安徽大学人工智能学院《操作系统》实验报告

学号___WA2214014__ 姓名___杨跃浙___ 年级__大三__

【实验名称】_____实验三　死锁避免_____

【实验内容】

1．实现银行家算法；

2．安全性检查算法；

1.银行家算法

银行家算法最初级为银行系统设计，以确保银行在发放现金贷款时，不会发生不能满足所有客户需要的情况。在 OS 设计中，用它来避免死锁。

为实现银行家算法，每个新进程在进入系统时它必须申明在运行过程中，可能需要的每种资源类型的最大单元数目，其数目不应超过系统所拥有的资源总量。当某一进程请求时，系统会自动判断请求量是否小于进程最大所需，同时判断请求量是否小于当前系统资源剩余量。若两项均满足，则系统试分配资源并执行安全性检查算法。

具体算法细节参考教材相关内容。

2.安全性检查算法 ：

.安全性检查算法用于检查系统进行资源分配后是否安全，若安全系统才可以执行此次分配;若不安全,则系统不执行此次分配。安全性检查算法原理为：在系统试分配资源后，算法从现有进程列表寻找出一个可执行的进程进行执行,执行完成后回收进程占用资源；进而寻找下一个可执行进程。当进程需求量大于系统可分配量时，进程无法执行。当所有进程均可执行，则产生一个安全执行序列，系统资源分配成功。若进程无法全部执行，即无法找到一条安全序列，则说明系统在分配资源后会不安全，所以此次分配失败。

具体算法细节参考教材相关内容。

【实验要求】

1、 根据两种算法的特点，分别设计数据结构。

2、 编写上述两种算法。

3、测试用例：教材 3.7 节（避免死锁） 中的“银行家算法之例”的（1）－（5）。

4、银行家算法的例子

假定系统中有 5 个进程 P0 到 P4，3 类资源及数量分别为 A(10 个）,B（5 个）,C（7 个），T0 时刻的资源分配情况。

	Max A B C	Allocation A B C	Need A B C	Available A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2
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	

表 1　T0 时刻的资源分配表

（1）T0 时刻的安全性

利用安全性算法对 T0 时刻的资源分配情况进行分析，可得下表

	Work A B C	Need A B C	Allocation A B C	Work+ Allocation A B C	Finish
P_1	3 3 2	1 2 2	2 0 0	5 3 2	true
P_3	5 3 2	0 1 1	2 1 1	7 4 3	true
P_4	7 4 3	4 3 1	0 0 2	7 4 5	true
P_2	7 4 5	6 0 0	3 0 2	10 4 7	true
P_0	10 4 7	7 4 3	0 1 0	10 5 7	true

表 2 T0 时刻的安全性检查表

分析得知：　T0 时刻存在着一个安全序列{P1 P3 P4 P2 P0}，故

系统是安全的。

(2) P1 请求资源 Request1(1,0,2)

P1 发出请求向量 Request1(1,0,2),系统按银行家算法进行检查:

1) Request1(1,0,2)≤Need1 (1,2,2)

2) Request1 (1,0,2) ≤ Available (3,3,2)

3) 系统试为 P1 分配资源,并修改相应的向量(见下表(3)所示)Avaliable, Need, Allocation

	Max A B C	Allocation A B C	Need A B C	Available A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2 (2 3 0)
P1	3 2 2	2 0 0 (3 0 2)	1 2 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	

表 3　P1 请求资源时的资源分配表

4) 利用安全性算法检查资源分配后此时系统是否安全.如表 4

	Work A B C	Need A B C	Allocation A B C	Work+ Allocation A B C	Finish
P_1	2 3 0	0 2 0	3 0 2	5 3 2	true
P_3	5 3 2	0 1 1	2 1 1	7 4 3	true
P_4	7 4 3	4 3 1	0 0 2	7 4 5	true
P_2	7 4 5	6 0 0	3 0 2	10 4 7	true
P_0	10 4 7	7 4 3	0 1 0	10 5 7	true

表 4　P1 请求资源时的安全性检查表

由安全性检查分析得知：　此时刻存在着一个安全序列{P1 P3 P4 P2 P0}，故系统是安全的，可以立即将 P1 所申请的资源分配给它。

（3）P4 请求资源 Request4(3,3,0)

P4 发出请求向量 Request4(3,3,0)，系统按银行家算法进行检查：

1）Request4(3,3,0)≤Need4(4,3,1)

2) Request4(3,3,0) ＞ Available(2,3,0),表示资源不够,则让 P4 等待

(4) P0 请求资源 Request0(0,2,0)

P0 发出请求向量 Request0(0,2,0)，系统按银行家算法进行检查：

1）Request0(0,2,0)≤Need0(7,4,3)

2）Request0 (0,2,0) ≤ Available(2,3,0)

3）系统试为 P0 分配资源，并修改相应的向量(见下表[5]所示)

	Max A B C	Allocation A B C	Need A B C	Available A B C
P0	7 5 3	0 1 0 [0 3 0]	7 4 3 [7 2 3]	3 3 2 (2 3 0) [2 1 0]
P1	3 2 2	2 0 0 (3 0 2)	1 2 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	

表 5　P0 请求资源时的资源分配表

4)L 因 Avaliable(2,1,0)

