華中農業大學
HUAZHONG AGRICULTURAL UNIVERSITY

# 实 验 报 告

# EXPERIMENT REPORT

姓名 高星杰

学号 2021307220712

专业 计科 2102

教师 任继平

科目 操作系统

信 息 学 院

COLLEGE OF INFORMATIC

2024 年 1 月

# 实验 3　死锁

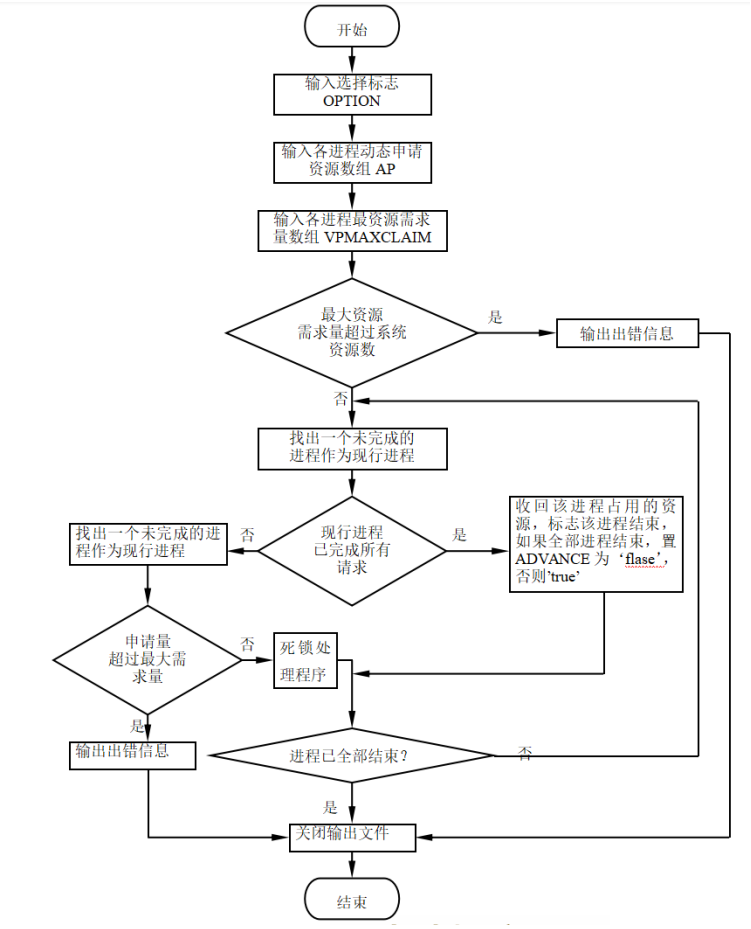
## 1．实验目的

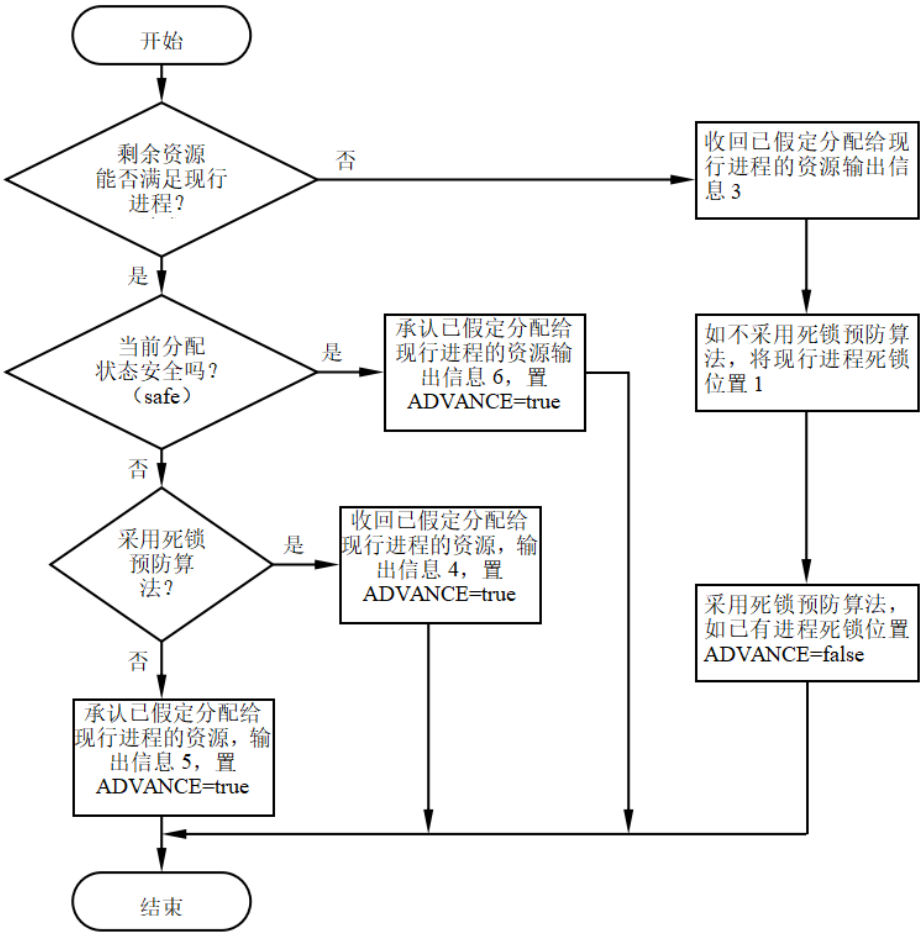
学生应独立的采用高级语言编写一个动态分配系统资源的程序，模拟死锁现象，观察死锁发生的条件，并采用适当的算法，有效地防止死锁的发生。学生应通过本次实验，更直观的了解死锁发生的原因，初步掌握防止死锁、解除死锁的简单方法，加深理解教材中有关死锁的内容。

