



# 廣東工業大學

## 实 验 报 告

课程名称 操作系统实验

学生学院 \_\_\_\_\_

专业班级 \_\_\_\_\_

学 号 \_\_\_\_\_

学生姓名 \_\_\_\_\_

2023 年 月 日



# 目录

前言 .....	1
一、课程的性质、目的和任务 .....	1
二、实验的意义和目的 .....	1
三、实验运行环境 .....	1
四、实验内容及安排 .....	1
实验 1 进程调度 .....	2
一、实验目的 .....	2
二、实验内容 .....	2
三、实验要求 .....	2
四、主要的数据结构 .....	2
五、算法流程图 .....	7
六、运行与测试（系统运行截图） .....	9
实验 2 银行家算法 .....	16
一、实验目的 .....	16
二、实验内容 .....	16
三、主要的数据结构 .....	16
四、程序流程图 .....	19
五、运行与测试（系统运行截图） .....	21
实验 3 内存管理实验 .....	23
一、实验目的 .....	23
二、实验内容 .....	23
三、相应的数据结构 .....	24
四、程序系统结构图 .....	33
五、运行与测试（系统运行截图） .....	35
六、分析、比较算法 .....	39
实验 4 磁盘调度算法 .....	41
一、实验目的 .....	41
二、实验内容 .....	41
三、分析算法性能 .....	41
四、画出实验内容 3 的性能曲线 .....	43

# 前言

## 一. 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

- 编程语言: C++语言
- 编程环境: Visual Studio 2017

## 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	

# 实验 1 进程调度

## 一、实验目的

编写并调试一个模拟的进程调度程序，加深对进程的概念及进程调度算法的理解。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。

## 二、实验内容

1. 分别采用抢占和非抢占调度方式
2. 实现“短进程优先”、“时间片轮转”、“高响应比优先”调度算法

## 三、实验要求

1. 由程序自动生成进程（最多 50 个），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。系统允许同时运行 5 个进程。
2. 当完成一个进程时，会创建一个新进程，直到进程总数达到 50 为止。
3. 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 信息。
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. 计算各调度算法的平均周转时间和带权平均周转时间。

## 四、主要的数据结构

### 1. 进程控制块 PCB 类

```
class PCB
{
private:
    std::string m_Id;
    //标识符,用于表示一个唯一的进程

    enum class State { W = 1, R, F };
    //W=Wait(就绪状态),R=Run(运行状态),F=Finish(完成状态)

    State m_state;
    //进程状态，指明进程的状态

    int m_ntime;
    //进程的要求服务时间
```

```

    int m_rtime;

    //已占用 CPU 服务时间

    int m_wtime;

    //进程已等待时间

public:
    PCB(void)

    {

        //不允许默认构造

        std::cout << "引发异常" << std::endl;

    }

    PCB(const std::string m_Id_argu , const int m_ntime_argu)

    {

        this->m_Id = std::string("");

        //先定义一个空字符串

        this->m_Id.append(m_Id_argu);

        this->m_state = PCB::State::W;

        this->m_ntime = m_ntime_argu;

        this->m_rtime = 0;

        this->m_wtime = 0;

    }

    ~PCB(void) {}

    Status IfProcessDone(void)const

    {

        return (this->m_rtime >= this->m_ntime);

    }

    //检测自身进程是否已完成运行

    friend class RunQueue;

    friend class ReadyQueue;

    //友元类

```

```

friend Status ProcessList::RunReadyProcess(PCB* p);

friend Status ProcessList::DiscontinueProcess(PCB* p);

//友元成员函数

friend std::ostream& operator<<(std::ostream& os, const ProcessList& P);

friend PCB* Non_ShortJobFirst(ReadyQueue* &Rq);

friend PCB* Non_HRRN(ReadyQueue* &Rq);

friend Status Pre_ShortJobFirst(ProcessList* &P);

friend Status Pre_HRRN(ProcessList* &P);

//友元函数

};

```

## 2. 就绪索引表类

```

class ReadyQueue
{
private:
    RdIndexType m_ReadyList;
public:
    Status AddRdProcess(PCB* p);
    //入队一个进程在就绪队列中
    Status DeleteRdProcess(PCB* p);
    //出队一个进程
    void WaitTimeAdd(void);
    //等待时间增长函数
    friend std::ostream& operator<<(std::ostream& os, const ProcessList& P);
    friend PCB* Non_ShortJobFirst(ReadyQueue* &Rq);
    friend PCB* Non_HRRN(ReadyQueue* &Rq);
    friend Status Pre_ShortJobFirst(ProcessList* &P);
    friend Status Pre_HRRN(ProcessList* &P);
    friend Status TimePieceRR(ProcessList* &P);

```

```

        friend      Status      NonPreemptive_Mode(ProcessList*      &P,      PCB*
(*ProcessSchedule)(ReadyQueue* &Rq));

        friend      Status      Preemptive_Mode(ProcessList*      &P,
Status(*Pre_ProcessSchedule)(ProcessList* &P));

        //友元函数
};

```

### 3. 执行索引表类

```

class RunQueue
{
private:
    RunIndexType m_RunList;
public:
    Status AddRunProcess(PCB* p);
    //把特定进程加入到执行队列里

    Status DeleteRunProcess(PCB* p,PCB::State S);
    //把特定进程从执行队列中移去，S 为进程的后续状态

    void RunTimeAdd(void);
    //执行时间增长函数

    inline Status FullRunQueue(void)const
    {
        return (this->m_RunList.size() == ProcessorNum);
    }
    //判断是否执行队列的进程达到最大数

    Status TurnProcessFinish(void);
    //检查执行索引表，把所有已运行完的进程移除

    friend std::ostream& operator<<(std::ostream& os, const ProcessList& P);

    friend Status Pre_ShortJobFirst(ProcessList* &P);

    friend Status Pre_HRRN(ProcessList* &P);

```



```

    friend Status TimePieceRR(ProcessList* &P);

    friend      Status      NonPreemptive_Mode(ProcessList*      &P,      PCB*
(*ProcessSchedule)(ReadyQueue* &Rq));

    friend      Status      Preemptive_Mode(ProcessList*      &P,
Status(*Pre_ProcessSchedule)(ProcessList* &P));

    //友元函数
};

```

#### 4. 进程表类

```

class ProcessList
{
private:
    RunQueue* m_Runp;
    //进程执行指针

    ReadyQueue* m_Readyp;
    //进程就绪表指针

    PCBListType m_PCBList;
    //所有 PCB 进程的列表

public:
    ProcessList(void);
    //构造函数声明

    ~ProcessList(void);
    //析构函数，需要释放内存空间

    Status CreateNewProcess(void);
    //创建新进程

    Status RunReadyProcess(PCB* p);
    //将特定的进程从就绪状态转入执行状态

    Status DiscontinueProcess(PCB* p);
    //将特定进程从执行队列中转入就绪队列(即发生了中断)

```

```

friend std::ostream& operator<<(std::ostream& os, const ProcessList& P);

//进程表显示

friend          Status          Preemptive_Mode(ProcessList*          &P,
Status(*Pre_ProcessSchedule)(ProcessList* &P));

friend          Status          NonPreemptive_Mode(ProcessList*          &P,          PCB*
(*ProcessSchedule)(ReadyQueue* &Rq));

friend Status Pre_ShortJobFirst(ProcessList* &P);

friend Status Pre_HRRN(ProcessList* &P);

friend Status TimePieceRR(ProcessList* &P);

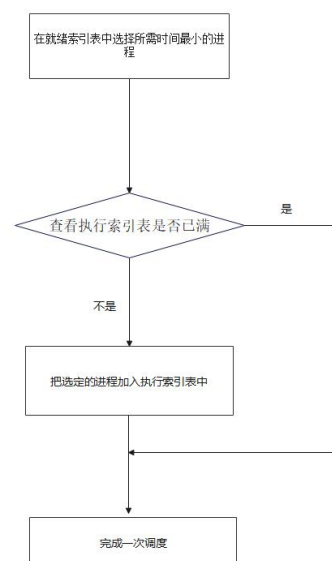
//友元函数

};

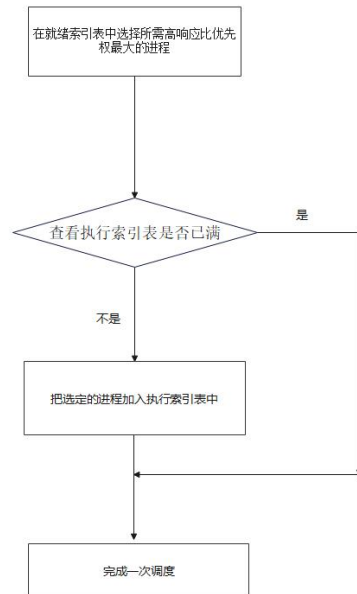
```

