

XI`AN TECHNOLOGICAL UNIVERSITY

课程设计报告

课程名称 计算机操作系统

专 业： 计算机科学与技术

班 级： 16060104

姓 名： 杨旭

学 号： 16060104117

指导教师： 姜虹

成 绩：

2019 年 1 月 4 日

**摘要**

避免死锁是处理死锁问题的一个有效策略，它是在资源的动态分配过程中，防止系统进入不安全状态，以避免发生死锁。这种方法施加的限制条件弱，可以获得较好的系统性能。因而，避免死锁算法的研究显得尤为重要

本文应用了最具代表性的避免死锁的算法——Dijkstra的银行家算法，通过在visual studio 2012编程环境编写C程序，并成功验证了该算法的有效性，证明了该算法完全可以满足避免死锁的要求

目录

[1. 绪论 4](#_Toc534494298)

[2. 需求分析 5](#_Toc534494299)

[2.1. 问题陈述 5](#_Toc534494300)

[2.2. 死锁的定义 5](#_Toc534494301)

[2.3. 死锁的必要条件 5](#_Toc534494302)

[2.4. 死锁的处理方法 6](#_Toc534494303)

[2.4.1. 避免死锁 6](#_Toc534494304)

[3. 概要设计 7](#_Toc534494305)

[3.1. 设计思路 7](#_Toc534494306)

[3.2. 执行步骤 7](#_Toc534494307)

[3.3. 安全性算法设计 7](#_Toc534494308)

[3.4. 数据结构 7](#_Toc534494309)

[4. 详细设计 9](#_Toc534494310)

[4.1. 主要函数 9](#_Toc534494311)

[4.1. 数据判断 9](#_Toc534494312)

[4.2. 银行家算法流程图 10](#_Toc534494313)

[4.3. 安全性算法流程 11](#_Toc534494314)

[5. 数据调试 12](#_Toc534494315)

[6. 总结 15](#_Toc534494316)

[7. 参考文献 16](#_Toc534494317)

[8. 附录（源代码） 17](#_Toc534494318)

1. 绪论

1．1前言  
Dijkstra (1965)提出了一种能够避免死锁的调度算法，称为银行家算法。  
它的模型基于一个小城镇的银行家，他向一群客户分别承诺了一定的贷款额度，每个客户都有一个贷款额度，银行家知道不可能所有客户同时都需要最大贷款额，所以他只保留一定单位的资金来为客户服务，而不是满足所有客户贷款需求的最大单位。  
这里将客户比作进程，贷款比作设备，银行家比作系统。  
客户们各自做自己的生意，在某些时刻需要贷款。在某一时刻，客户已获得的贷款和可用的最大数额贷款称为与资源分配相关的系统状态。一个状态被称为是安全的，其条件是存在一个状态序列能够使所有的客户均得到其所需的贷款。如果忽然所有的客户都申请，希望得到最大贷款额，而银行家无法满足其中任何一个的要求，则发生死锁。不安全状态并不一定导致死锁，因为客户未必需要其最大贷款额度，但银行家不敢抱这种侥幸心理。  
银行家算法就是对每一个请求进行检查，检查如果满足它是否会导致不安全状态。若是，则不满足该请求；否则便满足。  
检查状态是否安全的方法是看他是否有足够的资源满足一个距最大需求最近的客户。如果可以，则这笔投资认为是能够收回的，然后接着检查下一个距最大需求最近的客户，如此反复下去。  
如果所有投资最终都被收回，则该状态是安全的，最初的请求可以批准。  
1．2研究意义  
在多道程序系统中，多个进程的并发执行来改善系统的资源利用率，提高系统的吞吐量，但可能发生一种危险——死锁。所谓死锁(Deadlock)，是指多个进程在运行过程中因争夺资源而造成的一种僵局（DeadlyEmbrace），当进程处于这种状态时，若无外力作用，他们都无法在向前推进。  
要预防死锁，有摒弃“请求和保持”条件，摒弃“不剥夺”条件，摒弃“环路等待”条件等方法。  
但是，在预防死锁的几种方法之中，都施加了较强的限制条件；而在避免死锁的方法中，所施加的限制条件较弱，有可能获得令人满意的系统性能。在该方法中把系统状态分为安全状态和不安全状态，便可避免死锁的发生。  
而最具代表性的避免死锁的算法，便是Dijkstra的银行家算法。  
利用银行家算法，我们可以来检测CPU为进程分配资源的情况，决定CPU是否响应某进程的请求并为其分配资源，从而很好避免了死锁的产生。