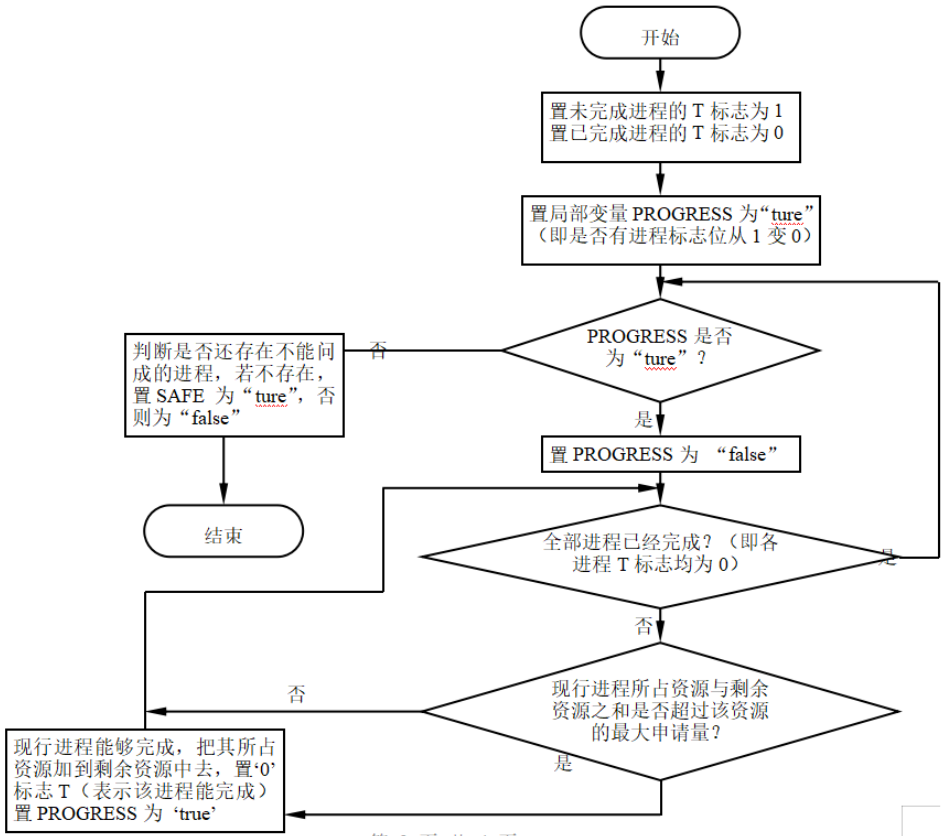
## 2．实验内容

本次实验采用银行算法防止死锁的发生。设有 3 个并发进程共享 10 个系统资源。在 3 个进程申请系统资源之和不超过 10 时，当然不可能发生死锁，因为各个进程资源都能满足。在有一个进程申请的系统资源数超过 10 时，必然会发生死锁。应该排队这两种情况。程序采用人工输入各进程的申请资源序列。如果随机给各进程分配资源，就可能发生死锁，这也就是不采用防止死锁算法的情况。假如，按照一定的规则，为各进程分配资源，就可以防止死锁的发生。示例采用银行算法。这是一种犹如“瞎子爬山”的方法，即探索一步，前进一步，行不通，再往其它方向试探，直至爬上山顶。这种方法是比较保守的，所花的代价也不小。

算法与框图




开始
剩余资源能否满足现行进程?
否
收回已假定分配给现行进程的资源输出信息 3
是
当前分配状态安全吗?（safe）
是
承认已假定分配给现行进程的资源输出信息 6，置 ADVANCE=true
如不采用死锁预防算法，将现行进程死锁位置 1
否
采用死锁预防算法?
是
收回已假定分配给现行进程的资源，输出信息 4，置 ADVANCE=true
采用死锁预防算法，如已有进程死锁位置 ADVANCE=false
否
承认已假定分配给现行进程的资源，输出信息 5，置 ADVANCE=true
结束


开始
置未完成进程的 T 标志为 1
置已完成进程的 T 标志为 0
置局部变量 PROGRESS 为“ture”（即是否有进程标志位从 1 变 0）
PROGRESS 是否为“ture”？
否
判断是否还存在不能问成的进程，若不存在，置 SAFE 为“ture”，否则为“false”
是
置 PROGRESS 为 “false”
结束
全部进程已经完成？（即各进程 T 标志均为 0）
是
否
现行进程所占资源与剩余资源之和是否超过该资源的最大申请量？
否
是
现行进程能够完成，把其所占资源加到剩余资源中去，置‘0’标志 T（表示该进程能完成）置 PROGRESS 为 ‘true’

