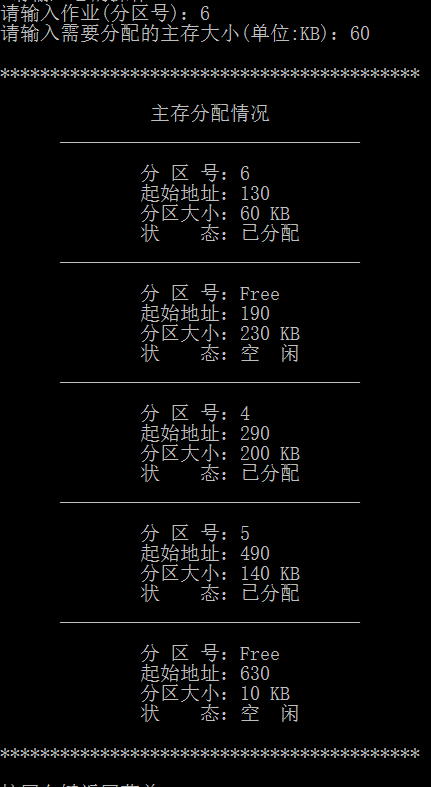
作业3申请主存100KB 作业2释放主存60KB



作业4申请主存200KB 作业3释放主存100KB

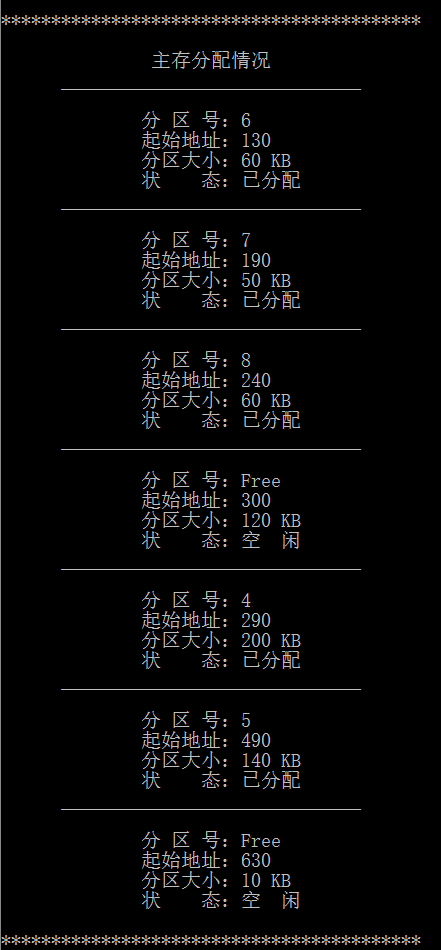


作业1释放主存130KB 作业5申请主存140KB



作业6申请主存60KB 作业7申请主存50KB





作业8申请主存60KB

## 6.总结

通过实验结果可以看到，首次适应算法最后第四个分区为空闲，大小为40KB，而最优适应算法第四个分区为空闲，大小为120KB！！

由此可以看到，最优适应算法能有效减少分区碎片，提高利用率。使用最优适应算法能是操作系统更好用。

实现的话，对于分配来说，根据三个算法来分配，基本用链表来模拟的话，应该没什么问题。而对于回收来说，要判断的条件就会比较多了，首先要考虑是不是第一个分区，再考虑是不是最后一个，两者都不是的话，就要考虑是否相邻的情况，对于第一个和最后一个，也同样需要考虑是否相邻的情况，所以用链表模拟的话，一定要细心，不然很容易会出现指针越界导致程序崩溃。

# 实验四 仿真各种磁盘调度算法

## 1.实验内容

了解磁盘调度的基本算法及性能

## 2.实验要求

由系统产生一系列磁盘请求（10个），分别给出先来先服务算法、最短寻道时间优先算法、扫描(SCAN)算法和循环扫描(CSCAN)算法时磁头移动顺序并计算磁头的平均移动磁道数。（ 假设磁头刚从80磁道移到100磁道）

## 3.算法原理相关知识

1、先来先服务算法（FCFS）

这是一种比较简单的磁盘调度算法。它根据进程请求访问磁盘的先后次序进行调度。此算法的优点是公平、简单，且每个进程的请求都能依次得到处理，不会出现某一进程的请求长期得不到满足的情况。此算法由于未对寻道进行优化，在对磁盘的访问请求比较多的情况下，此算法将降低设备服务的吞吐量，致使平均寻道时间可能较长，但各进程得到服务的响应时间的变化幅度较小。

