

操作系统实验报告



**实验[ 2 ]： 银行家算法的实现**

**姓 名：**  **於俊涛**

**学 号： 202007020625**

**专业班级： 计算机203**

**系 别： 计算机系**

**学 院： 电子信息与人工智能学院**

## 实验2 银行家算法的实现 预习报告

**一、实验内容**

银行家算法的实现。

**二、实验目的**

银行家算法是一种最有代表性的避免死锁的算法。在避免死锁方法中允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次分配资源的安全性，若分配不会导致系统进入不安全状态，则分配，否则等待。通过编写一个模拟动态资源分配的银行家算法程序，帮助学生进一步深入理解死锁、产生死锁的必要条件、安全状态等重要概念，并掌握避免死锁的具体实施方法。

**三、实验原理**

**3.1、银行家算法中的数据结构**

1）可利用资源向量Available

是个含有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数目。如果Available［j］=K，则表示系统中现有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，则表示进程i需要Rj类资源的最大数目为K。

3）分配矩阵Allocation

这也是一个n×m的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果Allocation［i,j］=K，则表示进程i当前已分得Rj类资源的数目为K。

4）需求矩阵Need。

这也是一个n×m的矩阵，用以表示每一个进程尚需的各类资源数。如果Need［i,j］=K，则表示进程i还需要Rj类资源K个，方能完成其任务。

Need［i,j］=Max［i,j］-Allocation［i,j］

**3.2、银行家算法**

设Requesti是进程Pi的请求向量，如果Requesti［j］=K，表示进程Pi需要K个Rj类型的资源。当Pi发出资源请求后，系统按下述步骤进行检查：

(1)如果Requesti［j］≤Need［i,j］，便转向步骤(2)；否则认为出错，因为它所需要的资源数已超过它所宣布最大值。

(2)如果Requesti［j］≤Available［j］，便转向步骤(3)；否则，表示尚无足够资源，Pi须等待。

(3)系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值：

Available［j］=Available［j］-Requesti［j］;

Allocation［i,j］=Allocation［i,j］+Requesti［j］;

Need［i,j］=Need［i,j］-Requesti［j］;

系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式将资源分配给进程Pi，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程Pi等待。

**3.3、安全性算法**

1）设置两个向量：

工作向量Work: 它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时，Work=Available;

工作向量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。开始时先做Finish［i］=false; 当有足够资源分配给进程时， 再令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,j］≤Work［j］；若找到，执行 (3)，否则，执行 (4)

3）当进程Pi获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

Work［j］=Work［j］+Allocation［i,j］;

Finish［i］=true;

go to step 2;

4）如果所有进程的Finish［i］=true都满足， 则表示系统处于安全状态；否则，系统处于不安全状态

## 实验2 银行家算法的实现

**一、实验内容**

银行家算法的实现。

**二、实验目的**

银行家算法是一种最有代表性的避免死锁的算法。在避免死锁方法中允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次分配资源的安全性，若分配不会导致系统进入不安全状态，则分配，否则等待。通过编写一个模拟动态资源分配的银行家算法程序，帮助学生进一步深入理解死锁、产生死锁的必要条件、安全状态等重要概念，并掌握避免死锁的具体实施方法。

**三、实验原理**

**3.1、银行家算法中的数据结构**

1）可利用资源向量Available

是个含有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数目。如果Available［j］=K，则表示系统中现有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，则表示进程i需要Rj类资源的最大数目为K。

3）分配矩阵Allocation

这也是一个n×m的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果Allocation［i,j］=K，则表示进程i当前已分得Rj类资源的数目为K。

4）需求矩阵Need。

这也是一个n×m的矩阵，用以表示每一个进程尚需的各类资源数。如果Need［i,j］=K，则表示进程i还需要Rj类资源K个，方能完成其任务。

Need［i,j］=Max［i,j］-Allocation［i,j］

**3.2、银行家算法**

设Requesti是进程Pi的请求向量，如果Requesti［j］=K，表示进程Pi需要K个Rj类型的资源。当Pi发出资源请求后，系统按下述步骤进行检查：

(1)如果Requesti［j］≤Need［i,j］，便转向步骤(2)；否则认为出错，因为它所需要的资源数已超过它所宣布最大值。

(2)如果Requesti［j］≤Available［j］，便转向步骤(3)；否则，表示尚无足够资源，Pi须等待。

(3)系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值：

Available［j］=Available［j］-Requesti［j］;

Allocation［i,j］=Allocation［i,j］+Requesti［j］;

Need［i,j］=Need［i,j］-Requesti［j］;