1. 需求分析

## **问题陈述**

在多道程序系统中，虽然能够借助于多个进程的并发执行，来改善系统资源的利用率，提高系统的吞吐量，但是依然有风险存在，那就是一锁死。 所谓锁死是指，多个进程在运行中因争夺资源而造成的一种僵局，当进程的这种僵持状态时，若无外力作用，它们将无法再向前推进。组程序中， 每个进程都无限等待被该组进程中的另进程所占有的资源，因而水远无法得到资源，这种现象就叫做进程死锁。

银行家算法是一种最具有代表性的避免死锁的算法。

要解释银行家算法，必须先解释操作系统的安全状态和不安全状态。所谓安全状态，是指系统能按照某种进程顺序{P1,P2. Pn}(称{P1,P2.., Pn }序列为安全序列)，来为每个进程Pi分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可以顺利完成。安全状态一定 没有死锁发生。

如果系统无法找到这样-一个安全序列，则称系统处于不安全状态。所谓安全序列，如果对每一个进程Pi(1<i<n)， 它以后尚需要的资源量不超过系统当前可利用的资源量与所有的进程Pj6<n)所占有的资源量之和，则称此进程序列{P1,2..，Pn}是安全的，称作安全序列。

## **死锁的定义**

如果一组进程中的每个进程都在等待仅有该组进程中的其他进程才能引发的事件，那么该组进程是死锁的（Deadlock）

## **死锁的必要条件**

1. 互斥条件。进程对所分配到的资源进行排他性使用，即在一段时间内，某资源只能被一个进程使用。如果此时还有其他进程请求该资源，则请求进程只能等待，直至占有该资源的进程用毕释放。
2. 请求和保持条件。进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其他进程占有，此时请求进程被阻塞，但对自己以获得的资源保持不放。
3. 不可抢占资源。进程以获得的资源在未被使用完之前不能被抢占，只能在进程使用完时由自己释放。
4. 循环等待条件。在发生死锁时，必然存在一个进程—资源循环链，即进程集合{P0,P1,P2,…,Pn}中的P0正在等待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。

## **死锁的处理方法**

1. 预防死锁。这是一种较简单和直观的事先预防方法。该方法是通过设置某些限制条件，去破坏产生死锁四个必要条件中的一个或几个来预防产生死锁。预防死锁是种较易实现的方法，已被广泛使用。
2. 避免死锁。同样是属于事先预防策略，但它并不是事先采取各种限制措施，去破坏产生死锁的四个必要条件，而是在资源的动态分配过程中，用某种方法防止系统进入不安全状态，从而可以避免发生死锁。
3. 检测死锁。这种方法无须事先采取任何限制性措施，允许进程在运行过程中发生死锁。但可通过检测机构及时地检测出死锁的发生，然后采取适当的措施，把进程从死锁中解脱出来。
4. 解除死锁。当检测到系统中已发生死锁时，就采取相应措施，将进程从死锁状态中解脱出来。常用的方法是撤消一些进程，回收它们的资源，将它们分配给已处于阻寨状态的进程，使其能继续运行。  
   上述的四种方法，从1到4对死锁的防范程度逐渐减弱，但对应的是资源利用率的提高，以及进程因资源因素而阻塞的频度下降（并发程度提高）

## 避免死锁

避免死锁同样属于实现与事先预防的策略但并不是实现采取某种限制措施，破坏产生死锁的必要条件，而是在资源动态分配过程中，防止系统进入不安全状态，以避免发生死锁。这种方法所施加的限制条件弱，可以获得较好的系统性能，目前常用此方法来避免死锁。

在死锁避免方法中，把系统状态分为安全状态和不安全状态。当系统处于安全状态时，可避免死锁。反之，当系统处于不安全状态时，则可能进入到死锁状态。

最具代表的避免死锁的算法是Dijkstra的银行家算法。

1. 概要设计

## **设计思路**

首先使用安全性算法检测当前系统中存在的进程是否安全，即此刻是否存在避免死锁的进程执行序列。若当前状态不安全需重新初始化系统中的进程。在确保安全后，用户可以提出进程请求，系统判断用户请求是否能够予以满足，若请求合法，则进行资源试分配，对试分配后的状态使用安全性算法进行检测，在确保安全后，对用户的请求进行资源分配，否则将本次的试探分配作废，恢复原来的资源分配状态，让用户请求的进程等待