已不能满足任何进程需要，故系统进入不安全状态，此时系统不分配资源。

【实验原理】

1. 死锁与死锁避免

在操作系统中，**死锁**是指多个进程因争夺资源而相互等待，从而无法继续执行的现象。死锁的主要原因是资源分配不合理或资源不足。为避免死锁，操作系统中需要采用策略确保资源分配过程始终处于安全状态。

银行家算法是一种经典的死锁避免算法，通过动态分配资源并检查系统安全性来避免死锁。其核心思想是，只有在分配资源后系统仍然处于安全状态时，才允许分配。

2. 银行家算法

银行家算法的核心目标是确保系统在资源分配后仍然处于**安全状态**。它通过以下两部分算法实现：

安全性检查算法：用于判断系统当前的资源分配是否安全，即是否存在一个可以完成所有进程的安全序列。

资源请求处理：当某个进程提出资源请求时，判断该请求是否可以满足，并尝试分配资源，调用安全性检查算法验证分配后的状态。

定义关键变量:

- Available：当前可用资源向量，表示系统中每种资源的可用数量。
- Max：最大需求矩阵，表示每个进程可能需要的每种资源的最大数量。

- Allocation: 已分配矩阵，表示系统已分配给每个进程的资源数量。
- Need: 需求矩阵，表示每个进程还需要的资源数量，用公式计算:

$$Need\ [i][j] = Max\ [i][j] - Allocation\ [i][j]$$

其中:i 表示进程编号，j 表示资源种类编号

3.安全性检查算法:

目标：检查当前资源分配是否安全，寻找一个安全序列。

步骤:

1)初始化工作向量 Work，设为 Available。

2)初始化完成标记向量 Finish，全部设为 false。

3)重复以下步骤，直到找到一个安全序列或无法分配:

- 寻找一个尚未完成 (Finish[i] == false) 的进程 P[i]，使得: $Need\ [i][j] \leq Work\ [j], \forall j$
- 如果找到，假定分配资源并更新:

$$Work\ [j]= Work\ [j]+ Allocation\ [i][j], \forall j$$

标记 Finish[i] = true。

- 如果未找到，系统状态不安全。

4)如果所有进程都完成 (Finish[i] == true)，则系统安全。

4.资源请求算法:

目标：判断是否满足一个进程的资源请求，并检查系统安全性。

步骤：

1)检查请求是否合法：

Request [i][j] ≤ Need [i][j] Request [i][j] ≤ Available [j]

2)假定分配资源，更新变量：

$$\text{Available}[j] = \text{Available}[j] - \text{Request}[i][j], \forall j$$
$$\text{Allocation}[i][j] = \text{Allocation}[i][j] + \text{Request}[i][j], \forall j$$
$$\text{Need}[i][j] = \text{Need}[i][j] - \text{Request}[i][j], \forall j$$

3) 调用安全性检查算法:如果安全，分配成功。如果不安全，回滚分配，拒绝请求。

5.核心公式

1.需求矩阵计算公式:

$$\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$$

2. 请求合法性判断公式:

Request [i][j] ≤ Need [i][j] Request [i][j] ≤ Available [j]

3. 试探性分配公式:

更新 Available:

$$\text{Available}[j] = \text{Available}[j] - \text{Request}[i][j]$$

更新 Allocation:

$$Allocation\ [i][j] = Allocation\ [i][j] + Request\ [i][j]$$

更新 Need:

$$Need\ [i][j] = Need\ [i][j] - Request\ [i][j]$$

4.安全性检查条件:

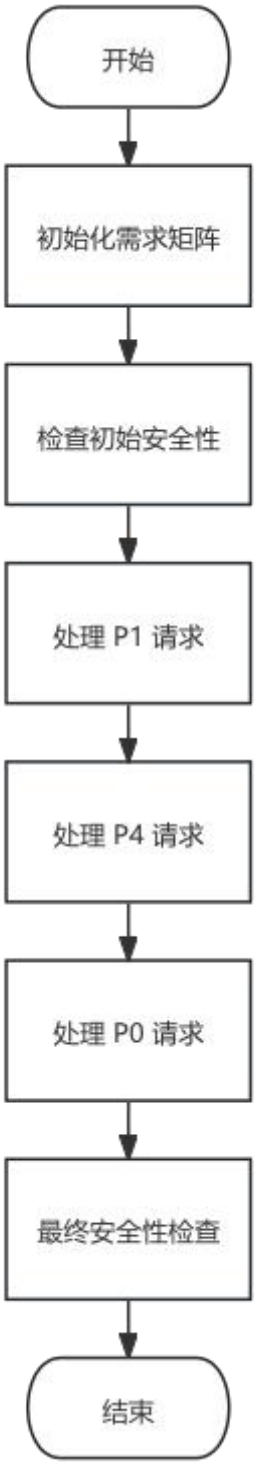
$$Need\ [i][j] \leq Work\ [j], \forall j$$

银行家算法是一种经典的死锁避免算法，其核心思想是通过动态地监控和管理系统资源分配，确保在任何时刻资源分配后系统都能找到一个安全的执行序列，从而避免死锁的发生。该算法通过对每个进程的最大需求、已分配资源和剩余需求进行精确计算，并结合系统的当前可用资源状态，判断资源请求是否合理。如果请求合法且分配后系统仍然安全，则允许资源分配；否则，拒绝请求或者让进程等待。银行家算法的应用不仅保证了资源分配的安全性，还提高了系统资源的利用率，其设计灵感来源于银行贷款管理的方式，在计算机操作系统中广泛用于多任务和多进程资源调度的场景，是操作系统中死锁避免的重要技术之一。