## 五、算法流程图

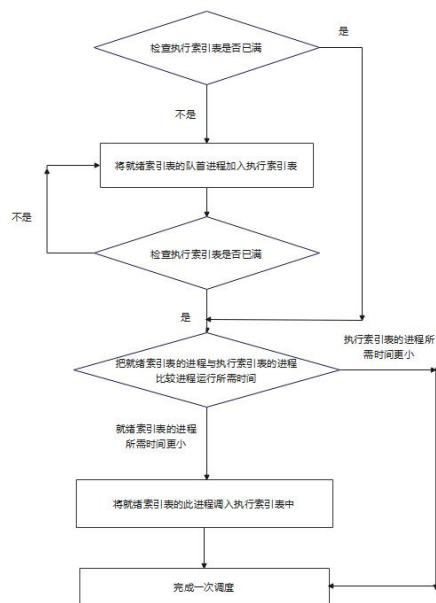
### 1. 非抢占方式的短进程优先调度算法



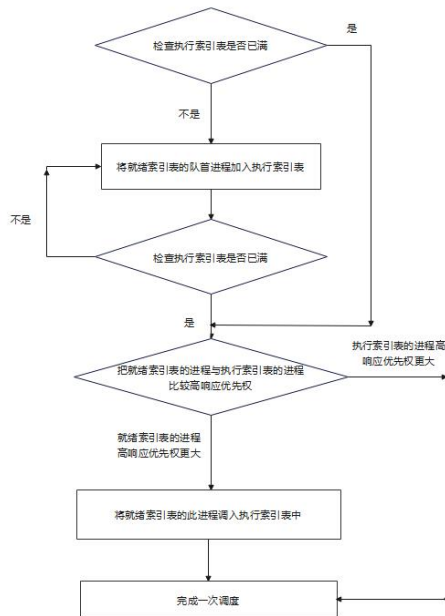
### 2. 非抢占方式的高响应比优先调度算法



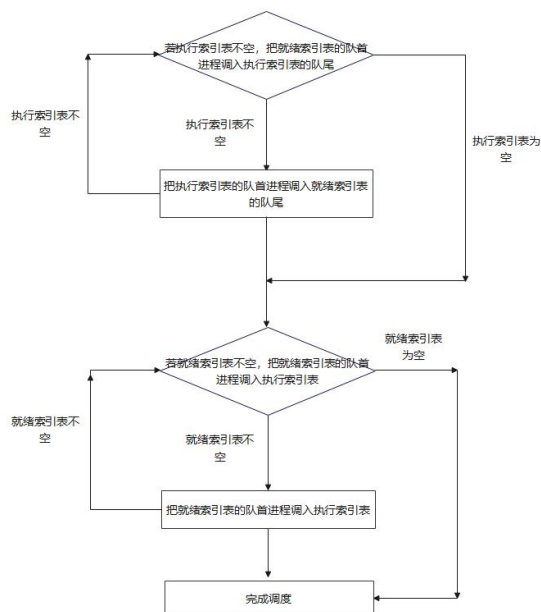
### 3. 抢占方式的短进程优先调度算法



### 4. 抢占方式的高响应比优先调度算法



## 5. 时间片轮转调度算法



## 六、运行与测试（系统运行截图）

1. 由程序自动生成进程（最多 50 个），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。系统允许同时运行 5 个进程。

```

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project_Experiment_1\Debug\Project_Experiment_1.exe
-----
第65个标准单位时间
正在运行的进程(以标识符显示进程):
所有就绪的进程(以标识符显示进程): a b c d e f g h i j k l m n o p q s t u v w x y z A B C D E F G H I J K L M N O P Q R
S T U V W X Y
所有完成的进程(以标识符显示进程):
所有进程的PCB信息:
进程标识符: a
进程状态: F(完成状态)
进程要求服务时间: 3
进程已经获得服务时间: 3
进程已等待时间: 0
进程周转时间: 3

```

```
C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project_Experiment_1\Debug\Project_Experiment_1.exe

-----
第6个标准单位时间
请输入新进程的标识符:
f
请输入新进程的要求服务时间:
10
正在运行的进程(以标识符显示进程): a b c d e
所有就绪的进程(以标识符显示进程): f
所有完成的进程(以标识符显示进程):
所有进程的PCB信息:
进程标识符: a
进程状态: R(执行状态)
进程要求服务时间: 10
进程已经获得服务时间: 7
进程已等待时间: 0
进程周转时间: 7
-----

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project_Experiment_1\Debug\Project_Experiment_1.exe

-----
第0个标准单位时间
请输入新进程的标识符:
a
请输入新进程的要求服务时间:
10
正在运行的进程(以标识符显示进程): a
所有就绪的进程(以标识符显示进程):
所有完成的进程(以标识符显示进程):
所有进程的PCB信息:
进程标识符: a
进程状态: R(执行状态)
进程要求服务时间: 10
进程已经获得服务时间: 1
进程已等待时间: 0
进程周转时间: 1
-----
```

2. 当完成一个进程时，会创建一个新进程，直到进程总数达到 50 为止。

```
C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project_Experiment_1\Debug\Project_Experiment_1.exe

-----
第1个标准单位时间
请输入新进程的标识符:
b
请输入新进程的要求服务时间:
2
正在运行的进程(以标识符显示进程): b
所有就绪的进程(以标识符显示进程):
所有完成的进程(以标识符显示进程): a
所有进程的PCB信息:
进程标识符: a
进程状态: F(完成状态)
进程要求服务时间: 2
进程已经获得服务时间: 2
进程已等待时间: 0
进程周转时间: 2

进程标识符: b
进程状态: R(执行状态)
进程要求服务时间: 2
进程已经获得服务时间: 1
进程已等待时间: 0
进程周转时间: 1
-----

第2个标准单位时间
请输入新进程的标识符:
c
```

3. 每进行一次调度，程序都要输出一一次运行结果：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 信息。

```

正在运行的进程(以标识符显示进程): c
所有就绪的进程(以标识符显示进程):
所有完成的进程(以标识符显示进程): a b
所有进程的PCB信息:
进程标识符: a
进程状态: F(完成状态)
进程要求服务时间: 2
进程已经获得服务时间: 2
进程已等待时间: 0
进程周转时间: 2

进程标识符: b
进程状态: F(完成状态)
进程要求服务时间: 2
进程已经获得服务时间: 2
进程已等待时间: 0
进程周转时间: 2

进程标识符: c
进程状态: R(执行状态)
进程要求服务时间: 2
进程已经获得服务时间: 1
进程已等待时间: 0
进程周转时间: 1

```

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。

进程标识符: a	进程标识符: d	进程标识符: i
进程状态: F(完成状态)	进程状态: R(执行状态)	进程状态: W(就绪状态)
进程要求服务时间: 2	进程要求服务时间: 10	进程要求服务时间: 10
进程已经获得服务时间: 2	进程已经获得服务时间: 8	进程已经获得服务时间: 0
进程已等待时间: 0	进程已等待时间: 0	进程已等待时间: 2
进程周转时间: 2	进程周转时间: 8	进程周转时间: 2

5. 计算各调度算法的平均周转时间和带权平均周转时间。

- i. 非抢占方式的短进程优先调度算法

进程数量为 10 的时候的平均周转时间和带权平均周转时间计算:

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project\_Experiment\_1\Debug\Project\_Experiment\_1.exe

```

第17个标准单位时间
正在运行的进程(以标识符显示进程):
所有就绪的进程(以标识符显示进程):
所有完成的进程(以标识符显示进程): a b c d e f g h i j
所有进程的PCB信息:
进程标识符: a
进程状态: F(完成状态)
进程要求服务时间: 10
进程已经获得服务时间: 10
进程已等待时间: 0
进程周转时间: 10

进程标识符: b
进程状态: F(完成状态)
进程要求服务时间: 8
进程已经获得服务时间: 8
进程已等待时间: 0
进程周转时间: 8

进程标识符: c
进程状态: F(完成状态)
进程要求服务时间: 6
进程已经获得服务时间: 6
进程已等待时间: 0
进程周转时间: 6

```

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project\_Experiment\_1\Debug\Project\_Experiment\_1.exe

```

进程标识符: d
进程状态: F(完成状态)
进程要求服务时间: 7
进程已经获得服务时间: 7
进程已等待时间: 0
进程周转时间: 7

进程标识符: e
进程状态: F(完成状态)
进程要求服务时间: 5
进程已经获得服务时间: 5
进程已等待时间: 0
进程周转时间: 5

进程标识符: f
进程状态: F(完成状态)
进程要求服务时间: 4
进程已经获得服务时间: 4
进程已等待时间: 0
进程周转时间: 4

进程标识符: g
进程状态: F(完成状态)
进程要求服务时间: 3
进程已经获得服务时间: 3
进程已等待时间: 0
进程周转时间: 3

```

```

进程标识符: h
进程状态: F(完成状态)
进程要求服务时间: 6
进程已经获得服务时间: 6
进程已等待时间: 0
进程周转时间: 10

进程标识符: i
进程状态: F(完成状态)
进程要求服务时间: 3
进程已经获得服务时间: 3
进程已等待时间: 0
进程周转时间: 3

进程标识符: j
进程状态: F(完成状态)
进程要求服务时间: 5
进程已经获得服务时间: 5
进程已等待时间: 0
进程周转时间: 6

-----
平均周转时间: 6.7
平均带权周转时间: 1.22
请按任意键继续. . .

```

ii. 非抢占方式的高响应比优先调度算法:

进程数量为 10 的时候的平均周转时间和带权平均周转时间计算:

```

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project_Experiment_1\Debug\Project_Experiment_1.exe

-----
第19个标准单位时间
正在运行的进程(以标识符显示进程):
所有就绪的进程(以标识符显示进程):
所有完成的进程(以标识符显示进程): a b c d e f g h i j
所有进程的PCB信息:
进程标识符: a
进程状态: F(完成状态)
进程要求服务时间: 10
进程已经获得服务时间: 10
进程已等待时间: 0
进程周转时间: 10

进程标识符: b
进程状态: F(完成状态)
进程要求服务时间: 9
进程已经获得服务时间: 9
进程已等待时间: 0
进程周转时间: 9

进程标识符: c
进程状态: F(完成状态)
进程要求服务时间: 8
进程已经获得服务时间: 8
进程已等待时间: 0
进程周转时间: 8

```

```

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project_Experiment_1\Debug\Project_Experiment_1.exe

进程标识符: d
进程状态: F(完成状态)
进程要求服务时间: 8
进程已经获得服务时间: 8
进程已等待时间: 0
进程周转时间: 8

进程标识符: e
进程状态: F(完成状态)
进程要求服务时间: 7
进程已经获得服务时间: 7
进程已等待时间: 0
进程周转时间: 7

进程标识符: f
进程状态: F(完成状态)
进程要求服务时间: 5
进程已经获得服务时间: 5
进程已等待时间: 0
进程周转时间: 9

进程标识符: g
进程状态: F(完成状态)
进程要求服务时间: 4
进程已经获得服务时间: 4
进程已等待时间: 0
进程周转时间: 8

```

```

进程标识符: h
进程状态: F(完成状态)
进程要求服务时间: 3
进程已经获得服务时间: 3
进程已等待时间: 0
进程周转时间: 7

进程标识符: i
进程状态: F(完成状态)
进程要求服务时间: 5
进程已经获得服务时间: 5
进程已等待时间: 0
进程周转时间: 9

进程标识符: j
进程状态: F(完成状态)
进程要求服务时间: 6
进程已经获得服务时间: 6
进程已等待时间: 0
进程周转时间: 10

-----
平均周转时间: 8.5
平均带权周转时间: 1.46
请按任意键继续. . .