## **执行步骤**

设进程Pi提出请求Request i [j]，则银行家算法按如下规则进行判断。

(1)如果Request i [j]<= Need[i,j]，则转（2)；否则，出错。

(2)如果Request i [j]<= Available[j]，则转（3)；否则，等待。

(3)系统试探分配资源，修改相关数据：

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

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

Need[i,j]-=Request[i,j];

(4)系统执行安全性检查，如安全，则分配成立；否则试探险性分配作废，系统恢复原状，进程等待。

## **安全性算法设计**

1)设置两个工作向量Work=Available;Finish

(2)从进程集合中找到一个满足下述条件的进程，

Finish[i]==false;

Need[i.j]<=Work[j];

如找到，执行（3)；否则，执行（4)

(3)设进程获得资源，可顺利执行，直至完成，从而释放资源。

Work[j]=Work[j]+Allocation[i.j];

Finish[i]=true;

GOTO 2

(4)如所有的进程Finish= true，则表示安全；否则系统不安全。

## **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]

1. 详细设计

## **主要函数**

1. void ProbeAlloc(int process, RESOURCE \*res) 资源试分配函数

在进程请求资源数合法的情况下，系统试探着把资源分配给请求进程，用此函数来修改资源数据结构的值

1. void RollBack(int process, RESOURCE \*res) 恢复资源分配状态函数

试分配后系统处于不安全状态，则将本次的试探分配作废，用此函数来恢复原来的资源分配状态

1. bool SafeCheck()安全性检测函数

执行安全性算法检查系统是否处于安全状态

1. bool request(int process, RESOURCE \*res) 进程资源请求函数

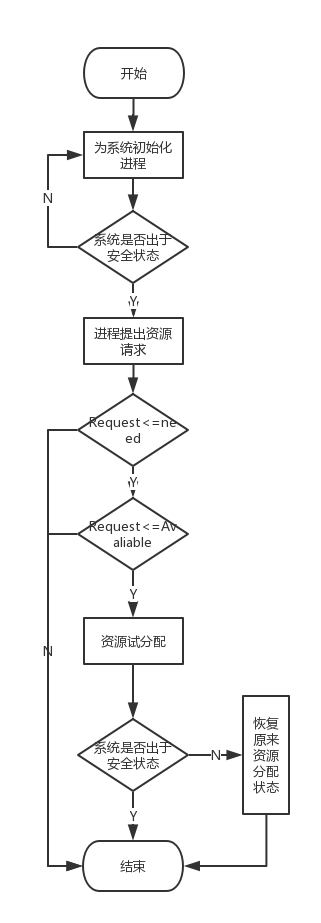
发出资源请求，并在确保请求合法的情况下调用void ProbeAlloc(int process, RESOURCE \*res) 系统试探着把资源分配给请求进程，用此函数来修改资源数据结构的值，并调用安全性检测函数bool SafeCheck()检查试探分配后系统是否处于安全状态，若不安全还需调用恢复资源分配状态函数void RollBack(int process, RESOURCE \*res)将本次的试探分配作废，恢复原来的资源分配状态

## **数据判断**

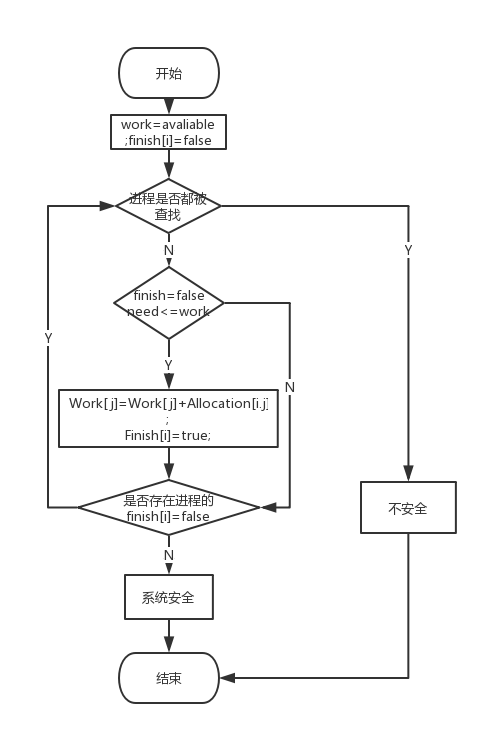
1. 对请求的进程名进行判断，当请求进程名0>i&&i>=PROCESS\_NUMBE时，提示该进程不存在，请重新输入。

2）对进程请求资源数进行判断，如果Request i [j]>= Need[i,j],提示进程请求资源数大于所需资源数。如果Request i [j]>= Available[j]，提示系统尚无足够资源满足进程要求，需等待。

## **银行家算法流程图**



## **安全性算法流程**



1. 数据调试

1）首先初始化系统可提供的资源ABC数分别为3，3，2，并初始化五个进程P0,P1,P2,P3,P4.