系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式将资源分配给进程Pi，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程Pi等待。

**3.3、安全性算法**

1）设置两个向量：

工作向量Work: 它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时，Work=Available;

工作向量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。开始时先做Finish［i］=false; 当有足够资源分配给进程时， 再令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,j］≤Work［j］；若找到，执行 (3)，否则，执行 (4)

3）当进程Pi获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

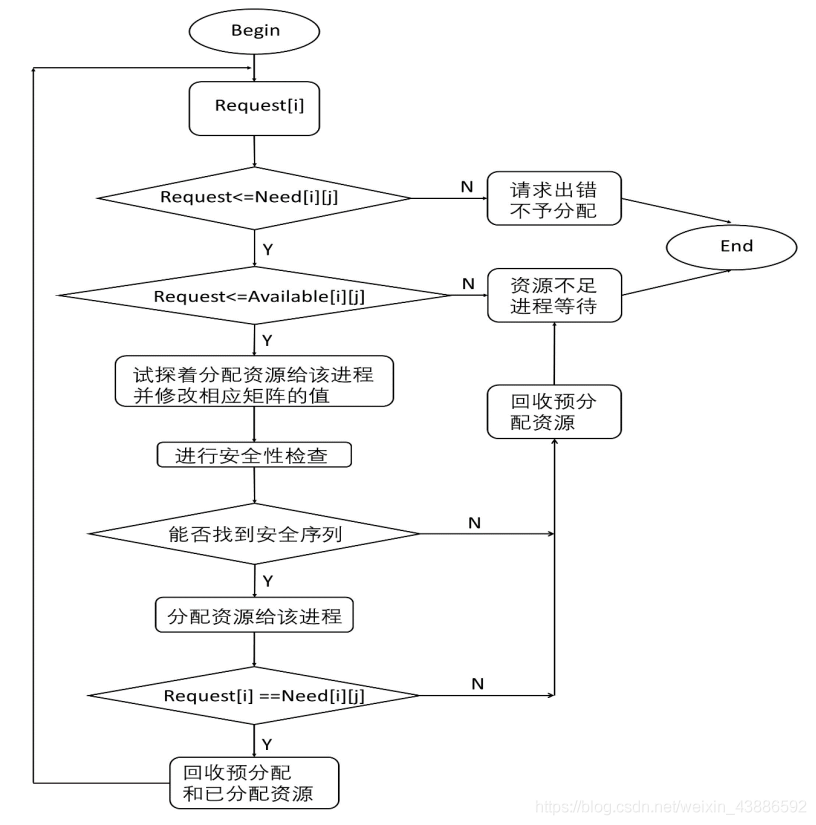
Work［j］=Work［i］+Allocation［i,j］;

Finish［i］=true;

go to step 2;

4）如果所有进程的Finish［i］=true都满足， 则表示系统处于安全状态；否则，系统处于不安全状态

**四、算法流程图**



**五、源程序及注释**

#include<stdio.h>

#define resource\_Num 100

#define process\_Num 100

int resourceNum;

int processNum;

int available[resource\_Num];//系统可用（剩余）资源

int maxRequest[process\_Num][resource\_Num];//进程的最大需求

int allocation[process\_Num][resource\_Num];//进程已经占有（分配）资源

int need[process\_Num][resource\_Num];//进程还需要资源

bool Finish[process\_Num];//是否安全

int safeSeries[process\_Num];//安全序列号

int request[resource\_Num];//进程请求资源量

int num;//资源数量计数

//打印输出系统信息

void showInfo()

{

printf("\n------------------------------------------------------------------------------------\n");

printf("当前系统各类资源剩余：");

for(int j = 0; j < resourceNum; j++)

{

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

}

printf("\n\n当前系统资源情况：\n");

printf(" PID\t Max\t\tAllocation\t Need\n");

for(int i = 0; i < processNum; i++)

{

printf(" P%d\t",i);

for(int j = 0; j < resourceNum; j++)

{

printf("%2d",maxRequest[i][j]);

}

printf("\t\t");

for(int j = 0; j < resourceNum; j++)

{

printf("%2d",allocation[i][j]);

}

printf("\t\t");

for(int j = 0; j < resourceNum; j++)

{

printf("%2d",need[i][j]);

}

printf("\n");

}

}

//打印安全检查信息

void SafeInfo(int \*work, int i)

{

int j;

printf(" P%d\t",i);

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

{

printf("%2d",work[j]);

}

printf("\t\t");

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

{

printf("%2d",allocation[i][j]);

}

printf("\t\t");