```

iii. 抢占方式的短进程优先调度算法:

进程数量为 10 的时候的平均周转时间和带权平均周转时间:

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project\_Experiment\_1\Debug\Project\_Experiment\_1.exe

```
-----
第16个标准单位时间
正在运行的进程(以标识符显示进程):
所有就绪的进程(以标识符显示进程): a b c d e f g h i j
所有完成的进程(以标识符显示进程):
所有进程的PCB信息:
进程标识符: a
进程状态: F(完成状态)
进程要求服务时间: 10
进程已经获得服务时间: 10
进程已等待时间: 0
进程周转时间: 10

进程标识符: b
进程状态: F(完成状态)
进程要求服务时间: 8
进程已经获得服务时间: 8
进程已等待时间: 0
进程周转时间: 8

进程标识符: c
进程状态: F(完成状态)
进程要求服务时间: 6
进程已经获得服务时间: 6
进程已等待时间: 0
进程周转时间: 6
```

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project\_Experiment\_1\Debug\Project\_Experiment\_1.exe

```
进程标识符: d
进程状态: F(完成状态)
进程要求服务时间: 7
进程已经获得服务时间: 7
进程已等待时间: 0
进程周转时间: 7

进程标识符: e
进程状态: F(完成状态)
进程要求服务时间: 8
进程已经获得服务时间: 8
进程已等待时间: 0
进程周转时间: 12

进程标识符: f
进程状态: F(完成状态)
进程要求服务时间: 4
进程已经获得服务时间: 4
进程已等待时间: 0
进程周转时间: 4

进程标识符: g
进程状态: F(完成状态)
进程要求服务时间: 3
进程已经获得服务时间: 3
进程已等待时间: 0
进程周转时间: 4
```

```
进程标识符: h
进程状态: F(完成状态)
进程要求服务时间: 5
进程已经获得服务时间: 5
进程已等待时间: 0
进程周转时间: 7

进程标识符: i
进程状态: F(完成状态)
进程要求服务时间: 3
进程已经获得服务时间: 3
进程已等待时间: 0
进程周转时间: 3

进程标识符: j
进程状态: F(完成状态)
进程要求服务时间: 4
进程已经获得服务时间: 4
进程已等待时间: 0
进程周转时间: 4

平均周转时间: 6.5
平均带权周转时间: 1.12333
请按任意键继续. . .
```

iv. 抢占方式的高响应比优先调度算法:

进程数量为 10 的时候的平均周转时间和带权平均周转时间:

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project\_Experiment\_1\Debug\Project\_Experiment\_1.exe

```
-----
第15个标准单位时间
正在运行的进程(以标识符显示进程):
所有就绪的进程(以标识符显示进程):
所有完成的进程(以标识符显示进程): a b c d e f g h i j
所有进程的PCB信息:
进程标识符: a
进程状态: F(完成状态)
进程要求服务时间: 10
进程已经获得服务时间: 10
进程已等待时间: 0
进程周转时间: 10

进程标识符: b
进程状态: F(完成状态)
进程要求服务时间: 8
进程已经获得服务时间: 8
进程已等待时间: 0
进程周转时间: 8

进程标识符: c
进程状态: F(完成状态)
进程要求服务时间: 9
进程已经获得服务时间: 9
进程已等待时间: 0
进程周转时间: 9
```



```

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project_Experiment_1\Debug\Project_Experiment_1.exe

进程标识符: d
进程状态: F(完成状态)
进程要求服务时间: 7
进程已经获得服务时间: 7
进程已等待时间: 0
进程周转时间: 7

进程标识符: e
进程状态: F(完成状态)
进程要求服务时间: 7
进程已经获得服务时间: 7
进程已等待时间: 0
进程周转时间: 7

进程标识符: f
进程状态: F(完成状态)
进程要求服务时间: 5
进程已经获得服务时间: 5
进程已等待时间: 0
进程周转时间: 8

进程标识符: g
进程状态: F(完成状态)
进程要求服务时间: 3
进程已经获得服务时间: 3
进程已等待时间: 0
进程周转时间: 6

进程标识符: h
进程状态: F(完成状态)
进程要求服务时间: 6
进程已经获得服务时间: 6
进程已等待时间: 0
进程周转时间: 8

进程标识符: i
进程状态: F(完成状态)
进程要求服务时间: 5
进程已经获得服务时间: 5
进程已等待时间: 0
进程周转时间: 7

进程标识符: j
进程状态: F(完成状态)
进程要求服务时间: 4
进程已经获得服务时间: 4
进程已等待时间: 0
进程周转时间: 6

-----
平均周转时间: 7.6
平均带权周转时间: 1.28333
请按任意键继续. . .

```

## v. 时间片轮转调度算法:

进程数量为 10 的时候的平均周转时间和带权平均周转时间:

```

C:\Users\lenovo\Desktop\学习\操作系统实验\实验一\代码\Project_Experiment_1\Debug\Project_Experiment_1.exe

-----
第16个标准单位时间
正在运行的进程(以标识符显示进程):
所有就绪的进程(以标识符显示进程): a b c d e f g h i j
所有完成的进程(以标识符显示进程):
所有进程的PCB信息:
进程标识符: a
进程状态: F(完成状态)
进程要求服务时间: 10
进程已经获得服务时间: 10
进程已等待时间: 0
进程周转时间: 14

进程标识符: b
进程状态: F(完成状态)
进程要求服务时间: 9
进程已经获得服务时间: 9
进程已等待时间: 0
进程周转时间: 14

进程标识符: c
进程状态: F(完成状态)
进程要求服务时间: 8
进程已经获得服务时间: 8
进程已等待时间: 0
进程周转时间: 10

进程标识符: d
进程状态: F(完成状态)
进程要求服务时间: 7
进程已经获得服务时间: 7
进程已等待时间: 0
进程周转时间: 10

进程标识符: e
进程状态: F(完成状态)
进程要求服务时间: 6
进程已经获得服务时间: 6
进程已等待时间: 0
进程周转时间: 9

进程标识符: f
进程状态: F(完成状态)
进程要求服务时间: 3
进程已经获得服务时间: 3
进程已等待时间: 0
进程周转时间: 3

进程标识符: g
进程状态: F(完成状态)
进程要求服务时间: 4
进程已经获得服务时间: 4
进程已等待时间: 0
进程周转时间: 7

```

```
进程标识符: h
进程状态: F(完成状态)
进程要求服务时间: 5
进程已经获得服务时间: 5
进程已等待时间: 0
进程周转时间: 7

进程标识符: i
进程状态: F(完成状态)
进程要求服务时间: 3
进程已经获得服务时间: 3
进程已等待时间: 0
进程周转时间: 6

进程标识符: j
进程状态: F(完成状态)
进程要求服务时间: 5
进程已经获得服务时间: 5
进程已等待时间: 0
进程周转时间: 7

-----
平均周转时间: 8.7
平均带权周转时间: 1.46841
请按任意键继续. . .
```

## 实验 2 银行家算法

### 一、实验目的

通过实验，加深理解死锁问题。

### 二、实验内容

1. 假定系统有 3 类资源 A（10 个）、B（15 个）、C（8 个），系统最多有 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. 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。
4. 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。
5. 一个进程执行完成后释放资源，并唤醒相应的阻塞进程，同时会随机创建一个新进程。
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。

### 三、主要的数据结构

#### 1. 系统资源类

```
//系统资源
class Resource
{
public:
    int A_num;
    //A 类资源的可用数量
    int B_num;
    //B 类资源的可用数量
    int C_num;
    //C 类资源的可用数量
    static const unsigned int A = 0;
    static const unsigned int B = 1;
    static const unsigned int C = 2;
    //指定 A、B、C 类资源在数组中的索引
    static const unsigned int A_maxnum = 10;
    static const unsigned int B_maxnum = 15;
    static const unsigned int C_maxnum = 8;
    //定义 A、B、C 类资源的最大可用数
```

```

Resource():A_num(A_maxnum),B_num(B_maxnum),C_num(C_maxnum)
{
//初始化资源数目
~Resource(){}
};

```

## 2. 进程表类

```

//进程表类
class ProcessList
{
private:
    ProcessListType m_PCBList;
    //进程线性表,采用索引方式组织 PCB

    ReadyListType* m_ReadyL;
    //就绪队列

    RunListType* m_RunL;
    //执行队列,实际上只有 1 个处理机

    BlockListType* m_BlockL;
    //阻塞队列

    Resource m_R;
    //进程资源池

public:
    //构造函数
    ProcessList(void)
    {
        m_PCBList.reserve(ProcessMaxNum);
        //预留足够的位置

        m_ReadyL = new ReadyListType;
        m_RunL = new RunListType;
        m_BlockL = new BlockListType;
        //分配内存
    }

    //析构函数
    ~ProcessList()
    {
        delete m_ReadyL;
        delete m_RunL;
        delete m_BlockL;
        //释放内存
    }

    Status CreatingProcess(void);
    //创建一个进程

    Status RunToReady(PCB* &p);
    //把特定进程从执行队列转入就绪队列中