最大需求矩阵MAX

{7,5,3},

{3,2,2},

{9,0,2},

{2,2,2},

{4,3,3}

已分配资源数矩阵 Allocation

{0,1,0},

{2,0,0},

{3,0,2},

{2,1,1},

{0,0,2}

需求矩阵 Need

{7,4,3},

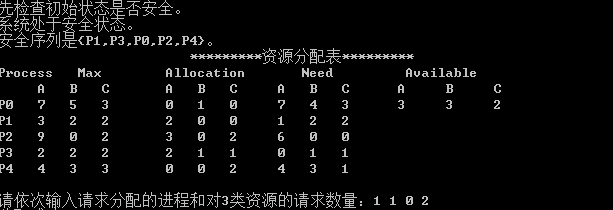
{1,2,2},

{6,0,0},

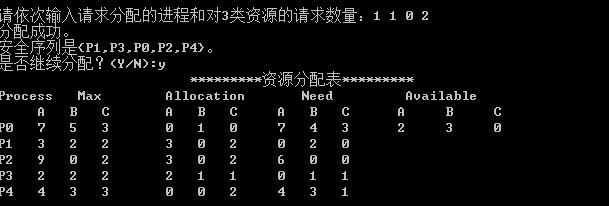
{0,1,1},

{4,3,1}

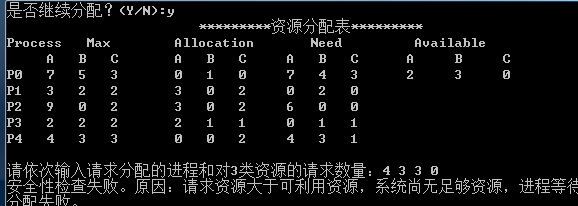
2）运行程序检测当前状态是否安全（下图为运行结果）



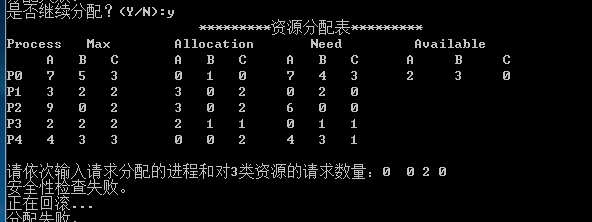
3）进程P1对资源ABC提出请求并成功分配，更新后的资源分配表成功显示。



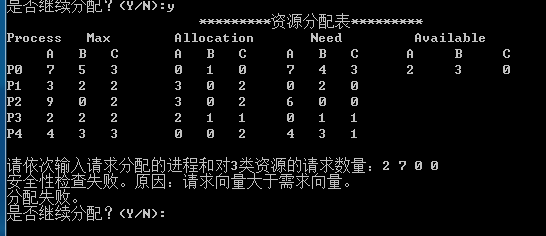
4）进程P4对三类资源进行请求，请求失败。失败原因：请求资源数过大，系统无法满足，进程必须等待。



5）进程P0对三类资源请求，虽满足了Request i [j]<= Need[i,j]，Request i [j]<= Available[j]，但是试探分配后，安全性算法检测到系统此时的进程已无安全序列，从何试探险性分配作废，系统恢复原状，进程等待。



6）进程P2提出资源请求分配失败，失败原因：请求资源数大于进程所需资源最大数。



1. 总结

在银行家算法这个系统之中，所采用的数据结构应是最基本的部分。银行家算法的数据结构我们采用了一维数组与二维数组来存储，比如最大需求量Max[][]、已分配资源数Allocation[][]、仍需求资源数Need[][]、以及系统可利用的资源数、申请各类资源等数组。

数据结构虽然重要但却只是基础，而最主要的用以实现系统功能的应该有两个部分，一是用银行家算法来判断，二是用安全性算法来检测系统的安全性。

首先，输入欲申请资源的进程以及其所申请的资源数，存放在Request数组中。

然后，判断进程请求的资源数是否大于其所需的资源数，若大于则报错并返回，若不大于则继续判断它是否大于系统在此时刻可利用的资源数，同样，如果大于则报错并反回，如果不大于则进行预分配，之后再调用安全型算法检查。

