****

操作系统原理

实验报告

|  |  |
| --- | --- |
| 学生姓名 | 吕环翥 |
| 学 号 | 8211220709 |
| 专业班级 | 计科T2102 |
| 指导教师 | 胡小龙 |
| 学 院 | 计算机学院 |
| 完成时间 | 2024-6-28 |

#### 一、实验目的

理解银行家算法的基本原理和其在操作系统资源分配中的应用。通过实现和模拟该算法，学习如何在多进程环境中动态管理资源，以避免系统死锁。实验的具体目标包括：

1. 掌握银行家算法的核心思想，即根据进程的最大需求、已分配资源和可用资源来判断资源请求的合理性。
2. 理解算法如何在进程请求资源时评估系统的安全性，确保所有进程都能在有限资源条件下顺利完成。
3. 模拟实际的资源分配过程，验证银行家算法在资源管理中的有效性。
4. 通过编程实现算法，提高在复杂逻辑和多维数组处理上的能力，增强对系统资源管理的理解。
5. 通过不同条件下的实验，观察资源分配对系统安全性的影响，分析系统在安全性与资源利用率之间的平衡。

#### 二、实验内容

**输入资源种类和每种资源的总数量**  
 首先输入系统中可用资源的种类数 N，然后逐个输入每种资源的名称（如 A, B, C）及其总数量。根据输入初始化系统的可用资源向量 Available，用于表示当前系统中各类资源的总数量。

**输入系统中的进程数**  
 输入系统中存在的进程数 M，表示需要模拟的进程数量。此信息将用于设置需求矩阵和分配矩阵的行数，确保每个进程都有对应的资源记录。

**输入每个进程对每种资源的最大需求矩阵 Max**  
 输入每个进程对每种资源的最大需求矩阵 Max。用户按行输入数据，表示每个进程在执行过程中可能需要的各类资源的最大数量。该矩阵用于记录每个进程对系统中各资源的最大需求。

**输入每个进程当前已分配的资源矩阵 Allocation，并计算每个进程的剩余需求矩阵 Need**  
 输入已分配给每个进程的资源矩阵 Allocation，表示当前已分配的各类资源数量。通过 Need = Max - Allocation 计算出每个进程的剩余需求矩阵 Need，用于表示每个进程在完成任务前仍需的各类资源量。此步骤同时更新系统可用资源量。

**显示当前系统状态**  
 输出当前系统状态，包括可用资源向量 Available、最大需求矩阵 Max、已分配矩阵 Allocation 和需求矩阵 Need。通过这些矩阵，直观展示当前系统的资源分配情况，便于后续的安全性检查和资源请求模拟。

#### 三、实验要点说明

**数据结构**

* 1. Max[M][N]：二维数组，记录每个进程对每种资源的最大需求。初始化时由用户输入，表示进程在执行过程中可能需要的最大资源量。
  2. Allocation[M][N]：二维数组，记录当前已分配给每个进程的资源量。由用户输入，表示系统已分配的各类资源数量。
  3. Need[M][N]：通过 Need[i][j] = Max[i][j] - Allocation[i][j] 计算，表示每个进程还需要的资源量。该数组在初始化时自动计算，用于判断进程是否还需额外资源。
  4. Available[N]：一维数组，表示系统当前可用的各类资源数量。初始化时计算为各资源的总数量减去已分配给各进程的数量。
  5. Request[N]：一维数组，用户输入，表示某个进程请求的各类资源数量，用于模拟资源请求的处理。

**初始化过程**  
在程序启动时，首先输入资源的种类数 N，然后输入每种资源的名称和总数量，初始化 Available 数组。接着，输入系统中进程数 M，用于定义矩阵的维度。随后，用户输入 Max 矩阵，表示每个进程的最大资源需求。之后，输入 Allocation 矩阵，表示当前各进程已分配的资源数量。通过 Max 和 Allocation 计算得到 Need 矩阵。此时，系统根据已分配资源更新 Available 数组，计算公式为 Available[j] = Available[j] - sum(Allocation[i][j])。

**安全性算法**  
安全性算法用于判断当前资源分配是否会导致死锁。算法通过模拟资源分配过程，检查是否存在一个安全序列，使得每个进程都能在其最大需求范围内完成执行。算法步骤如下：

* 1. 初始化 Work 数组为当前 Available，Finish 数组为 False。
  2. 遍历所有进程，对于每个尚未完成的进程，检查其 Need 是否小于等于 Work。
  3. 若满足条件，将 Allocation 加入 Work，标记 Finish 为 True，并将该进程加入安全序列。
  4. 若所有进程均能满足条件，系统处于安全状态；否则，不安全。

