银行家算法

——《操作系统》课程实验四：

处理机调度问题

黄嘉浩　　无 27　　2022010666

huang-jh22@mails.tsinghua.edu.cn

2025 年 5 月 18 日

目录

1 问题描述及实现要求

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。

实现要求

1. 对实现的算法通过流程图进行说明;

2. 设计不少于三组测试样例，需包括资源分配成功和失败的情况；

3. 能够展示系统资源占用和分配情况的变化及安全性检测的过程；

4. 结合操作系统课程中银行家算法理论对实验结果进行分析，验证结果的正确性；

5. 分析算法的鲁棒性及算法效率。

2 实验环境

编程语言 Python 3.12

编程环境 Windows 11

3 设计思路及程序结构

3.1 设计思路与核心流程

为了保证所有进程正常运行完成，在系统运行过程中，对进程提出的每一个（系统能够满足的）资源申请进行动态检查（安全性算法），并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否

则予以分配。

安全性算法的步骤是，遍历每个进程的需求资源量，判断当前系统的动态可分配资源是否满足该进程的需求量。若满足，则将该进程的资源分配给它，并将其状态改为完成状态（True），释放它所占有的资源；否则，继续遍历下一个进程。若所有进程都处于完成状态，则说明系统处于安全状态。否则，系统处于不安全状态。

核心流程

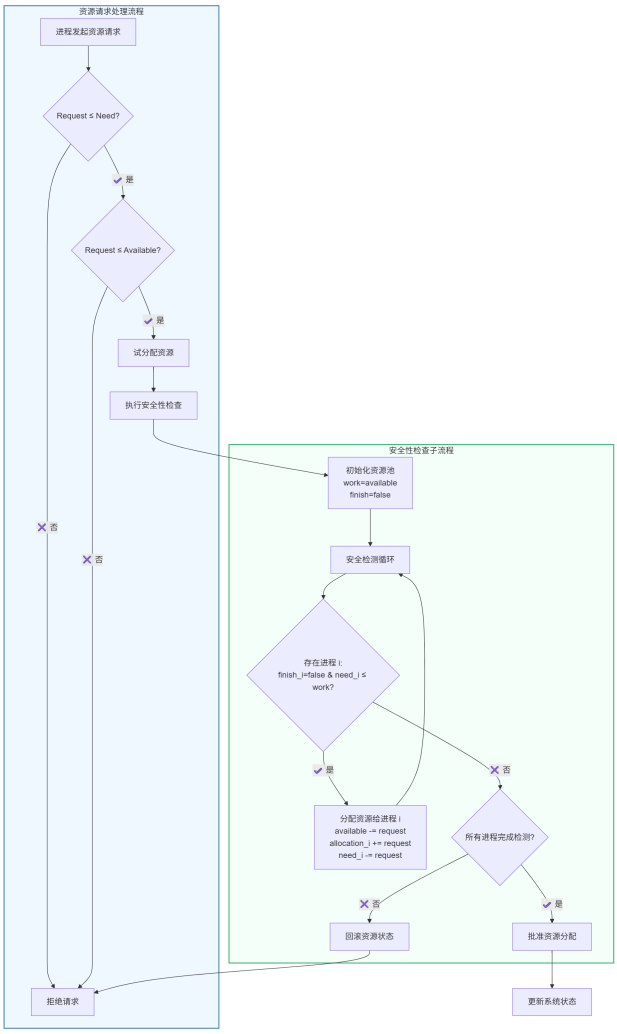

图 1: 银行家算法核心流程

3.2 程序结构与关键代码解释

银行家算法的数据结构共有四类，如下所示。

类别	功能
可利用资源向量 available	含有 m 个元素的数组，其中每个元素代表一类可利用资源的数目
最大需求矩阵 max	n*m 矩阵，表示 n 个进程的每一个对 m 类资源的最大需求
分配矩阵 allocation	n*m 矩阵，表示每个进程已分配的每类资源的数目
需求矩阵 need	n*m 矩阵，表示每个进程还需要各类资源数

3.2.1 安全性算法

Listing 1: 安全性算法代码

```
def is_safe_state(self):
    work = self.available.copy() #动态可分配资源
    finish = [False] * self.num_processes # 是否有足够的资源分配给进程，初值为 false
    safe_sequence = []

    while True:
        found = False
        for i in range(self.num_processes):
            if not finish[i] and np.all(self.need[i] <= work):
                work += self.allocation[i]
                finish[i] = True
                safe_sequence.append(i)
                found = True
                break
        if not found:
            break
```