最后，无论此次分配是否成功，我们都可以选择继续分配或者退出系统。

首先，Finish[]为布尔型，默认是False，即该进程未完成。而Work——即该系统中可以用来工作的资源数——最开始为系统最初可以用的资源数。

然后，我们从第一个进程开始判断该进程未完成且其所需求的资源量不大于该系统中可以用来工作的资源量这个条件是否成立，即Finish[]=False且Need[][]<=Work[]是否成立。成立的话则将当前在工作的资源量与该进程已分配的资源量相加，存放于当前可用来工作的资源量当中，即Work[]=Work[]+Allocation，并将Finish[]的值改为True。否则便将此进程的优先级减一，排在队位，然后开始往后循环。

待所有的进程循环完毕，我们再次判断是否还存在进程的Finish[]=False，如果仍存在，则说明系统没有安全序列，处于不安全状态，不可以进行分配；否则，系统处于安全状态，将预分配变为实际分配，求出安全序列并且将实际分配后的资源分配情况打印输出。

除此之外，在程序当中，我们也得强调一下对输入的合法性的判断。比如，我们输入的欲申请资源的进程号没有在系统已存在的进程当中，或者进程号定义为整型，但是却错输成字母等情况，我们需要对这些情况进行判断，让程序报错返回而并非因错误而中断。

这样的情况处理起来比较麻烦，相当于对每次输入针对各种不同的情况都得做判断。我也没有涵盖全部的输入，仅仅只是对输入的进程号不在已存在进程当中、以及输入的操作选择不存在两种情况分别作了判断，并且针对第二种情况设定了五次输入错误的话系统关闭的功能。而因为对于某些——比如进程号——本来设定就是整型，因此对输入的是字母的判别因比较复杂而未能加上。

总之，银行家算法是避免死锁的主要方法，其思路在很多方面都非常值得我们来学习借鉴。

[1]汤小丹,梁红兵,哲凤屏,汤子瀛.计算机操作系统.4版.西安:西安电子科技大学出版社,2014.

[2]谭浩强.C程序设计.4版.北京:清华大学出版社,2010.

[3] 严蔚敏,吴伟民.数据结构.北京:清华大学出版社,2006.

1. 附录（源代码）

1.banker.cpp

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <stdio.h>

#include <string.h>

#include "banker.h"

//试探分配

void ProbeAlloc(int process, RESOURCE \*res)

{

Available.A -= res->A;

Available.B -= res->B;

Available.C -= res->C;

Allocation[process].A += res->A;

Allocation[process].B += res->B;

Allocation[process].C += res->C;

Need[process].A -= res->A;

Need[process].B -= res->B;

Need[process].C -= res->C;

}

//若试探分配后进入不安全状态，将分配回滚

void RollBack(int process, RESOURCE \*res)

{

Available.A += res->A;

Available.B += res->B;

Available.C += res->C;

Allocation[process].A -= res->A;

Allocation[process].B -= res->B;

Allocation[process].C -= res->C;

Need[process].A += res->A;

Need[process].B += res->B;

Need[process].C += res->C;

}

bool SafeCheck()//安全性检查

{

RESOURCE Work = Available;

bool Finish[PROCESSES\_NUMBER] = { false,false,false,false,false };

int i;

int j = 0;

for (i = 0; i < PROCESSES\_NUMBER; i++)

{

//是否已检查过

if (Finish[i] == false)

{

//是否有足够的资源分配给该进程

if (Need[i].A <= Work.A && Need[i].B <= Work.B && Need[i].C <= Work.C)

{

//有则使其执行完成，并将已分配给该进程的资源全部回收

Work.A += Allocation[i].A;

Work.B += Allocation[i].B;

Work.C += Allocation[i].C;

Finish[i] = true;

safe[j++] = i;

i = -1; //重新进行遍历

}

}

}

//如果所有进程的Finish向量都为true则处于安全状态，否则为不安全状态

for (i = 0; i < PROCESSES\_NUMBER; i++)

{

if (Finish[i] == false)

{

return false;

}

}

return true;

}

//资源分配请求

bool request(int process, RESOURCE \*res)

{

//request向量需小于Need矩阵中对应的向量

if (res->A <= Need[process].A && res->B <= Need[process].B && res->C <= Need[process].C)

