**江苏科技大学**

**课程实验报告**

课 程： 操作系统原理与安全

课 题： 实验二·银行家算法

学 院： 计算机学院

姓 名： 陈四贵

班 级： 1822107101

学 号： 182210710119

开发语言： Java

目 录

[一、 实验目的 1](#_Toc41677755)

[二、 实验要求 1](#_Toc41677756)

[三、 实验所需数据结构 1](#_Toc41677757)

[四、 银行家算法 1](#_Toc41677758)

[五、 安全性算法 2](#_Toc41677759)

[六、 流程图 2](#_Toc41677760)

[七、 实验代码及其说明 3](#_Toc41677761)

[1.总体结构 3](#_Toc41677762)

[2.分支结构 3](#_Toc41677763)

[⑴全局数据 3](#_Toc41677764)

[核心思想： 3](#_Toc41677765)

[附·源代码 3](#_Toc41677766)

[⑵主函数 3](#_Toc41677767)

[核心思想 3](#_Toc41677768)

[附·源代码 3](#_Toc41677769)

[⑶applyResource()函数 4](#_Toc41677770)

[核心思想 4](#_Toc41677771)

[附·源代码 4](#_Toc41677772)

[⑷isFinished()函数 6](#_Toc41677773)

[核心思想 6](#_Toc41677774)

[附·源代码 6](#_Toc41677775)

[⑸isSafe()函数 7](#_Toc41677776)

[核心思想 7](#_Toc41677777)

[附·源代码 7](#_Toc41677778)

[⑹printStatus()函数 8](#_Toc41677779)

[说明 8](#_Toc41677780)

[附·源代码 8](#_Toc41677781)

[八、 运行结果 9](#_Toc41677782)

[九、 实验心得 10](#_Toc41677783)

# 实验目的

死锁会引起计算机工作僵死，因此操作系统中必须防止。本实验的目的在于让学生独立的使用高级语言编写和调试一个系统动态分配资源的简单模拟程序，了解死锁产生的条件和原因，并采用银行家算法有效地防止死锁的发生，以加深对课堂上所讲授的知识的理解。

# 实验要求

设计有n个进程共享m个系统资源的系统，进程可动态的申请和释放资源，系统按各进程的申请动态的分配资源。

系统能显示各个进程申请和释放资源，以及系统动态分配资源的过程，便于用 户观察和分析；

# 实验所需数据结构

1．可利用资源向量 Available ，它是一个含有 m 个元素的数组，其中的每一个元素代表一类可利用的资源的数目，其初始值是系统中所配置的该类全部可用资源数目。其数值随该类资源的分配和回收而动态地改变。如果Available（j）=k，表示系统中现有 Rj 类资源 k 个。

2．最大需求矩阵 Max，这是一个 n×m 的矩阵，它定义了系统中 n 个进程中的每一 个进程对 m 类资源的最大需求。如果 Max（i，j）=k，表示进程 i 需要 Rj 类资源的最大数目为 k。

3．分配矩阵 Allocation，这是一个 n×m 的矩阵，它定义了系统中的每类资源当前一分配到每一个进程的资源数。如果 Allocation（i，j）=k，表示进程 i 当前已经分到 Rj 类资源的数目为 k。Allocation i 表示进程 i 的分配向量，有矩阵 Allocation的第 i 行构成。

4．需求矩阵 Need，这是一个 n×m 的矩阵，用以表示每个进程还需要的各类资源的 数目。如果 Need（i，j）=k，表示进程 i 还需要 Rj 类资源 k 个，才能完成其任务。Need i 表示进程 i 的需求向量，由矩阵 Need 的第 i 行构成。

上述三个矩阵间存在关系：Need（i，j）=Max（i，j）-Allocation（i，j）；

# 银行家算法

见教材P96.