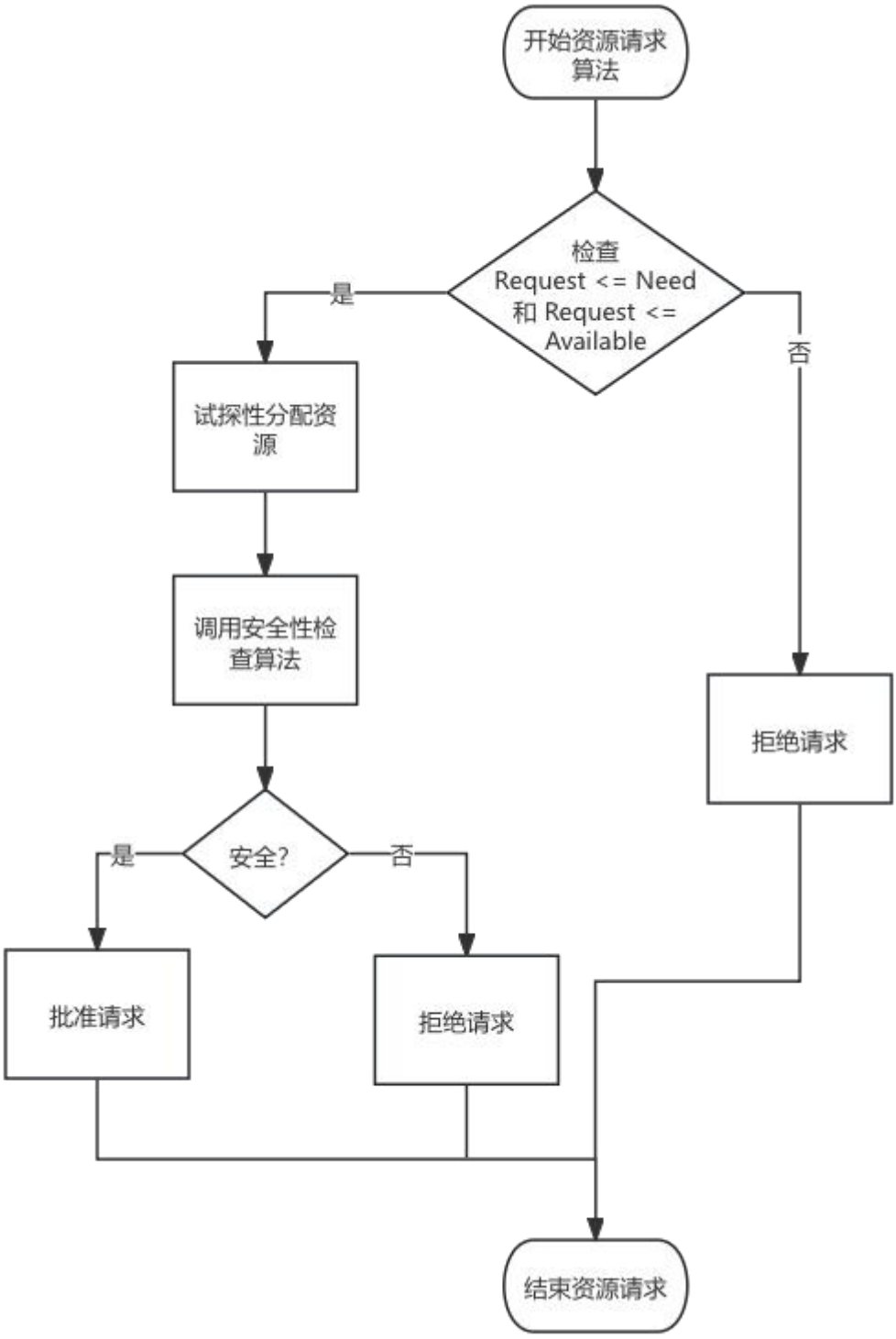
【实验步骤】

程序流程图：

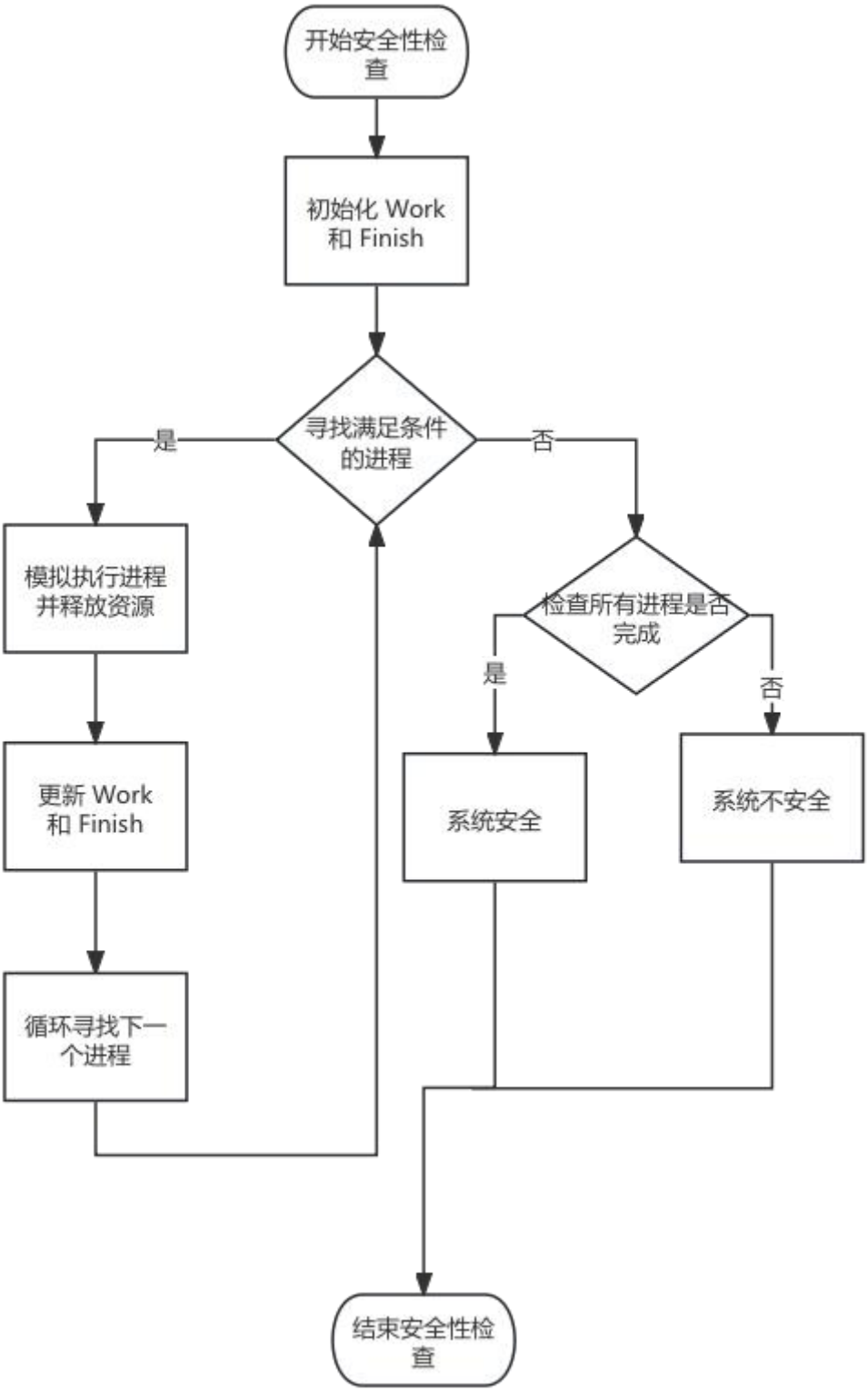
主程序：



资源请求算法：


开始资源请求算法
检查 Request <= Need 和 Request <= Available
是
否
试探性分配资源
调用安全性检查算法
拒绝请求
安全?
是
否
批准请求
拒绝请求
结束资源请求

安全性检查算法：


开始安全性检查
初始化 Work 和 Finish
寻找满足条件的进程
是
否
模拟执行进程并释放资源
更新 Work 和 Finish
循环寻找下一个进程
检查所有进程是否完成
是
否
系统安全
系统不安全
结束安全性检查