2、最短寻道时间优先算法（SSTF）

该算法选择这样的进程，其要求访问的磁道与当前磁头所在的磁道距离最近，以使每次的寻道时间最短，该算法可以得到比较好的吞吐量，但却不能保证平均寻道时间最短。其缺点是对用户的服务请求的响应机会不是均等的，因而导致响应时间的变化幅度很大。在服务请求很多的情况下，对内外边缘磁道的请求将会无限期的被延迟，有些请求的响应时间将不可预期。

3、扫描算法（SCAN）

扫描算法不仅考虑到欲访问的磁道与当前磁道的距离，更优先考虑的是磁头的当前移动方向。例如，当磁头正在自里向外移动时，扫描算法所选择的下一个访问对象应是其欲访问的磁道既在当前磁道之外，又是距离最近的。这样自里向外地访问，直到再无更外的磁道需要访问才将磁臂换向，自外向里移动。这时，同样也是每次选择这样的进程来调度，即其要访问的磁道，在当前磁道之内，从而避免了饥饿现象的出现。由于这种算法中磁头移动的规律颇似电梯的运行，故又称为电梯调度算法。此算法基本上克服了最短寻道时间优先算法的服务集中于中间磁道和响应时间变化比较大的缺点，而具有最短寻道时间优先算法的优点即吞吐量较大，平均响应时间较小，但由于是摆动式的扫描方法，两侧磁道被访问的频率仍低于中间磁道。

4、循环扫描算法（CSCAN）

循环扫描算法是对扫描算法的改进。如果对磁道的访问请求是均匀分布的，当磁头到达磁盘的一端，并反向运动时落在磁头之后的访问请求相对较少。这是由于这些磁道刚被处理，而磁盘另一端的请求密度相当高，且这些访问请求等待的时间较长，为了解决这种情况，循环扫描算法规定磁头单向移动。例如，只自里向外移动，当磁头移到最外的被访问磁道时，磁头立即返回到最里的欲访磁道，即将最小磁道号紧接着最大磁道号构成循环，进行扫描。

## 4.实验实现

### 4.1先来先服务算法

#### 4.1.1流程图



#### 4.1.2重要算法

void FCFS(int cidao[],int m,int now) //磁道号数组，个数为m

{

int sum=0; //总寻道长度

int j,i;

float ave; //平均寻道长度

sum+=abs(cidao[0]-now);

printf("磁盘扫描序列为：");

for( i=0; i<m; i++) //输出磁盘扫描序列

{

printf("%d ", cidao[i]);

}

for(i=0,j=1; j<m; i++,j++) //求平均寻道长度

{

sum+=abs(cidao[j]-cidao[i]);

ave=(float)(sum)/(float)(m);

}

printf("\n");

printf("平均移动磁道数：%f\n\n按回车键进行最短寻道时间优先算法调度\n", ave);

}

### 4.2最短寻道时间优先算法

#### 4.2.1流程图



#### 4.2.2重要算法

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*最短寻道时间优先调度算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//思路新颖之处：用l（left）和r（right）标志来扫描（有序）数组，模仿快排。

void SSTF(int cidao[],int m,int now)

{

int k=1;

int l,r;

int i,j,sum=0;

float ave;

cidao=bubble(cidao,m); //调用冒泡排序算法排序

printf("磁盘扫描序列为：");

if(cidao[m-1]<=now) //若当前磁道号大于请求序列中最大者，则直接由外向内依次给予各请求服务

{

for(i=m-1; i>=0; i--)

printf("%d ", cidao[i]);

sum=now-cidao[0];

}

if(cidao[0]>=now) //若当前磁道号小于请求序列中最小者，则直接由内向外依次给予各请求服务

{

for(i=0; i<m; i++)

printf("%d ", cidao[i]);

sum=cidao[m-1]-now;

}

if(now>cidao[0]&&now<cidao[m-1]) //若当前磁道号大于请求序列中最小者且小于最大者

{

while(cidao[k]<now) //确定当前磁道在已排的序列中的位置，后面的算法都用到了，可以直接复制后少量修改，节省时间。

{

k++;

}//找到等于大于当前磁道的位置

//用左右两个标记来扫描有序数组（从中间向左右）

l=k-1;//当前左

r=k;//当前或当前右

while((l>=0)&&(r<m)) //当前磁道在请求序列范围内

{

//选择与当前磁道最近的请求给予服务

if((now-cidao[l])<=(cidao[r]-now))//左近

{

printf("%d ", cidao[l]);

sum+=now-cidao[l];

now=cidao[l];

l=l-1;

