中山大学移动信息工程学院本科生实验报告

（2017 学年春季学期）

课程名称：Operating System　　任课教师：饶洋辉　　批改人(此处为 TA 填写)：

年级+班级	1506	专业（方向）	移动信息工程
学号	15352116	姓名	洪子淇
电话	13726205766	Email	1041915206@qq.com
开始日期	2017.6.6	完成日期	2017.6.8

1. 实验目的

1. 了解死锁的含义
2. 了解线程产生死锁现象的原因，以及了解如何解决死锁的方法。
3. 实现安全性检测算法。

2. 实验过程

(一)实验思路简述

每次从第一个线程开始遍历，找到第一个没有遍历过并且空闲资源满足需求的线程，就将该线程的状态设置为“已分配”，并且释放该线程所占有的资源，更新当前的空闲资源数。然后再回到第一个线程开始遍历，直到遍历了 n 次。（目的是：使前面的线程在该线程释放资源后满足了要求，并对此可以进行分配）

(二)伪代码

```
AVAILABLE[MAX];    //记录当前系统空闲的资源数
ALLOCATION[MAX][MAX];  //记录线程目前所占有的资源数
NEED[MAX][MAX];    //记录线程申请的资源数
safety[MAX];  //安全序列
```

```
finish[MAX];  //线程是否被分配资源
for k = 0 ： N
        for i = 0 ： N
        {
            flag = 0;
            if (!finish[i]){
                for j = 0 : M
                    if (NEED[i][j] > AVAILABLE[j]){
                        flag = 1;
                        break;
                    }
                if (flag == 0)
                {
                    safety[count++]=i;
                    finish[i] = 1;
                    for j = 0 : M
                        AVAILABLE[j] += ALLOCATION[i][j];
                    break;
                }
            }
        }
If （count == N） prinft the safe list;
```

(三)具体实现

✧ 参数变量声明:

```
int AVAILABLE[MAX];          系统可用的资源数
int ALLOCATION[MAX][MAX];    线程目前占据的资源数
int NEED[MAX][MAX];          线程还需要的资源数
int safety[MAX];  安全序列
int n, m; //n线程数量; m资源种类
```

```
bool finish[MAX];    //表示线程是否被分配
int count = 0;   //计算分配的线程数量
bool flag = 0;   //找到是否有线程需求的资源数量大于所提供的资源数量
```

✧ 函数实现：

根据思路分析，首先是进入 n 次循环，每一次循环从第一个线程开始找起，如果这个线程是没有被分配的（finish == false），并且系统可以满足要求的线程的资源需求（sign == 0），那么便更新线程的状态（finish = true），以及把该线程所占据的资源释放（AVAILABLE += ALLOCATION），将线程记录到安全序列，并且再从第一个线程开始寻找；如果该线程没有被分配，可以系统的资源数不满足他的需求，则从他的下一个开始寻找；如果该线程是已经分配的，则也是从他的下一个线程开始寻找。其中用了一个 count 记录了安全序列的个数，等到 n 次循环过后，函数返回了 count 的值。

最后只要在 main 函数里面判断 count 的值是否等于线程的数量，即可判断是否为安全状态，如果安全就输出安全序列。

bankman 函数：

```
for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
    {
        flag = 0;
        if (!finish[i]){
            for (int j = 0; j < m; ++j)
                if (NEED[i][j] > AVAILABLE[j]){
                    flag = 1;
                    break;
                }

            if (flag == 0)
            {
                safety[count++]=i;
                finish[i] = 1;
                for (int j = 0; j < m; ++j)
                    AVAILABLE[j] += ALLOCATION[i][j];
                break;
            }
        }
    }
```

每次从第一个线程开始寻找，重复n次操作

不满足要求，则去寻找这个线程的下一个

满足要求，记录安全序列以及分配状态，并且释放资源

回到下一个k循环，从第一个线程开始寻找

main 函数：

```
if (n != bankman())
    cout << "No" << endl;
else{
    cout << "Yes";
    for(int i = 0;i < n; ++i)
    {
        cout << " " << safety[i];
    }
    cout << endl;
}
```