核心代码：

```
#include <stdio.h>
#include <stdbool.h>

#define P 5 // 进程数量
#define R 3 // 资源种类数量

int Allocation[P][R] = {
{0, 1, 0},
{2, 0, 0},
{3, 0, 2},
{2, 1, 1},
{0, 0, 2}}; // 已分配矩阵

int Max[P][R] = {
{7, 5, 3},
{3, 2, 2},
{9, 0, 2},
{2, 2, 2},
{4, 3, 3}}; // 最大需求矩阵

int Need[P][R]; // 需求矩阵
int Available[R] = {3, 3, 2}; // 可用资源向量

// 函数声明
void calculateNeed();
bool isSafeState(const char* phase); // 安全性检查
bool requestResources(int processID, int request[], const
char* phase); // 资源请求
void printSystemState(const char* message); // 输出当前系统
状态

int main() {
// 初始化需求矩阵
calculateNeed();

// (1) 检查系统初始安全性
printf("步骤 (1): 检查系统初始安全性\n");
```

```
isSafeState("(1) 初始安全性检查");

// (2) P1 请求资源
printf("\n步骤 (2): P1 请求资源 Request1(1, 0, 2)\n");
int request1[R] = {1, 0, 2};
requestResources(1, request1, "(2) P1 请求资源");

// (3) P4 请求资源
printf("\n步骤 (3): P4 请求资源 Request4(3, 3, 0)\n");
int request4[R] = {3, 3, 0};
requestResources(4, request4, "(3) P4 请求资源");

// (4) P0 请求资源
printf("\n步骤 (4): P0 请求资源 Request0(0, 2, 0)\n");
int request0[R] = {0, 2, 0};
requestResources(0, request0, "(4) P0 请求资源");

// (5) 最终安全性检查
printf("\n步骤 (5): 进行最终安全性检查\n");
isSafeState("(5) 最终安全性检查");

return 0;
}

// 计算需求矩阵 Need = Max - Allocation
void calculateNeed() {
for (int i = 0; i < P; i++) {
for (int j = 0; j < R; j++) {
Need[i][j] = Max[i][j] - Allocation[i][j];
}
}
}

// 输出当前系统状态
void printSystemState(const char* message) {
printf("[%s] 系统状态:\n", message);
printf("Available: ");
for (int i = 0; i < R; i++) {
printf("%d ", Available[i]);
```

```
}
printf("\n");

printf("Allocation:\n");
for (int i = 0; i < P; i++) {
printf("P%d: ", i);
for (int j = 0; j < R; j++) {
printf("%d ", Allocation[i][j]);
}
printf("\n");
}

printf("Need:\n");
for (int i = 0; i < P; i++) {
printf("P%d: ", i);
for (int j = 0; j < R; j++) {
printf("%d ", Need[i][j]);
}
printf("\n");
}
printf("\n");
}

// 安全性检查算法
bool isSafeState(const char* phase) {
int Work[R]; // 工作向量
bool Finish[P] = {false}; // 是否完成
int safeSequence[P]; // 保存安全序列

// 初始化 Work = Available
for (int i = 0; i < R; i++) {
Work[i] = Available[i];
}

int count = 0;

printf("[%s] 开始安全性检查...\n", phase);
while (count < P) {
bool allocated = false;
```

```
for (int i = 0; i < P; i++) {
if (!Finish[i]) {
bool canAllocate = true;

for (int j = 0; j < R; j++) {
if (Need[i][j] > Work[j]) {
canAllocate = false;
break;
}
}

if (canAllocate) {
printf(" P%d 可以安全执行。释放资源后:\n", i);
for (int j = 0; j < R; j++) {
Work[j] += Allocation[i][j];
}
Finish[i] = true;
safeSequence[count++] = i;

printf(" Work: ");
for (int j = 0; j < R; j++) {
printf("%d ", Work[j]);
}
printf("\n");

allocated = true;
}
}
}

if (!allocated) {
printf("[%s] 不安全状态，无法找到完整的安全序列。\n", phase);
return false;
}
}

// 打印安全序列
printf("[%s] 系统安全。安全序列: ", phase);
for (int i = 0; i < P; i++) {
```

```
printf("P%d ", safeSequence[i]);
}
printf("\n");
return true;
}

// 资源请求算法
bool requestResources(int processID, int request[], const
char* phase) {
printf("[%s] 进程 P%d 请求资源: ", phase, processID);
for (int i = 0; i < R; i++) {
printf("%d ", request[i]);
}
printf("\n");

// 检查 Request <= Need 和 Request <= Available
for (int i = 0; i < R; i++) {
if (request[i] > Need[processID][i]) {
printf("[%s] 请求超过需求，拒绝请求。\n", phase);
return false;
}
if (request[i] > Available[i]) {
printf("[%s] 请求超过可用资源，拒绝请求。\n", phase);
return false;
}
}

// 试探性分配资源
printf("[%s] 试探性分配资源...\n", phase);
for (int i = 0; i < R; i++) {
Available[i] -= request[i];
Allocation[processID][i] += request[i];
Need[processID][i] -= request[i];
}

printSystemState(phase);

// 检查分配后是否安全
if (isSafeState(phase)) {
```

```
printf("[%s] 请求被批准。\n", phase);
return true;
} else {
// 回滚
printf("[%s] 请求导致不安全状态，回滚资源分配。\n", phase);
for (int i = 0; i < R; i++) {
Available[i] += request[i];
Allocation[processID][i] -= request[i];
Need[processID][i] += request[i];
}
printSystemState("(回滚后)");
return false;
}
}
```

