

操作系统实验四实验报告

银行家算法（基准分：100）

学号： 2019011008

班级： 无 92

姓名： 刘雪枫

目 录

一、	问题描述.....	1
二、	实验平台.....	1
三、	算法原理.....	1
四、	算法流程图.....	1
五、	代码实现.....	2
	(一) 数学运算.....	2
	(二) 银行家算法实现.....	3
六、	测试用例.....	6
七、	运行测例.....	7
八、	算法鲁棒性和效率分析.....	10
九、	实验心得.....	11
十、	思考题.....	11

一、 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。

二、 实验平台

本实验在 x86_64 平台的 Ubuntu 20.04 LTS 操作系统上进行，编程语言采用 C++11，构建工具采用 GNU Make 4.2.1。

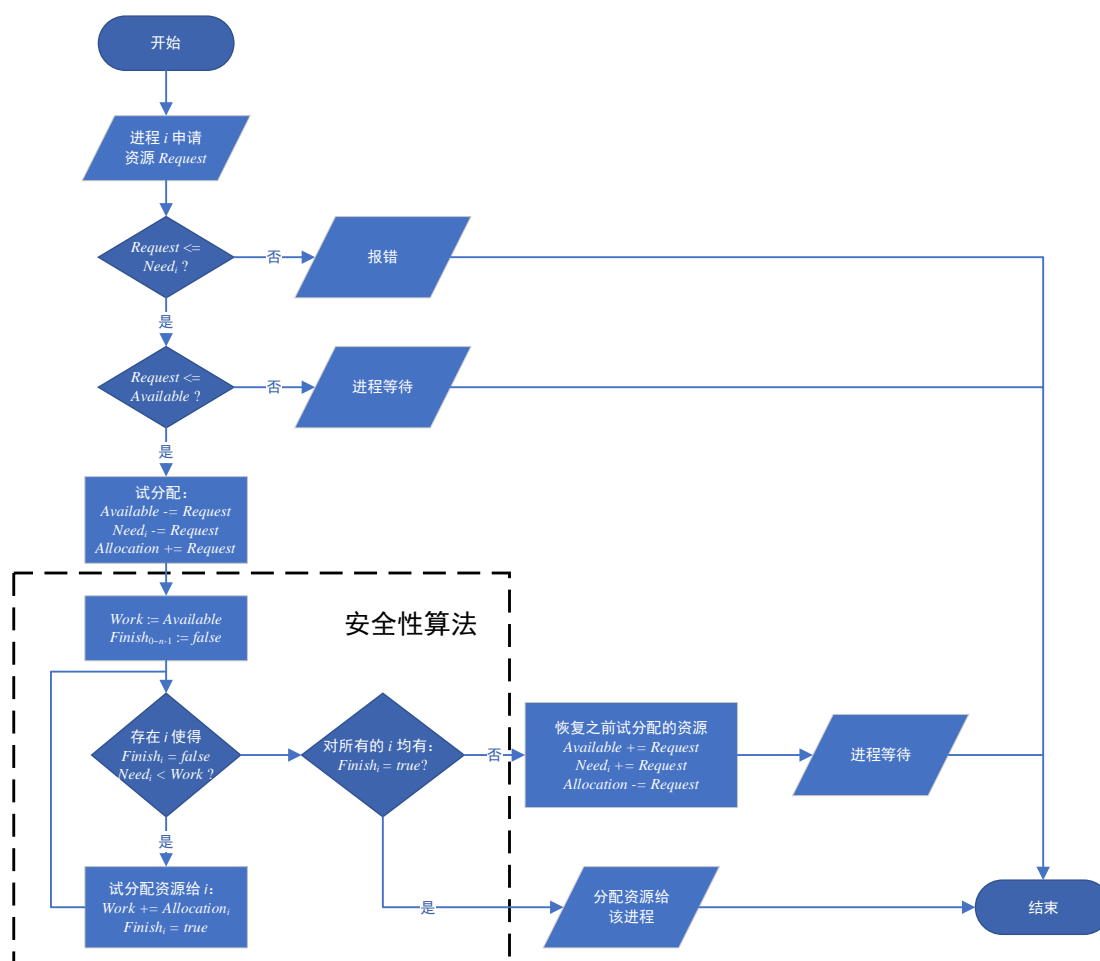
三、 算法原理

银行家算法是避免死锁的一种方法。当有进程申请资源时，如果资源申请超过了其需要的资源数，则会直接报错。否则，银行家算法先假装把资源分配给该进程，再检测分配后的状态是否是安全状态。如果是安全状态，则把资源分配给该进程；如果不是，则该进程开始等待。

