实验二：死锁观察与避免

姓名：汪江豪　学号：22121630　实验日期：2024.12.2

实验环境：

实验设备：Lenovo Legion R7000P2021H
开发环境：VScode

实验目的：

死锁会引起计算机工作僵死，造成整个系统瘫痪。因此，死锁现象是操作系统特别是 大型系统中必须设法防止的。学生应独立的使用高级语言编写和调试一个系统动态分配资 源的简单模拟程序，观察死锁产生的条件，并采用适当的算法，有效的防止死锁的发生。 通过实习，更直观地了解死锁的起因，初步掌握防止死锁的简单方法，加深理解课堂上讲 授过的知识。

实验要求：

1. 设计一个 n 个并发进程共享 m 个系统资源的系统。进程可动态地申请资源和释放资 源。系统按各进程的申请动态地分配资源
2. 系统应能显示各进程申请和释放资源以及系统动态分配资源的过程，便于用户观察 和分析
3. 系统应能选择是否采用防止死锁算法或选用何种防止算法(如有多种算法)。在不采用 防止算法时观察死锁现象的发生过程。在使用防止死锁算法时，了解在同样申请条件下，防 止死锁的过程.

实验内容：

1. 题目

本示例采用银行算法防止死锁的发生。假设有三个并发进程共享十个系统。在三个进程申请的系统资源之和不超过 10 时，当然不可能发生死锁，因为各个进程申请的资源都能满足。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。应该排除这二种情况。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。示例中采用了银行算法。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，再往其他方向试探，直至爬上山顶。这种方法是比较保守的。所花的代价也不小。

2. 算法与框图

银行算法，顾名思义是来源于银行的借贷业务，一定数量的本金要应付各种客户的借贷 周转，为了防止银行因资金无法周转而倒闭，对每一笔贷款，必须考察其最后是否能归还。 研究死锁现象时就碰到类似的问题，有限资源为多个进程共享，分配不好就会发生每个进程 都无法继续下去的死锁僵局。 银行算法的原理是先假定每一次分配成立，然后检查由于这次分配是否会引起死锁，即 剩下的资源是不是能满足任一进程完成的需要。如这次分配是安全的(不会引起死锁)，就实 施这次分配，再假定下一次分配。如果不安全，就不实施，再作另一种分配试探，一直探索 到各进程均满足各自的资源要求，防止了死锁的

发生。

程序框图如下图所示：

图 2 防止死锁程序框图

图 3 死锁处理程序框图

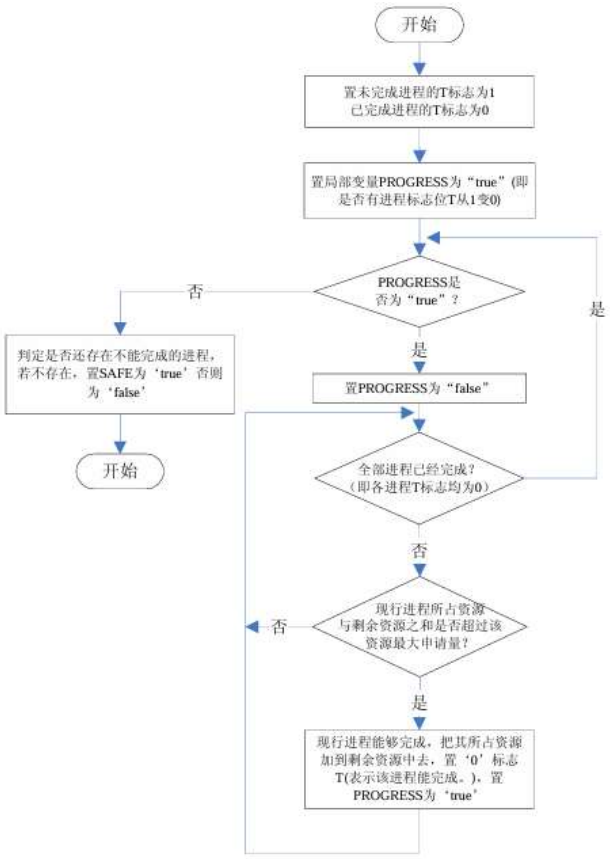


图 4　safe 函数框图

核心代码(安全性检查和资源请求函数实现)：

```
// 执行安全性检查
bool Banker::check(string &list)
{
    // cout << "成功进入 check 函数,开始安全性检查" << endl;
    int n = Max.size();
    int m = Available.size();
    vector<int> work = Available;
    bool finish[n] = {false};
    int num = 0;
    // cout << "预处理完成，即将进入循环" << endl;
    for (int i = 0; i < n; i++)
    {
        if (!finish[i])
        {
            if (larger(work, Need[i]))
            {
                add(work, Allocation[i]);
                finish[i] = true;
                list += "P" + to_string(i) + " ";
                num++;
```

```
                // cout << "当前 finish:";
                // for (auto i : finish)
                //     cout << i << " ";
                // cout << endl;
                i = -1;
            }
        }
        if (num == n)
            return true;
    }
    return false;
}
// 进程请求资源实现
bool Banker::request(int pid, vector<int> &r)
{
    // 进程号过大，请求资源数量与可用资源数量不等
    if (pid < 0 || pid >= Max.size() || r.size() != Available.size())
    {
        cout << "Invalid request" << endl;
        return false;
    }

    // 请求资源数量超过最大需求或可用资源数量
    for (int i = 0; i < r.size(); i++)
    {
        if (r[i] > Need[pid][i] || r[i] > Available[i])
        {
            cout << "Request exceeds need or available resources." << endl;
            return false;
        }
    }

    // 请求合法，尝试分配资源
    cout << "Request is legal, trying allocation" << endl;
    sub(Available, r);
    add(Allocation[pid], r);
    sub(Need[pid], r);
    cout << "Trying finished, starting to check security" << endl;
    string list = "";
    // 如果不安全，恢复资源，报告信息
    if (!check(list))
    {
        add(Available, r);
        sub(Allocation[pid], r);
```

```
        add(Need[pid], r);
        cout << "Request Failed! it would lead to an unsafe state." <<
endl;
        return false;
    }
    cout << "Request Succeed!" << endl;
    cout << "After allocation, one safe sequence: " << list << endl;
    return true;
}
```

