**背景分析**

多级反馈队列调度算法既能使高优先级的作业得到响应又能使短作业（进程）迅速完成。（对比一下FCFS与高响应比优先调度算法的缺陷）。该算法为操作系统的核心功能，通过模拟该功能的实现，了解操作系统特性。

**系统设计**

1. 实验目的：分析操作系统的核心功能模块，理解相关功能模块实现的数据结构和算法，并加以实现， 加深对操作系统原理和实现过程的理解。
2. 实验环境：

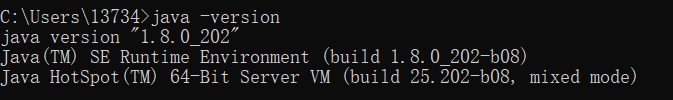
操作系统：windows 10 家庭版

Cpu: i5-8300H

Gpu: gtx 1060 6g

内存:8g

Java jdk:



编译器: Intellij IDEA 2022.2.2

**详细设计**

1. 实验内容：

多级反馈队列调度的模拟：

本算法设置了三个优先级从高到低的队列，时间片长度分别为：

队列一：2

队列二：4

队列三：8

1. 实现思路：

3.1 采用自定义类Progress实例化进程对象，组成进程数组；

3.2 初始化三个优先级从高到低（时间片长：2、4、8）的队列；

3.3 按照：

3.3.1 首次加入的进程入最高优先级队列尾等待；

3.3.2 某队列内时间片完而进程还没结束就立即调入次级优先级队列尾等待继续运行；

3.3.3 只要有进程进入高优先级队列内，便对低优先级队列中的进程进行抢占式运行；

的思想每一秒对每个队列内的进程具有的进程标识符、到达时间、运行时间、仍需运行时间（方便显示）等属性进行更新和输出，以此模拟实际操作系统的调度状态。

1. 主要的数据结构

/\*三个队列\*/

private static Queue<Progress> firstQueue = new LinkedList<>();

private static Queue<Progress> secondQueue = new LinkedList<>();

private static Queue<Progress> thirdQueue = new LinkedList<>();

/\*\*

\* 内部进程类：模拟进程

\*/

private static class Progress implements Comparable<Progress> {

String id; //进程标识符

int reachTime; //到达时间

int cpuTime; //运行时间

int needTime; //仍需时间

char state; //进程状态

/\*重写比较器\*/

@Override

public int compareTo( Progress b ) {

//按reachTime从小到大排序

return Float.compare(reachTime, b.reachTime);

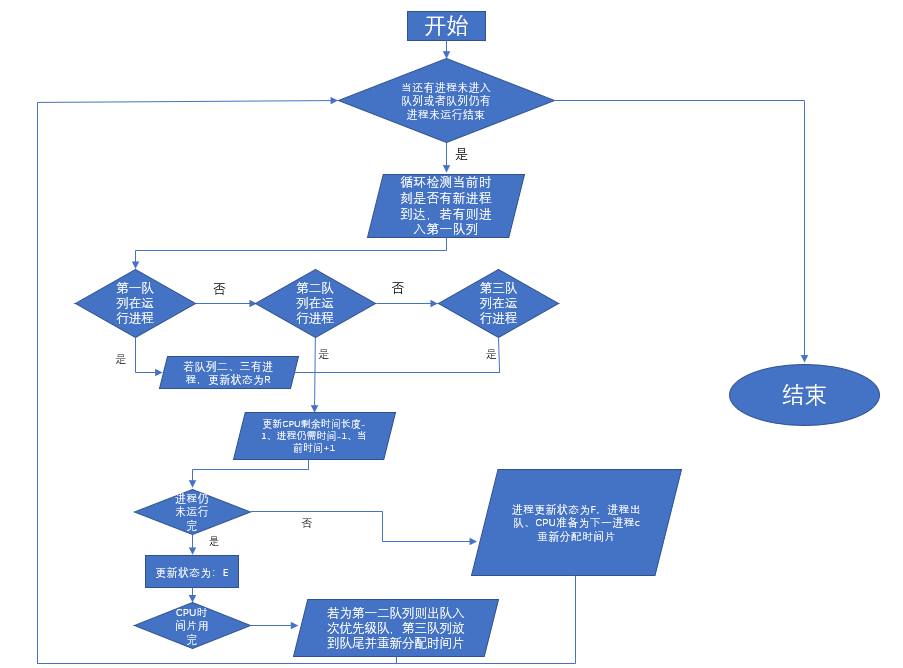
}

}

/\*进程数组\*/

Progress[] pro = new Progress[proNum];