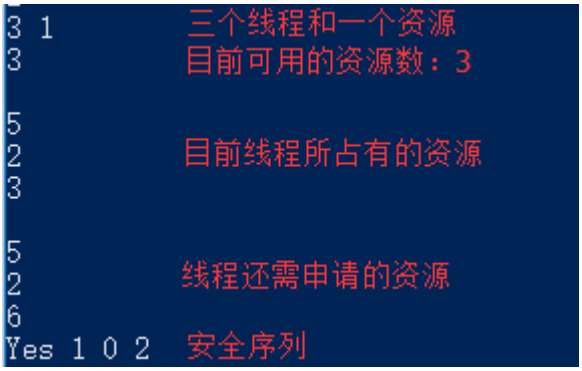
3. 实验结果

1986		smie15352116	1000	C++	Accepted	0sec	312 KB	1774 Bytes	2017-06-07 22:06:26

- 安全输入：

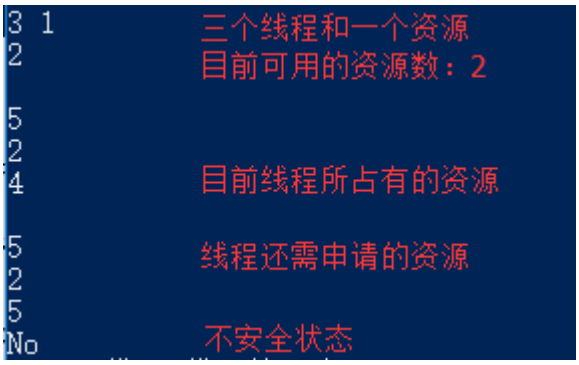
考虑一个系统，一共有 1 种资源和 3 个线程，此时还有 3 个空闲的资源，而目前 p0 占有 5 个资源，p1 占有 2 个资源，p2 占有 2 个资源；他们分别在申请

5，2，7 个资源。此时系统处于安全状态，顺序 p0->p1->p2 满足安全条件。这是因为 p1 立即得到资源并且释放资源，这是系统就会有 5 个资源，此时 p0 就可以得到资源并且释放，最后轮到 p2 的得到资源并且释放出所有的资源。



- 不安全的输入：

考虑一个系统，一共有 1 种资源和 3 个线程，此时还有 2 个空闲的资源，而目前 p0 占有 5 个资源，p1 占有 2 个资源，p2 占有 3 个资源；他们分别在申请 5，2，6 个资源。这时候系统处于不安全状态。这时只有 p1 可得到其所有的资源，当释放这些资源的时候，系统只有 4 个资源。所以 p0 和 p2 都进入了等待状态，导致了死锁。



- 思考：

对比上两个输入，有一点不同就是，对于线程 p2，在安全状态的时候他申请的资源数是 6，目前具有的资源数是 3；而在安全状态的时候他申请的资源数是 5，目前具有的资源数是 4。换句话可以说，在安全状态下他并没有按照

安全序列进行分配，而是先给了 p2 分配多一个资源，再进行分配，这时候就会导致了系统的不安全状态，最后导致死锁状态。从侧面看出了安全序列的重要性。

4.回答问题

- 1.没有安全序列，一定会导致死锁吗，为什么?

答：不一定。死锁的产生：由于对于一个申请资源的线程，该资源暂时不可以用进入了等待状态，如果该资源是被其他等待线程所占有，那么这个线程就有可能再也无法改变他现在的等待状态而导致死锁。可以说死锁是一种不安全的序列。以下分情况讨论：

a) 假设这个资源数很多，满足任何一个线程的需求的时候，在这种情况下是无论哪一个线程先运行都没关系，其实换一句话也可以说成，任何线程的组合都是一个安全序列。

b) 安全序列可以保证系统一直处于安全的状态。在不安全状态下，操作系统不能阻止进程以会导致死锁的方式申请资源，所以说不安全状态不一定会导致死锁。具体来说就是，线程不一定需要满足最大的资源数的时候才可以运行，当它不需要那么多的时候也可以实现用户的要求的时候，久可能降低死锁发生的概率。

c) 在不安全的状态下，如果线程忽然之间被中断，释放资源的时候，可用的资源数就会变多，而满足了其他线程的需求。这种情况下也会降低死锁发生的概率。

- 2. 银行家算法实用吗，为什么?