}

else//右近

{

printf("%d ", cidao[r]);

sum+=cidao[r]-now;

now=cidao[r];

r=r+1;

}

}

if(l==-1) //磁头移动到序列的最小号，返回外侧扫描仍未扫描的磁道

{

for(j=r; j<m; j++)

{

printf("%d ", cidao[j]);

}

sum+=cidao[m-1]-cidao[0];

}

else //磁头移动到序列的最大号，返回内侧扫描仍未扫描的磁道

{

for(j=l; j>=0; j--)

{

printf("%d ", cidao[j]);

}

sum+=cidao[m-1]-cidao[0];

}

}

ave=(float)(sum)/(float)(m);

printf("\n");

printf("平均移动磁道数：%f\n\n按回车键进行扫描算法调度\n", ave);

}

### 4.3扫描算法

#### 4.3.1流程图



#### 4.3.2重要算法

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*扫描调度算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void SCAN(int cidao[],int m,int now) //先要给出当前磁道号和移动臂的移动方向

{

int k=1;

int l,r,d;

int i,j,sum=0;

float ave;

cidao=bubble(cidao,m); //调用冒泡排序算法排序

if(cidao[m-1]<=now) //若当前磁道号大于请求序列中最大者，则直接由外向内依次给予各请求服务,此情况同最短寻道优先

{

printf("磁盘扫描序列为：");

for(i=m-1; i>=0; i--)

printf("%d ", cidao[i]);

sum=now-cidao[0];

}

if(cidao[0]>=now) //若当前磁道号小于请求序列中最小者，则直接由内向外依次给予各请求服务,此情况同最短寻道优先

{

printf("磁盘扫描序列为：");

for(i=0; i<m; i++)

printf("%d ", cidao[i]);

sum=cidao[m-1]-now;

}

if(now>cidao[0]&&now<cidao[m-1]) //若当前磁道号大于请求序列中最小者且小于最大者

{

while(cidao[k]<now)

{

k++;

}

l=k-1;

r=k;

printf("请输入扫描调度算法的当前移动臂的移动的方向 (1 表示向外 ，0表示向内) : ");

d = geti();

printf("磁盘扫描序列为：");

if(d==0) //选择移动臂方向向内，则先向内扫描

{

for(j=l; j>=0; j--)

{

printf("%d ", cidao[j]);//输出向内扫描的序列

}

for(j=r; j<m; j++) //磁头移动到最小号，则改变方向向外扫描未扫描的磁道

{

printf("%d ", cidao[j]);//输出向外扫描的序列

}

sum=now-2\*cidao[0]+cidao[m-1];

}

else //选择移动臂方向向外，则先向外扫描

{

for(j=r; j<m; j++)

{

printf("%d ", cidao[j]);//输出向外扫描的序列

}

for(j=l; j>=0; j--) //磁头移动到最大号，则改变方向向内扫描未扫描的磁道

{

printf("%d ", cidao[j]);

}

sum=-now-cidao[0]+2\*cidao[m-1];

}

}

ave=(float)(sum)/(float)(m);

printf("\n");

printf("平均移动磁道数：%f\n\n按回车键进行循环扫描算法调度\n", ave);

}

### 4.4循环扫描算法

#### 4.4.1流程图



#### 4.4.2重要算法

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*循环扫描调度算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void CSCAN(int cidao[],int m,int now)

{

int k=1;

int l,r;

int i,j,sum=0;

float ave;

cidao=bubble(cidao,m); //调用冒泡排序算法排序

printf("磁盘扫描序列为：");

if(cidao[m-1]<=now) //若当前磁道号大于请求序列中最大者，则直接将移动臂移动到最小号磁道依次向外给予各请求服务

{

for(i=0; i<m; i++)

printf("%d ", cidao[i]);

sum=now-2\*cidao[0]+cidao[m-1];

}

if(cidao[0]>=now) //若当前磁道号小于请求序列中最小者，则直接由内向外依次给予各请求服务,此情况同最短寻道优先

{

for(i=0; i<m; i++)

printf("%d ", cidao[i]);

sum=cidao[m-1]-now;

}

if(now>cidao[0]&&now<cidao[m-1]) //若当前磁道号大于请求序列中最小者且小于最大者

{

while(cidao[k]<now) //单向反复地从内向外扫描

{

k++;

}

l=k-1;

r=k;

for(j=r; j<m; j++)