1. 流程图



**编码**

package com.algorithm.multiStageFeedback;

import java.util.\*;

/\*\*

\* @Class MSFQS

\* @Description 多级反馈队列调度算法

\* @Author Naren

\* @Date 2020/5/30 10:46

\* @Version 1.0

\*/

public class MSFQS {

/\*三个队列\*/

private static Queue<Progress> firstQueue = new LinkedList<>();

private static Queue<Progress> secondQueue = new LinkedList<>();

private static Queue<Progress> thirdQueue = new LinkedList<>();

private static int firstTime; //第一队列cpu时间片

private static int secondTime; //第二队列cpu时间片

private static int thirdTime; //第三队列cpu时间片

private static int proNum; //进程数量

private static Scanner sc = new Scanner(System.in);

/\*\*

\* 内部进程类：模拟进程

\*/

private static class Progress implements Comparable<Progress> {

String id; //进程标识符

int reachTime; //到达时间

int cpuTime; //运行时间

int needTime; //仍需时间

char state; //进程状态

/\*重排输出格式\*/

@Override

public String toString() {

System.out.println();

return String.format("进程%s: %10d %7d %8d %7c\n", id, reachTime, cpuTime, needTime, state);

}

/\*重写比较器\*/

@Override

public int compareTo( Progress b ) {

//按reachTime从小到大排序

return Float.compare(reachTime, b.reachTime);

}

}

/\*\*

\* 进程调度算法：Multi-stage feedback queue scheduling algorithm

\*/

private static void progressScheduling(Progress[] pro){

int firstCpu = firstTime;

int secondCpu = secondTime;

int thirdCpu = thirdTime;

int currentTime = 0;

int num = 0;

//System.out.println(Arrays.toString(pro));

/\*当有进程未运行时或进程队列不为空时，以每1时间片为单位\*/

while(num < proNum || !firstQueue.isEmpty() || !secondQueue.isEmpty() || !thirdQueue.isEmpty()){

/\*当前时刻有进程到达，则添加入第一队列\*/

while(num < proNum && pro[num].reachTime == currentTime)

firstQueue.offer(pro[num++]);

//打印上一秒各队列进程状态

viewMenu(currentTime);

/\*当前为队列1在运行进程\*/

if(!firstQueue.isEmpty()){

if (secondQueue.peek() != null) secondQueue.peek().state = 'R';

if (thirdQueue.peek() != null) thirdQueue.peek().state = 'R';

//仍需时间：-1

firstQueue.peek().needTime -= 1;

//CPU剩余时间片：-1

firstTime -= 1;

//更新当前时间：+1

currentTime++;

//进程正在运行，状态：E.

if(firstQueue.peek().needTime > 0){

firstQueue.peek().state = 'E';

//当前队列CPU时间片用完而进程仍未运行完时，进程出队，入次优先级队尾

if(firstTime == 0) {

firstQueue.peek().state = 'R';

secondQueue.offer(firstQueue.poll());

firstTime = firstCpu;

}

}

//进程运行完毕，状态：F，记录完成时刻并出队

else if(firstQueue.peek().needTime == 0){

firstQueue.peek().state = 'F';

System.out.printf("\n当前时刻：%d,此进程运行结束：\n",currentTime);

System.out.println(firstQueue.peek());

Objects.requireNonNull(firstQueue.poll());

firstTime = firstCpu;