```

```

    Status ReadyToRun(PCB* &p, const ResourceType Request);
    //把特定进程从就绪队列转入执行队列中

    Status ReadyToBlock(const PCB* p);
    //把特定进程从就绪状态转入阻塞状态

    Status CheckUpFinish(void);
    //检查执行进程是否达到完成标志

    Status WakeUpBlock(void);
    //把阻塞队列的队首进程转入到就绪队列中

    Status RoundRobin(void);
    //时间片轮转调度算法

    Status Banker(PCB* &p, ResourceType& Request);
    //银行家算法

    Status Security_algo(PCB* & p,const ResourceType& Available, int** Need, int**
Allocation, std::vector<PCB*>& IndexList);
    //安全性算法

    Status ProcessScheduling(void);
    //进程调度函数

    friend std::ostream& operator<<(std::ostream& os, ProcessList& P);
    //友元函数

};

```

### 3. 进程控制块 PCB 类

```

//进程控制块 PCB 类
class PCB
{
public:
    enum class State
    {
        W = 1,
        //Wait,就绪状态

        R,
        //Run, 运行状态

        B,
        //Block, 阻塞状态

        F
        //Finish, 完成状态
    };

private:
    State m_state;
    //进程状态

    std::string m_Id;
    //进程标识符(进程名)

    ResourceType m_maxneedR;
    //需要的资源总数

```

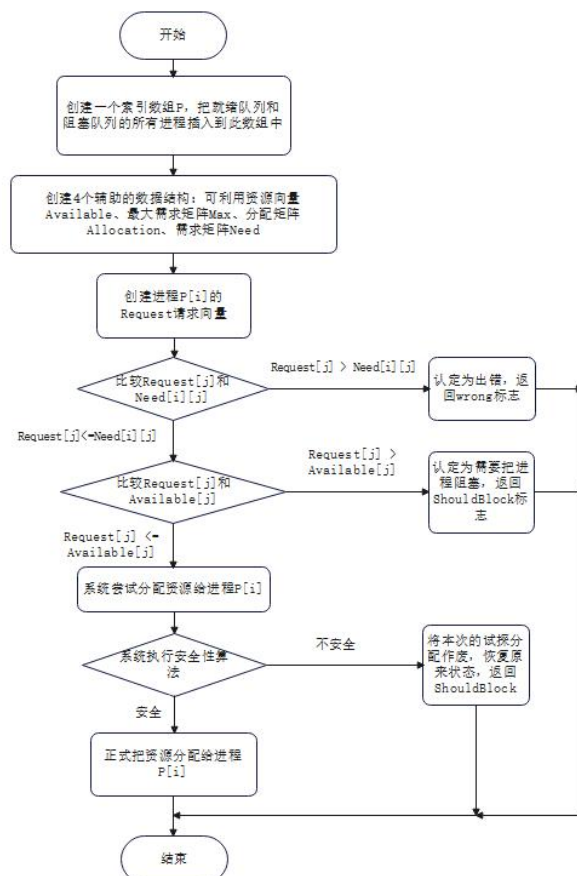
```

    ResourceType m_hadR;
    //已分配的资源数
public:
    PCB(void)
    {
        std::cout << "引发异常" << std::endl;
        //不允许默认构造
    }
    PCB(const std::string Id_argu)
    {
        m_Id = std::string("");
        m_Id.append(Id_argu);
        //初始化进程标识符
        m_state = State::W;
        //将状态设置为就绪状态
        m_hadR[Resource::A] = m_hadR[Resource::B]
        = m_hadR[Resource::C] = 0;
        //初始化已分配资源
        static std::default_random_engine e;
        //随机数引擎
        m_maxneedR[Resource::A] = e() % (Resource::A_maxnum + 1);
        m_maxneedR[Resource::B] = e() % (Resource::B_maxnum + 1);
        m_maxneedR[Resource::C] = e() % (Resource::C_maxnum + 1);
        //随机申请资源
    }
    ~PCB(void){}
    Status IfFinish(void)const;
    //判断进程是否达到完成条件
    friend class ProcessList;
    //友元类
    friend std::ostream& operator<<(std::ostream& os, ProcessList& P);
    friend int GetListIndex(PCB* &p, std::vector<PCB*>& List);
    //友元函数
};

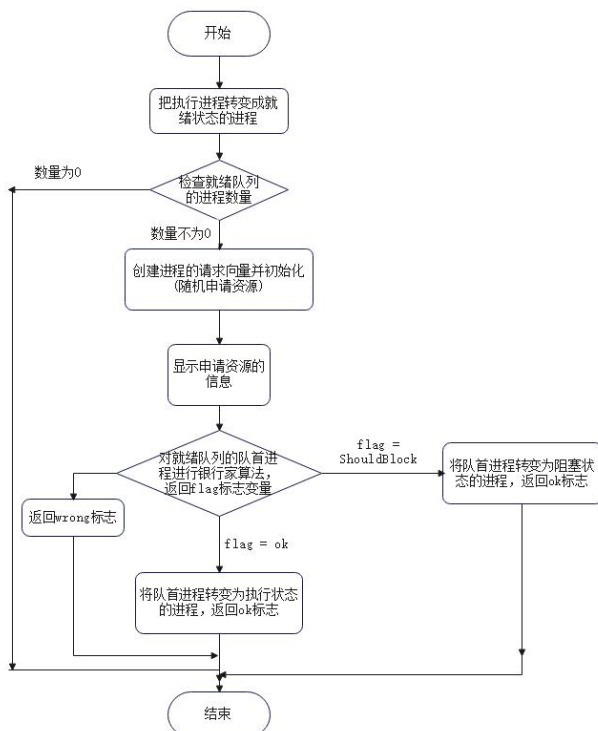
```

#### 四、程序流程图

##### 1. 银行家算法流程图



## 2. 时间片轮转调度算法流程图



## 五、运行与测试（系统运行截图）

- 1) 假定系统有 3 类资源 A（10 个）、B（15 个）、C（8 个），系统最多有 5 个进程并发执行，进程调度采用时间片轮转调度算法。

```
-----
第0个标准时间
请输入创建进程的标识符: a
a进程请求资源: (A:1 B:0 C:2)
系统中的资源可用数: (A:9 B:15 C:6)
正在运行的进程(以标识符显示): a
就绪队列中的进程(以标识符显示):
阻塞队列中的进程(以标识符显示):
完成的进程(以标识符显示):
所有进程的PCB信息:
进程标识符: a
进程状态: R(运行状态)
最大需要的资源总数:
A:6 B:6 C:2
已经获得的资源总数:
A:1 B:0 C:2
-----
```

```
-----
第6个标准时间
a进程请求资源: (A:1 B:6 C:0)
系统中的资源可用数: (A:4 B:8 C:5)
正在运行的进程(以标识符显示):
就绪队列中的进程(以标识符显示): d e b
阻塞队列中的进程(以标识符显示): c a
完成的进程(以标识符显示):
所有进程的PCB信息:
进程标识符: a
进程状态: B(阻塞状态)
最大需要的资源总数:
A:6 B:6 C:2
已经获得的资源总数:
A:1 B:0 C:2

进程标识符: b
进程状态: W(就绪状态)
最大需要的资源总数:
A:9 B:12 C:1
已经获得的资源总数:
A:5 B:7 C:1
-----
```

```
-----
第8个标准时间
d进程请求资源: (A:0 B:2 C:5)
系统中的资源可用数: (A:4 B:6 C:0)
正在运行的进程(以标识符显示): d
就绪队列中的进程(以标识符显示): e b
阻塞队列中的进程(以标识符显示): c a
完成的进程(以标识符显示):
所有进程的PCB信息:
进程标识符: a
进程状态: B(阻塞状态)
最大需要的资源总数:
A:6 B:6 C:2
已经获得的资源总数:
A:1 B:0 C:2

进程标识符: b
进程状态: W(就绪状态)
最大需要的资源总数:
A:9 B:12 C:1
已经获得的资源总数:
A:5 B:7 C:1
-----
```

- 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。

```
-----
进程标识符: b
进程状态: W(就绪状态)
最大需要的资源总数:
A:9 B:12 C:1
已经获得的资源总数:
A:5 B:7 C:1
-----
```

- 3) 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。

```
-----
第6个标准时间
a进程请求资源: (A:1 B:6 C:0)
-----
```

```
-----
第10个标准时间
e进程请求资源: (A:0 B:2 C:4)
-----
```

```
-----
第14个标准时间
d进程请求资源: (A:4 B:1 C:0)
-----
```



```

第21个标准时间
系统中的资源可用数: (A:2 B:1 C:0)
正在运行的进程(以标识符显示): b
就绪队列中的进程(以标识符显示):
阻塞队列中的进程(以标识符显示): c a e d
完成的进程(以标识符显示):
所有进程的PCB信息:
进程标识符: a
进程状态: B(阻塞状态)
最大需要的资源总数:
A:6 B:6 C:2
已经获得的资源总数:
A:1 B:0 C:2

进程标识符: b
进程状态: R(运行状态)
最大需要的资源总数:
A:9 B:12 C:1
已经获得的资源总数:
A:7 B:12 C:1

```

```

第22个标准时间
b进程请求资源: (A:2 B:0 C:0)
请输入创建进程的标识符: f
系统中的资源可用数: (A:9 B:13 C:1)
正在运行的进程(以标识符显示):
就绪队列中的进程(以标识符显示): f c
阻塞队列中的进程(以标识符显示): a e d
完成的进程(以标识符显示): b
所有进程的PCB信息:
进程标识符: a
进程状态: B(阻塞状态)
最大需要的资源总数:
A:6 B:6 C:2
已经获得的资源总数:
A:1 B:0 C:2

进程标识符: b
进程状态: F(完成状态)
最大需要的资源总数:
A:9 B:12 C:1
已经获得的资源总数:
A:0 B:0 C:0