## 3 实验步骤

### （1） 任务分析：

本实验采用银行算法来防止死锁的发生。假设有 3 个并发进程需要共享 10 个系统资源。当 3 个进程申请的系统资源总和不超过 10 时，不会发生死锁，因为每个进程的资源需求都可以得到满足。然而，当至少一个进程申请的系统资源数量超过 10 时，必然会发生死锁。在这种情况下，需要排队等待资源分配。

为了防止死锁的发生，采用了银行算法。这种算法的思想来源于银行的借贷业务。银行需要确保一定数量的资金可以满足各种客户的借贷需求，以防止因资金周转不灵而导致银行倒闭。在银行算法中，对于每一笔贷款申请，必须先考虑最终是否可以偿还。在研究死锁现象时，我们面临类似的问题，即有限的资源需要被多个进程共享，如果分配不当，就可能导致进程陷入无法继续执行的死锁状态。

银行算法的原理是首先假设一次资源分配是有效的，然后检查这次分配是否会导致死锁，即剩余的资源是否能满足任何一个进程完成其执行所需的资源。如果这次分配是安全的（不会导致死锁），那么就执行这次分配，并继续进行下一次资源分配的试探。这个过程会一直进行下去，直到所有进程的资源请求都得到满足，从而防止死锁的发生。

银行算法是一种比较保守的方法，其代价也相对较高。它需要进行多次试探和资源的重新分配，以确保系统能够避免死锁的发生。

**① 输入的形式和输入值的范围：**

- 用户通过菜单选择选项进行交互。
- 初始化数据要求用户输入以下内容：
  - 资源数量和进程数量（正整数）。
  - 每种资源的总量（非负整数）。
  - 每个进程的最大资源需求（非负整数）。
  - 当前资源分配给每个进程的数量（非负整数）。
  - 请求资源时，用户需要输入进程号和请求的每种资源数量（非负整数）。

**② 输出的形式：**

- 初始化数据后，显示当前的资源分配、最大需求和可用资源。
- 在请求资源时，如果请求可以满足并且系统仍然安全，显示安全的进程序列。
- 在请求资源时，如果请求无法满足或者系统不再安全，显示相应的错误消息。
- 显示菜单选项供用户选择。

**③ 程序所能达到的功能：**

- 初始化资源分配数据。
- 检查当前的资源分配是否安全。

- 模拟进程发出的资源请求，并检查系统是否仍然安全。
- 显示当前的资源分配、最大需求和可用资源。

**④ 测试数据：**

- 正确的输入及其输出结果：
  - 初始化数据后，显示正确的资源分配、最大需求和可用资源。
  - 模拟进程发出的资源请求，并显示正确的安全进程序列。
- 含有错误的输入及其输出结果：
  - 输入无效的资源数量或进程数量，显示错误消息。
  - 输入超出范围的资源数量或进程数量，显示错误消息。
  - 请求超过可用资源的数量，显示错误消息。
  - 请求超过进程的最大资源需求，显示错误消息。
  - 请求使系统不再安全，显示错误消息。

**（2） 概要设计：**

**2.1 数据类型**

数组（基本数据类型）：

- int Resource[MAX]：代表系统中每种资源的总数。
- int Max[MAX][MAX]：最大需求矩阵。定义每个进程对每种资源的最大需求。
- int Allocation[MAX][MAX]：分配矩阵。显示当前分配给每个进程的资源。
- int Need[MAX][MAX]：需求矩阵。计算每个进程剩余的资源需求。
- int Available[MAX]：可用向量。跟踪系统中当前可用的资源。
- int Work[MAX]：工作向量。在安全检查算法中用于跟踪检查过程中的分配。
- bool Finish[MAX]：布尔数组，用来指示进程是否已完成执行。