该安全性算法的主要流程如下：

1. 首先，复制当前可用资源向量 `available` 到临时变量 `work`，并初始化所有进程的完成标志 `finish` 为 `False`，表示所有进程都尚未完成。
2. 进入循环，每次循环尝试寻找一个尚未完成且其需求 `need[i]` 小于等于当前可用资源 `work` 的进程 i。
3. 如果找到这样的进程 i，则认为该进程可以顺利完成，将其分配的资源 `allocation[i]` 归还给系统（即加到 `work` 上），并将 `finish[i]` 置为 `True`，同时将该进程编号加入安全序列 `safe_sequence`。
4. 如果本轮循环没有找到任何可以完成的进程（即 `found` 仍为 `False`），则跳出循环。
5. 最后，判断所有进程的 `finish` 是否都为 `True`。若是，则系统处于安全状态，并返回安全序列；否则，系统处于不安全状态。

3.2.2 资源申请算法

Listing 2: 资源申请算法代码

```
def request_resources(self, process_id, request):
    request = np.array(request)

    # 检查请求是否超过需求
    if np.any(request > self.need[process_id]):
        print(f"错误：进程 {process_id} 的请求超过了其最大需求")
        return False

    # 检查请求是否超过可用资源
    if np.any(request > self.available):
        print(f"错误：进程 {process_id} 的请求超过了可用资源")
        return False

    # 尝试分配资源
```

```
15        old_available = self.available.copy()
16        old_allocation = self.allocation.copy()
17        old_need = self.need.copy()
18
19        self.available -= request
20        self.allocation[process_id] += request
21        self.need[process_id] -= request
22
23        # 检查安全性
24        safe, sequence = self.is_safe_state()
```

该资源申请算法的主要流程如下：

1. 首先，将请求向量 `request` 转换为数组，方便后续计算。
2. 检查请求是否超过了该进程的最大需求（`need[process_id]` ）。如果超过，直接拒绝请求并返回 `False`。
3. 检查请求是否超过了当前系统可用资源（`available`）。如果超过，直接拒绝请求并返回 `False`。
4. 如果请求合法，尝试分配资源。为防止分配后系统进入不安全状态，先保存当前的 `available`、`allocation` 和 `need` 状态，便于回滚。
5. 执行资源分配操作：将请求的资源从 `available` 中减去，分配到对应进程的 `allocation`，并更新 `need`。
6. 调用前述安全性算法（`is_safe_state`）判断分配后系统是否安全。
7. 如果系统安全，则分配成功，返回 `True`；否则，回滚到分配前的状态，拒绝本次请求，返回 `False`。

这样，便保证了每次资源分配都不会让系统进入不安全状态，从而有效避免死锁的发生。

4 程序运行情况

笔者设计了三组测试用例，分别展示了资源分配成功、资源分配失败（超过需求及超过可用资源导致不安全状态）和多资源类型的情况。

4.1 测试用例一：资源分配成功

4.1.1 测试用例一分析

基本信息：

- 进程数：5
- 资源种类数：3
- 总资源向量：`[10, 5, 7]`

分配矩阵（allocation）：

进程	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

最大需求矩阵（max）：

进程	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

请求序列：

1. 进程 1 请求 `[1, 0, 2]`
2. 进程 4 请求 `[3, 3, 0]`
3. 进程 0 请求 `[0, 2, 0]`

算法执行过程分析：

- 初始可用资源向量：`[3, 3, 2]`
- 进程 1 请求 `[1, 0, 2]`，满足需求且系统安全，分配成功，此时找到一条系统安全序列：`[1, 3, 0, 2, 4]`。
- 分配后可用资源向量：`[2, 3, 0]`
- 进程 4 请求 `[3, 3, 0]`，请求超过可用资源，分配失败。
- 进程 0 请求 `[0, 2, 0]`，资源分配会导致不安全状态（死锁），分配失败。
- 最终系统安全序列：`[1, 3, 0, 2, 4]`。