检测安全状态使用的是安全性算法。其流程是，贪心地将目前已经有的资源分配给能够得到全部所需资源的进程使其能够运行完成归还资源，循环进行此操作，直到操作不能继续进行。如果所有进程都能够完成，则处于安全状态，否则处于不安全状态。

四、 算法流程图

使用流程图描述该银行家如下：



五、 代码实现

(一) 数学运算

实验使用 C++ 内置的 `std::vector<int>` 来表示各种向量，使用嵌套的 `std::vector<std::vector<int>>` 来表示矩阵。由于算法需要各种矩阵和向量运算，因此先定义向量的加减运算。代码位于 `arithmetic.hpp` 中。定义的函数如下：

```

1. template <typename T>
2. std::vector<T>
3. operator+(const std::vector<T>& v1, const std::vector<T>& v2);
4.
5. template <typename T>

```

```
6. std::vector<T>
7. operator-(const std::vector<T>& v1, const std::vector<T>& v2);
8.
9. template <typename T>
10. std::vector<T>&
11. operator+=(std::vector<T>& v1, const std::vector<T>& v2);
12.
13. template <typename T>
14. std::vector<T>&
15. operator-=(std::vector<T>& v1, const std::vector<T>& v2)
```

(二) 银行家算法实现

银行家算法需要知道进程还需要的资源数矩阵 `need`、已经分配的资源矩阵 `allocated`、剩余的资源 `available`、要申请资源的进程号 `id` 和要申请的资源向量 `acquire`。银行家算法的执行结果有三种：分配资源成功、分配资源失败、进程需等待。因此定义枚举类型作为返回结果，位于 `banker.hpp` 中：

```
1. enum class allocate_result
2. {
3.     FAIL    = 0,
4.     WAIT    = 1,
5.     SUCCESS = 2
6. };
```

下面将之前流程图中的银行家算法使用 C++ 语言表述。编写 `can_allocate` 函数执行银行家算法判断是否可以分配。值得一提的是，本实验中 `can_allocate` 函数与前述算法有细微差别，该函数只用来判断是否可以分配，而不真正执行分配，真正分配资源的工作交由调用者。这样考虑的原因是基于实际使用情况的。因为一般来说，操作系统都会在某一个统一的地方进行资源管理，而分配资源的过程绝不只有修改分配矩阵的值这么简单。将分配资源和修改分配矩阵的值放在统一的地方处理的设计会更加合理。同时，也保证了函数的单一职责性，有利于代码的模块化。如果要更换死锁检测算法，只需要更改 `can_allocate` 即可，可扩展性更强。代码的具体实现在 `banker.cpp` 中，核心代码如下：

```
1. allocate_result
2. can_allocate(const std::vector<std::vector<int>>& need, const std::vector<std::vector<int>>& allocated,
3.             const std::vector<int>& available, const std::vector<int>
4.             & acquire, std::size_t id)
5. {
6.     if (id >= need.size()) {
7.         throw std::invalid_argument("Process ID is invalid!");
8.     }
9.     for (std::size_t i = 0; i < acquire.size(); ++i) {
10.        if (acquire[i] > need[id][i]) {
11.            SAFE_LOG("Fail to allocate to %d\n", (int)id);
12.            return allocate_result::FAIL;
13.        }
14.    }
15.
16.    for (std::size_t i = 0; i < acquire.size(); ++i) {
17.        if (acquire[i] > available[i]) {
18.            SAFE_LOG("Process: %d begins to wait\n", (int)id);
19.            return allocate_result::WAIT;
20.        }
21.    }
22.
23.    auto new_need = need;
24.    new_need[id] -= acquire;
25.    auto new_available = available - acquire;
26.    auto new_allocated = allocated;
27.    new_allocated[id] += acquire;
28.
29.    if (is_safe_state(std::move(new_need), std::move(new_allocated),
30.                      std::move(new_available))) {
31.        SAFE_LOG("Success to allocate to %d\n", (int)id);
32.        return allocate_result::SUCCESS;
33.    }
34.    SAFE_LOG("Process: %d begins to wait\n", (int)id);
35.    return allocate_result::WAIT;
36. }
```