向量（标准模板库数据类型）：

- vector<int> Safeorder：存储可以执行的进程的安全序列，这些进程的执行不会导致死锁。

**2.2 主程序流程：**

初始化：提示用户输入系统的初始状态，包括资源和进程的数量，以及分配和最大矩阵。

用户交互：通过菜单驱动的界面，用户可以选择初始化数据、请求资源、显示当前状态或退出程序。

资源请求和分配：当发出资源请求时，程序会检查是否可以在不导致死锁的情况下满足请求，使用安全算法进行检查。

显示状态：用户可以随时查看资源分配的当前状态、可用资源和待定需求。

退出：用户可以选择退出程序。

**2.3 层次关系（函数调用）：**

**main 函数：**核心控制函数。根据用户的选择调用其他函数。

- 调用 Menu 显示选项。
- 根据用户输入调用 Init、Order、Display。

**Menu 函数：**向用户显示交互菜单。在每次操作后由 main 重复调用以显示选项。

**Init 函数：**初始化数据结构。当用户选择初始化数据时，从 main 调用。

- 内部调用 checkInit 来验证输入数据。

**Order 函数：**处理资源请求。当用户想要请求资源时，从 main 调用。

- 调用 Safecheck 执行安全算法。
- 可能会更改 Available、Allocation 和 Need 的状态。

**Safecheck 函数：**执行安全算法。在 Order 内部调用，以检查当前状态是否安全。

- 修改 Work 和 Finish 数组并更新 Safeorder。

**Display 函数：**显示系统的当前状态。当用户想要查看当前资源分配时，从 main 调用。

checkInit 函数：验证初始输入数据。在 Init 内部调用，以确保数据的完整性。

（3） 详细设计

**3.1 数据类型实现及操作伪代码**

- int Resource[MAX]: 存储每种资源的总数量。
- int Max[MAX][MAX]: 存储每个进程对每种资源的最大需求。
- int Allocation[MAX][MAX]: 存储每个进程已分配的资源数量。
- int Need[MAX][MAX]: 存储每个进程尚需的资源数量。
- int Available[MAX]: 存储当前可用的资源数量。
- int Work[MAX]: 在安全性检查中使用的临时数组，存储工作时的资源数量。
- bool Finish[MAX]: 标记每个进程是否完成。
- vector<int> Safeorder: 存储可能的安全序列。

```
// 相关数据结构
int n_process;              // 表示进程的个数
int n_resource;             // 表示资源的个数
int Resource[MAX];          // 表示资源的总数
int Max[MAX][MAX];          // 最大需求矩阵
int Allocation[MAX][MAX]; // 分配矩阵
int Need[MAX][MAX];         // 需求矩阵
int Available[MAX];         // 可用资源向量
int Work[MAX];              // 工作向量
bool Finish[MAX];           // 完成标志
vector<int> Safeorder;      // 安全序列记录
```

**操作的伪码算法：**
**初始化（Init）：**
输入：无
输出：初始化系统状态
算法：
1. 输入资源种类数 n_resource 和进程数 n_process。
2. 对于每种资源，输入总量 Resource[i]。
3. 对于每个进程，输入其对每种资源的最大需求 Max[process][resource] 和当前分配 Allocation[process][resource]。
4. 计算每个进程的需求 Need[process][resource] = Max[process][resource] - Allocation[process][resource]。
5. 计算系统的初始可用资源 Available[resource]。
6. 调用 checkInit() 检查输入的合法性。
**安全性检查（Safecheck）：**
输入：无
输出：是否有安全序列
算法：
1. 初始化 Work = Available。
2. 初始化所有进程的 Finish[process] = false。
3. 循环直到所有进程都被标记为完成或无法进行下一步：
   a. 对于每个进程 process：
      i. 如果 Finish[process] = false 且 Need[process] <= Work，则分配资源，更新 Work，标记 Finish[process] = true，并将 process 添加到 Safeorder。
**请求资源（Order）：**
输入：请求资源的进程编号和数量
输出：是否成功分配资源
算法：
1. 输入请求资源的进程编号 process 和请求数量 Request[resource]。
2. 检查 Request 是否小于等于 Need[process] 和 Available。
3. 如果满足条件，尝试分配资源并调用 Safecheck() 检查安全性。
4. 如果安全，则更新 Allocation, Need, Available。否则，拒绝请求并恢复原状态。
**显示当前状态（Display）：**
输入：无
输出：显示当前资源分配状态
算法：
1. 显示每个进程的 Max, Allocation, Need。
2. 显示系统的 Available 资源。

### 3.2 主程序和其他模块伪代码

为了确保清晰和可实现性，我将为主程序及其他模块提供详细的伪码算法。这些伪码将涵盖程序的主要流程和关键功能。

**主程序 (main)**

开始

  初始化 n_resource, n_process, Resource, Max, Allocation, Need, Available

  循环

    调用 Menu() 显示菜单

    输入用户选择 choose

根据 choose 的值执行相应操作：

case 1: 调用 Init()

case 2: 调用 Order()

case 3: 调用 Display()

case 4: 打印退出信息并结束程序

default: 打印错误信息

结束循环

结束

**菜单 (Menu)**

函数 Menu

打印菜单选项

结束函数

**初始化 (Init)**

函数 Init

输入 n_resource, n_process

对于每种资源 i:

输入 Resource[i]

对于每个进程 j:

对于每种资源 k:

输入 Max[j][k]

输入 Allocation[j][k]

计算 Need[j][k] = Max[j][k] - Allocation[j][k]

计算初始的 Available[resource]

调用 checkInit() 检查输入的合法性

结束函数

**安全性检查 (Safecheck)**

函数 Safecheck

初始化 Work = Available

初始化所有 Finish[process] = false

重复

对于每个进程 i:

如果 Finish[i] = false 且 Need[i] <= Work:

更新 Work = Work + Allocation[i]

设置 Finish[i] = true

将 i 添加到 Safeorder

直到所有进程都被标记为完成或没有新的进程可以完成

如果所有进程 Finish[process] = true:

返回 true

否则:

返回 false

结束函数

**请求资源 (Order)**

函数 Order

输入请求资源的进程编号 process 和数量 Request[resource]

如果 Request <= Need[process] 且 Request <= Available:

尝试分配资源：Allocation[process] += Request, Available -= Request, Need[process] -= Request

如果 Safecheck() = true:

打印成功分配信息和安全序列

否则:

撤销分配：Allocation[process] -= Request, Available += Request, Need[process] += Request

打印分配失败信息

否则:

打印请求无效信息

结束函数

**显示当前状态 (Display)**

函数 Display

打印每个进程的 Max, Allocation, Need, 和 Available

结束函数

输入检查 (checkInit)

less

Copy code

函数 checkInit

对于每个进程 i 和每种资源 j:

检查 Max[i][j] 和 Allocation[i][j] 的有效性

对于每种资源 i:

检查 Available[i] 的有效性

结束函数

### 3.3 函数和过程调用关系图

创建一个函数和过程调用关系图可以帮助清晰地展示程序中不同模块之间的相互调用关系。在您提供的程序中，各函数的调用关系如下：

main

调用 Menu

根据用户选择调用 Init, Order, Display

Init

调用 checkInit

Order

调用 Safecheck

Safecheck

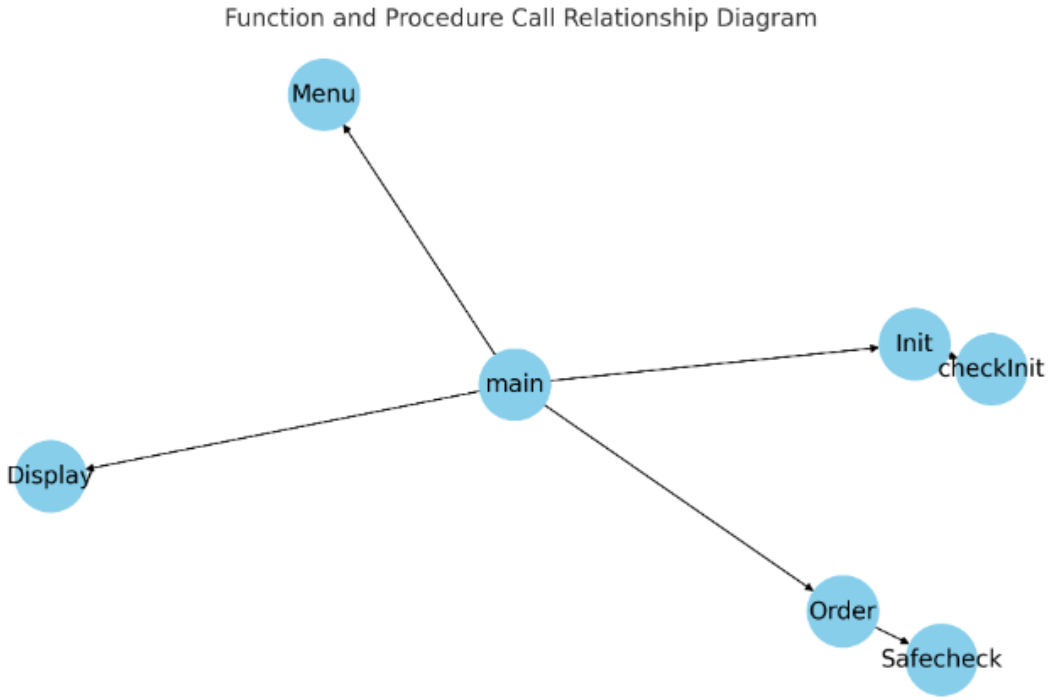
(不调用其他函数)

Display

(不调用其他函数)

checkInit

(不调用其他函数)




这个图形展示了各个函数和过程之间的调用关系。每个节点代表一个函数或过程，箭头指向表示一个函数调用另一个函数。例如，从 `main` 出发的箭头指向 `Menu`, `Init`, `Order`, 和

`Display`，表示 `main` 函数根据用户输入调用这些函数。同样，`Init` 调用 `checkInit`，而 `Order` 调用 `Safecheck`。这种可视化有助于理解程序的结构和流程。

（4） 调试分析：

a. **遇到的问题及解决办法**

**输入验证问题：**

问题描述：用户输入可能不符合预期，如负数、非数字字符或超出预定范围的值。

解决方法：在接收输入之后进行严格的验证。确保所有输入的数据符合预期范围和类型。如果输入无效，提供错误信息并要求用户重新输入。

**数组越界问题：**

问题描述：在处理数组时，可能会访问数组界限之外的内存，导致程序崩溃或不可预测的行为。

解决方法：在访问数组前，始终检查索引是否在数组的有效范围内。可以使用断言或条件语句来确保索引的有效性。

**死锁检测逻辑错误：**

问题描述：实现安全性算法时的逻辑错误可能导致错误地判断系统是否安全，进而错误地允许或拒绝资源请求。

解决方法：仔细检查并测试安全性算法的实现。确保逻辑正确处理了所有可能的情况。可以通过编写单元测试来验证算法的正确性。

**资源分配和回收的一致性问题：**

问题描述：在资源请求和释放过程中，可能会不一致地更新资源状态，导致资源计数错误。

解决方法：确保在资源分配和回收时，所有相关的数据结构（如 Available, Allocation, Need）都同步更新。考虑使用事务式的更新方法，即只有当所有更新都可以完成时才进行实际更新。