# 安全性算法

1．设置两个向量。 Work：它表示系统可提供给进程继续运行的各类资源数目，它包含 m 个元素，开始执行安全性算法时，Work = Available。 Finish：它表示系统是否有足够的资源分配给进程，使之运行完成，开始 Finish（i）=false；当有足够资源分配给进程 Pi 时，令 Finish（i）=true；

2．从进程集合中找到一个能满足下述条件的进程:

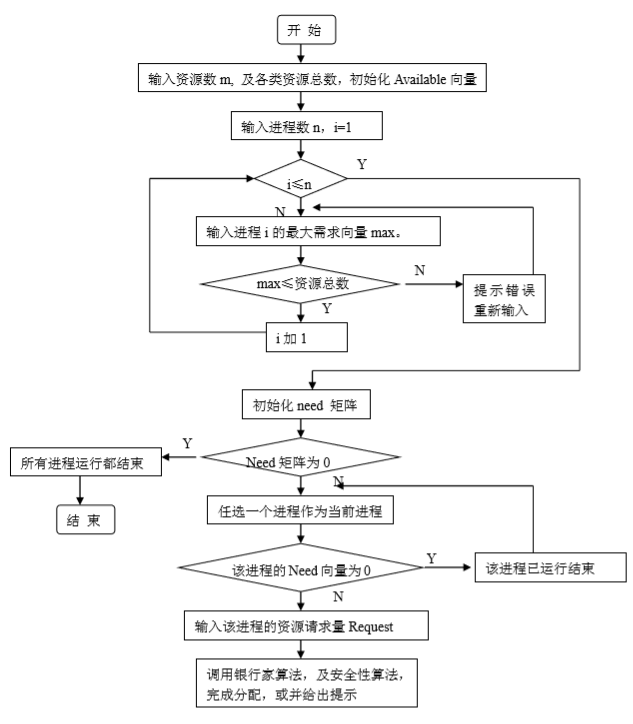
Finish（i）= = false； Need i ≤work；

如找到则执行步骤 3；否则，执行步骤 4；

3．当进程 Pi 获得资源后，可顺利执行直到完成，并释放出分配给它的资源，故应执行Work = work + Allocation(i) Finish（i）=true；转向步骤 2；

4．若所有进程的 Finish（i）都为 true，则表示系统处于安全状态；否则，系统处于不安全状态。

# 流程图



# 实验代码及其说明

## 1.总体结构

编写代码时根据实验指导书的要求做出适当的调整，模块化实现银行家算法与安全性检查。主要组成部分有主函数、applyResource()函数、isFinished()函数、isSafe()函数、printStatus()函数等。

## 2.分支结构

### ⑴全局数据

### 核心思想：

对在全局中都需要的数据赋予全局属性，减小程序开销，增加程序可读性。

#### 附·源代码：

1. **static** **int** m,n;   //m个系统资源，n个进程数
2. **static** **int**[] Available;  //可利用资源向量，是一个含有m个元素的数组。每一个元素代表每一类资源的数目
3. **static** **int**[][] Max;   //最大需求矩阵（n×m）。定义了系统中n个进程对m类资源的最大需求
4. **static** **int**[][] Allocation;   //分配矩阵（n×m）。表示进程i已经分到Rj资源的数目为k个
5. **static** **int**[][] Need;   //需求矩阵(n×m）。表示进程i还需要Rj类资源的数目为k个
6. **static** **boolean**[] isReleased;
7. **static** Scanner scanner=**new** Scanner(System.in);

### ⑵主函数

核心思想：初始化系统资源数、进程数、进程对资源的需求数，并控制程序流程，不断申请资源(applyResource)。

附·源代码：

1. **public** **static** **void** main(String[] args){
2. System.out.print("请您输入系统资源数：");
3. m=scanner.nextInt();
4. Available=**new** **int**[m];
5. System.out.print("请您输入各个资源的数量（用空格隔开）：");
6. **for**(**int** i=0;i<=m-1;i++){
7. Available[i]=scanner.nextInt();
8. }
9. System.out.print("请您输入进程数：");
10. n=scanner.nextInt();
11. Max=**new** **int**[n][m];
12. Allocation=**new** **int**[n][m];  //初始化，分配矩阵默认值为0
13. Need=**new** **int**[n][m];
14. isReleased=**new** **boolean**[n];
15. **for**(**int** i=0;i<=n-1;i++){
16. isReleased[i]=**false**;
17. }
18. **for**(**int** i=0;i<=n-1;i++){
19. System.out.print("请您输入进程" + i + "对各资源的需求数（用空格隔开,请不要越界）：");
20. **for**(**int** j=0;j<=m-1;j++) {
21. Max[i][j]=scanner.nextInt();
22. Need[i][j]=Max[i][j]-Allocation[i][j];
23. }
24. }
25. **while**(**true**){
26. applyResource();
27. }
28. }