运行结果：

```
youngbean@YoungBeans ~ % /Users/youngbean/Documents/Github/Misc-P
步骤 (1): 检查系统初始安全性
[(1) 初始安全性检查] 开始安全性检查...
  P1 可以安全执行。释放资源后:
  Work: 5 3 2
  P3 可以安全执行。释放资源后:
  Work: 7 4 3
  P4 可以安全执行。释放资源后:
  Work: 7 4 5
  P0 可以安全执行。释放资源后:
  Work: 7 5 5
  P2 可以安全执行。释放资源后:
  Work: 10 5 7
[(1) 初始安全性检查] 系统安全。安全序列: P1 P3 P4 P0 P2

步骤 (2): P1 请求资源 Request1(1, 0, 2)
[(2) P1 请求资源] 进程 P1 请求资源: 1 0 2
[(2) P1 请求资源] 试探性分配资源...
[(2) P1 请求资源] 系统状态:
Available: 2 3 0
Allocation:
P0: 0 1 0
P1: 3 0 2
P2: 3 0 2
P3: 2 1 1
P4: 0 0 2
Need:
P0: 7 4 3
P1: 0 2 0
P2: 6 0 0
P3: 0 1 1
P4: 4 3 1

[(2) P1 请求资源] 开始安全性检查...
  P1 可以安全执行。释放资源后:
  Work: 5 3 2
  P3 可以安全执行。释放资源后:
  Work: 7 4 3
  P4 可以安全执行。释放资源后:
  Work: 7 4 5
  P0 可以安全执行。释放资源后:
  Work: 7 5 5
  P2 可以安全执行。释放资源后:
  Work: 10 5 7
[(2) P1 请求资源] 系统安全。安全序列: P1 P3 P4 P0 P2
[(2) P1 请求资源] 请求被批准。

步骤 (3): P4 请求资源 Request4(3, 3, 0)
[(3) P4 请求资源] 进程 P4 请求资源: 3 3 0
[(3) P4 请求资源] 请求超过可用资源, 拒绝请求。

步骤 (4): P0 请求资源 Request0(0, 2, 0)
[(4) P0 请求资源] 进程 P0 请求资源: 0 2 0
[(4) P0 请求资源] 试探性分配资源...
[(4) P0 请求资源] 系统状态:
Available: 2 1 0
Allocation:
P0: 0 3 0
P1: 3 0 2
P2: 3 0 2
P3: 2 1 1
P4: 0 0 2
Need:
P0: 7 2 3
P1: 0 2 0
P2: 6 0 0
P3: 0 1 1
P4: 4 3 1

[(4) P0 请求资源] 开始安全性检查...
[(4) P0 请求资源] 不安全状态, 无法找到完整的安全序列。
[(4) P0 请求资源] 请求导致不安全状态, 回滚资源分配。
[(回滚后)] 系统状态:
Available: 2 3 0
Allocation:
P0: 0 1 0
P1: 3 0 2
P2: 3 0 2
P3: 2 1 1
P4: 0 0 2
Need:
P0: 7 4 3
P1: 0 2 0
P2: 6 0 0
P3: 0 1 1
P4: 4 3 1


步骤 (5): 进行最终安全性检查
[(5) 最终安全性检查] 开始安全性检查...
  P1 可以安全执行。释放资源后:
  Work: 5 3 2
  P3 可以安全执行。释放资源后:
  Work: 7 4 3
  P4 可以安全执行。释放资源后:
  Work: 7 4 5
  P0 可以安全执行。释放资源后:
  Work: 7 5 5
  P2 可以安全执行。释放资源后:
  Work: 10 5 7
[(5) 最终安全性检查] 系统安全。安全序列: P1 P3 P4 P0 P2

Saving session...
...copying shared history...
```

步骤 (1): 检查系统初始安全性
[(1) 初始安全性检查] 开始安全性检查...
  P1 可以安全执行。释放资源后:
  Work: 5 3 2
  P3 可以安全执行。释放资源后:
  Work: 7 4 3
  P4 可以安全执行。释放资源后:
  Work: 7 4 5
  P0 可以安全执行。释放资源后:
  Work: 7 5 5
  P2 可以安全执行。释放资源后:
  Work: 10 5 7
[(1) 初始安全性检查] 系统安全。安全序列: P1 P3 P4 P0 P2

步骤 (2): P1 请求资源 Request1(1, 0, 2)
[(2) P1 请求资源] 进程 P1 请求资源: 1 0 2
[(2) P1 请求资源] 试探性分配资源...
[(2) P1 请求资源] 系统状态:
Available: 2 3 0
Allocation:
P0: 0 1 0
P1: 3 0 2
P2: 3 0 2
P3: 2 1 1
P4: 0 0 2
Need:
P0: 7 4 3
P1: 0 2 0
P2: 6 0 0
P3: 0 1 1
P4: 4 3 1

[(2) P1 请求资源] 开始安全性检查...
  P1 可以安全执行。释放资源后:
  Work: 5 3 2
  P3 可以安全执行。释放资源后:
  Work: 7 4 3
  P4 可以安全执行。释放资源后:
  Work: 7 4 5
  P0 可以安全执行。释放资源后:
  Work: 7 5 5
  P2 可以安全执行。释放资源后:

```
  Work: 10 5 7
[(2) P1 请求资源] 系统安全。安全序列: P1 P3 P4 P0 P2
[(2) P1 请求资源] 请求被批准。


步骤 (3): P4 请求资源 Request4(3, 3, 0)
[(3) P4 请求资源] 进程 P4 请求资源: 3 3 0
[(3) P4 请求资源] 请求超过可用资源，拒绝请求。


步骤 (4): P0 请求资源 Request0(0, 2, 0)
[(4) P0 请求资源] 进程 P0 请求资源: 0 2 0
[(4) P0 请求资源] 试探性分配资源...
[(4) P0 请求资源] 系统状态:
Available: 2 1 0
Allocation:
P0: 0 3 0
P1: 3 0 2
P2: 3 0 2
P3: 2 1 1
P4: 0 0 2
Need:
P0: 7 2 3
P1: 0 2 0
P2: 6 0 0
P3: 0 1 1
P4: 4 3 1


[(4) P0 请求资源] 开始安全性检查...
[(4) P0 请求资源] 不安全状态，无法找到完整的安全序列。
[(4) P0 请求资源] 请求导致不安全状态，回滚资源分配。
[(回滚后)] 系统状态:
Available: 2 3 0
Allocation:
P0: 0 1 0
P1: 3 0 2
P2: 3 0 2
P3: 2 1 1
P4: 0 0 2
Need:
P0: 7 4 3
P1: 0 2 0
P2: 6 0 0
```

```
P3: 0 1 1
P4: 4 3 1
```

```
步骤（5）：进行最终安全性检查
[（5）最终安全性检查] 开始安全性检查...
  P1 可以安全执行。释放资源后：
  Work: 5 3 2
  P3 可以安全执行。释放资源后：
  Work: 7 4 3
  P4 可以安全执行。释放资源后：
  Work: 7 4 5
  P0 可以安全执行。释放资源后：
  Work: 7 5 5
  P2 可以安全执行。释放资源后：
  Work: 10 5 7
[（5）最终安全性检查] 系统安全。安全序列：P1 P3 P4 P0 P2
```