**资源请求过程**  
用户可以输入进程号和请求的资源数量，模拟资源请求。程序首先检查请求是否合法，即 Request[j] <= Need[i][j] 和 Request[j] <= Available[j]。若不合法，则拒绝请求。若合法，进入试探性分配阶段。

**试探性分配与回滚**  
在试探性分配中，暂时将 Request 中的资源分配给相应进程，更新 Available、Allocation 和 Need 数组。然后调用安全性算法检查系统是否安全：

* 1. 如果安全，确认分配并更新系统状态。
  2. 如果不安全，撤销此次分配，恢复 Available、Allocation 和 Need 数组至原状态，保证系统不会陷入不安全状态。

这种试探性分配与回滚机制确保了系统资源管理的动态性和安全性，使得系统在面对多进程资源请求时能有效避免死锁。

银行家算法实例

#### 四、银行家算法实例

假定系统中有五个进程{P0、P1、P2、P3、P4}和三种类型资源{A、B、C}，每一种资源的数量分别为10、5、7。各进程的最大需求、T0时刻资源分配情况如下所示。



试问：

①T0时刻是否安全？

② T0之后的T1时刻P1请求资源Request1(1,0,2)是否允许？

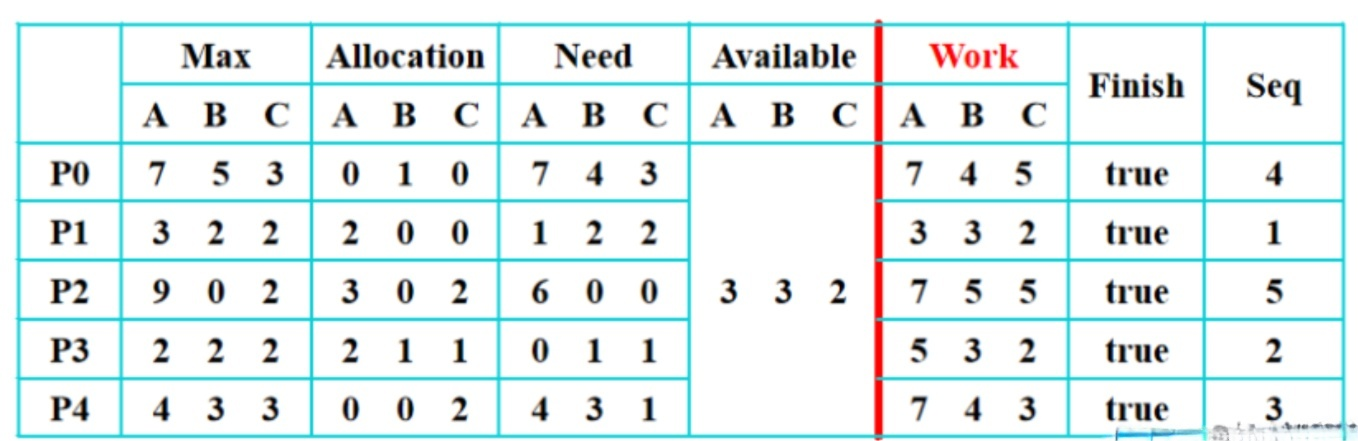
③ T1之后的T2时刻P4请求资源Request4(3,3,0)是否允许？

④ T2之后的T3时刻P0请求资源Request0(0,2,0)是否允许？

解：

① T0时刻是否安全？ 工作向量Work.它表示系统可提供给进程继续运行所需要的各类资源的数目

(1) T0时刻安全性分析



存在安全序列{P1, P3, P4, P0, P2}，系统安全。

(2) T0之后的T1时刻P1请求资源Request1(1,0,2)可否允许？

①Request1(1,0,2) ≤ Need1(1,2,2)，P1请求在最大需求范围内

②Request1(1,0,2) ≤ Available1(3,3,2)，可用资源可满足P1请求需要

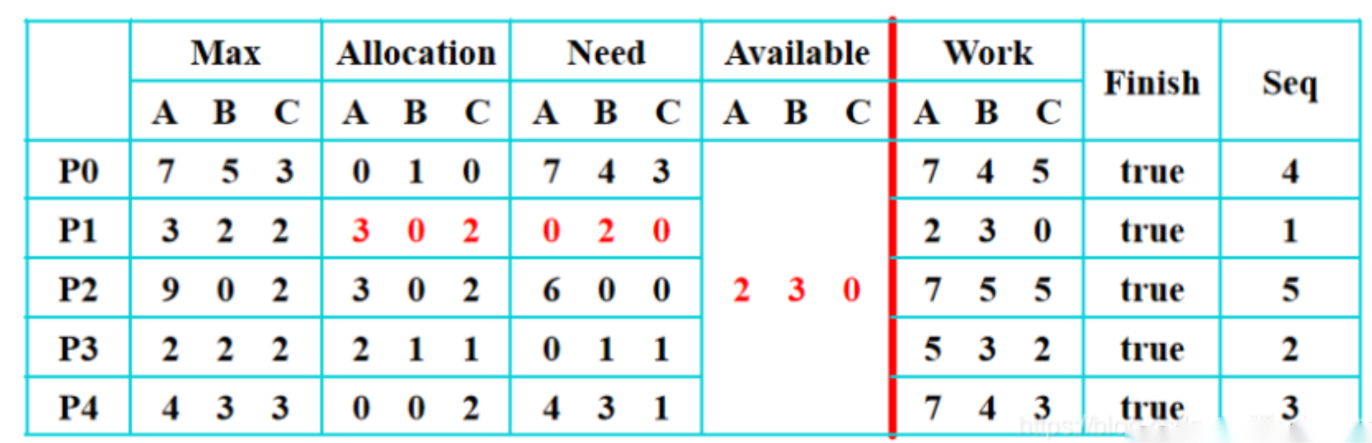
③假定可为P1分配，修改Available,Allocation1,Need1向量

Available(2,3,0) = Available(3,3,2)-Request1(1,0,2);

Need1(0,2,0) = Need1(1,2,2)-Request1(1,0,2);

Allocation1(3,0,2) =Allocation1(2,0,0)+Request1(1,0,2);