}

}

/\*当前为队列2在运行进程\*/

else if(!secondQueue.isEmpty()){

if (thirdQueue.peek() != null) thirdQueue.peek().state = 'R';

//仍需时间：-1

secondQueue.peek().needTime -= 1;

//CPU剩余时间片：-1

secondTime -= 1;

//更新当前时间：+1

currentTime++;

//进程运行完毕，状态：F，记录完成时刻并出队

if(secondQueue.peek().needTime == 0){

secondTime = secondCpu;

secondQueue.peek().state = 'F';

System.out.printf("\n当前时刻：%d,此进程运行结束：\n",currentTime);

System.out.println(secondQueue.peek());

Objects.requireNonNull(secondQueue.poll());

}

//进程正在运行，状态：E.

else if(secondQueue.peek().needTime > 0){

secondQueue.peek().state = 'E';

//当前队列CPU时间片用完而进程仍未运行完时，进程出队，入次优先级队尾

if(secondTime == 0) {

secondQueue.peek().state = 'R';

thirdQueue.offer(secondQueue.poll());

secondTime = secondCpu;

}

}

}

/\*当前为队列3在运行进程\*/

else if(!thirdQueue.isEmpty()){

//仍需时间：-1

thirdQueue.peek().needTime -= 1;

//CPU剩余时间片：-1

thirdTime -= 1;

//更新当前时间：+1

currentTime++;

//进程正在运行，状态：R.

if(thirdQueue.peek().needTime > 0){

thirdQueue.peek().state = 'E';

//当前队列CPU时间片用完而进程仍未运行完时，进程出队，入次优先级队尾

if(thirdTime == 0) {

thirdQueue.peek().state = 'R';

thirdQueue.offer(thirdQueue.poll());

thirdTime = thirdCpu;

}

}

//进程运行完毕，状态：F，记录完成时刻并出队

else{

firstTime = firstCpu;

thirdQueue.peek().state = 'F';

System.out.printf("\n当前时刻：%d,此进程运行结束：\n",currentTime);

System.out.println(thirdQueue.peek());

Objects.requireNonNull(thirdQueue.poll());

}

}

}

}

/\*\*

\* 输入面板：获取到进程数组

\*/

private static Progress[] operator(){

System.out.println("-----------------朱哲琛+张星宇+段雨辰-----------------\n");

System.out.println("欢迎进入多级队列反馈调度模拟系统，队列个数：3。\n\n");

System.out.println("请按队列优先级从高到低的顺序输入各个队列的时间片长度：");

firstTime = sc.nextInt();

secondTime = sc.nextInt();

thirdTime = sc.nextInt();

System.out.print( "请输入进程数:" );

proNum = sc.nextInt();

/\*获取到进程数组\*/

Progress[] pro = new Progress[proNum];

System.out.println( "请依次输入进程标识符,进程到达时间,进程运行时间:" );

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

pro[i] = new Progress();

pro[i].id = sc.next();

pro[i].reachTime = sc.nextInt();

pro[i].cpuTime = sc.nextInt();

pro[i].needTime = pro[i].cpuTime;

pro[i].state = 'R';

}

//对进程按照compareTo()的要求按照到达时间排序

Arrays.sort(pro);

return pro;

}

/\*\*

\* 输出面板：实时输出运行结果

\*/

private static void viewMenu(int currentTime){

System.out.printf("\n当前时刻：%d\n",currentTime);

System.out.println("---------------------------------------------");

System.out.println(" 到达时间 运行时间 剩余时间 状态");

if(firstQueue.isEmpty()) System.out.println("队列一：空");

else System.out.println("队列一：\n"+ firstQueue.toString()

.replace("[", "").replace("]", "")

.replace(", ", ""));

if(secondQueue.isEmpty()) System.out.println("队列二：空");

else System.out.println("队列二：\n"+ secondQueue.toString()

.replace("[", "").replace("]", "")

.replace(", ", ""));

if(thirdQueue.isEmpty()) System.out.println("队列三：空");

else System.out.println("队列三：\n"+ thirdQueue.toString()