其中，`is_safe_state` 是安全性算法，用于判断是否是安全状态。其实现也在 `banker.cpp` 中。此算法仍然是安全性算法，但使用了队列优化了时间复杂度，即

去掉了 Finish 数组，而是改为将尚未完成的进程放在队列中。这种设计可以避免当搜索进程时需要将全部进程遍历，改为队列后每次遍历只需要遍历未完成的进程了，因此在平均时间性能上会有提升。核心代码如下：

```
1. bool
2. is_safe_state(std::vector<std::vector<int>> need, std::vector<std::vector<int>> allocated, std::vector<int> available)
3. {
4.     if (need.size() == 0) { // Trivial circumstance, safe
5.         return true;
6.     }
7.
8.     std::queue<std::size_t> left_process;
9.
10.    // Push all processes into the queue
11.
12.    for (std::size_t i = 0; i < need.size(); ++i) {
13.        left_process.emplace(i);
14.    }
15.
16.    do {
17.
18.        // Check for once
19.
20.        bool has_allocated = false;
21.        std::size_t left_count = left_process.size();
22.
23.        for (std::size_t i = 0; i < left_count; ++i) {
24.            auto id = left_process.front();
25.            left_process.pop();
26.
27.            // Check if process 'pid' can satisfy
28.
29.            bool satisfy = true;
30.            for (std::size_t j = 0; j < available.size(); ++j) {
31.                if (available[j] < need[id][j]) {
32.                    satisfy = false;
33.                    break;
34.                }
35.            }
36.
37.            if (satisfy) { // Can finish task, allocate to it.
```

```

38.         available += allocated[id];
39.         SAFE_LOG("In safe check: try to allocate to: %d\n", (
            int)id);
40.         has_allocated = true;
41.     } else { // Cannot finish task, push back the process.
42.         left_process.emplace(id);
43.     }
44. }
45.
46.     if (!has_allocated) { // No process allocated
47.         return false;
48.     }
49. } while (!left_process.empty());
50. return true;
51. }

```

六、测试用例

设计四组测试用例，分别对四种情况进行测试：成功分配、当前资源不足而等待、因会产生死锁而等待、因申请资源超出需求而报错。

1. 成功分配

首先，设计成功分配的测试用例。有 5 个进程、3 个资源。参数为：

$$Need = \begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}, Allocation = \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix},$$

$$Available = [3 \quad 3 \quad 2], Acquire = [1 \quad 0 \quad 2]$$

且申请资源的进程为进程 1（仅从号为 0~4）。此时，根据银行家算法，若将资源分配给进程 1，则矩阵变为：

$$Need = \begin{bmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}, Allocation = \begin{bmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix},$$

$$Available = [2 \quad 3 \quad 0]$$

再执行安全性算法，存在一个可能的序列：1 -> 3 -> 4 -> 0 -> 2。因此分配后为安全状态，因此可以成功分配。

2. 申请的资源超出可用资源而等待

设计用例，使得可用资源向量为 $Available = [2 \ 3 \ 0]$ ，进程 4 申请资源 $Acquire = [3 \ 3 \ 0]$ ，并使得需求大于申请资源向量，并且不存在死锁即可。

