操作系统实验 处理机调度 银行家算法

姓名：陈彦旭

班级：无24

1 问题描述

银行家算法是避免死锁的一种重要方法，将操作系统视为银行家，操作系统管理的资源视为银行家管理的资金，进程向操作系统请求分配资源即企业向银行申请贷款的过程。

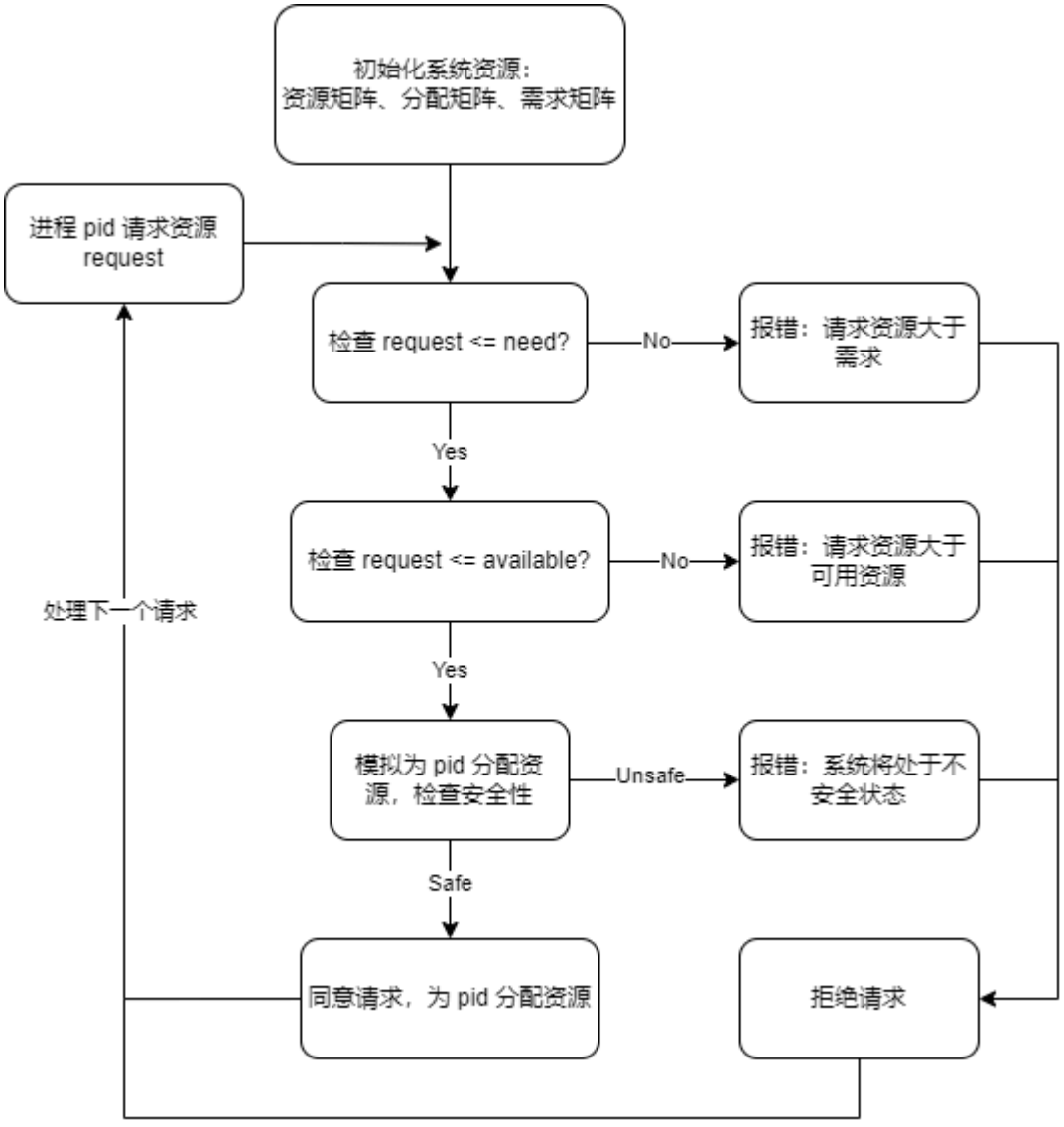
请根据银行家算法的思想，编写程序模拟实现动态资源分配，并能够有效避免死锁的发生。

2 设计思路

OS: Ubuntu 24.04.2 LTS x86_64.