④利用安全性算法检查试探将资源分配后状态的安全性



存在安全序列{P1, P3, P4, P0, P2}，所以试探将资源分配给进程P1后的状态是安全的，可将资源分配给进程P1。

③ T1之后的T2时刻P4请求资源Request4(3,3,0)是否允许？

Request4(3,3,0)≤Need4(4,3,1)，P4请求在最大需求范围内。

Request4(3,3,0)≤Available(2,3,0)不成立，即可用资源暂不能满足P4请求资源需要，P4阻塞等待。

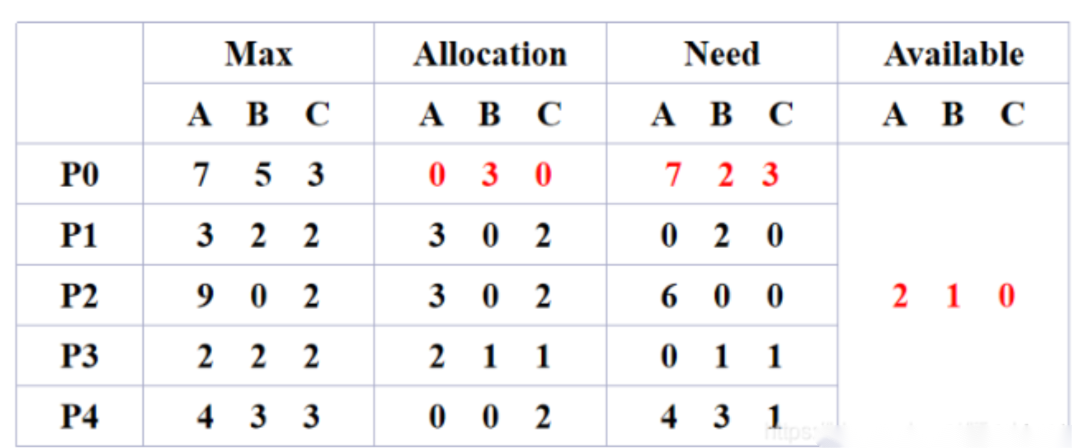
P4请求资源Request4(3,3,0)不允许。

④ T2之后的T3时刻P0请求资源Request0(0,2,0)是否允许？

Request0(0，2，0)≤Need0(7，4，3)；

Request0(0，2，0)≤Available(2，3，0)；

系统暂时先假定可为P0分配资源，并修改有关数据，如下图所示：



进行安全性检查：可用资源Available(2，1，0)已不能满足任何进程的需要，故系统进入不安全状态，此时系统不分配资源。

#### 五、实验流程

**初始化（init()）**  
程序开始时，初始化系统资源，包括可用资源、最大需求、已分配资源和需求矩阵。用户输入进程和资源的信息，计算可用资源向量。

**显示数据（showdata()）**  
输出当前系统的资源状态，包括每个进程的最大需求、已分配资源和剩余需求矩阵，以及系统的可用资源。

**安全性检查（safe()）**  
检查系统当前状态是否安全。若存在安全序列，系统处于安全状态，否则输出“不安全”。

**输出安全序列**  
如果系统安全，输出安全序列，显示可以顺序执行的进程。

**提出请求（request[i]）**  
用户输入某进程对资源的请求。

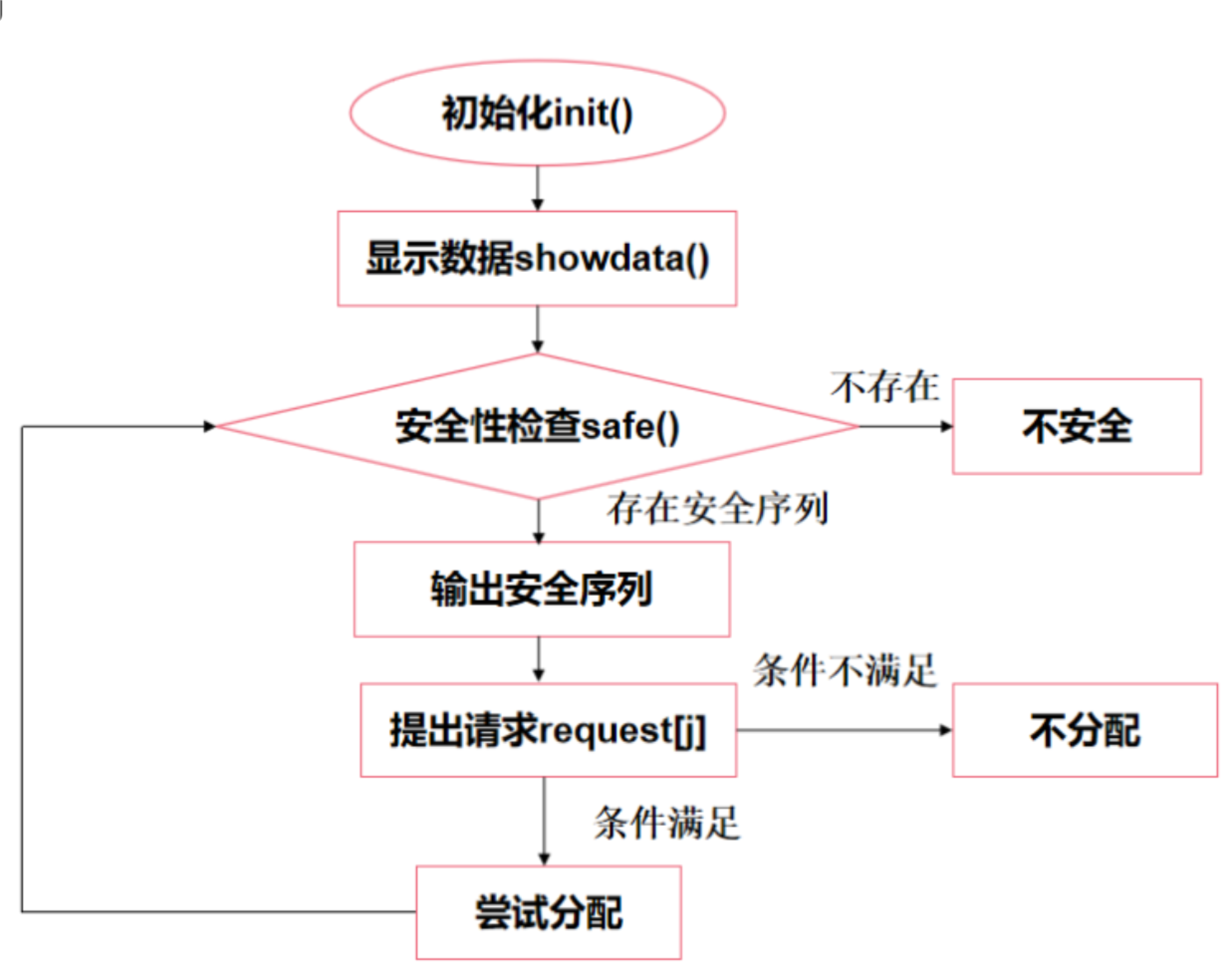
**条件检查**  
检查请求是否合理：

* 请求的资源是否超过进程的需求。
* 请求的资源是否超过系统的可用资源。 若不满足条件，则拒绝分配。

**尝试分配**  
如果请求满足条件，尝试分配资源。更新系统状态并再次进行安全性检查。

**安全性检查**  
检查分配后的系统状态，若仍然安全，则确认分配，否则回滚分配，保持系统安全。

整个流程确保在资源分配过程中，系统始终保持安全，避免死锁发生。



#### 六、实验代码

#include <stdio.h>

#include <stdlib.h>

#define FALSE 0

#define TRUE 1

// 主要数据结构

char resourceName[100] = {0}; // 资源名称

int max[100][100] = {0}; // 最大需求矩阵

int allocation[100][100] = {0}; // 系统已分配矩阵

int need[100][100] = {0}; // 还需要资源矩阵

int available[100] = {0}; // 可用资源矩阵

int request[100] = {0}; // 请求资源向量

int work[100] = {0}; // 存放系统可提供资源量

int finish[100] = {0}; // 标记系统是否有足够的资源分配给各个进程

int safeSequence[100] = {0}; // 存放安全序列

int processCount, resourceCount;

// 初始化数据

void initialize() {

int i, j;

int temp[100] = {0}; // 临时存储分配的资源总数

// 输入资源数量和初始可用资源

printf("请输入资源的种类数: ");

scanf("%d", &resourceCount);