结果分析：

初始资源分配状态（T0 时刻）

- **系统状态：**

- Available = (3, 3, 2) （初始可用资源向量）

- Allocation：

P0：(0，1，0)

P1：(2，0，0)

P2：(3，0，2)

P3：(2，1，1)

P4：(0，0，2)

• Max：

P0：(7，5，3)

P1：(3，2，2)

P2：(9，0，2)

P3：(2，2，2)

P4：(4，3，3)

• Need = Max - Allocation：

P0：(7，4，3)

P1：(1，2，2)

P2：(6，0，0)

P3：(0，1，1)

P4：(4，3，1)

安全性检查（T0 时刻）

1. Available = (3, 3, 2)：

• 检查进程 P1：Need1 = (1, 2, 2) <= Available。

• 分配成功，Available = Available + Allocation1 = (5, 3, 2)。

• 检查进程 P3：Need3 = (0, 1, 1) <= Available。

• 分配成功，Available = Available + Allocation3 = (7, 4, 3)。

• 检查进程 P4：Need4 = (4, 3, 1) <= Available。

• 分配成功，Available = Available + Allocation4 = (7, 4, 5)。

• 检查进程 P2：Need2 = (6, 0, 0) <= Available。

• 分配成功，Available = Available + Allocation2 = (10, 4, 7)。

• 检查进程 P0：Need0 = (7, 4, 3) <= Available。

• 分配成功，Available = Available + Allocation0 = (10, 5, 7)。

2. 安全序列为：{P1, P3, P4, P2, P0}。

• 系统安全。

步骤 (2): P1 请求资源 Request1(1, 0, 2)

1. **检查请求合法性：**

• Request1(1, 0, 2) <= Need1(1, 2, 2) → 合法。

• Request1(1, 0, 2) <= Available(3, 3, 2) → 合法。

2. **试探性分配：**

• 更新 Available = (3, 3, 2) - (1, 0, 2) = (2, 3, 0)。

- 更新 Allocation1 = (2, 0, 0) + (1, 0, 2) = (3, 0, 2)。
- 更新 Need1 = (1, 2, 2) - (1, 0, 2) = (0, 2, 0)。

3. **安全性检查：**

- 初始 Available = (2, 3, 0)：
- 检查 P1: Need1 = (0, 2, 0) <= Available。
- 分配成功，Available = (2, 3, 0) + Allocation1 = (5, 3, 2)。
- 检查 P3: Need3 = (0, 1, 1) <= Available。
- 分配成功，Available = (5, 3, 2) + Allocation3 = (7, 4, 3)。
- 检查 P4: Need4 = (4, 3, 1) <= Available。
- 分配成功，Available = (7, 4, 3) + Allocation4 = (7, 4, 5)。
- 检查 P2: Need2 = (6, 0, 0) <= Available。
- 分配成功，Available = (7, 4, 5) + Allocation2 = (10, 4, 7)。
- 检查 P0: Need0 = (7, 4, 3) <= Available。
- 分配成功，Available = (10, 5, 7)。

4. 安全序列为：{P1, P3, P4, P2, P0}。

- 系统安全。

• P1 的请求被批准。

步骤 (3): P4 请求资源 Request4(3, 3, 0)

1. **检查请求合法性：**

• Request4(3, 3, 0) <= Need4(4, 3, 1) → 合法。

• Request4(3, 3, 0) > Available(2, 3, 0) → 不合法（资源不足）。

2. **结果：**

• 由于当前 Available 无法满足 P4 的请求，系统将 **拒绝 P4 的请求**，让 P4 进入等待状态。

步骤 (4): P0 请求资源 Request0(0, 2, 0)

1. **检查请求合法性：**

• Request0(0, 2, 0) <= Need0(7, 4, 3) → 合法。

• Request0(0, 2, 0) <= Available(2, 3, 0) → 合法。

2. **试探性分配：**

• 更新 Available = (2, 3, 0) - (0, 2, 0) = (2, 1, 0)。

• 更新 Allocation0 = (0, 1, 0) + (0, 2, 0) = (0, 3, 0)。

• 更新 Need0 = (7, 4, 3) - (0, 2, 0) = (7, 2, 3)。

3. **安全性检查：**

- 初始 Available = (2, 1, 0)：
- 检查 P1: Need1 = (0, 2, 0)，但 Need1[1] > Available[1] → 不满足。
- 检查 P3: Need3 = (0, 1, 1)，但 Need3[2] > Available[2] → 不满足。
- 检查 P4: Need4 = (4, 3, 1)，但 Need4[0] > Available[0] → 不满足。
- 检查 P2: Need2 = (6, 0, 0)，但 Need2[0] > Available[0] → 不满足。
- 检查 P0: Need0 = (7, 2, 3)，但 Need0[2] > Available[2] → 不满足。
- 没有进程可以安全执行，系统状态不安全。