**性能问题：**

问题描述：对于大量的进程和资源，算法可能会变得低效，尤其是在频繁的安全性检查时。

解决方法：优化安全性算法的实现。例如，使用更高效的数据结构或减少不必要的迭代。

**用户界面和体验问题：**

问题描述：控制台程序可能不够直观或用户友好。

解决方法：改进用户界面，例如提供清晰的指示和错误消息，确保易于使用。考虑实现图形用户界面以提高可用性。

b. **算法的时空分析和改进设想：**

**时间复杂度分析：**

初始化（Init）：

时间复杂度：O(n*m)，其中 n 是进程数，m 是资源种类数。这是因为需要为每个进程的每种资源输入最大需求和分配。

安全性检查（Safecheck）：

时间复杂度：O(n^2*m)，其中 n 是进程数，m 是资源种类数。在最坏情况下，对于每个进程都需要检查每种资源，且可能需要多次迭代直到找到安全序列。

资源请求（Order）：

时间复杂度：O(n*m)，这包括了对请求的有效性的检查，以及可能的一次安全性检查。

显示状态（Display）：

时间复杂度：O(n*m)，需要遍历每个进程和每种资源来显示当前状态。

**空间复杂度分析：**

所有矩阵和向量:

空间复杂度：O(n*m)，其中 n 是进程数，m 是资源种类数。这是由于 Max, Allocation, Need, Resource, Available 等数组的存储需求。

安全性检查中的额外空间:

需要 O(n) 的额外空间来存储 Work 和 Finish 数组以及 Safeorder 向量。

**改进设想：**

**优化安全性检查:**

可以通过优化数据结构（如使用位图）或减少不必要的迭代来提高安全性检查的效率。

**输入验证的优化:**

实现更高效的输入验证机制，例如，通过异常处理和输入缓冲区的管理，提高输入处理的效率。

**减少空间占用:**

考虑使用更紧凑的数据结构来存储进程和资源信息，如使用位向量而不是整数数组。

**并行处理:**

在可能的情况下，实现并行或多线程处理，特别是在处理大量进程和资源时。

**用户界面优化:**

实现图形用户界面以提高易用性和交互性。

**c．经验和体会**

编写实现银行家算法的代码是一次极具启发性和挑战性的经历。它不仅需要对操作系统的资源分配和死锁避免机制有深入的理解，还要求将这些理论知识应用到实际的编程中。这个过程既是对我的编程技能的一次锻炼，也是对理论知识的实践检验。

在编码过程中，我意识到细节的重要性，特别是在处理用户输入和错误处理方面。合理的

输入验证和全面的错误处理对于保证程序的健壮性和可靠性至关重要。同时，实现如银行家算法这样复杂的算法，让我对编程逻辑和算法设计的理解更加深刻。

调试和测试也是编写此代码过程中的重要部分。通过细致的调试和全面的测试，我能够确保算法在各种不同条件下都能正常运行。此外，我也学到了代码优化的重要性，不仅在于提升性能，也在于保证代码的可读性和可维护性。

总的来说，这个任务是一次宝贵的学习经历，它不仅提升了我的编程能力，还加深了我对操作系统核心概念的理解。这次经历强化了我在面对复杂编程任务时的问题解决能力，让我更加认识到理论知识与实践技能的密切联系。

**（5） 测试结果：**

**测试用例 1：正常情况**

输入:

进程数 (n_process) = 3

资源种类数 (n_resource) = 2

资源总量 (Resource) = {10, 5}

最大需求 (Max) = {{7, 5}, {3, 2}, {9, 0}}

已分配资源 (Allocation) = {{0, 1}, {2, 0}, {3, 0}}

请求资源 (Request from Process 1) = {1, 0}

预期输出:

分配前的可用资源 (Available) = {7, 4}

分配后的可用资源 = {6, 4}

安全序列存在，如 {0, 2, 1}

**测试用例 2：边界条件**

输入:

进程数 = 3

资源种类数 = 2

资源总量 = {10, 5}

最大需求 = {{10, 5}, {5, 3}, {7, 2}}

已分配资源 = {{5, 3}, {3, 2}, {2, 2}}

请求资源 (Request from Process 2) = {2, 0}

预期输出:

分配前的可用资源 = {0, 0}

请求被拒绝，因为无法满足需求

**测试用例 3：异常情况**

输入:

进程数 = 3

资源种类数 = 2

资源总量 = {9, 6}

最大需求 = {{3, 2}, {9, 0}, {2, 2}}

已分配资源 = {{1, 2}, {3, 0}, {2, 1}}

请求资源 (Request from Process 1) = {3, 3}（超过最大需求）

预期输出:

请求被拒绝，因为请求超过了进程的最大需求

**测试用例 4：无安全序列的情况**

输入:

进程数 = 3

资源种类数 = 2

资源总量 = {10, 5}

最大需求 = {{7, 5}, {3, 2}, {9, 0}}

已分配资源 = {{0, 1}, {3, 2}, {3, 2}}

请求资源 (Request from Process 0) = {7, 4}

预期输出:

分配前的可用资源 = {7, 4}

请求被拒绝，因为分配后没有安全序列

**（6） 使用说明：**

**步骤 1：启动程序**

运行编译好的银行家算法程序。

**步骤 2：初始化数据（Init）**

- 程序启动后，首先需要初始化数据。
- 根据提示输入资源的种类数（n_resource）和进程的数量（n_process）。
- 为每种资源输入其总量（Resource[i]）。
- 对于每个进程，输入其对每种资源的最大需求量（Max[process][resource]）和当前已分配的资源量（Allocation[process][resource]）。
- 程序将自动计算每个进程的需求量（Need[process][resource]）和系统的初始可用资源（Available[resource]）。

**步骤 3：处理资源请求（Order）**

- 在程序的主菜单中选择处理资源请求的选项。
- 输入请求资源的进程编号（比如，进程 1、2 等）。
- 为该进程输入希望请求的各种资源的数量。
- 程序将检查此次请求是否会导致系统进入不安全状态。如果不会，它将批准该请求并更新资源分配情况；否则，请求将被拒绝。

**步骤 4：显示资源分配情况（Display）**

- 在主菜单中选择显示资源分配情况的选项。
- 程序将展示每个进程的最大需求、当前分配、剩余需求以及当前系统的可用资源。

**步骤 5：退出程序**

- 在主菜单中选择退出选项来结束程序。

**注意事项：**

- 确保在每一步正确地输入数据。无效或错误的输入可能导致程序运行不正确。
- 在处理资源请求时，注意程序的反馈。如果请求被拒绝，应检查当前资源分配情况和进程的需求。

## 4. 实验总结

通过本次实验，我学习了死锁及其预防的相关概念和方法。实验中采用了银行算法来防止死锁的发生，并编写了一个动态分配系统资源的程序进行模拟。在实验过程中，我了解到死锁是指多个进程因竞争有限资源而陷入无法继续执行的状态。为了防止死锁的发生，银行算法被应用于资源的分配。该算法通过安全性检查来确保资源分配不会导致死锁。如果分配是安全的，即剩余资源可以满足任何一个进程完成其执行所需的资源，那么就执行该分配，并继续进行下一次资源分配的试探。

在编写程序的过程中，我采用了合适的数据结构来表示资源数量、最大需求、分配情况等信息，并设计了相应的算法来模拟进程的资源请求和分配过程。通过交互菜单，用户可以进行初始化数据、请求资源、显示当前状态等操作。

在实验中，我遇到了一些挑战和问题，例如如何正确处理用户输入、如何进行安全性检查等。通过仔细分析问题、查阅相关资料并进行调试，我逐渐解决了这些问题，并最终成功完成了实验任务。

通过本次实验，我更深入地了解了死锁的原因和防止死锁的方法。我学会了如何使用银行算法进行资源的安全分配，以避免死锁的发生。此外，我还提高了编程和问题解决能力，加深了对操作系统的理解。

总而言之，本次实验对我来说是一次很有收获的实践。通过实际编写程序和模拟死锁场景，我加深了对死锁及其预防机制的理解，并提升了自己的编程技能和解决问题的能力。这将对我今后的学习和工作中有很大的帮助。