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

{

printf("%2d",need[i][j]);

}

printf("\t\t");

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

{

printf("%2d",allocation[i][j]+work[j]);

}

printf("\n");

}

//判断一个进程的资源是否全为零

bool isAllZero(int kang)

{

num = 0;

for(int i = 0; i < resourceNum; i++ )

{

if(need[kang][i] == 0)

{

num ++;

}

}

if(num == resourceNum)

{

return true;

}

else

{

return false;

}

}

//安全检查

bool isSafe()

{

//int resourceNumFinish = 0;

int safeIndex = 0;

int allFinish = 0;

int work[resource\_Num] = {0};

int r = 0;

int temp = 0;

int pNum = 0;

//预分配为了保护available[]

for(int i = 0; i < resourceNum; i++)

{

work[i] = available[i];

}

//把未完成进程置为false

for(int i = 0; i < processNum; i++)

{

bool result = isAllZero(i);

if(result == true)

{

Finish[i] = true;

allFinish++;

}

else

{

Finish[i] = false;

}

}

//预分配开始

while(allFinish != processNum)

{

num = 0;

for(int i = 0; i < resourceNum; i++)

{

if(need[r][i] <= work[i] && Finish[r] == false)

{

num ++;

}

}

if(num == resourceNum)

{

for(int i = 0; i < resourceNum; i++ )

{

work[i] = work[i] + allocation[r][i];

}

allFinish ++;

SafeInfo(work,r);

safeSeries[safeIndex] = r;

safeIndex ++;

Finish[r] = true;

}

r ++;//该式必须在此处

if(r >= processNum)

{

r = r % processNum;

if(temp == allFinish)

{

break;

}

temp = allFinish;

}

pNum = allFinish;

}

//判断系统是否安全

for(int i = 0; i < processNum; i++)

{

if(Finish[i] == false)

{

printf("\n当前系统不安全！\n\n");

return false;

}

}

//打印安全序列

printf("\n当前系统安全！\n\n安全序列为：");

for(int i = 0; i < processNum; i++)

{

bool result = isAllZero(i);

if(result == true)

{

pNum --;

}

}

for(int i = 0; i < pNum; i++)

{

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

}

return true;

}

//主函数

int main()

{

int i,j;

printf("请输入进程数和资源数:\n");

scanf("%d %d",&processNum,&resourceNum);

printf("请输入进程可用(剩余)资源数:\n");

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

scanf("%d",&available[resource\_Num]);

}

printf("请输入进程安全序列号:\n");

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

scanf("%d",&safeSeries[process\_Num]);

}

printf("请输入进程最大需求数(按[%d,%d]的格式):\n",processNum,resourceNum);

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

for(j=0;j<resourceNum;j++){

scanf("%d",&maxRequest[process\_Num][resource\_Num]);

}

printf("请输入进程已占有资源数(按[%d,%d]的格式):\n",processNum,resourceNum);

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

for(j=0;j<resourceNum;j++){

scanf("%d",&allocation[process\_Num][resource\_Num]);

}

printf("请输入进程还需资源数(按[%d,%d]的格式):\n",processNum,resourceNum);

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

for(j=0;j<resourceNum;j++){

scanf("%d",&need[process\_Num][resource\_Num]);

}

int curProcess = 0;

int a = -1;

showInfo();

printf("\n系统安全情况分析\n");

printf(" PID\t Work\t\tAllocation\t Need\t\tWork+Allocation\n");

bool isStart = isSafe();

//用户输入或者预设系统资源分配合理才能继续进行进程分配工作

while(isStart)

{

//限制用户输入，以防用户输入大于进程数量的数字，以及输入其他字符（乱输是不允许的）

do

{

if(curProcess >= processNum || a == 0)

{

printf("\n请不要输入超出进程数量的值或者其他字符：\n");

while(getchar() != '\n'){};//清空缓冲区

a = -1;

}

printf("\n------------------------------------------------------------------------------------\n");

printf("\n输入要分配的进程：");

a = scanf("%d",&curProcess);

printf("\n");

}while(curProcess >= processNum || a == 0);

//限制用户输入，此处只接受数字，以防用户输入其他字符（乱输是不允许的）

for(int i = 0; i < resourceNum; i++)

{

do

{

if(a == 0)

{

printf("\n请不要输入除数字以外的其他字符，请重新输入：\n");

while(getchar() != '\n'){};//清空缓冲区

a = -1;

}

printf("请输入要分配给进程 P%d 的第 %d 类资源：",curProcess,i+1);

a = scanf("%d", &request[i]);

}while( a == 0);

