课程名称 计算机操作系统

实验成绩

指导教师 曹勇



实 验 报 告

院系 信息工程学院

班级 物联网工程

学号 1601141019

姓名 晏沈威

日期 2018/10/17 地点：505

**­** 实验二、银行家算法

一、 目的和要求

银行家算法是由 Dijkstra 设计的最具有代表性的避免死锁的算法。本实验要求用高级语言编写一

个银行家的模拟算法。通过本实验可以对预防死锁和银行家算法有更深刻的认识。

二、 实验内容

1、 设置数据结构

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

2、 设计安全性算法

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

够的资源分配给进程

三、 实验环境

1、 pc

2、 vc++

四、程序源代码：

//算法函数类

package os;

/\*\*

\* @Author 晏沈威

\* @Email wsyjlly@qq.com

\* @Type\_name BankerClass

\* @Createtime 2018年10月16日 下午12:25:11

\*/

import java.util.Scanner;

public class BankerClass {

int[] Available = { 10, 8, 7 };

int[][] Max = new int[3][3];

int[][] Alloction = new int[3][3];

int[][] Need = new int[3][3];

int[][] Request = new int[3][3];

int[] Work = new int[3];

int num = 0;// 进程编号

Scanner in = new Scanner(System.in);

public BankerClass() {

// Max={{6,3,2},{5,6,1},{2,3,2}};

}

public void setSystemVariable() {// 设置各初始系统变量，并判断是否处于安全状态。

setMax();

setAlloction();

printSystemVariable();

SecurityAlgorithm();

}

public void setMax() {// 设置Max矩阵

System.out.println("请设置各进程的最大需求矩阵Max：");

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

System.out.println("请输入进程P" + i + "的最大资源需求量：");

for (int j = 0; j < 3; j++) {

Max[i][j] = in.nextInt();

}

}

}

public void setAlloction() {// 设置已分配矩阵Alloction

System.out.println("请设置请各进程分配矩阵Alloction：");

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

System.out.println("晴输入进程P" + i + "的分配资源量：");

for (int j = 0; j < 3; j++) {

Alloction[i][j] = in.nextInt();

}

}

System.out.println("Available=Available-Alloction.");

System.out.println("Need=Max-Alloction.");

for (int i = 0; i < 3; i++) {// 设置Alloction矩阵

for (int j = 0; j < 3; j++) {

Available[i] = Available[i] - Alloction[j][i];

}

}

for (int i = 0; i < 3; i++) {// 设置Need矩阵

for (int j = 0; j < 3; j++) {

Need[i][j] = Max[i][j] - Alloction[i][j];

}

}

}

public void printSystemVariable() {

System.out.println("此时资源分配量如下：");

System.out.println("进程 " + " Max " + " Alloction " + " Need " + " Available ");

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

System.out.print("P" + i + " ");

for (int j = 0; j < 3; j++) {

System.out.print(Max[i][j] + " ");

}

System.out.print("| ");

for (int j = 0; j < 3; j++) {

System.out.print(Alloction[i][j] + " ");

}

System.out.print("| ");

for (int j = 0; j < 3; j++) {

System.out.print(Need[i][j] + " ");

}

System.out.print("| ");

if (i == 0) {

for (int j = 0; j < 3; j++) {

System.out.print(Available[j] + " ");

}

}

System.out.println();

}

}

public void setRequest() {// 设置请求资源量Request

System.out.println("请输入请求资源的进程编号：");

num = in.nextInt();// 设置全局变量进程编号num

System.out.println("请输入请求各资源的数量：");

for (int j = 0; j < 3; j++) {

Request[num][j] = in.nextInt();

}

System.out.println("即进程P" + num + "对各资源请求Request：(" + Request[num][0] + "," + Request[num][1] + ","+ Request[num][2] + ").");

BankerAlgorithm();

}

public void BankerAlgorithm() {// 银行家算法

boolean T = true;

if (Request[num][0] <= Need[num][0] && Request[num][1] <= Need[num][1] && Request[num][2] <= Need[num][2]) {// 判断Request是否小于Need

if (Request[num][0] <= Available[0] && Request[num][1] <= Available[1] && Request[num][2] <= Available[2]) {// 判断Request是否小于Alloction