4.1.2 测试用例一运行结果

Listing 3: 测试用例一运行结果

```
===== 测试用例 资源分配成功案例 =====

-> 当前系统状态:
可用资源: [3 3 2]

分配矩阵:
进程    资源0   资源1   资源2
0       0       1       0
1       2       0       0
2       3       0       2
3       2       1       1
4       0       0       2

需求矩阵:
进程    资源0   资源1   资源2
0       7       4       3
1       1       2       2
2       6       0       0
3       0       1       1
4       4       3       1

系统处于安全状态，安全序列为: [1, 3, 0, 2, 4]

进程 1 请求资源: [1, 0, 2]
资源分配成功，安全序列为: [1, 3, 0, 2, 4]

-> 当前系统状态:
可用资源: [2 3 0]

分配矩阵:
进程    资源0   资源1   资源2
0       0       1       0
1       3       0       2
2       3       0       2
3       2       1       1
4       0       0       2

```

```
需求矩阵:
进程    资源0   资源1   资源2
0       7       4       3
1       0       2       0
2       6       0       0
3       0       1       1
4       4       3       1

系统处于安全状态，安全序列为: [1, 3, 0, 2, 4]

进程 4 请求资源: [3, 3, 0]
错误：进程 4 的请求超过了可用资源

-> 当前系统状态:
可用资源: [2 3 0]

分配矩阵:
进程    资源0   资源1   资源2
0       0       1       0
1       3       0       2
2       3       0       2
3       2       1       1
4       0       0       2

需求矩阵:
进程    资源0   资源1   资源2
0       7       4       3
1       0       2       0
2       6       0       0
3       0       1       1
4       4       3       1

系统处于安全状态，安全序列为: [1, 3, 0, 2, 4]

进程 0 请求资源: [0, 2, 0]
资源分配会导致不安全状态，请求被拒绝

-> 当前系统状态:
可用资源: [2 3 0]

分配矩阵:
```

```
进程    资源0   资源1   资源2
0       0       1       0
1       3       0       2
2       3       0       2
3       2       1       1
4       0       0       2

需求矩阵:
进程    资源0   资源1   资源2
0       7       4       3
1       0       2       0
2       6       0       0
3       0       1       1
4       4       3       1

系统处于安全状态，安全序列为: [1, 3, 0, 2, 4]
```

最终，系统找到了安全序列 `[1，3，0，2，4]`，说明系统处于安全状态，这是符合前述分析的。

4.2 测试用例二：资源分配失败

4.2.1 测试用例二分析

基本信息：

- 进程数：5
- 资源种类数：3
- 总资源向量：`[10，5，7]`

分配矩阵（allocation）：

进程	A	B	C
P0	0	4	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

最大需求矩阵（max）：

进程	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

请求序列：

1. 进程 1 请求 `[3, 0, 2]`
2. 进程 4 请求 `[3, 0, 3]`
3. 进程 0 请求 `[0, 0, 2]`

算法执行过程分析：

- 初始可用资源向量：`[3, 0, 2]`
- 进程 1 请求 `[3, 0, 2]`，请求量超过其最大需求 `[1, 2, 2]`，系统拒绝分配。
- 进程 4 请求 `[3, 0, 3]`，请求量超过当前可用资源，系统拒绝分配。

- 进程 0 请求 [0, 0, 2]，虽然请求量合法且不超过可用资源，但分配后系统进入不安全状态，系统拒绝分配。
- 最终系统状态未发生变化，系统始终处于不安全状态。

4.2.2 测试用例二运行结果

Listing 4: 测试用例二运行结果

```
===== 测试用例 资源分配失败案例 =====

-> 当前系统状态:
可用资源: [3 0 2]

分配矩阵:
进程    资源0   资源1   资源2
0       0       4       0
1       2       0       0
2       3       0       2
3       2       1       1
4       0       0       2

需求矩阵:
进程    资源0   资源1   资源2
0       7       1       3
1       1       2       2
2       6       0       0
3       0       1       1
4       4       3       1

系统处于不安全状态

进程 1 请求资源: [3, 0, 2]
错误：进程 1 的请求超过了其最大需求

-> 当前系统状态:
可用资源: [3 0 2]

分配矩阵:
进程    资源0   资源1   资源2
```

```
0       0       4       0
1       2       0       0
2       3       0       2
3       2       1       1
4       0       0       2

需求矩阵:
进程    资源0   资源1   资源2
0       7       1       3
1       1       2       2
2       6       0       0
3       0       1       1
4       4       3       1

系统处于不安全状态

进程 4 请求资源: [3, 0, 3]
错误：进程 4 的请求超过了其最大需求

-> 当前系统状态:
可用资源: [3 0 2]

分配矩阵:
进程    资源0   资源1   资源2
0       0       4       0
1       2       0       0
2       3       0       2
3       2       1       1
4       0       0       2

需求矩阵:
进程    资源0   资源1   资源2
0       7       1       3
1       1       2       2
2       6       0       0
3       0       1       1
4       4       3       1

系统处于不安全状态

进程 0 请求资源: [0, 0, 2]
```

```
资源分配会导致不安全状态，请求被拒绝

—> 当前系统状态:
可用资源: [3 0 2]

分配矩阵:
进程    资源0   资源1   资源2
0       0       4       0
1       2       0       0
2       3       0       2
3       2       1       1
4       0       0       2

需求矩阵:
进程    资源0   资源1   资源2
0       7       1       3
1       1       2       2
2       6       0       0
3       0       1       1
4       4       3       1

系统处于不安全状态
```