```

```

第12个标准时间
b进程请求资源: (A:2 B:4 C:0)
系统中的资源可用数: (A:2 B:2 C:0)
正在运行的进程(以标识符显示): b
就绪队列中的进程(以标识符显示): d
阻塞队列中的进程(以标识符显示): c a e
完成的进程(以标识符显示):
所有进程的PCB信息:
进程标识符: a
进程状态: B(阻塞状态)
最大需要的资源总数:
A:6 B:6 C:2
已经获得的资源总数:
A:1 B:0 C:2

进程标识符: b
进程状态: R(运行状态)
最大需要的资源总数:
A:9 B:12 C:1
已经获得的资源总数:
A:7 B:11 C:1

```

```

第13个标准时间
系统中的资源可用数: (A:2 B:2 C:0)
正在运行的进程(以标识符显示): b
就绪队列中的进程(以标识符显示): d
阻塞队列中的进程(以标识符显示): c a e
完成的进程(以标识符显示):
所有进程的PCB信息:
进程标识符: a
进程状态: B(阻塞状态)
最大需要的资源总数:
A:6 B:6 C:2
已经获得的资源总数:
A:1 B:0 C:2

进程标识符: b
进程状态: R(运行状态)
最大需要的资源总数:
A:9 B:12 C:1
已经获得的资源总数:
A:7 B:11 C:1

```

```

第14个标准时间
d进程请求资源: (A:4 B:1 C:0)
系统中的资源可用数: (A:2 B:2 C:0)
正在运行的进程(以标识符显示):
就绪队列中的进程(以标识符显示): b
阻塞队列中的进程(以标识符显示): c a e d
完成的进程(以标识符显示):
所有进程的PCB信息:
进程标识符: a
进程状态: B(阻塞状态)
最大需要的资源总数:
A:6 B:6 C:2
已经获得的资源总数:
A:1 B:0 C:2

进程标识符: b
进程状态: W(就绪状态)
最大需要的资源总数:
A:9 B:12 C:1
已经获得的资源总数:
A:7 B:11 C:1

```

- 4) 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。

```

进程标识符: b
进程状态: R(运行状态)

```

```

进程标识符: c
进程状态: B(阻塞状态)

```

```

进程标识符: f
进程状态: W(就绪状态)

```

```

进程标识符: a
进程状态: F(完成状态)

```

- 5) 一个进程执行完成后释放资源，并唤醒相应的阻塞进程，同时会随机创建一个新进程。

```

第21个标准时间
系统中的资源可用数: (A:2 B:1 C:0)
正在运行的进程(以标识符显示): b
就绪队列中的进程(以标识符显示):
阻塞队列中的进程(以标识符显示): c a e d
完成的进程(以标识符显示):

```

```

第22个标准时间
b进程请求资源: (A:2 B:0 C:0)
请输入创建进程的标识符: f
系统中的资源可用数: (A:9 B:13 C:1)
正在运行的进程(以标识符显示):
就绪队列中的进程(以标识符显示): f c
阻塞队列中的进程(以标识符显示): a e d
完成的进程(以标识符显示): b

```

- 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。

```

第14个标准时间
d进程请求资源: (A:4 B:1 C:0)
系统中的资源可用数: (A:2 B:2 C:0)
正在运行的进程(以标识符显示):
就绪队列中的进程(以标识符显示): b
阻塞队列中的进程(以标识符显示): c a e d
完成的进程(以标识符显示):
所有进程的PCB信息:
进程标识符: a
进程状态: B(阻塞状态)
最大需要的资源总数:
A:6 B:6 C:2
已经获得的资源总数:
A:1 B:0 C:2

进程标识符: b
进程状态: W(就绪状态)
最大需要的资源总数:
A:9 B:12 C:1
已经获得的资源总数:
A:7 B:11 C:1

```

```

进程标识符: c
进程状态: B(阻塞状态)
最大需要的资源总数:
A:0 B:1 C:6
已经获得的资源总数:
A:0 B:0 C:0

进程标识符: d
进程状态: B(阻塞状态)
最大需要的资源总数:
A:4 B:3 C:5
已经获得的资源总数:
A:0 B:2 C:5

进程标识符: e
进程状态: B(阻塞状态)
最大需要的资源总数:
A:3 B:10 C:5
已经获得的资源总数:
A:0 B:0 C:0

```

## 实验3 内存管理实验

### 一、实验目的

通过实验，加深理解动态分区、基本分页和基本分段三种内存管理方式的原理。

### 二、实验内容

1. 创建一个模拟用户内存空间（1MB），在这个空间进行内存管理。
2. 根据不同算法创建相应的作业队列（20个作业，构建作业控制块，包含必要的信息）。
3. 按照先来先服务算法为这些作业分配内存，当作业完成后回收内存。
4. 对于动态分区管理方法，实现首次适应算法（其他算法选做）；对于基本分段管理方法，采用最佳适应算法分配内存；对于基本分页管理方法，按地址顺序分配内存块。（自选是否进行“紧凑”）
5. 每当有作业进入内存或释放内存，画出内存状态图（作业分配内存情况，空闲内存）。

6.内存最多允许同时运行 5 个作业，用时间片轮转算法运行作业。

### 三、相应的数据结构

#### 1. 动态分区管理方式(首次适应算法)

##### 1.1 空闲分区表项类

```
//动态分区管理(空闲分区表项类)
struct IdleListElemType
{
    int m_num;
    //分区号
    int m_size;
    //分区大小
    int m_OgAdr;
    //分区始址
    enum class State
    {
        F = 1,
        //空闲状态
        O
        //占用状态
    };
    IdleListElemType::State m_state;
    //分区状态
};
```

##### 1.2 作业调度器类

```
//作业调度器类
class JobScheduler
{
private:
    QueueType* m_BackupQueue;
    //后备队列
    QueueType* m_RunQueue;
    //运行队列
    ListType m_JobList;
    //作业数组
    IdleListType* m_ML;
    //动态分区空间分区表
public:
    JobScheduler(void)
    {
        m_BackupQueue = new QueueType;
        m_RunQueue = new QueueType;
```

```

        m_JobList.reserve(JobMaxNum);
        m_ML = new IdleListType;
        m_ML->reserve(MemorySize / PartitionSize);
    }
    //构造函数
    ~JobScheduler(void)
    {
        delete m_BackupQueue;
        delete m_RunQueue;
        delete m_ML;
    }
    //析构函数

    Status Init_ML(void);
    //初始化空闲分区表

    Status CreatingJob(void);
    //创建作业函数

    Status AllocateMemory(JCB* &j);
    //分配内存

    Status DeallocateMemory(JCB* &j);
    //回收内存

    void RunTimeIncreasing(void);
    //执行时间增长函数

    Status ReadyToRun(JCB* &j);
    //把就绪作业转入执行状态

    Status RunToReady(JCB* &j);
    //把执行作业转入就绪状态

    Status TurnToFinish(void);
    //检查作业是否完成

    friend void JobScheduling_DM(JobScheduler* &J);
    friend Status Round_Robin_DM(JobScheduler* &J);
    friend std::ostream& operator<<(std::ostream& os, IdleListType* ML);
    friend std::ostream& operator<<(std::ostream& os, QueueType* RunQueue);
    //友元函数
};

```

### 1.3 作业控制块类

```
//JCB 作业控制块类
class JCB
{
public:
    enum class State
    {
        W = 1,
        //就绪状态
        F,
        //完成状态
        R
        //运行状态
    };
private:
    int m_ntime;
    //作业还需运行的时间
    std::string m_Id;
    //作业标识符
    JCB::State m_state;
    //作业状态
    int m_PartGetAdr;
    //获得的分区首址；若没有，那么为-1
public:
    JCB(void)
    {
        std::cout << "引发异常" << std::endl;
        //不允许默认构造
    }
    JCB(const std::string Id_argu)
    {
        m_ntime = JobRunTime;
        m_Id = std::string("");
        m_Id += Id_argu;
        m_state = JCB::State::W;
        m_PartGetAdr = -1;
        //初始化成员变量
    }
    ~JCB(void){}
    inline Status IfFinish(void)
    {
        return (m_ntime <= 0);
    }
    //检查作业是否已经完成
```

```

friend class JobScheduler;
//友元类
friend std::ostream& operator<<(std::ostream& os, QueueType* RunQueue);
};

```

## 2. 分页存储管理方式(地址顺序算法)

### 2.1 作业调度器类

```

//作业调度器类
class JobScheduler
{
private:
    QueueType* m_BackUpQueue;
    //后备队列
    QueueType* m_RunQueue;
    //执行队列
    ListType m_JobList;
    //作业数组
    MemoryList* m_MS;
    //内存状态数组
    //PTRType* m_PTR;
    ///页表寄存器 PTR
public:
    //默认构造
    JobScheduler(void)
    {
        m_BackUpQueue = new QueueType;
        m_RunQueue = new QueueType;
        m_JobList.reserve(JobMaxNum);
        m_MS = new MemoryList;
        //m_PTR = new PTRType;
        for (auto i = m_MS->begin(); i != m_MS->end(); i++)
        {
            *i = Free;
        }
    }

    //析构
    ~JobScheduler(void)
    {
        delete m_BackUpQueue;
        delete m_RunQueue;
        delete m_MS;
        //delete m_PTR;
    }
}

```

```

//创建作业，并加入到后备队列
Status JobCreating(void);

//回收内存
Status DeAllocateMemory(JCB* &j);

//把 j 作业从执行状态转为就绪状态
Status RunToReady(JCB* &j);

//把特定作业从就绪状态转入执行状态
Status ReadyToRun(JCB* &j);

Status AllocateMemory(JCB* &j);
//为作业 j 分配内存

void RunTimeIncrease(void);
//运行时间增长

void FindHasFinish(void);
//把完成的作业转入完成状态

//友元函数
friend Status RobinRound_Paging(JobScheduler* &J);
friend Status JobScheduling_Paging(JobScheduler* &J);
friend std::ostream& operator<<(std::ostream& os, Paging::JobScheduler*
&J);
};