### ⑶applyResource()函数

核心思想：模拟银行家算法申请资源，限制非法变量输入，并不断通过isReleased[]数组检查进程是否已释放（获取到全部所需资源），还不断调用isFinished()函数检查系统分配资源后是否处于不安全状态。

附·源代码：

1. **static** **void** applyResource(){//申请资源
2. printStatus();
3. **int**[] Request=**new** **int**[m];   //某进程对m个资源的需求量
4. System.out.print("请您输入您要申请资源的进程编号(0~n-1)：");
5. **int** i=scanner.nextInt();
6. **while**(isReleased[i]){
7. System.out.print("进程"+i+"已释放，不能再申请资源。请您重新选择进程：");
8. i=scanner.nextInt();
9. }
10. System.out.print("请您输入为进程"+i+"的申请资源（不同种资源用空格隔开，0表示不申请该类资源）：");
11. **int** step=0;   //判断应跳转步骤
12. **while**(step==0) {
13. **for** (**int** j = 0; j <= m - 1; j++) {
14. Request[j] = scanner.nextInt();
15. **if** (Request[j] > Available[j]) {
16. step = 1;
17. }
18. **if** (Request[j] > Max[i][j]) {
19. step = 2;
20. }
21. }
22. **switch** (step) {
23. **case** 1 :{
24. System.out.print("您的请求超过了系统的最大资源量，请重新输入：");
25. step = 0;
26. **break**;
27. }
28. **case** 2 :{
29. System.out.print("您不需要这么多资源，不要太贪心哦。请重新输入：");
30. step = 0;
31. **break**;
32. }
33. **default**: {
34. step = 3;
35. }
36. }
37. }
38. **if**(isFinished(i,Request)){
39. System.out.println("所有进程完成资源分配，分配结束。");
40. System.exit(0);
41. }
42. }

### ⑷isFinished()函数

核心思想：检查系统分配资源后是否处于不安全状态，由于是模拟操作，不能对原相关纪录数组作更改，因而采用数组克隆（复制），模拟相关操作。若不能满足资源分配，则该操作是不安全的，不能分配资源。

附·源代码：

1. **static** **boolean** isFinished(**int** index,**int**[] Request){ //判断某进程是否完成所有资源分配
2. **int**[] work=Available.clone();
3. **boolean**[] finish=isReleased.clone();
4. **int**[][] need\_temp=**new** **int**[n][m];
5. **int**[][] allocate\_temp=**new** **int**[n][m];
6. **for**(**int** i=0;i<n;i++){
7. need\_temp[i]=Need[i].clone();
8. allocate\_temp[i]=Allocation[i].clone();
9. }
10. **for**(**int** i=0;i<m;i++){
11. **if**(Need[index][i]<Request[i]){
12. Request[i]=Need[index][i];
13. work[i]-=Request[i];
14. allocate\_temp[index][i]+=Request[i];
15. }
16. }
17. //重置need\_temp
18. **for**(**int** i=0;i<m;i++){
19. need\_temp[index][i]-=Request[i];
20. }
21. **if**(!isSafe(work,finish,need\_temp,allocate\_temp)){
22. System.out.println("分配会造成系统不安全，取消分配");
23. **return** **false**;
24. }
25. **else**{
26. System.out.println("分配成功");
27. **for**(**int** i=0;i<m;i++){
28. Available[i]-=Request[i];
29. Allocation[index][i]+=Request[i];
30. Need[index][i]-=Request[i];
31. }
32. **if**(!isReleased[index]){
33. **boolean** judge=**false**;
34. **for**(**int** j=0;j<m;j++){
35. **if**(Need[index][j]!=0) judge=**true**;
36. }
37. **if**(!judge){
38. isReleased[index]=**true**;
39. **for**(**int** j=0;j<m;j++){
40. Available[j]+=Allocation[index][j];
41. }
42. }
43. }
44. }
45. **boolean** isFinished=**true**;
46. **for**(**int** i=0;i<n;i++){
47. **for**(**int** j=0;j<m;j++){
48. **if**(Need[i][j]!=0){
49. isFinished=**false**;
50. }
51. }
52. }
53. **return** isFinished;
54. }