最终，系统无法找到安全序列，这是符合前述分析的。

4.3 测试用例三：多资源类型

4.3.1 测试用例三分析

基本信息：

- 进程数：4
- 资源种类数：4
- 总资源向量：`[10, 5, 7, 8]`

分配矩阵（allocation）：

进程	A	B	C	D
P0	0	1	1	0
P1	2	0	0	1
P2	3	0	2	0
P3	2	1	1	2

最大需求矩阵（max）：

进程	A	B	C	D
P0	7	5	3	2
P1	3	2	2	2
P2	9	0	2	2
P3	2	2	2	4

请求序列：

1. 进程 1 请求 `[1, 0, 1, 0]`
2. 进程 3 请求 `[0, 1, 0, 1]`
3. 进程 2 请求 `[2, 0, 0, 1]`

分析过程：

- 初始可用资源向量：`[3, 3, 3, 5]`
- 进程 1 请求 `[1, 0, 1, 0]`，当前可用资源满足其需求，资源分配后系统存在安全序列 `[1, 3, 0, 2]`，资源分配成功。
- 分配后可用资源向量：`[2, 3, 2, 5]`
- 进程 3 请求 `[0, 1, 0, 1]`，当前可用资源满足其需求，资源分配后系统存在安全序列 `[1, 3, 0, 2]`，资源分配成功。

- 分配后可用资源向量：[2，2，2，4]
- 进程 2 请求 [2，0，0，1]，当前可用资源满足其需求，资源分配后系统存在安全序列 [1，3，0，2]，资源分配成功。
- 分配后可用资源向量：[0，2，2，3]
- 最终系统安全序列：[1，3，0，2]。

4.3.2 测试用例三运行结果

Listing 5: 测试用例三运行结果

```
===== 测试用例 多资源类型案例 =====

-> 当前系统状态:
可用资源: [3 3 3 5]

分配矩阵:
进程    资源0   资源1   资源2   资源3
0       0       1       1       0
1       2       0       0       1
2       3       0       2       0
3       2       1       1       2

需求矩阵:
进程    资源0   资源1   资源2   资源3
0       7       4       2       2
1       1       2       2       1
2       6       0       0       2
3       0       1       1       2

系统处于安全状态，安全序列为: [1, 3, 0, 2]

进程 1 请求资源: [1, 0, 1, 0]
资源分配成功，安全序列为: [1, 3, 0, 2]

-> 当前系统状态:
可用资源: [2 3 2 5]
```

```
27
28  分配矩阵:
29  进程    资源0   资源1   资源2   资源3
30  0       0       1       1       0
31  1       3       0       1       1
32  2       3       0       2       0
33  3       2       1       1       2
34
35  需求矩阵:
36  进程    资源0   资源1   资源2   资源3
37  0       7       4       2       2
38  1       0       2       1       1
39  2       6       0       0       2
40  3       0       1       1       2
41
42  系统处于安全状态，安全序列为: [1, 3, 0, 2]
43
44  进程 3 请求资源: [0, 1, 0, 1]
45  资源分配成功，安全序列为: [1, 3, 0, 2]
46
47  -> 当前系统状态:
48  可用资源: [2 2 2 4]
49
50  分配矩阵:
51  进程    资源0   资源1   资源2   资源3
52  0       0       1       1       0
53  1       3       0       1       1
54  2       3       0       2       0
55  3       2       2       1       3
56
57  需求矩阵:
58  进程    资源0   资源1   资源2   资源3
59  0       7       4       2       2
60  1       0       2       1       1
61  2       6       0       0       2
62  3       0       0       1       1
63
64  系统处于安全状态，安全序列为: [1, 3, 0, 2]
65
66  进程 2 请求资源: [2, 0, 0, 1]
67  资源分配成功，安全序列为: [1, 3, 2, 0]
```

```
68
69  -> 当前系统状态:
70  可用资源: [0 2 2 3]
71
72  分配矩阵:
73  进程    资源0   资源1   资源2   资源3
74  0       0       1       1       0
75  1       3       0       1       1
76  2       5       0       2       1
77  3       2       2       1       3
78
79  需求矩阵:
80  进程    资源0   资源1   资源2   资源3
81  0       7       4       2       2
82  1       0       2       1       1
83  2       4       0       0       1
84  3       0       0       1       1
85
86  系统处于安全状态，安全序列为: [1, 3, 2, 0]
```