}

//判断用户输入的分配是否合理，如果合理，开始进行预分配

num = 0;

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

{

if(request[i] <= need[curProcess][i] && request[i] <= available[i])

{

num ++;

}

else

{

printf("\n发生错误！可能原因如下：\n(1)您请求分配的资源可能大于该进程的某些资源的最大需要！\n(2)系统所剩的资源已经不足了！\n");

break;

}

}

if(num == resourceNum)

{

num = 0;

for(int j = 0; j < resourceNum; j++)

{

//分配资源

available[j] = available[j] - request[j];

allocation[curProcess][j] = allocation[curProcess][j] + request[j];

need[curProcess][j] = need[curProcess][j] - request[j];

//记录分配以后，是否该进程需要值为0了

if(need[curProcess][j] == 0)

{

num ++;

}

}

//如果分配以后出现该进程对所有资源的需求为0了，即刻释放该进程占用资源（视为完成）

if(num == resourceNum)

{

//释放已完成资源

for(int i = 0; i < resourceNum; i++ )

{

available[i] = available[i] + allocation[curProcess][i];

}

printf("\n\n本次分配进程 P%d 完成,该进程占用资源全部释放完毕！\n",curProcess);

}

else

{

//资源分配可以不用一次性满足进程需求

printf("\n\n本次分配进程 P%d 未完成！\n",curProcess);

}

showInfo();

printf("\n系统安全情况分析\n");

printf(" PID\t Work\t\tAllocation\t Need\t\tWork+Allocation\n");

//预分配完成以后，判断该系统是否安全，若安全，则可继续进行分配，若不安全，将已经分配的资源换回来

if(!isSafe())

{

for(int j = 0; j < resourceNum; j++)

{

available[j] = available[j] + request[j];

allocation[curProcess][j] = allocation[curProcess][j] - request[j];

need[curProcess][j] = need[curProcess][j] +request[j];

}

printf("资源不足，等待中...\n\n分配失败！\n");