{

printf("%d ", cidao[j]);//输出从当前磁道向外扫描的序列

}

for(j=0; j<r; j++) //当扫描完最大号磁道，磁头直接移动到最小号磁道，再向外扫描未扫描的磁道

{

printf("%d ", cidao[j]);

}

sum=2\*cidao[m-1]+cidao[l]-now-2\*cidao[0];

}

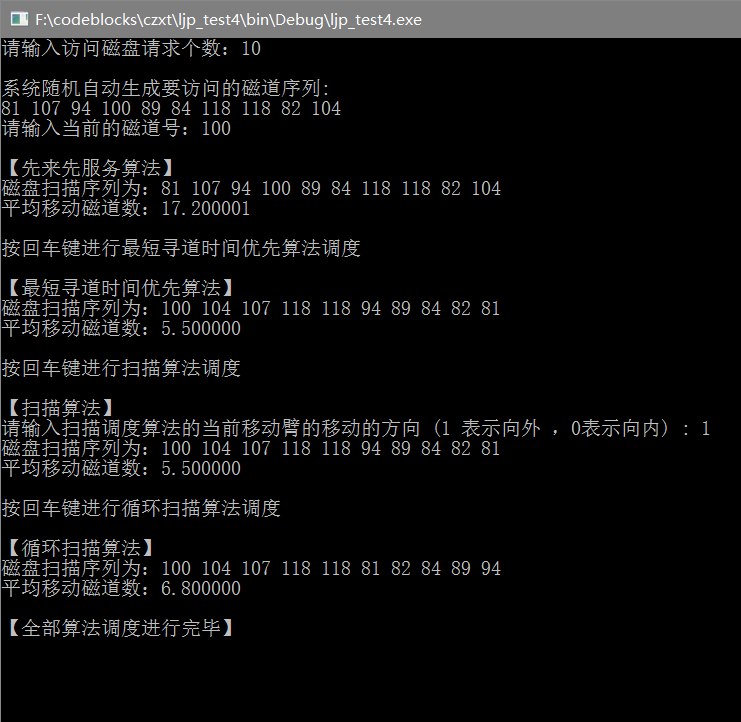
ave=(float)(sum)/(float)(m);

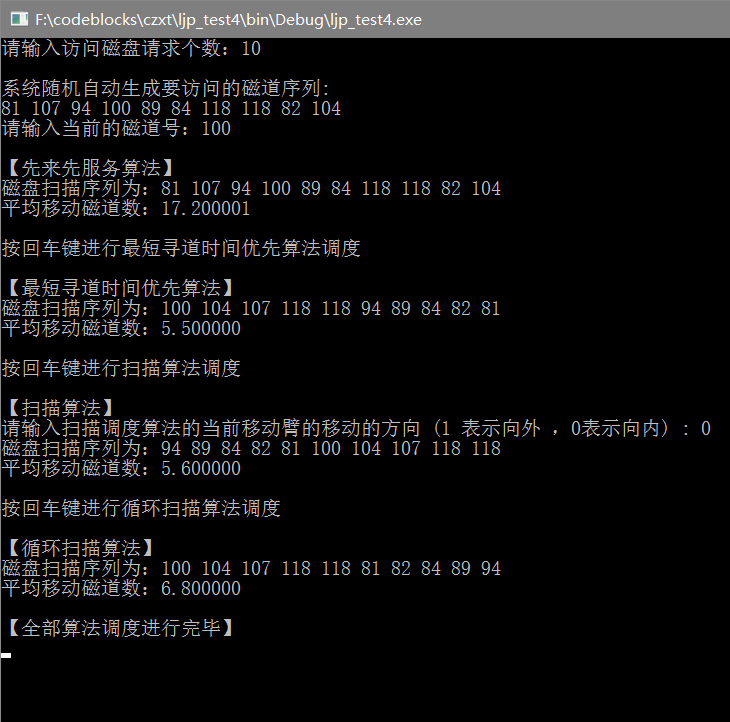
printf("\n");

printf("平均移动磁道数：%f\n\n【全部算法调度进行完毕】\n", ave);

}

## 5.测试结果





## 6.总结

FCFS算法公平、简单平均寻道距离大，仅应用在磁盘I/O较少的场合。

SSTF算法性能比“先来先服务”好不能保证平均寻道时间最短，可能出现“饥饿”现象。

SCAN算法寻道性能较好，可避免“饥饿”现象不利于远离磁头一端的访问请求。

C-SCAN算法消除了对两端磁道请求的不公平。