```

## 2.2 作业控制块类

```

//作业控制块类
class JCB
{
public:
    enum class State
    {
        W = 1,
        //就绪状态
        F,
        //完成状态
        R
        //运行状态
    };
private:

```

```

std::string m_Id;
//作业标识符

int m_ntime;
//作业还需运行的时间

JCB::State m_state;
//作业状态

PTHeaderType m_pt;
//抽象页表,first 是页表始址, second 是页表长度

public:
//默认构造
JCB(void)
{
    std::cout << "引发异常" << std::endl;
    //不允许默认构造
}

//带标识符的参数构造
JCB(const std::string Id_argu)
{
    m_Id = std::string("");
    m_Id += Id_argu;
    //标识符
    m_ntime = JobRunTime;
    //运行时间
    m_state = State::W;
    //作业状态
    m_pt.first = new PageTableType;
    m_pt.second = (JobSize / Sectionsize); //页表长度
    //页表头初始化
    for (int i = 0; i < m_pt.second; i++)
    {
        (*(m_pt.first))[i] = -1;
    }
    //初始化获得的块号
}

//复制构造函数, 因为要深拷贝(在 CreatingJob 函数的 vector 的 push_back)
JCB(const JCB& j)
{
    m_Id = std::string("");
    m_Id += j.m_Id;
    m_ntime = j.m_ntime;
    m_state = j.m_state;
    m_pt.first = new PageTableType;

```



```

        m_pt.second = j.m_pt.second;
        for (int i = 0; i < m_pt.second; i++)
        {
            (*(m_pt.first))[i] = (*(j.m_pt.first))[i];
        }
    }

    //析构
    ~JCB(void)
    {
        delete m_pt.first;
        m_pt.first = nullptr;
        //释放空间
    }

    //检查是否作业完成
    inline Status IfFinish(void) const
    {
        return (m_ntime <= 0);
    }

    friend class JobScheduler;
    //友元类

    friend std::ostream& Paging::operator<<(std::ostream& os, Paging::JobScheduler* &J);
};

```

### 3. 基本分段存储管理(最佳适应算法)

#### 3.1 空闲分区链结点类

```

//空闲分区链结点类
struct Node
{
    int m_id;
    //分区号
    int m_size;
    //分区大小/KB
    int m_addr;
    //分区始址
    Node* front , *next;
    //前、后向指针
};

```

#### 3.2 作业调度器类

```

//作业调度器类

```

```

class JobScheduler
{
private:
    QueueType* m_BackUpQueue;
    //后备队列
    QueueType* m_RunQueue;
    //执行队列
    ListType m_JCBList;
    //JCB 列表
    Node* m_head;
    //空闲分区链的头结点
public:
    //默认构造
    JobScheduler(void);

    //析构
    ~JobScheduler(void);

    //创建作业，并加入到后备队列
    Status JobCreating(void);

    //回收内存
    Status DeAllocateMemory(JCB* &j);

    //把 j 作业从执行状态转为就绪状态
    Status RunToReady(JCB* &j);

    //把特定作业从就绪状态转入执行状态
    Status ReadyToRun(JCB* &j);

    Status AllocateMemory(JCB* &j);
    //为作业 j 分配内存

    void RunTimeIncrease(void);
    //运行时间增长

    void FindHasFinish(void);
    //把完成的作业转入完成状态

    //友元函数
    friend Status RobinRound_Segment(JobScheduler* &J);
    friend Status JobScheduling_Segment(JobScheduler* &J);
    friend      std::ostream&      operator<<(std::ostream&      os,
Segment::JobScheduler* &J);

```

```
};
```

### 3.3 作业控制块类

```
//作业控制块类
class JCB
{
public:
    enum class State
    {
        W = 1,
        //就绪
        F,
        //完成
        R
        //执行
    };
private:
    std::string m_Id;
    //作业标识符
    State m_state;
    //作业状态
    Node* m_node;
    //获得的分区链节点值
    int m_ntime;
    //还需运行的时间
public:
    //默认构造
    JCB(void)
    {
        std::cout << "引发异常" << std::endl;
    }
    //构造
    JCB(std::string Id_argu);
    //析构
    ~JCB(void);

    //检查是否作业完成
    inline Status IfFinish(void)const
    {
        return (m_ntime <= 0);
    }

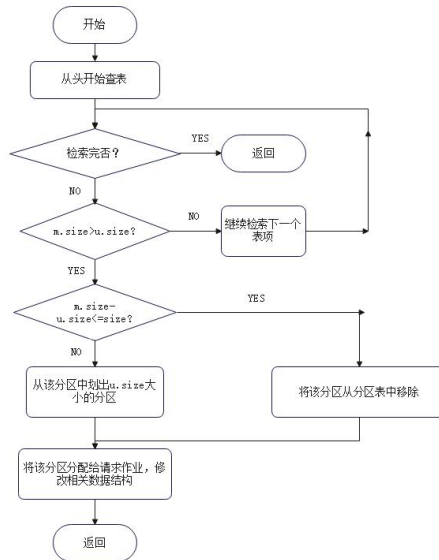
    friend class JobScheduler;
    //友元类
```

```
friend std::ostream& operator<<(std::ostream& os, Segment::JobScheduler* &J);
};
```

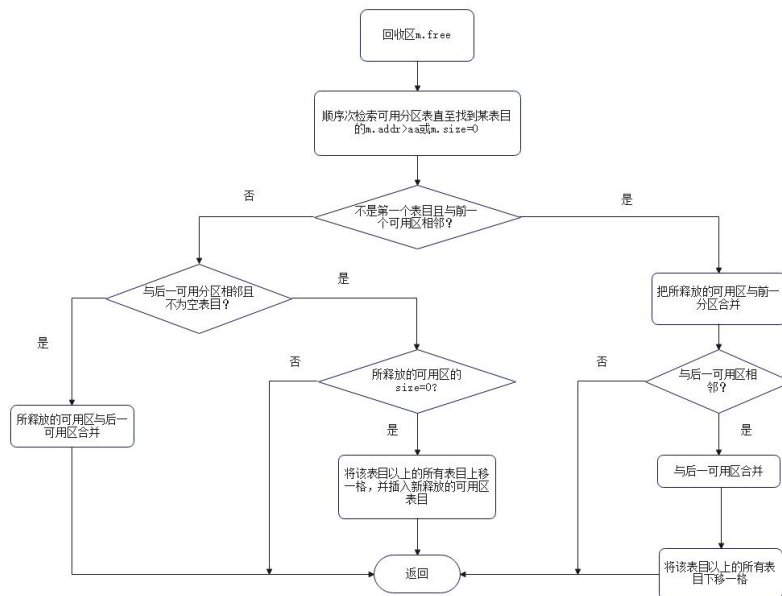
## 四、程序系统结构图

### 1. 动态分区管理

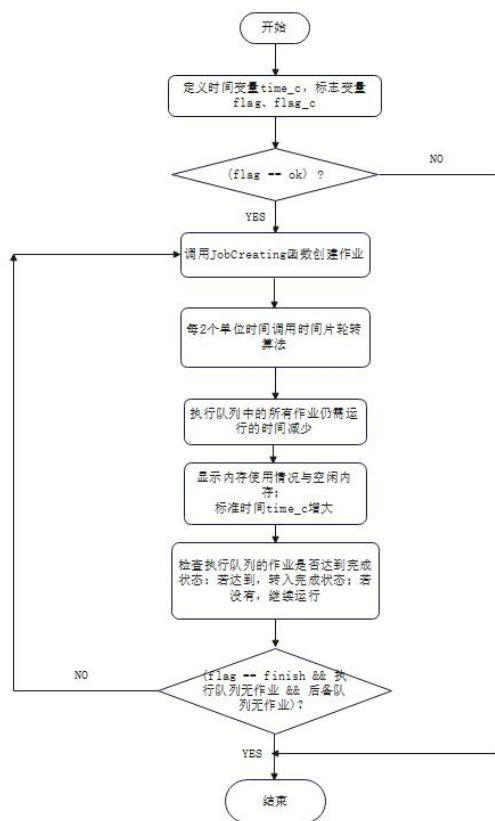
#### 1.1 分配内存流程图



#### 1.2 回收内存流程图

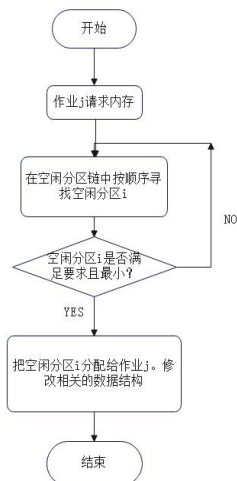


### 2. 分页存储管理

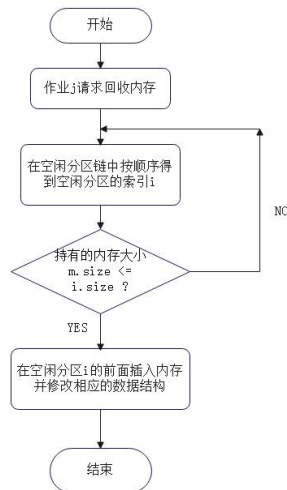


### 3. 分段存储管理

#### 3.1 分配内存(最佳适应算法)流程图



#### 3.2 回收内存(最佳适应算法)流程图



## 五、运行与测试（系统运行截图）

1.创建一个模拟用户内存空间（1MB），在这个空间进行内存管理。

```

分区号：32
分区大小：32
分区始址：992
分区状态：F(空闲状态)
  
```

2.根据不同算法创建相应的作业队列（20 个作业，构建作业控制块，包含必要的信息）。

第0个标准时间 请输入创建的作业标识符：a	第1个标准时间 请输入创建的作业标识符：b
--------------------------	--------------------------

第2个标准时间 请输入创建的作业标识符：c	第3个标准时间 请输入创建的作业标识符：d
--------------------------	--------------------------