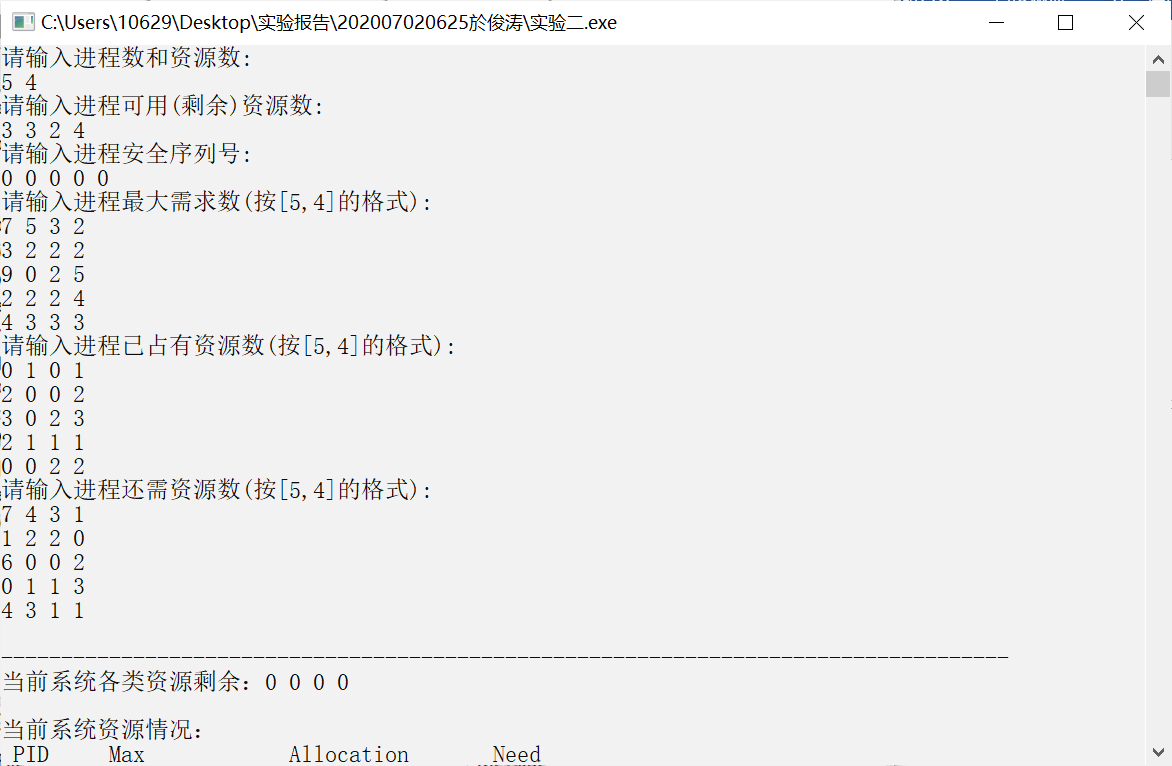
}

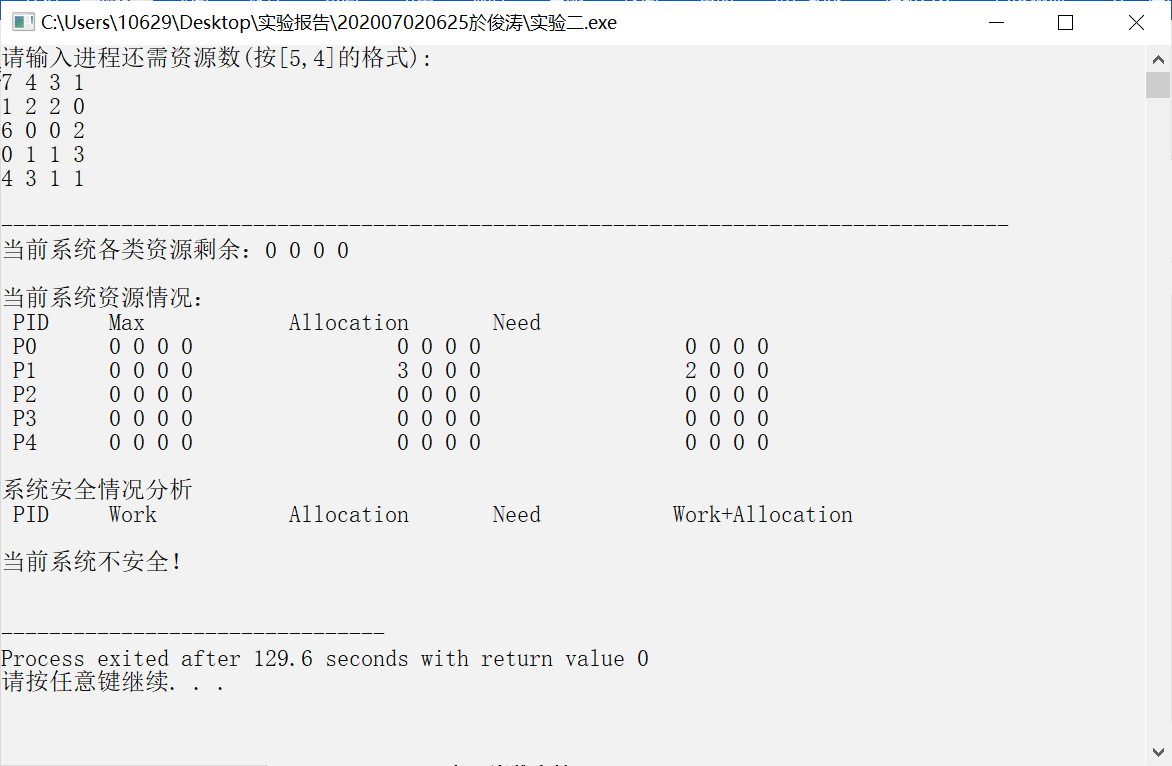
}

}

}

**六、打印的程序运行时初值和运行结果**





**七、实验小结**

1.银行家算法是一种用来避免操作系统死锁出现的有效算法。

1. 死锁：是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

3.死锁的发生必须具备以下四个必要条件：

1）互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

1. 请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。
2. 不抢占条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
3. 循环等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

4.避免死锁算法中最有代表性的算法就是Dijkstra E.W 于1968年提出的银行家算法，银行家算法是避免死锁的一种重要方法，防止死锁的机构只能确保上述四个条件之一不出现，则系统就不会发生死锁。

1. 为实现银行家算法，系统必须设置若干数据结构，同时要解释银行家算法，必须先解释操作系统安全状态和不安全状态。
2. 安全序列:是指一个进程序列{P1，…，Pn}是安全的，即对于每一个进程Pi(1≤i≤n），它以后尚需要的资源量不超过系统当前剩余资源量与所有进程Pj (j < i )当前占有资源量之和。
3. 安全状态：如果存在一个由系统中所有进程构成的安全序列P1，…，Pn，则系统处于安全状态。

8.安全状态一定是没有死锁发生。不安全状态：不存在一个安全序列。不安全状态不一定导致死锁。