Language: Python 3.12.

流程图：



设计一个银行家类 `Banker`，实现对系统资源的管理和调度功能。主要成员变量：

1. `total`：资源总数。
2. `allocation`：分配矩阵，表示每个进程已经分配的资源数。
3. `max_demand`：最大需求矩阵，表示每个进程对资源的最大需求。
4. `available`：可用资源，表示当前系统中可用的资源数。
5. `need`：需求矩阵，表示每个进程还需要的资源数。
6. `num_processes`：进程个数。
7. `num_resources`：资源种类数。

主要的方法有：

1. `validate_max_demand()`：在初始化时检查是否所有进程的需求都小于系统资源总数。
2. `print_state()`：输出当前系统状态，包括资源总数、分配矩阵等5个重要指标。
3. `is_request_valid()`：判断请求是否合法，包括检查进程号是否存在、请求资源是否超过最大需求、请求资源是否超过可用资源。
4. `is_system_safe()`：判断系统是否处于安全状态，使用安全性算法检查当前资源分配是否会导致死锁，并返回一个安全序列。如果系统处于不安全状态，则返回空序列。其中，安全性算法的思路大致为：现根据当前系统可用资源数，尝试全部分配给其中某一个进程，如果能满足该进程的最大需求，则模拟该进程分配得到所有资源之后释放所有资源，然后再次寻找是否存在其他这样的进程，最后如果所有进程都满足并能释放资源，说明不会出现死锁，系统是安全的。
5. `process_request()`：处理一次进程的资源分配请求。
6. `release_resources()`：处理一次进程的资源释放请求。

在主文件 `main.py` 中，首先初始化系统资源，从 JSON 文件中读取存储的信息，创建银行家 `Banker` 对象，得到资源总数 `total`、分配矩阵 `allocation`、最大需求矩阵 `max_demand`，由此可以计算出资源种类数、进程个数、可用资源 `available` 和需求矩阵 `need`。

对于系统运行过程中每一个进程的每一个请求，首先判断请求是否合法。若合法，则尝试进行资源分配，更新 `allocation, available, need` 矩阵，并判断系统是否处于安全状态。若处于安全状态，则允许分配；否则，拒绝此次请求，恢复刚刚分配的资源，系统回滚到请求前的状态。

3 文件结构说明

```
.
|-- data/
|   |-- example_1.json        # 测试样例1
|   |-- example_1_out.txt     # 测试样例1的输出
|   |-- example_2.json        # 测试样例2
|   |-- example_2_out.txt     # 测试样例2的输出
|   |-- example_3.json        # 测试样例3
|   `-- example_3_out.txt     # 测试样例3的输出
|-- report/
|   |-- report.md             # 实验报告
```

```
|   `-- report.pdf            # 实验报告
`-- src/
    |-- banker.py             # 银行家类
    `-- main.py               # 主文件
```

4 样例测试

设计三个不同的测试样例，位于目录 `data/` 下，分别为 `example_1.json, example_2.json. example_3.json` ，测试样例包括资源分配成功和失败的情况。