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

printf("请输入资源 %d 的名称: ", i + 1);

fflush(stdin);

scanf(" %c", &resourceName[i]);

printf("请输入资源 %c 的初始总数量: ", resourceName[i]);

scanf("%d", &available[i]);

}

// 输入进程数量

printf("\n请输入进程的数量: ");

scanf("%d", &processCount);

// 输入每个进程的最大需求矩阵

printf("请输入每个进程的最大需求矩阵 [Max]:\n");

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

printf("进程 P%d:\n", i);

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

printf("资源 %c: ", resourceName[j]);

scanf("%d", &max[i][j]);

}

}

// 输入每个进程已分配的资源矩阵

printf("请输入每个进程已分配的资源矩阵 [Allocation]:\n");

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

printf("进程 P%d:\n", i);

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

printf("资源 %c: ", resourceName[j]);

scanf("%d", &allocation[i][j]);

need[i][j] = max[i][j] - allocation[i][j]; // 计算需求矩阵

temp[j] += allocation[i][j]; // 更新已分配总数

}

}

// 计算系统可用资源向量

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

available[j] -= temp[j];

}

}

// 显示资源分配矩阵

void displayData() {

int i, j;

printf("\n当前系统资源状态:\n");

printf("Available: ");

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

printf("%c=%d ", resourceName[i], available[i]);

}

printf("\n\n进程 | Max | Allocation | Need \n");

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

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

printf("P%d | ", i);

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

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

}

printf(" | ");

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

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

}

printf(" | ");

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