最终，系统找到了安全序列 `[1, 3, 2, 0]`，说明系统处于安全状态，这是符合前述分析的。

5 算法鲁棒性及效率分析

本算法在资源请求时，首先检查请求是否超过进程的最大需求（need）和系统当前可用资源（available）。若请求不合法，立即拒绝，防止非法操作导致系统异常。这保证了算法对异常输入的容错能力。

在尝试分配资源后，算法会调用安全性检测（is_safe_state）。若检测到系统进入不安全状态，会将所有资源分配操作回滚到分配前的状态，确保系统始终处于安全状态。这一机制极大增强了算法对潜在死锁和资源分配错误的防护能力。

对于超过最大需求或可用资源的请求，算法会输出明确的错误提示，便于用户定位问题。

总之，在输入合法性检查、资源分配回滚和错误提示等方面，算法展现了较强的鲁棒性。

时间复杂度分析

在每次资源请求时，算法需要进行两步主要操作：（1）合法性检查（与进程数和资源种类线性相关）；（2）安全性检测（is_safe_state），其核心是尝试找到一个安全序列。安全性检测的最坏情况是每次都只能完成一个进程，总共需要遍历 n 次（n 为进程数），每次判断需遍历所有进程和资源，因此时间复杂度为 $O(n^2 \cdot m)$，其中 m 为资源种类数。整体来看，单次资源请求的复杂度为 $O(n^2 \cdot m)$，对于中小规模系统（进程数和资源种类不大）是可以接受的。

空间复杂度分析

算法主要维护分配矩阵、最大需求矩阵、需求矩阵和可用资源向量，空间复杂度为 $O(n \cdot m)$。

6 思考题

Q1 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？

1. **请求合法性检查：**每次进程请求资源时，必须首先判断请求量是否超过其最大需求（need）以及当前系统可用资源（available）。只有在请求合法且系统有足够资源时，才考虑分配。

2. **安全性检测：**在资源实际分配前，需通过安全性算法（is_safe_state）模拟分配，判断分配后系统是否仍处于安全状态。只有在分配后系统依然安全时，才真正分配资源。

3. **状态回滚机制：**若分配后系统不安全，必须将所有资源分配操作回滚到分配前的状态，确保系统不会进入不安全状态。

4. **动态更新数据结构：**每次资源分配或释放后，需及时更新分配矩阵（allocation）、需求矩阵（need）和可用资源向量（available），保证数据的准确性。

只有严格遵循上述事项，才能确保银行家算法在动态资源分配过程中有效避免死锁，保证系统安全运行。

7 实验总结

本实验实现了银行家算法，模拟了资源分配和安全性检测的过程。通过多组测试用例验证了算法的正确性和鲁棒性。实验中，笔者深入理解了银行家算法的核心思想和实现细节，并掌握了如何在实际系统中应用该算法来避免死锁和确保系统安全。

总体来看，实验过程进展顺利。本实验采用的银行家算法步骤参考了操作系统课件，遇到的问题主要集中在算法实现的细节上，如资源请求的合法性检查和安全性检测的实现。由于 Python numpy 库直接支持矩阵向量乘法（MVM）运算，故笔者采用 Python 语言编写了本实验。通过不断调试和测试，最终成功实现了预期功能。

最后，感谢老师与助教的辛勤付出，使得笔者获益匪浅。再次对老师和助教表示衷心感谢！