第4个标准时间 请输入创建的作业标识符：e	第5个标准时间 请输入创建的作业标识符：f
--------------------------	--------------------------

第6个标准时间 请输入创建的作业标识符：g	第7个标准时间 请输入创建的作业标识符：h
--------------------------	--------------------------

第8个标准时间 请输入创建的作业标识符：i	第9个标准时间 请输入创建的作业标识符：j
--------------------------	--------------------------

第10个标准时间 请输入创建的作业标识符：k	第11个标准时间 请输入创建的作业标识符：l
---------------------------	---------------------------

第12个标准时间 请输入创建的作业标识符：m	第13个标准时间 请输入创建的作业标识符：n
---------------------------	---------------------------

第14个标准时间 请输入创建的作业标识符： o	第15个标准时间 请输入创建的作业标识符： p
第16个标准时间 请输入创建的作业标识符： q	第17个标准时间 请输入创建的作业标识符： r
第18个标准时间 请输入创建的作业标识符： s	第19个标准时间 请输入创建的作业标识符： t

4.对于动态分区管理方法，实现首次适应算法；对于基本分段管理方法，采用最佳适应算法分配内存；对于基本分页管理方法，按地址顺序分配内存块。（自选是否进行“紧凑”）

4.1 对于动态分区管理方法，实现首次适应算法；采用空闲分区表结构用以描述空闲分区和已分配分区的情况；

空闲内存：	分区使用情况：
分区号： 1	分区始址： 128
分区大小： 12	分区大小： 20
分区始址： 20	分区状态： 0(占用状态)
分区状态： F(空闲状态)	

4.2 对于基本分页管理方法，按地址顺序分配内存块。

作业标识符： f	作业标识符： g
作业页表：	作业页表：
页号0———段号0	页号0———段号10
页号1———段号1	页号1———段号11
页号2———段号2	页号2———段号12
页号3———段号3	页号3———段号13
页号4———段号4	页号4———段号14
页号5———段号5	页号5———段号15
页号6———段号6	页号6———段号16
页号7———段号7	页号7———段号17
页号8———段号8	页号8———段号18
页号9———段号9	页号9———段号19

4.3 对于基本分段管理方法，采用最佳适应算法分配内存；

作业标识符： c  
作业状态： R(执行状态)  
作业占用内存情况：  
段号： c1 ， 段长： 5KB  
段号： c2 ， 段长： 15KB

5.每当有作业进入内存或释放内存，画出内存状态图（作业分配内存情况，空闲内存）。

5.1 动态分区管理



空闲内存:	分区号: 6	分区号: 12	分区号: 18
分区号: 1	分区大小: 32	分区大小: 32	分区大小: 32
分区大小: 12	分区始址: 160	分区始址: 352	分区始址: 544
分区始址: 20	分区状态: F(空闲状态)	分区状态: F(空闲状态)	分区状态: F(空闲状态)
分区状态: F(空闲状态)	分区号: 7	分区号: 13	分区号: 19
分区号: 2	分区大小: 32	分区大小: 32	分区大小: 32
分区大小: 12	分区始址: 192	分区始址: 384	分区始址: 576
分区始址: 52	分区状态: F(空闲状态)	分区状态: F(空闲状态)	分区状态: F(空闲状态)
分区状态: F(空闲状态)	分区号: 8	分区号: 14	分区号: 20
分区号: 3	分区大小: 32	分区大小: 32	分区大小: 32
分区大小: 12	分区始址: 224	分区始址: 416	分区始址: 608
分区始址: 84	分区状态: F(空闲状态)	分区状态: F(空闲状态)	分区状态: F(空闲状态)
分区状态: F(空闲状态)	分区号: 9	分区号: 15	分区号: 21
分区号: 4	分区大小: 32	分区大小: 32	分区大小: 32
分区大小: 12	分区始址: 256	分区始址: 448	分区始址: 640
分区始址: 116	分区状态: F(空闲状态)	分区状态: F(空闲状态)	分区状态: F(空闲状态)
分区状态: F(空闲状态)	分区号: 10	分区号: 16	分区号: 22
分区号: 5	分区大小: 32	分区大小: 32	分区大小: 32
分区大小: 32	分区始址: 288	分区始址: 480	分区始址: 672
分区始址: 128	分区状态: F(空闲状态)	分区状态: F(空闲状态)	分区状态: F(空闲状态)
分区状态: F(空闲状态)	分区号: 11	分区号: 17	分区号: 23
	分区大小: 32	分区大小: 32	分区大小: 32
	分区始址: 320	分区始址: 512	分区始址: 704
	分区状态: F(空闲状态)	分区状态: F(空闲状态)	分区状态: F(空闲状态)

分区号: 24	分区号: 30
分区大小: 32	分区大小: 32
分区始址: 736	分区始址: 928
分区状态: F(空闲状态)	分区状态: F(空闲状态)
分区号: 25	分区号: 31
分区大小: 32	分区大小: 32
分区始址: 768	分区始址: 960
分区状态: F(空闲状态)	分区状态: F(空闲状态)
分区号: 26	分区号: 32
分区大小: 32	分区大小: 32
分区始址: 800	分区始址: 992
分区状态: F(空闲状态)	分区状态: F(空闲状态)
分区号: 27	分区使用情况:
分区大小: 32	分区始址: 32
分区始址: 832	分区大小: 20
分区状态: F(空闲状态)	分区状态: 0(占用状态)
分区号: 28	分区始址: 64
分区大小: 32	分区大小: 20
分区始址: 864	分区状态: 0(占用状态)
分区状态: F(空闲状态)	
分区号: 29	分区始址: 96
分区大小: 32	分区大小: 20
分区始址: 896	分区状态: 0(占用状态)
分区状态: F(空闲状态)	

分区始址: 0  
分区大小: 20  
分区状态: 0(占用状态)

## 5.2 基本分页存储管理



内存使用情况:	页号5-----段号15	段号42 段号483
作业标识符: b	页号6-----段号16	段号43 段号484
作业页表:	页号7-----段号17	段号44 段号485
页号0-----段号0	页号8-----段号18	段号45 段号486
页号1-----段号1	页号9-----段号19	段号46 段号487
页号2-----段号2	作业标识符: a	段号47 段号488
页号3-----段号3	作业页表:	段号48 段号489
页号4-----段号4	页号0-----段号20	段号49 段号490
页号5-----段号5	页号1-----段号21	段号50 段号491
页号6-----段号6	页号2-----段号22	段号51 段号492
页号7-----段号7	页号3-----段号23	段号52 段号493
页号8-----段号8	页号4-----段号24	段号53 段号494
页号9-----段号9	页号5-----段号25	段号54 段号495
作业标识符: c	页号6-----段号26	段号55 段号496
作业页表:	页号7-----段号27	段号56 段号497
页号0-----段号10	页号8-----段号28	段号57 段号498
页号1-----段号11	页号9-----段号29	段号58 段号499
页号2-----段号12	空闲内存:	段号59 段号500
页号3-----段号13	段号30	段号60 段号501
页号4-----段号14	段号31	段号61 段号502
页号5-----段号15	段号32	段号62 段号503
页号6-----段号16	段号33	段号63 段号504
页号7-----段号17	段号34	段号64 段号505
	段号35	段号65 段号506
	段号36	段号66 段号507
	段号37	段号67 段号508
	段号38	段号68 段号509
	段号39	段号69 段号510
	段号40	段号70 段号511
	段号41	段号71

### 5.3 基本分段管理

作业标识符: c  
 作业状态: R(执行状态)  
 作业占用内存情况:  
 段号: c1 , 段长: 5KB  
 段号: c2 , 段长: 15KB

作业标识符: d  
 作业状态: R(执行状态)  
 作业占用内存情况:  
 段号: d1 , 段长: 10KB  
 段号: d2 , 段长: 10KB

空闲内存情况:

段号1 大小: 4KB  
 段号2 大小: 5KB  
 段号3 大小: 5KB  
 段号4 大小: 5KB

段号163 大小: 15KB  
 段号164 大小: 15KB  
 段号165 大小: 15KB

6.内存最多允许同时运行 5 个作业，用时间片轮转算法运行作业。

#### 6.1 动态分区管理

分区使用情况:  
 分区始址: 160  
 分区大小: 20  
 分区状态: 0(占用状态)

分区始址: 128  
 分区大小: 20  
 分区状态: 0(占用状态)

分区始址: 32  
 分区大小: 20  
 分区状态: 0(占用状态)

分区始址: 64  
 分区大小: 20  
 分区状态: 0(占用状态)

分区始址: 96  
 分区大小: 20  
 分区状态: 0(占用状态)

## 6.2 基本分页存储管理

内存使用情况:	段号17	作业标识符: a
作业标识符: d	段号18	作业页表:
作业页表:	段号19	页号0——段号40
页号0——段号0	作业标识符: b	页号1——段号41
页号1——段号1	作业页表:	页号2——段号42
页号2——段号2	页号0——段号20	页号3——段号43
页号3——段号3	页号1——段号21	页号4——段号44
页号4——段号4	页号2——段号22	页号5——段号45
页号5——段号5	页号3——段号23	页号6——段号46
页号6——段号6	页号4——段号24	页号7——段号47
页号7——段号7	页号5——段号25	页号8——段号48
页号8——段号8	页号6——段号26	页号9——段号49
页号9——段号9	页号7——段号27	
作业标识符: e	页号8——段号28	
作业页表:	页号9——段号29	
页号0——段号10	作业标识符: c	
页号1——段号11	作业页表:	
页号2——段号12	页号0——段号30	
页号3——段号13	页号1——段号31	
页号4——段号14	页号2——段号32	
页号5——段号15	页号3——段号33	
页号6——段号16	页号4——段号34	
	页号5——段号35	
	页号6——段号36	
	页号7——段号37	
	页号8——段号38	
	页号9——段号39	
	作业标识符: a	
	作业页表:	
	页号0——段号40	