### ⑸isSafe()函数

核心思想：判断此状态下系统是否处于安全状态。

附·源代码：

1. **static** **boolean** isSafe(**int**[] work,**boolean**[] finish,**int**[][] need\_temp,**int**[][] allocation\_temp){
2. Queue<Integer> queue=**new** ArrayDeque<Integer>();  //假设能完成的队列
3. **int** time=0;
4. **while**(**true**){
5. **boolean** loop=**true**;
6. **for**(**int** i=0;i<n;i++){
7. time++;
8. **if**(!finish[i]){
9. **boolean** b=**false**;
10. **for**(**int** j=0;j<m;j++){
11. **if**(work[j]<need\_temp[i][j]){
12. b=**true**;
13. }
14. **if**(b) **break**;
15. }
16. **if**(!b){
17. time=0;
18. queue.add(i);
19. finish[i]=**true**;
20. **for**(**int** j=0;j<m;j++){
21. work[j]+=allocation\_temp[i][j];
22. allocation\_temp[i][j]+=need\_temp[i][j];
23. need\_temp[i][j]=0;
24. }
25. System.out.println();
26. }
27. }
28. }
29. **boolean** isFinish=**false**;
30. **for**(**int** i=0;i<n;i++){
31. **if**(!finish[i]){
32. isFinish=**true**;
33. **break**;
34. }
35. }
36. **if**(!isFinish){**return** **true**;}
37. **if**(time>n){**return** **false**;}
38. }
39. }