.replace("[", "").replace("]", "")

.replace(", ", ""));

System.out.println("=============================================");

}

/\*\*

\* main()

\*/

public static void main(String[] args) {

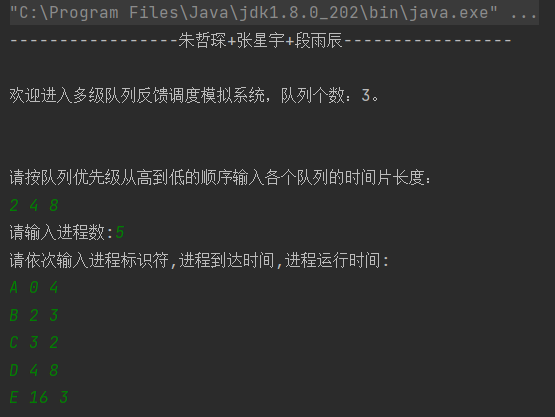
progressScheduling(operator());

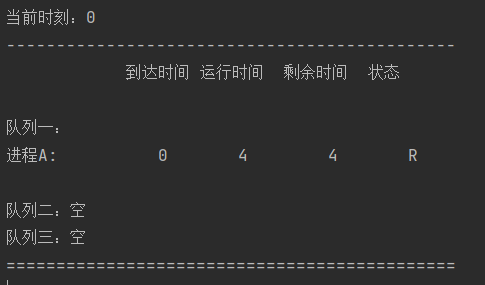
}

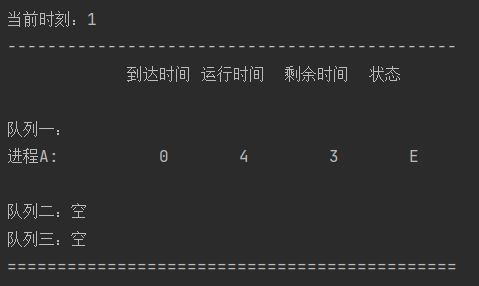
}

}

**测试**



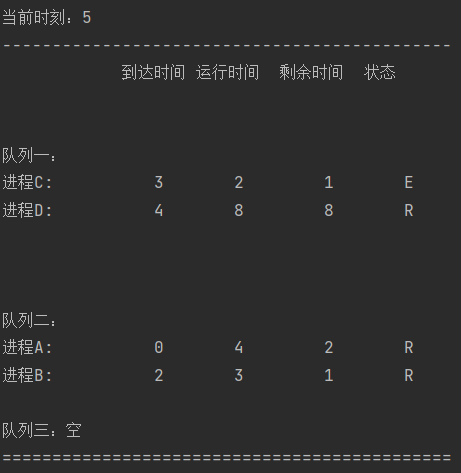