其中使用样例 `example_3.json` 的输出结果如下：

```
Read data from file /home/derrick/study/operating-system-
project/lab3_banker_algorithm/data/example_1.json.
Current state of the system resources:
Total: [10  8  7  9]
Allocated:
[[1 0 1 2]
 [2 1 0 1]
 [1 2 2 1]
 [0 1 1 2]]
Maximum demand:
[[5 4 3 5]
 [3 2 2 3]
 [7 5 4 6]
 [4 3 3 4]]
Need:
[[4 4 2 3]
 [1 1 2 2]
 [6 3 2 5]
 [4 2 2 2]]
Available: [6 4 3 3]
------------------------------------------------------------
Process 1 is requesting resources: [1 0 1 1]
Beginning system safety check...
Initial Work (Available Resources): [5, 4, 2, 2]
Process 1 can proceed. Need: [0, 1, 1, 1], Work: [5, 4, 2, 2]
Process 1 has finished. Updated Work: [8, 5, 3, 4]
Process 0 can proceed. Need: [4, 4, 2, 3], Work: [8, 5, 3, 4]
Process 0 has finished. Updated Work: [9, 5, 4, 6]
Process 2 can proceed. Need: [6, 3, 2, 5], Work: [9, 5, 4, 6]
Process 2 has finished. Updated Work: [10, 7, 6, 7]
Process 3 can proceed. Need: [4, 2, 2, 2], Work: [10, 7, 6, 7]
Process 3 has finished. Updated Work: [10, 8, 7, 9]
The system will be in a safe state. Safe sequence: [1, 0, 2, 3]
Request accepted. Allocation successful.
Current state of the system resources:
Total: [10  8  7  9]
Allocated:
[[1 0 1 2]
 [3 1 1 2]
```

```
 [1 2 2 1]
 [0 1 1 2]]
Maximum demand:
[[5 4 3 5]
 [3 2 2 3]
 [7 5 4 6]
 [4 3 3 4]]
Need:
[[4 4 2 3]
 [0 1 1 1]
 [6 3 2 5]
 [4 2 2 2]]
Available: [5 4 2 2]
------------------------------------------------------------
Process 0 is requesting resources: [5 0 0 0]
Process 0 is requesting more than its maximum demand:
Request: [5 0 0 0]
Need:    [4 4 2 3]
Invalid request. Allocation rejected.
Current state of the system resources:
Total: [10  8  7  9]
Allocated:
[[1 0 1 2]
 [3 1 1 2]
 [1 2 2 1]
 [0 1 1 2]]
Maximum demand:
[[5 4 3 5]
 [3 2 2 3]
 [7 5 4 6]
 [4 3 3 4]]
Need:
[[4 4 2 3]
 [0 1 1 1]
 [6 3 2 5]
 [4 2 2 2]]
Available: [5 4 2 2]
------------------------------------------------------------
Process 2 is requesting resources: [1 1 1 3]
Process 2 is requesting more than available resources:
Request:   [1 1 1 3]
Available: [5 4 2 2]
Invalid request. Allocation rejected.
Current state of the system resources:
Total: [10  8  7  9]
Allocated:
[[1 0 1 2]
 [3 1 1 2]
 [1 2 2 1]
 [0 1 1 2]]
Maximum demand:
[[5 4 3 5]
 [3 2 2 3]
```

```
 [7 5 4 6]
 [4 3 3 4]]
Need:
[[4 4 2 3]
 [0 1 1 1]
 [6 3 2 5]
 [4 2 2 2]]
Available: [5 4 2 2]
------------------------------------------------------------
Process 3 is requesting resources: [1 0 1 0]
Beginning system safety check...
Initial Work (Available Resources): [4, 4, 1, 2]
Process 1 can proceed. Need: [0, 1, 1, 1], Work: [4, 4, 1, 2]
Process 1 has finished. Updated Work: [7, 5, 2, 4]
Process 0 can proceed. Need: [4, 4, 2, 3], Work: [7, 5, 2, 4]
Process 0 has finished. Updated Work: [8, 5, 3, 6]
Process 2 can proceed. Need: [6, 3, 2, 5], Work: [8, 5, 3, 6]
Process 2 has finished. Updated Work: [9, 7, 5, 7]
Process 3 can proceed. Need: [3, 2, 1, 2], Work: [9, 7, 5, 7]
Process 3 has finished. Updated Work: [10, 8, 7, 9]
The system will be in a safe state. Safe sequence: [1, 0, 2, 3]
Request accepted. Allocation successful.
Current state of the system resources:
Total: [10  8  7  9]
Allocated:
[[1 0 1 2]
 [3 1 1 2]
 [1 2 2 1]
 [1 1 2 2]]
Maximum demand:
[[5 4 3 5]
 [3 2 2 3]
 [7 5 4 6]
 [4 3 3 4]]
Need:
[[4 4 2 3]
 [0 1 1 1]
 [6 3 2 5]
 [3 2 1 2]]
Available: [4 4 1 2]
------------------------------------------------------------
Process 0 is requesting resources: [1 0 1 0]
Beginning system safety check...
Initial Work (Available Resources): [3, 4, 0, 2]
The system will be in an unsafe state.
Request denied. Back to the previous state.
Current state of the system resources:
Total: [10  8  7  9]
Allocated:
[[1 0 1 2]
 [3 1 1 2]
 [1 2 2 1]
 [1 1 2 2]]
```

```
Maximum demand:
[[5 4 3 5]
 [3 2 2 3]
 [7 5 4 6]
 [4 3 3 4]]
Need:
[[4 4 2 3]
 [0 1 1 1]
 [6 3 2 5]
 [3 2 1 2]]
Available: [4 4 1 2]
------------------------------------------------------------
```