3. 由于发生死锁而等待

设计用例，有 5 个进程、3 个资源。参数为：

$$Need = \begin{bmatrix} 2 & 4 & 4 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 6 \\ 6 & 3 & 1 \end{bmatrix}, Allocation = \begin{bmatrix} 0 & 1 & 0 \\ 3 & 0 & 0 \\ 1 & 0 & 1 \\ 6 & 1 & 2 \\ 0 & 0 & 6 \end{bmatrix},$$

$$Available = [2 \ 3 \ 4], Acquire = [0 \ 2 \ 4]$$

且申请资源的进程为进程 0（仅从号为 0~4）。

原始数据存在安全序列 1 -> 2 -> 4 -> 3 -> 0。假定将资源分配给进程 0，则矩阵变为：

$$Need = \begin{bmatrix} 2 & 2 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 6 \\ 6 & 3 & 1 \end{bmatrix}, Allocation = \begin{bmatrix} 0 & 3 & 4 \\ 3 & 0 & 0 \\ 1 & 0 & 1 \\ 6 & 1 & 2 \\ 0 & 0 & 6 \end{bmatrix},$$

$$Available = [2 \ 1 \ 0], Acquire = [0 \ 2 \ 4]$$

此时，执行安全性算法，首先可以分配给进程 1，剩余资源变为 $[5 \ 1 \ 0]$ ，然后可以分配给进程 2，剩余资源变为 $[6 \ 1 \ 1]$ 。此时便不能分配给任意一个进程而出现死锁。因此是不安全状态，故不应当分配资源给进程 0。

4. 因申请资源超出需求而报错

此样例很好设计，只需要让请求的资源数很大即可。

七、 运行测例

下面对样例进行测试。编写 main 函数（位于 main.cpp）中，进行数据处理。该 main 函数中，先读取数据组数，对每组数据，先读取进程数和资源数，再一次读取需求矩阵、已分配矩阵、剩余资源矩阵、请求进程号和请求向量。然后执行银行家算法。结束后根据结果进行资源分配，并输出分配后的各个矩阵和向量。

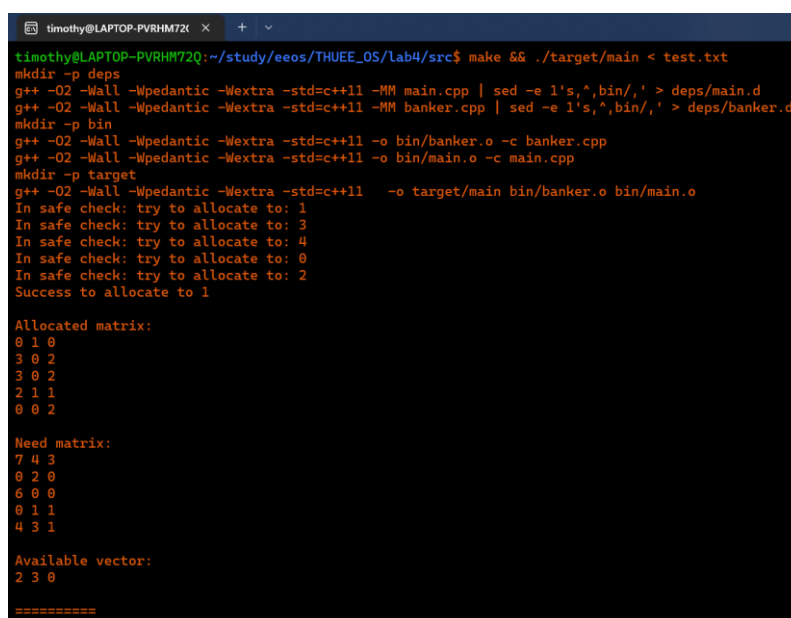
其中，银行家算法和安全性算法的执行过程均输出了 log 信息，外加 main 函数中输出的分配后的矩阵和向量，可以直观地观察程序的运行情况和结果，检验是否正确。