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

}

printf("\n");

}

printf("\n");

}

// 检查系统是否安全

int checkSafety() {

int i, j, k = 0;

int canProceed;

// 初始化 Work 和 Finish

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

work[j] = available[j];

}

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

finish[i] = FALSE;

}

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

canProceed = TRUE;

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

if (need[i][j] > work[j]) {

canProceed = FALSE;

break;

}

}

if (canProceed && !finish[i]) {

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

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

}

finish[i] = TRUE;

safeSequence[k++] = i;

i = -1; // 重新检查所有进程

}

}

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

if (!finish[i]) {

printf("系统不安全!\n");

return FALSE;

}

}

printf("系统是安全的，安全序列为: ");

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

printf("P%d ", safeSequence[i]);

if (i < k - 1) {

printf("-> ");

}

}

printf("\n");

return TRUE;

}

// 银行家算法处理资源请求

void bankerAlgorithm() {

int i, j, processNum;

printf("请输入请求资源的进程号 (0-%d): ", processCount - 1);

scanf("%d", &processNum);

printf("请输入进程 P%d 的请求资源数:\n", processNum);

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

printf("资源 %c: ", resourceName[j]);

scanf("%d", &request[j]);

}

// 检查请求是否合法

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

if (request[j] > need[processNum][j]) {

printf("请求的资源超过进程需要的资源，不予分配！\n");

return;

}

if (request[j] > available[j]) {

printf("请求的资源超过系统可用资源，不予分配！\n");

return;

}

}

// 试探性分配

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

available[j] -= request[j];