在这组测试样例中：

1. 第1次请求成功，分配资源后系统处于安全状态，安全序列为 `[1 3 0 2]` 。
2. 第2次请求失败，不合法，进程0请求的资源 `[5 0 0 0]` 超过了其最大需求 `[4 4 2 3]` 。
3. 第3次请求失败，不合法，进程2请求的资源 `[1 1 1 3]` 超过了可用资源 `[5 4 2 2]` 。
4. 第4次请求成功，分配资源后系统处于安全状态，安全序列为 `[1 3 0 2]` 。
5. 第5次请求失败，因为系统处于不安全状态，如果同意请求，则剩余资源 `[3 4 0 2]` 无法满足其他所有进程需求，导致各个进程等待而又无法分配资源，出现死锁，因此系统不安全。

测试样例1中 `example_1.json` 包含4种资源、4个进程、5次请求，测试样例2 `example_2.json` 包含10种资源、15个进程、12次请求，测试样例3 `example_3.json` 包含16种资源、20个进程、20次请求。运行测试之后，结果均符合预期，且包含请求不合法的各种情况、请求合法但是不安全、请求合法且安全的情况，能够正确处理各种异常，程序鲁棒性较好。

代码中使用 `np.array` 的相关操作进行矩阵运算，计算效率较高，对于判断请求是否合法、分配资源、释放资源的操作非常方便迅速，稍微复杂一点的安全性检查算法，使用了两层循环，外层循环用于遍历循环第一个进行分配资源的进程，也就是安全序列的第一个进程，内存循环用于判断剩余进程是否出现需求大于可用资源的情况，因此循环的复杂度均为进程的个数，总体复杂度为 $O(n^2)$ ，其中 n 为进程个数。实际操作系统进程数在几十至几百量级，因此复杂度并不高。

5 思考题

（1）银行家算法在实现过程中需注意资源分配的哪些事项才能避免死锁?

答：初始化时，应保证每个进程的最大需求量小于系统中资源总数，否则该进程永远无法得到所需资源。检查每一个请求的合法性：请求资源不能超过最大需求、请求资源不能超过可用资源。如果合法，应先尝试分配资源，再检查系统安全性，只有系统仍处于安全状态才能真正分配资源，否则拒绝请求，回滚到请求前的状态，防止进入潜在死锁状态。最后还应该在进程结束后即使回收和释放资源，避免其他进程等待时间过长。只有在每一次资源分配都确保系统处于安全状态，才能真正避免死锁的发生。

6 实验感想

银行家算法实现起来并不难，本质上只是一些矩阵的比较和运算，但是它蕴含的思想比较重要，其一是检查每次请求的合法性，其二是即使请求合法，如果分配资源也可能导致系统在未来出现死锁问题，也就是进入不安全状态，因此安全性检查是必不可少的。安全性检查的核心，是产生尝试分配资源，模拟各种分配情况，找到能够避免死锁的安全序列，才能保证不会出现“无论怎样分配资源都会导致死锁”的情况。银行家算法是一种简单而朴素的思想，编写程序观察它的操作流程，也增进了我对操作系统资源分配与调度的理解和掌握。