主体框架代码：

```
void test()
{
    Banker banker;
    while (true)
    {
        printf("-----银行家算法-----\n");
        printf("1. Init\n");
        printf("2. Request\n");
        printf("3. IsSafe\n");
        printf("4. PrintMatrix\n");
        printf("5. Exit\n");
        printf("--------------------\n");
        printf("Please select: ");
        char c;
        cin >> c;
        switch (c)
        {
        case '1':
        {
            banker = Init();
            break;
        }
```

```
        case '2':
        {
            Request(banker);
            break;
        }
        case '3':
        {
            string list = "";
            if (banker.check(list))
            {
                cout << "Safe now!" << endl;
                cout << "One safe sequence:" << list << endl;
            }
            else
                cout << "Unsafe now!" << endl;
            break;
        }
        case '4':
        {
            PrintMatrix(banker);
            break;
        }
        case '5':
            exit(0);
        }
    }
}
```

结果：

初始化部分：

```
-----银行家算法-----
1. Init
2. Request
3. IsSafe
4. PrintMatrix
5. Exit
--------------------
Please select: 1
Please input the available(A B C):3 3 2
Please enter the number of processes:5
Initialize Max matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Initialize Allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
已初始化完成
```

资源矩阵显示：

```
-----银行家算法-----
1. Init
2. Request
3. IsSafe
4. PrintMatrix
5. Exit
--------------------
Please select: 4
Available(A B C):3 3 2
Process    Max     Allocation  Need
  P0      7 5 3      0 1 0     7 4 3
  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
```

安全性检查：

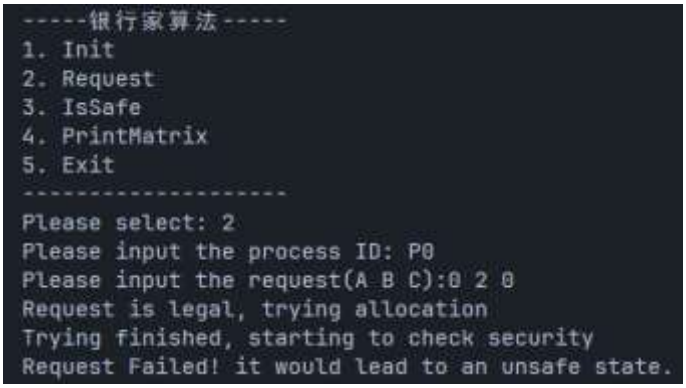
```
-----银行家算法-----
1. Init
2. Request
3. IsSafe
4. PrintMatrix
5. Exit
--------------------
Please select: 3
Safe now!
One safe sequence:P1 P3 P0 P2 P4
```

请求资源 P1(1,0,2)：

```
-----银行家算法-----
1. Init
2. Request
3. IsSafe
4. PrintMatrix
5. Exit
--------------------
Please select: 2
Please input the process ID: P1
Please input the request(A B C):1 0 2
Request is legal, trying allocation
Trying finished, starting to check security
Request Succeed!
After allocation, one safe sequence: P1 P3 P0 P2 P4
```

```
-----银行家算法-----
1. Init
2. Request
3. IsSafe
4. PrintMatrix
5. Exit
--------------------
Please select: 4
Available(A B C):2 3 0
Process    Max     Allocation  Need
  P0      7 5 3      0 1 0     7 4 3
  P1      3 2 2      3 0 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
```

请求资源 P0(0,2,0)：



体会：

本次实验，我基于 C++，实现了银行家算法来避免进程死锁。同时，我对于操作系统中进程资源分配有了更深的理解。银行家算法的核心在于如何安全地分配资源，以避免系统进入不安全状态从而导致死锁。在理解了算法的基本原理后，我着手设计了数据结构，并定义了相关的成员函数，如 request 请求资源函数，check 安全性检查函数，以及各种基于向量运算的成员函数，来实现进程资源类目之间的运算。此次实验让我感受到了操作系统避免进程死锁算法的精妙之处，也让我对计算机科学产生了更浓厚的兴趣。

然而，银行家算法在现代操作系统中，其实并不常见。其实现要求高，性能开销大等等。然而，作为死锁避免的经典算法之一，学好它是十分必要的，这种算法的思想是可以应用到特定的程序或系统当中的，此次实验让我受益匪浅。