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

Available[i] -= Request[num][i];

Alloction[num][i] += Request[num][i];

Need[num][i] -= Request[num][i];

}

} else {

System.out.println("当前没有足够的资源可分配，进程P" + num + "需等待。");

T = false;

}

} else {

System.out.println("进程P" + num + "请求已经超出最大需求量Need.");

T = false;

}

if (T == true) {

printSystemVariable();

System.out.println("现在进入安全算法：");

SecurityAlgorithm();

}

}

public void SecurityAlgorithm() {// 安全算法

boolean[] Finish = { false, false, false };// 初始化Finish

int count = 0;// 完成进程数

int circle = 0;// 循环圈数

int[] S = new int[3];// 安全序列

for (int i = 0; i < 3; i++) {// 设置工作向量

Work[i] = Available[i];

}

boolean flag = true;

while (count < 3) {

if (flag) {

System.out.println("进程 " + " Work " + " Alloction " + " Need " + " Work+Alloction ");

flag = false;

}

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

if (Finish[i] == false && Need[i][0] <= Work[0] && Need[i][1] <= Work[1] && Need[i][2] <= Work[2]) {// 判断条件

System.out.print("P" + i + " ");

for (int k = 0; k < 3; k++) {

System.out.print(Work[k] + " ");

}

System.out.print("| ");

for (int j = 0; j < 3; j++) {

Work[j] += Alloction[i][j];

}

Finish[i] = true;// 当当前进程能满足时

S[count] = i;// 设置当前序列排号

count++;// 满足进程数加1

for (int j = 0; j < 3; j++) {

System.out.print(Alloction[i][j] + " ");

}

System.out.print("| ");

for (int j = 0; j < 3; j++) {

System.out.print(Need[i][j] + " ");

}

System.out.print("| ");

for (int j = 0; j < 3; j++) {

System.out.print(Work[j] + " ");

}

System.out.println();

}

}

circle++;// 循环圈数加1

if (count == 3) {// 判断是否满足所有进程需要

System.out.print("此时存在一个安全序列：");

for (int i = 0; i < 3; i++) {// 输出安全序列

System.out.print("P" + S[i] + " ");

}

System.out.println("故当前可分配！");

break;// 跳出循环

}

if (count < circle) {// 判断完成进程数是否小于循环圈数

count = 5;

System.out.println("当前系统处于不安全状态，故不存在安全序列。");

break;// 跳出循环

}

}

}

}

//测试类

/\*\*

\*

\*/

package os;

/\*\*

\* @Author 晏沈威

\* @Email wsyjlly@qq.com

\* @Type\_name TestBankerClass

\* @Createtime 2018年10月16日 下午12:39:25

\*/

import java.util.Scanner;

public class TestBankerClass {

public static void main(String[] args) { // TODO code application logic here

boolean Choose = true;

String C;

@SuppressWarnings("resource")

Scanner in = new Scanner(System.in);

BankerClass T = new BankerClass();

System.out.println("这是一个三个进程，初始系统可用三类资源为{10,8,7}的银行家算法：");

T.setSystemVariable();

while (Choose == true) {

T.setRequest();

System.out.println("您是否还要进行请求：y/n?");

C = in.nextLine();

if (C.endsWith("n")) {

Choose = false;