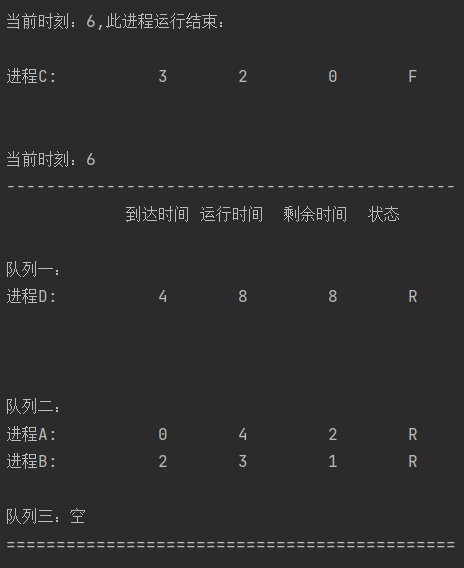




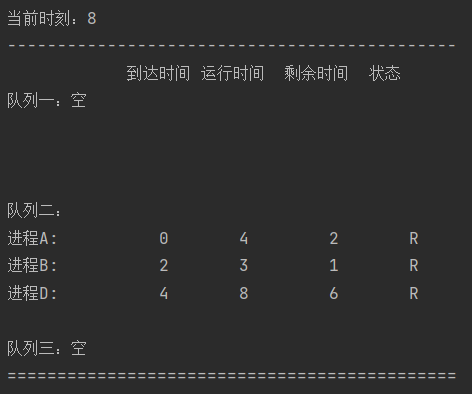


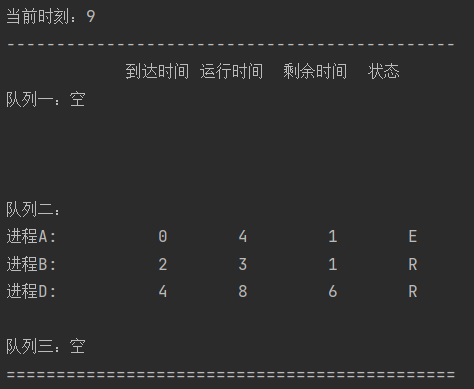


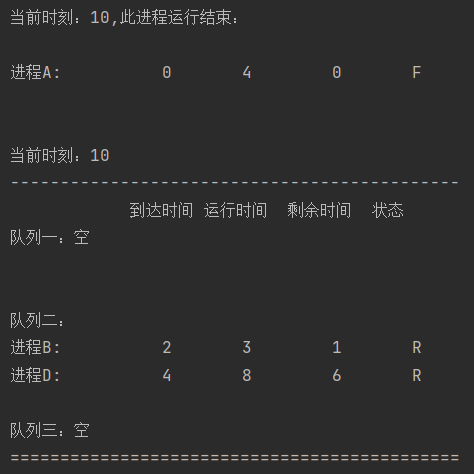


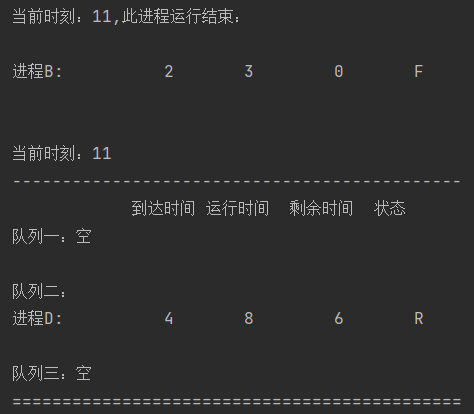


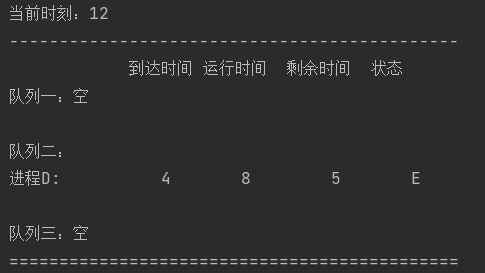


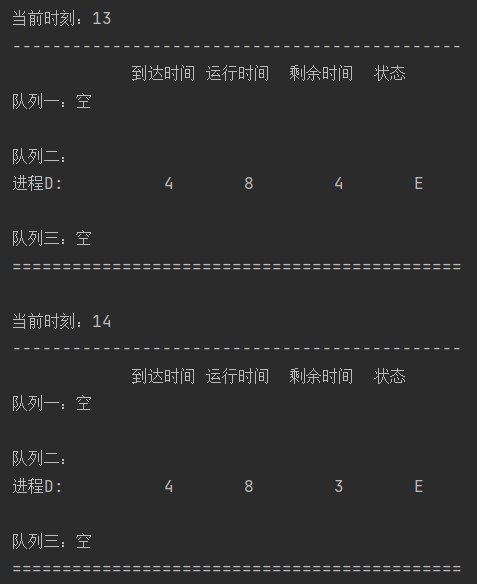


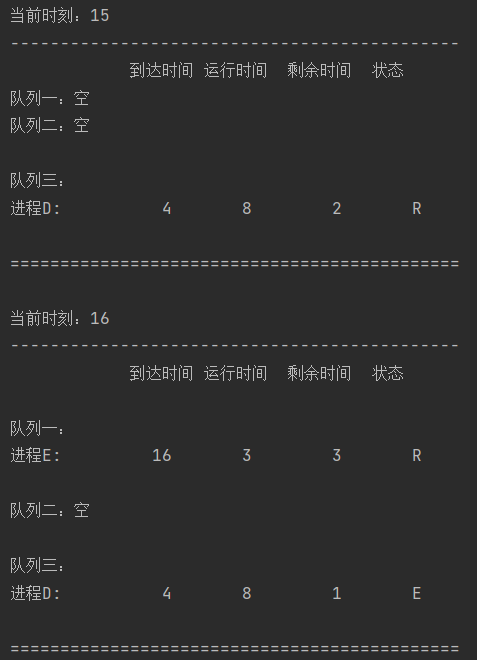


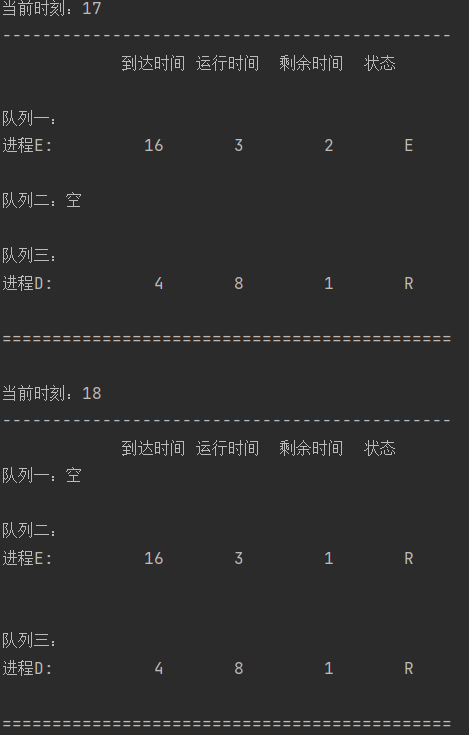