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

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

}

// 检查系统是否安全

if (checkSafety()) {

printf("请求已被批准，资源已分配。\n");

} else {

// 回滚

printf("请求导致系统不安全，回滚。\n");

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

available[j] += request[j];

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

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

}

}

}

int main() {

char choice;

initialize();

displayData();

if (!checkSafety()) {

printf("初始状态不安全，程序退出。\n");

exit(0);

}

do {

printf("\n银行家算法模拟:\n");

printf("R(r) - 请求资源\n");

printf("E(e) - 退出\n");

printf("请选择操作: ");

fflush(stdin);

scanf(" %c", &choice);

switch (choice) {

case 'r':

case 'R':

bankerAlgorithm();

displayData();

break;

case 'e':

case 'E':

printf("退出程序。\n");

break;

default:

printf("无效选择，请重新输入。\n");

break;

}

} while (choice != 'e' && choice != 'E');

return 0;

}

#### 七、结果展示

**初始状态**  
输入资源种类、每种资源的总数量、进程数，以及每个进程的最大需求和已分配资源。系统计算并显示初始的 Available、Max、Allocation 和 Need 矩阵。

当前系统资源状态:

Available: A=3 B=3 C=2

进程 | Max | Allocation | Need

-----|---------|------------|--------

P0 | 7 5 3 | 0 1 0 | 7 4 3

P1 | 3 2 2 | 2 0 0 | 1 2 2

P2 | 9 0 2 | 3 0 2 | 6 0 0

P3 | 2 2 2 | 2 1 1 | 0 1 1

P4 | 4 3 3 | 0 0 2 | 4 3 1

**安全性检查**  
运行安全性算法，判断初始状态是否安全，并输出安全序列。

系统是安全的，安全序列为: P1 -> P3 -> P4 -> P0 -> P2

**资源请求模拟**  
用户输入进程号及其请求的资源数量，系统检查请求合法性。

请输入请求资源的进程号 (0-4): 1

请输入进程 P1 的请求资源数:

资源 A: 1

资源 B: 0

资源 C: 2

**请求处理结果**  
检查请求是否满足条件，若满足，尝试分配资源，并进行安全性检查。

请求已被批准，资源已分配。

系统是安全的，安全序列为: P3 -> P4 -> P0 -> P2 -> P1

**回滚处理**  
如果分配导致系统不安全，程序回滚到初始状态。

请求导致系统不安全，回滚。

**最终状态展示**  
经过若干次资源请求后，系统输出当前资源状态，显示更新后的 Available、Max、Allocation 和 Need 矩阵。

当前系统资源状态:

Available: A=2 B=3 C=0

进程 | Max | Allocation | Need

-----|---------|------------|--------

P0 | 7 5 3 | 0 1 0 | 7 4 3

P1 | 3 2 2 | 3 0 2 | 0 2 0

P2 | 9 0 2 | 3 0 2 | 6 0 0

P3 | 2 2 2 | 2 1 1 | 0 1 1

P4 | 4 3 3 | 0 0 2 | 4 3 1

#### 八、实验心得与总结

通过本次实验，我深入理解了银行家算法在资源分配中的应用，以及如何利用该算法有效避免系统死锁。实验过程中，我实现了资源的动态管理，模拟了多进程环境下的资源分配，并验证了系统的安全性。

##### 实验心得：

**算法理解**： 银行家算法是确保资源分配安全性的重要机制。在模拟过程中，通过安全性检查，能够有效识别潜在的死锁情况，确保系统始终处于安全状态。

**编程技巧**： 在编写代码时，我加深了对多维数组的理解，特别是在资源矩阵的初始化和更新方面。试探性分配和回滚机制的实现，使我更加熟练地处理复杂的条件判断。

**系统安全性**： 通过多次请求和分配模拟，验证了银行家算法在确保系统安全性方面的有效性。实验中，通过输出安全序列，可以清晰地了解哪些进程能够按序执行，避免了资源争夺引发的死锁。

**用户体验**： 为了提升程序的用户体验，我增加了输入校验和友好的提示信息，帮助用户更好地理解操作步骤和结果。这也提高了代码的可读性和易用性。

##### 实验总结：

本次实验不仅加强了我对银行家算法的理解，也让我体会到了算法在实际系统中的重要性。在多进程操作系统中，合理的资源管理和死锁避免是系统稳定运行的关键。通过本实验，我进一步提升了在资源分配和系统安全性分析方面的能力，为日后更复杂的操作系统设计奠定了坚实的基础。