{

//request向量需小于Available向量

if (res->A <= Available.A && res->B <= Available.B && res->C <= Available.C)

{

//试探分配

ProbeAlloc(process, res);

//如果安全检查成立，则请求成功，否则将分配回滚并返回失败

if (SafeCheck())//执行安全算法并能找到安全序列

{

return true;

}

else

{

printf("安全性检查失败。\n");

printf("正在回滚...\n");

RollBack(process, res);

}

}

else

{

printf("安全性检查失败。原因：请求资源大于可利用资源，系统尚无足够资源，进程等待\n");

}

}

else

{

printf("安全性检查失败。原因：请求向量大于需求向量。\n");

}

return false;

}

//输出资源分配表

void PrintTable()

{

printf("\t\t\t\*\*\*\*\*\*\*\*\*资源分配表\*\*\*\*\*\*\*\*\*\n");

printf("Process Max Allocation Need Available\n");

printf(" A B C A B C A B C A B C \n");

printf("P0 %d %d %d %d %d %d %d %d %d %d %d %d \n", Max[0].A, Max[0].B, Max[0].C,Allocation[0].A, Allocation[0].B, Allocation[0].C, Need[0].A, Need[0].B, Need[0].C, Available.A, Available.B, Available.C);

printf("P1 %d %d %d %d %d %d %d %d %d\n", Max[1].A, Max[1].B, Max[1].C, Allocation[1].A, Allocation[1].B, Allocation[1].C, Need[1].A, Need[1].B, Need[1].C);

printf("P2 %d %d %d %d %d %d %d %d %d\n", Max[2].A, Max[2].B, Max[2].C, Allocation[2].A, Allocation[2].B, Allocation[2].C, Need[2].A, Need[2].B, Need[2].C);

printf("P3 %d %d %d %d %d %d %d %d %d\n", Max[3].A, Max[3].B, Max[3].C, Allocation[3].A, Allocation[3].B, Allocation[3].C, Need[3].A, Need[3].B, Need[3].C);

printf("P4 %d %d %d %d %d %d %d %d %d\n", Max[4].A, Max[4].B, Max[4].C, Allocation[4].A, Allocation[4].B, Allocation[4].C, Need[4].A, Need[4].B, Need[4].C);

printf("\n");

}

int main()

{

int ch;

printf("先检查初始状态是否安全。\n");

if (SafeCheck())

{

printf("系统处于安全状态。\n");

printf("安全序列是{P%d,P%d,P%d,P%d,P%d}。\n", safe[0], safe[1], safe[2], safe[3], safe[4]);

}

else

{

printf("系统处于不安全状态。程序将退出...\n");

goto over;

}

do

{

int process;

RESOURCE res;

PrintTable();

printf("请依次输入请求分配的进程和对3类资源的请求数量：");

scanf("%d%d%d%d", &process, &res.A, &res.B, &res.C);

if (request(process, &res))

{

printf("分配成功。\n");

printf("安全序列是{P%d,P%d,P%d,P%d,P%d}。\n", safe[0], safe[1], safe[2], safe[3], safe[4]);

}

else

{

printf("分配失败。\n");

}

printf("是否继续分配？(Y/N):");

fflush(stdin); //虽然C标准不支持这种用法，但是VC++支持

ch = getchar();

} while (ch == 'Y' || ch == 'y');

over:

printf("执行完毕。");

return 0;

}

1. banker.h

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#pragma warning(disable:4996)

//typedef int bool;

#define false 0

#define true !false

//系统中所有进程数量

#define PROCESSES\_NUMBER 5

typedef struct {

int A;

int B;

int C;

int D;

}RESOURCE;

//最大需求矩阵 Max

RESOURCE Max[PROCESSES\_NUMBER] =

{

{7,5,3},

{3,2,2},

{9,0,2},

{2,2,2},

{4,3,3}

};

//已分配资源数矩阵 Allocation

RESOURCE Allocation[PROCESSES\_NUMBER] =

{

{0,1,0},

{2,0,0},

{3,0,2},

{2,1,1},

{0,0,2}

};

//需求矩阵

RESOURCE Need[PROCESSES\_NUMBER] =

{

{7,4,3},

{1,2,2},

{6,0,0},

{0,1,1},

{4,3,1}

};

RESOURCE Available = { 3,3,2};//可用资源向量

int safe[PROCESSES\_NUMBER];