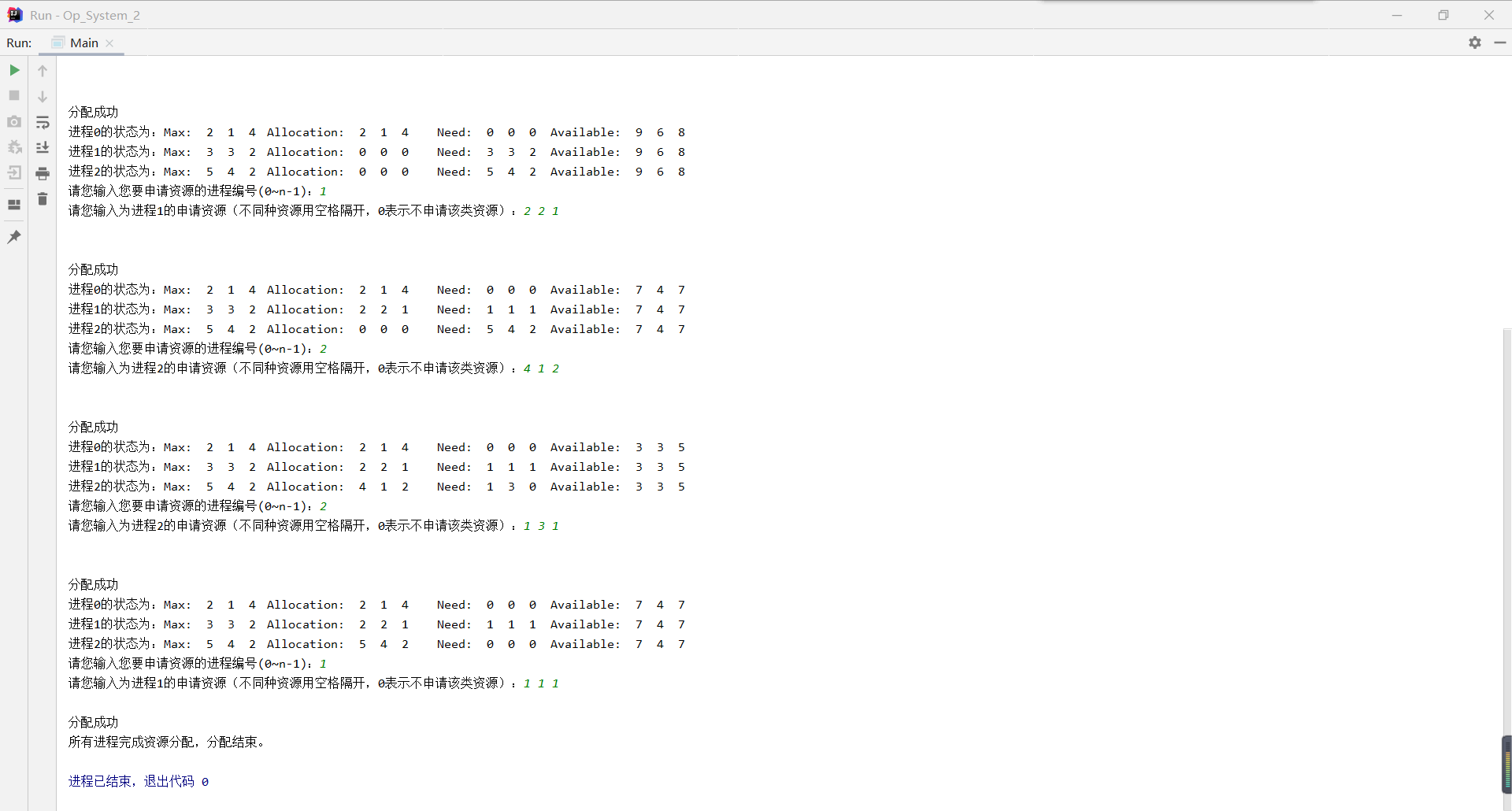
### ⑹printStatus()函数

说明：实时打印各记录数组信息。

附·源代码：

1. **static** **void** printStatus(){ //打印当前状态
2. **for**(**int** i=0;i<n;i++)
3. {
4. **for**(**int** j=0;j<m;j++)
5. {
6. Need[i][j]=Max[i][j]-Allocation[i][j];
7. }
8. }
9. **for**(**int** i=0;i<n;i++)
10. {
11. System.out.print("进程"+(i)+"的状态为：Max: ");
12. **for**(**int** j=0;j<m;j++) {System.out.print(" "+Max[i][j]+" ");}
13. System.out.print("\tAllocation: ");
14. **for**(**int** j=0;j<m;j++) {System.out.print(" "+Allocation[i][j]+" ");}
15. System.out.print("\tNeed: ");
16. **for**(**int** j=0;j<m;j++) {System.out.print(" "+Need[i][j]+" ");}
17. System.out.print("\tAvailable: ");
18. **for**(**int** j=0;j<m;j++) {System.out.print(" "+Available[j]+" ");}
19. System.out.println();
20. }
21. }

# 运行结果



# 实验心得

计算机一旦陷入死锁的状态，就很浪费资源与时间。为尽量避免死锁的发生，我们可以预防死锁，也可以避免死锁。虽然预防死锁能取得非常好的防范效果，但是检测的信息太多，系统开销太大，大大减慢系统运行速度，得不偿失。在这种情况下，避免死锁就显得尤为重要，我们只需要判断下一步操作是否会引起死锁，如果会，避免该操作即可。其中，银行家算法是比较有代表性的算法，它能很好的判断进程会不会引起死锁，然后避免它。在生活中各方面的管理上都得到了很广泛的应用，但银行家算法每次进行申请资源都要进行下一步的模拟操作，相比检测死锁而言较为繁琐，因而在作业量较大的系统中，银行家算法就显然不是最优解，检测死锁，若发生死锁就采取新的调度算法，优先满足其中一个，解除死锁，这样，就既能解决死锁，又能尽可能不影响系统运转速度。总之，运转量较小时可采取预防死锁、避免死锁的方式，运转量较大时宜采用检测死锁、解除死锁的方式。