4. **结果：**

- **回滚分配**：
- 恢复 Available = (2, 1, 0) + (0, 2, 0) = (2, 3, 0)。
- 恢复 Allocation0 = (0, 3, 0) - (0, 2, 0) = (0, 1, 0)。
- 恢复 Need0 = (7, 2, 3) + (0, 2, 0) = (7, 4, 3)。
- 系统拒绝 P0 的请求。

步骤 (5)：最终安全性检查

1. **当前系统状态：**

• Available = (2, 3, 0)。

• Allocation：

P0：(0，1，0)

P1：(3，0，2)

P2：(3，0，2)

P3：(2，1，1)

P4：(0，0，2)

• Need：

P0：(7，4，3)

P1：(0，2，0)

P2：(6，0，0)

P3：(0，1，1)

P4：(4，3，1)

2. **安全性检查：**

• 初始 Available = (2, 3, 0)：

• 检查 P1: Need1 = (0, 2, 0) <= Available。

• 分配成功，Available = (2, 3, 0) + Allocation1 = (5, 3, 2)。

• 检查 P3: Need3 = (0, 1, 1) <= Available。

- 分配成功，Available = (5, 3, 2) + Allocation3 = (7, 4, 3)。
- 检查 P4: Need4 = (4, 3, 1) <= Available。
- 分配成功，Available = (7, 4, 3) + Allocation4 = (7, 4, 5)。
- 检查 P2: Need2 = (6, 0, 0) <= Available。
- 分配成功，Available = (7, 4, 5) + Allocation2 = (10, 4, 7)。
- 检查 P0: Need0 = (7, 4, 3) <= Available。
- 分配成功，Available = (10, 5, 7)。

3. **安全序列：**

- 安全序列为：{P1, P3, P4, P2, P0}。

4. **结果：**

- 系统安全。

在银行家算法中，**安全序列不唯一**，只要符合安全性检查的条件，所有的安全序列都是正确的。因此，**程序输出的安全序列** {P1, P3, P4, P0, P2} 和手算得到的安全序列 {P1, P3, P4, P2, P0} 都是正确的。

所以，程序运行结果正确。

【实验总结】

通过本次实验，我深入学习并实现了银行家算法及其核心部分：安全性检查算法和资源请求算法。我了解到，银行家算法是一种通过动态

监控资源分配来避免死锁的经典算法，其核心在于确保系统在任何时刻的资源分配都能找到一个安全的执行序列，从而保证系统的安全性。在实验中，我通过安全性检查算法判断系统当前状态是否安全，并通过资源请求算法验证进程资源请求是否合法，进而动态调整系统资源的分配情况。

实验让我明白了银行家算法的逻辑性和实用性。通过实时更新系统的可用资源（Available）、已分配资源（Allocation）和剩余需求（Need），算法能够动态判断资源请求是否满足条件，并试探性地分配资源以验证分配后的安全性。银行家算法的设计思想不仅保障了系统的安全性，还有效提高了资源利用率，让我更加理解操作系统中资源管理的复杂性与精确性。

在实验的实现和调试过程中，我遇到了以下问题，并逐一解决：

1. **需求矩阵计算错误**

- **问题描述**：在初始化需求矩阵（Need）时，公式 Need[i][j] = Max[i][j] - Allocation[i][j] 中，部分资源种类的计算结果错误。
- **修改过程**：经过检查，我发现问题出在矩阵的初始化和数组索引的范围控制上。为解决这一问题，我重新设计了矩阵初始化的逻辑，并引入调试输出，逐步验证每个 Need 元素的计算结果。

2. **安全性检查逻辑遗漏**

- **问题描述**：在安全性检查算法中，条件判断 Need[i][j] <= Work[j] 有时会漏掉部分资源种类，导致算法错误地判定某些进程无法执行。

- **修改过程**：我仔细检查代码后，发现是循环终止条件设置错误，导致部分资源种类未被正确判断。我修改了循环范围，确保遍历所有资源种类，同时加入调试打印来验证每一步的条件判断。

3. **资源请求算法回滚问题**

- **问题描述**：在资源请求算法中，当分配后系统状态不安全时，回滚逻辑未正确恢复初始状态，导致后续资源状态紊乱。

- **修改过程**：我在分配失败的分支中补充了回滚逻辑，确保 Available、Allocation 和 Need 恢复到分配前的状态，并通过逐步调试验证回滚的正确性。

4. **安全序列输出问题**

- **问题描述**：安全性检查中输出的安全序列偶尔会出现顺序混乱。

- **修改过程**：我发现是因为安全序列数组的赋值和完成标记的更新顺序不一致导致的。我重新调整了标记完成和记录序列的步骤，确保每次安全序列的输出顺序正确。

在实验过程中，我学会了如何通过银行家算法动态管理系统资源，包括初始化资源状态、计算需求矩阵以及判断资源请求的合法性和安全

性。同时，我深入理解了安全性检查算法的具体实现，通过工作向量和完成标记判断进程是否能够顺利执行，并正确输出安全序列。在实现资源请求算法的过程中，我掌握了试探性分配和回滚机制的运用，明确了如何在资源分配后验证系统的安全性，并在状态不安全时及时撤销不合理的分配操作。这次实验让我不仅熟悉了银行家算法的核心原理，还提升了对算法实现细节的把控能力，以及分析问题和优化程序逻辑的综合能力。