## 6.3 基本分段存储管理

作业标识符: g	作业标识符: d
作业状态: R(执行状态)	作业状态: R(执行状态)
作业占用内存情况:	作业占用内存情况:
段号: g1, 段长: 5KB	段号: d1, 段长: 10KB
段号: g2, 段长: 5KB	段号: d2, 段长: 10KB
段号: g3, 段长: 5KB	
段号: g4, 段长: 5KB	
作业标识符: e	
作业状态: R(执行状态)	
作业占用内存情况:	
段号: e1, 段长: 5KB	
段号: e2, 段长: 5KB	
段号: e3, 段长: 10KB	
作业标识符: f	
作业状态: R(执行状态)	
作业占用内存情况:	
段号: f1, 段长: 5KB	
段号: f2, 段长: 15KB	
作业标识符: c	
作业状态: R(执行状态)	
作业占用内存情况:	
段号: c1, 段长: 5KB	
段号: c2, 段长: 15KB	

## 六、分析、比较算法

对于动态分区管理方式，其主要是通过空闲分区表/链的管理对一个作业整个的装入或调出。运用的首次适应算法在分配内存时，从队首开始顺序查找，直至找到一个满足要求的空闲分区。

对于分页存储管理方式，其主要是把作业分成固定大小的页，把内存空间分成固定大小的段，允许作业离散地装入或调出。运用的地址顺序查找算法是最简单的分配内存算法。

对于分段存储管理方式，其主要是把作业分成不定大小的段，允许作业离散且不定大小地装入或调出内存空间。其运用的最佳适应算法在分配内存时，从队首开

始查找(空闲分区链从小到大顺序形成), 总能找到能满足要求、又是最小的空闲分区。

这三种存储管理方式相对比来说:

1. 内存装入或调出的灵活性; 动态分区管理方式最低, 分页存储管理方式其次, 分段存储管理方式最高。
2. 管理方式实现的难度; 动态分区管理方式最低, 分页存储管理方式其次, 分段存储管理方式最高。
3. 管理方式的内存开销; 动态分区管理方式最少, 分段存储管理方式其次, 分页存储管理方式最高。

三种内存分配算法相对比:

1. 算法实现的难度; 地址顺序算法最低, 首次适应算法其次, 最佳适应算法最高。
2. 算法留下碎片的难易; 地址顺序算法最难, 首次适应算法其次, 最佳适应算法最易。

## 实验 4 磁盘调度算法

### 一、实验目的

通过实验，理解磁盘 I/O 调度。

### 二、实验内容

1. 磁盘共划分 256 个磁道，当前磁头在 100 号磁道，并向磁道增加的方向移动。
2. 随机产生一组（共 10 个）磁盘 I/O 任务（即要访问的磁道号）。分别计算 FCFS、SSTF、SCAN、CSCAN 算法的平均寻道长度。
3. 在 I/O 过程中，随机产生新的 I/O 任务，最多生成 100 个 I/O 任务。比较 FCFS、SSTF、SCAN、CSCAN 算法的平均寻道长度。
4. 实现 FSCAN 算法，并用数据进行测试。

### 三、分析算法性能

(i) 一组(共 10 个)磁盘 I/O 任务，分别计算四种算法的平均寻道长度。

#### 1. FCFS 算法的平均寻道长度

```
请求IO:
磁道号: 92
磁道号: 246
磁道号: 238
磁道号: 121
磁道号: 44
磁道号: 223
磁道号: 5
磁道号: 225
磁道号: 186
磁道号: 43
FCFS平均寻道长度: 116.3
```

#### 2. SSTF 算法的平均寻道长度

```
请求IO:
磁道号: 92
磁道号: 246
磁道号: 238
磁道号: 121
磁道号: 44
磁道号: 223
磁道号: 5
磁道号: 225
磁道号: 186
磁道号: 43
SSTF平均寻道长度: 35.8
```

#### 3. SCAN 算法的平均寻道长度



```

请求IO:
磁道号: 92
磁道号: 246
磁道号: 238
磁道号: 121
磁道号: 44
磁道号: 223
磁道号: 5
磁道号: 225
磁道号: 186
磁道号: 43
SCAN平均寻道长度: 38.7
请按任意键继续. . .

```

#### 4. CSCAN 算法的平均寻道长度

```

请求IO:
磁道号: 92
磁道号: 246
磁道号: 238
磁道号: 121
磁道号: 44
磁道号: 223
磁道号: 5
磁道号: 225
磁道号: 186
磁道号: 43
CSCAN平均寻道长度: 47.4

```

(ii) 在 I/O 过程中, 随机产生新的 I/O 任务, 最多生成 100 个 I/O 任务。比较 FCFS、SSTF、SCAN、CSCAN 算法的平均寻道长度。

请求IO:	请求IO:	请求IO:
磁道号: 92	磁道号: 177	磁道号: 165
磁道号: 246	磁道号: 172	磁道号: 101
磁道号: 238	磁道号: 205	磁道号: 158
磁道号: 121	磁道号: 151	磁道号: 70
磁道号: 44	磁道号: 107	磁道号: 171
磁道号: 223	磁道号: 207	磁道号: 231
磁道号: 5	磁道号: 53	磁道号: 20
磁道号: 225	磁道号: 57	磁道号: 109
磁道号: 186	磁道号: 128	磁道号: 109
磁道号: 43	磁道号: 95	磁道号: 52
FCFS平均寻道长度: 116.3	FCFS平均寻道长度: 57.5	FCFS平均寻道长度: 79.2
SSTF平均寻道长度: 35.8	SSTF平均寻道长度: 27.3	SSTF平均寻道长度: 30.9
SCAN平均寻道长度: 38.7	SCAN平均寻道长度: 26.1	SCAN平均寻道长度: 34.2
CSCAN平均寻道长度: 47.4	CSCAN平均寻道长度: 30.3	CSCAN平均寻道长度: 39.2

请求IO:	请求IO:	请求IO:
磁道号: 120	磁道号: 147	磁道号: 245
磁道号: 2	磁道号: 30	磁道号: 183
磁道号: 73	磁道号: 132	磁道号: 250
磁道号: 112	磁道号: 109	磁道号: 178
磁道号: 94	磁道号: 22	磁道号: 150
磁道号: 113	磁道号: 43	磁道号: 215
磁道号: 27	磁道号: 101	磁道号: 247
磁道号: 152	磁道号: 186	磁道号: 159
磁道号: 91	磁道号: 136	磁道号: 93
磁道号: 146	磁道号: 82	磁道号: 173
FCFS平均寻道长度: 61.2	FCFS平均寻道长度: 64.4	FCFS平均寻道长度: 70.5
SSTF平均寻道长度: 25.6	SSTF平均寻道长度: 25	SSTF平均寻道长度: 16.4
SCAN平均寻道长度: 20.2	SCAN平均寻道长度: 25	SCAN平均寻道长度: 30.7
CSCAN平均寻道长度: 29.4	CSCAN平均寻道长度: 31	CSCAN平均寻道长度: 30.7

请求IO:	请求IO:	请求IO:	请求IO:
磁道号: 123	磁道号: 165	磁道号: 168	磁道号: 201
磁道号: 186	磁道号: 91	磁道号: 15	磁道号: 207
磁道号: 53	磁道号: 132	磁道号: 237	磁道号: 56
磁道号: 158	磁道号: 152	磁道号: 72	磁道号: 4
磁道号: 203	磁道号: 207	磁道号: 22	磁道号: 48
磁道号: 17	磁道号: 97	磁道号: 43	磁道号: 112
磁道号: 27	磁道号: 85	磁道号: 210	磁道号: 120
磁道号: 248	磁道号: 48	磁道号: 75	磁道号: 163
磁道号: 222	磁道号: 77	磁道号: 104	磁道号: 188
磁道号: 19	磁道号: 237	磁道号: 7	磁道号: 98
FCFS平均寻道长度: 101.5	FCFS平均寻道长度: 60.3	FCFS平均寻道长度: 110.7	FCFS平均寻道长度: 58.4
SSTF平均寻道长度: 35.9	SSTF平均寻道长度: 24.1	SSTF平均寻道长度: 33.1	SSTF平均寻道长度: 28
SCAN平均寻道长度: 37.9	SCAN平均寻道长度: 32.6	SCAN平均寻道长度: 36.7	SCAN平均寻道长度: 31
CSCAN平均寻道长度: 41.5	CSCAN平均寻道长度: 37.5	CSCAN平均寻道长度: 43.5	CSCAN平均寻道长度: 40.4

性能分析：

从总体上看，SSTF 算法的平均寻道长度更小，SCAN 算法其次，CSCAN 算法再其次，FCFS 算法平均寻道长度最大。

FCFS 算法是最简单的算法，其算法实现难度也是最低的。但由于此算法未对寻道进行优化，致使平均寻道时间可能较长，故 FCFS 算法仅适用于请求任务数目较少的场合。

SSTF 算法实质上是基于优先级的算法，有可能导致优先级低的任务发生“饥饿”现象，这种算法不能保证平均寻道时间最短。SSTF 算法平均寻道长度明显低于 FCFS 算法，较之有更好的寻道性能。

SCAN 算法是 SSTF 算法的修改后一种可防止出现“饥饿”现象的算法。SCAN 算法既能获得较好的寻道性能，又能防止“饥饿”现象，故被广泛用于大、中、小型机器。

CSCAN 算法是 SCAN 算法的改进。SCAN 算法存在这样的问题：当磁头刚从里向外移动而越过了某一磁道时，恰好又有一进程请求访问此磁道，致使该进程的请求被大大地推迟。CSCAN 算法通过磁头单向移动来减少这种延迟。

#### 四、画出实验内容 3 的性能曲线

(横坐标 I/O 任务，纵坐标寻道长度，比较四个算法的曲线)