}

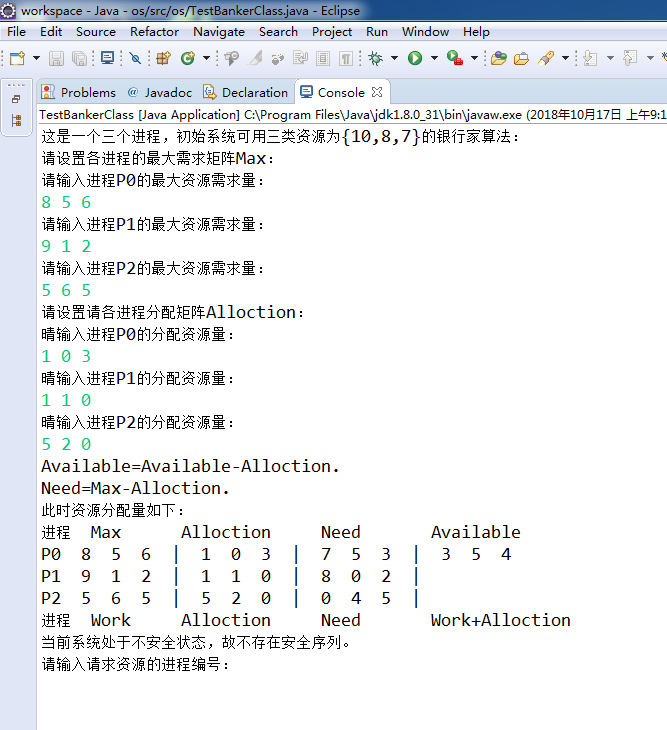
}

}

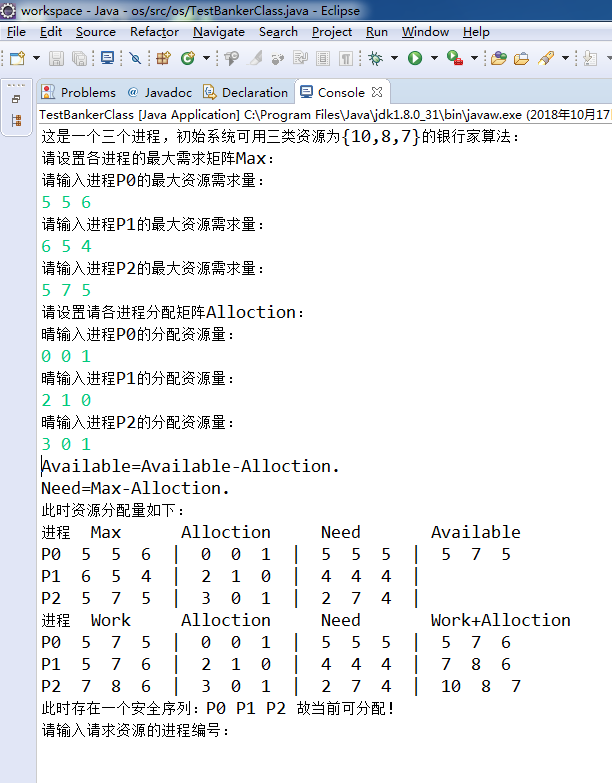
}

五、实验截图

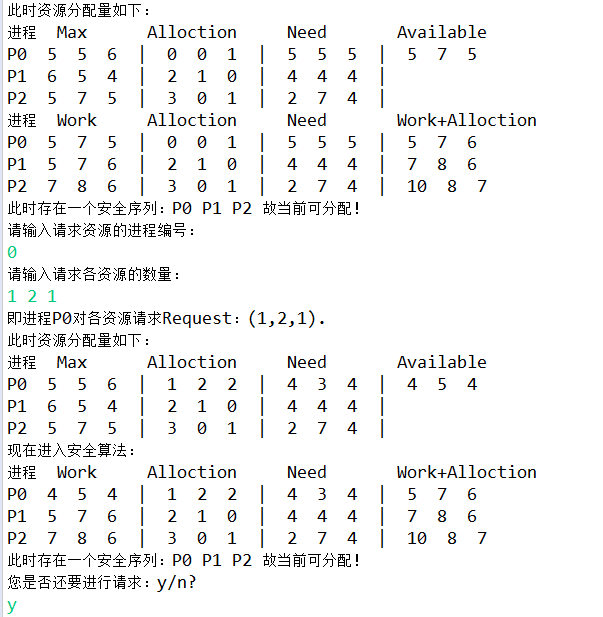
不安全状态：



安全状态：



安全状态请求：



六、实验总结

通过本实验我对预防死锁和银行家算法有更深刻的认识，并且了解的银行家算法的具体思想和运算过程