测试用例已经写在 src/main.txt 中。在 src 中执行：

```
$ make && ./target/main < test.txt
```

即可得到结果。

第一个测试用例结果：



```
timothy@LAPTOP-PVRHM72Q:~/study/eeos/THUEE_05/Lab4/src$ make && ./target/main < test.txt
mkdir -p deps
g++ -O2 -Wall -Wpedantic -Wextra -std=c++11 -MM main.cpp | sed -e 1's,^,bin/' > deps/main.d
g++ -O2 -Wall -Wpedantic -Wextra -std=c++11 -MM banker.cpp | sed -e 1's,^,bin/' > deps/banker.d
mkdir -p bin
g++ -O2 -Wall -Wpedantic -Wextra -std=c++11 -o bin/banker.o -c banker.cpp
g++ -O2 -Wall -Wpedantic -Wextra -std=c++11 -o bin/main.o -c main.cpp
mkdir -p target
g++ -O2 -Wall -Wpedantic -Wextra -std=c++11 -o target/main bin/banker.o bin/main.o
In safe check: try to allocate to: 1
In safe check: try to allocate to: 3
In safe check: try to allocate to: 4
In safe check: try to allocate to: 0
In safe check: try to allocate to: 2
Success to allocate to 1

Allocated matrix:
0 1 0
3 0 2
3 0 2
2 1 1
0 0 2

Need matrix:
7 4 3
0 2 0
6 0 0
0 1 1
4 3 1

Available vector:
2 3 0

=====
```

可以看到，如之前所言，成功找到了安全序列 1 -> 3 -> 4 -> 0 -> 2，且分配后矩阵和向量均正确。

第二个测试用例结果：

```
=====
Process: 4 begins to wait

Allocated matrix:
0 1 0
3 0 2
3 0 2
2 1 1
0 0 2

Need matrix:
7 4 3
0 2 0
6 0 0
0 1 1
4 3 1

Available vector:
2 3 0
=====
```

可以看到，进程 4 的结果也是正确的。

第三个测试用例结果：

```
=====
In safe check: try to allocate to: 1
In safe check: try to allocate to: 2
Process: 0 begins to wait

Allocated matrix:
0 1 0
3 0 0
1 0 1
6 1 2
0 0 6

Need matrix:
2 4 4
0 1 0
1 1 0
1 1 6
6 3 1

Available vector:
2 3 4
=====
```

可以看到，该用例也是执行安全性算法是试分配给进程 1 和 2，然后发现不安全状态的。

第四个测试用例结果：

```
=====
Fail to allocate to 0

Allocated matrix:
0 1 0
3 0 0
1 0 1
6 1 2
0 0 6

Need matrix:
2 4 4
0 1 0
1 1 0
1 1 6
6 3 1

Available vector:
2 3 4
=====
```

可以看到算法也成功报错。

八、 算法鲁棒性和效率分析

本算法对多种边界情况都进行了判断，例如进程数为零直接返回安全、向量之间维度不一致会在向量相加操作时会做出判断并抛出 `std::invalid_argument` 异常，鲁棒性可以说是比较不错的，对异常情况都做了不同的处理。

在算法效率上，如果要计算最坏复杂度。设资源数为 m 、进程数为 n ，执行安全性算法时，考虑最坏情况，每轮检查只能检查出一个能够满足的进程，考虑到本实验使用了队列进行优化，因此检查的总进程数为 $n + (n - 1) + (n - 2) + \dots + 1 = \frac{n(n+1)}{2}$ （若不使用队列进行优化，则每次要扫描所有进程，该数值为 n^2 ）。最坏情况下，每次对资源满足性进行检查时，都是最后一个资源不满足，因此对每个进程的检查都检查 m 次，故最坏复杂度为 $O\left(\frac{n(n+1)}{2}m\right) = O(mn^2)$ 。

注意到由于本算法使用了队列进行优化，该最坏情况出现的条件非常苛刻，因此实际使用时，平均复杂度应当会小于上面的值。

九、 实验心得

本次实验中,我自己动手实现了银行家算法,熟悉了银行家算法的具体过程,巩固了课上所学的知识。综合来看,实验过程还是很顺利的,我独立地完成了本次实验。非常感谢老师在课上的辛勤讲授和助教的辛苦付出,老师对知识的课上讲解和助教对实验的精心准备,都让我在此次实验中受益匪浅,再次对老师和助教表示衷心的感谢。

十、 思考题

Q: 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?

A: 首先,初始状态必须是安全的状态;第二,申请的资源数不能超过其需求数;第三,申请的资源数不能超过现有的资源总数;第四,分配资源后,不能进入不安全状态。以上条件都满足后,再分配资源,才能尽可能避免死锁。