## 5. 附录

```
#include <iostream>
#include <vector>
using namespace std;

#define MAX 20

// 相关数据结构
int n_process;           // 表示进程的个数
int n_resource;          // 表示资源的个数
int Resource[MAX];       // 表示资源的总数
int Max[MAX][MAX];       // 最大需求矩阵
int Allocation[MAX][MAX]; // 分配矩阵
int Need[MAX][MAX];      // 需求矩阵
int Available[MAX];      // 可用资源向量
int Work[MAX];           // 工作向量
bool Finish[MAX];        // 完成标志
vector<int> Safeorder;    // 安全序列记录

void Menu()
{
    cout << "******************银行家算法******************" << endl;
    cout << "*              1.初始化数据                   *" << endl;
    cout << "*              2.申请资源                     *" << endl;
    cout << "*              3.显示资源分配情况             *" << endl;
    cout << "*              4.退出                         *" << endl;
    cout << "**********************************************" << endl;
    cout << "请选择>";
}

/********输入检查********/
void checkInit()
{
    if (n_resource)
        for (int i = 0; i < n_process; i++)
        {
            for (int j = 0; j < n_resource; j++)
            {
                if (Max[i][j] < 0)
```

```
                    cout << "Max[" << i << "][" << j << "]输入值小于0! "
<< endl;
                if (Allocation[i][j] < 0)
                    cout << "Allocation[" << i << "][" << j << "]输入值小
于0! " << endl;
                if (Allocation[i][j] > Max[i][j])
                    cout << "Allocation[" << i << "][" << j << "]的值大于
Max[" << i << "][" << j << "]输入值" << endl;
            }
        }
    for (int i = 0; i < n_resource; i++)
    {
        if (Available[i] < 0)
            cout << "Available[" << i << "]的值小于0! " << endl;
    }
    cout << "输入检查完毕! " << endl;
}

/********初始化********/
int Init()
{
    if (n_resource != 0 && n_process != 0)
    {
        cout << "你已经初始化过了! " << endl;
        return 1;
    }
    cout << "请分别输入资源个数和进程个数，中间用空格隔开：";
    cin >> n_resource >> n_process;
    cout << "请输入各个资源的总拥有量：";
    for (int i1 = 0; i1 < n_resource; i1++)
        cin >> Resource[i1];
    for (int i = 0; i < n_process; i++)
    {
        cout << "P" << i << "对各个资源的最大需求量：";
        for (int j = 0; j < n_resource; j++)
            cin >> Max[i][j];
        cout << "P" << i << "各个资源已分配量：";
        for (int j1 = 0; j1 < n_resource; j1++)
            cin >> Allocation[i][j1];
        for (int j2 = 0; j2 < n_resource; j2++)
            Need[i][j2] = Max[i][j2] - Allocation[i][j2];
    }
    for (int i2 = 0; i2 < n_resource; i2++)
    {
```

```
        int sum[MAX] = {0};
        for (int j = 0; j < n_process; j++)
        {
            if (i2 == 0)
                sum[i2] += Allocation[j][i2];
            if (i2 == 1)
                sum[i2] += Allocation[j][i2];
            if (i2 == 2)
                sum[i2] += Allocation[j][i2];
        }
        Available[i2] = Resource[i2] - sum[i2];
    }
    checkInit();
    return 1;
}

/********安全性检查********/
bool Safecheck()
{
    Safeorder.clear();
    for (int i = 0; i < n_resource; i++)
        Work[i] = Available[i];
    for (int i3 = 0; i3 < n_process; i3++)
        Finish[i3] = false;
    // 开始安全性检查
    int count = 0;
    for (int k = 0; k < n_process; k++)
    {
        for (int i = 0; i < n_process; i++)
        {
            if (Finish[i] == false)
            {
                count = 0;
                for (int j = 0; j < n_resource; j++)
                {
                    if (Need[i][j] <= Work[j])
                        count++;
                }
                if (count == n_resource)
                {
                    for (int j = 0; j < n_resource; j++)
                    {
                        Work[j] = Work[j] + Allocation[i][j];
                    }
```

```
                    Finish[i] = true;
                    Safeorder.push_back(i);
                }
            }
        }
    }
    count = 0;
    for (int i4 = 0; i4 < n_process; i4++)
    {
        if (Finish[i4] == true)
            count++;
    }
    if (count == n_process)
        return true;
    else
        return false;
}

/********资源分配********/
int Order()
{
    int n = -1;                          // 请求资源的进程号
    int *Request = new int[n_resource]; // 表示请求的各个资源数量
    cout << "请输入你要请求的进程号：";
    cin >> n;
    cout << "请输入你要请求各个资源的数量，中间用空格隔开：";
    for (int i6 = 0; i6 < n_resource; i6++)
        cin >> Request[i6];
    // 开始判断
    for (int i = 0; i < n_resource; i++)
    {
        if (Need[n][i] < Request[i])
        {
            cout << "需求量>最大需求量，分配失败" << endl;
            return 1;
        }
    }
    for (int i7 = 0; i7 < n_resource; i7++)
    {
        if (Available[i7] < Request[i7])
        {
            cout << "系统资源无法满足要求，分配失败" << endl;
            return 1;
        }
```

```
    }
    // 试分配资源给请求进程，并做安全性检查
    for (int i8 = 0; i8 < n_resource; i8++)
    {
        Available[i8] -= Request[i8];
        Allocation[n][i8] += Request[i8];
        Need[n][i8] -= Request[i8];
    }
    bool Is_safe = Safecheck();
    if (Is_safe == true)
    {
        cout << "系统已经分配资源给 P" << n << "进程了！ " << endl;
        cout << "其中一个安全序列为： " << endl;
        for (int i = 0; i < Safeorder.size(); i++)
            cout << "P" << Safeorder.at(i) << "->";
        cout << "End" << endl;
    }
    else
    {
        cout << "系统会处于不安全状态，分配失败" << endl;
        // 恢复试分配之前的现场
        for (int i = 0; i < n_resource; i++)
        {
            Available[i] += Request[i];
            Allocation[n][i] -= Request[i];
            Need[n][i] += Request[i];
        }
    }
    return 1;
}

/********显示********/
void Display()
{
    cout << endl;
    cout << "进程 \t Max \t Allocation\tNeed\tAvailable" << endl;
    for (int i = 0; i < n_process; i++)
    {
        cout << " P" << i << " \t";
        for (int j = 0; j < n_resource; j++)
        {
            cout << Max[i][j] << " ";
        }
        cout << "\t   ";
```

```
        for (int j4 = 0; j4 < n_resource; j4++)
        {
            cout << Allocation[i][j4] << " ";
        }
        cout << "\t";
        for (int j5 = 0; j5 < n_resource; j5++)
        {
            cout << Need[i][j5] << " ";
        }
        cout << "\t  ";
        for (int j6 = 0; i == 0 && j6 < n_resource; j6++)
        {
            cout << Available[j6] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

/********主控函数********/
int main()
{
    int choose = 0;
    while (1)
    {
        Menu();
        cin >> choose;
        switch (choose)
        {
        case 1:
            Init();
            break;
        case 2:
            Order();
            break;
        case 3:
            Display();
            break;
        case 4:
            cout << "系统已退出！";
            return 1;
        default:
            cout << "查无此项" << endl;
            break;
```

```
        }
    }
}
```