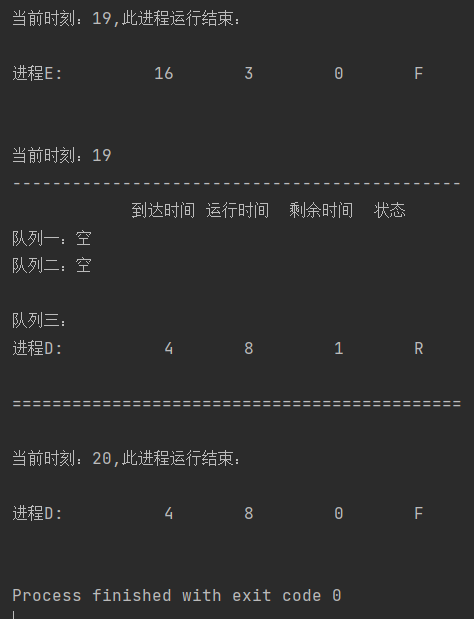












**总结**

多级反馈调度算法非常神奇，关键在于这三点：

1. 首次加入的进程入最高优先级队列尾等待；
2. 某队列内时间片完而进程还没结束就立即调入次级优先级队列尾等待继续运行；
3. 只要有进程进入高优先级队列内，便对低优先级队列中的进程进行抢占式 运行；

3.1 多级反馈队列调度算法的特点是适合大中小进程同时到达时的情况，对于大进程多的情况，其性能与资源的使用远优于短作业优先、先来先服务等算法。并且对于时间片完而未结束的进程、会在次优先级队列中为其提供倍增的时间等待继续运行。

3.2运行的结果中会使得到达时间晚的进程可能也会先运行结束。它对于各个进程的统筹调度安排非常合理，使得每个进程都有机会执行。我模拟的是三级反馈调度，在此过程中其实可以将其拓展为更多级，只需要新建一个自定义队列类，最后按照用户意愿的个数初始化队列数组即可。