答：对于以前的操作系统来说是实用的，由于以前的计算机的进程都需要获得所有的资源的时候才可以启动，这个算法就可以保证进程不进入死锁状态。

不过对于目前的计算机来说可能没那么实用。第一，很多进程不需要一次性获得最大的资源需求数量，当它的需求没有这么大的时候，银行家算法也会对它分配最大的申请数量，就会过高地评估系统风险，可能会导致了由于申请的资源数太大，从而一直没有申请到资源，使得该线程一直处于等待状态的结果，线程响应时间变长，速度变慢，不满足用户需求。第二，银行家算法每次给进程分配资源的时候都会进行一次安全状态的检测，满足安全条件之后才会对该资源进行分配，这样子就会耗费很多时间。虽然稳定性提升了，可是时间也减慢了。如果不使用银行家算法，只是可能导致死锁发生。而且处理死锁的时间也不一定会比银行家算法实现的时间长。。。

- 3. 查阅并介绍现代操作系统是如何处理死锁的。

答：主要有四种：鸵鸟算法、预防算法、避免死锁、检测和解除死锁。

- ✧ 鸵鸟算法

形象地说就像鸵鸟一样忽略该问题。当死锁发生的概率小，而且预防死锁比解决死锁现象的成本（或时间）还要高（长），就不需要管这种情况。

- ✧ 检测和解除死锁

这种技术指的是，系统不去阻止死锁的产生，而是允许死锁产生，当检测到死锁发生之后再采取措施对其进行恢复。有点类似鸵鸟算法都不在乎死锁的产生，不同的是，这个方案实现了对死锁的检测。检测之后就会恢

复，而鸵鸟算法没有对死锁现象进行检测。死锁恢复大概有三种方法：

1. 抢占恢复：临时将某个资源进行转移，使得等待的线程可以运行，运行之后结束释放资源，再将资源转移回原来的所有者身上。（前提是，被转走资源的线程对该资源的需求不是很强烈）
2. 回滚恢复：类似游戏通关保存，下一关没过，就回到上一关的状态。这个方法主要是周期性对进程检测点检测（游戏保存点），一当检测到死锁，就回到上一个检测点。
3. 杀死进程恢复：就是字面上得意思，通过结束掉对该资源占有的一个线程，释放资源，进行死锁恢复。

✧ 避免死锁

这次我们的实验就是一种典型的避免死锁的算法，就是通过某种策略使得系统进入安全状态，避免死锁的发生。

✧ 预防死锁

预防和避免不同，死锁是一个病毒，预防是采取了打疫苗的方式使得病毒不能繁殖，而避免是选择了远离病毒源，病毒还具备感染能力。所以预防死锁，是打破了死锁产生了四个必要条件，让死锁不能发生。可以采取的措施是：破坏互斥条件；破坏占有和等待条件；破坏不可抢占条件；破坏环路等待条件。第一个和第三个直观上容易理解。第二个有两种情况，①在线程开始运行前，必须一次性获得请求的所有资源，否则不能运行；②当线程处于请求资源的状态的时候，不能占有资源，得把资源放回到锅里之后再从锅里一次性拿全自己需要的资源。

（参考了 yw8355507 的博客的“现代操作系统---死锁”，链接：

http://blog.csdn.net/yw8355507/article/details/50493525)

4. 实验感想

通过这次实验，回顾了老师上课讲到的银行家算法的相关内容，对于银行家的安全性算法有了更深刻的理解，同时了解了死锁的含义，以及死锁产生的四个必要条件，只要有一个条件不能实现的话就可以避免死锁的产生。这次实验比之前的实验要简单一点，主要是对课本内容的回顾以及应用。加深了对知识的印象。