操作系统实验三

银行家算法的实现

姓名：周嘉莹

学号：71118321

报告日期 2019/5/6

一、实验目的

通过实验，加深对多实例资源分配系统中死锁避免方法——银行家算法的理解，掌握Windows环境下银行家算法的实现方法，同时巩固利用Windows API进行共享数据互斥访问和多线程编程的方法。。

二、实验内容

1. 在 Windows 操作系统上，利用 Win32API 编写多线程应用程序实现银行家算法。
2. 创建 n 个线程来申请或释放资源，只有保证系统安全，才会批准资源申请。
3. 通过 Win32 API 提供的信号量机制，实现共享数据的并发访问。
4.实验环境：Windows XP + Virtualbox4.04+Ubuntu10.04

三、实验步骤

1.编写程序

```
#include <iostream>
#include <stdio.h>
#include <Windows.h>
#include <process.h>
#include<time.h>
using namespace std;
const int m = 3;//m 为每个进程需要的资源种类，3 个
const int n = 5;//n 为进程数量，5 个
HANDLE mutex1, mutex2;//互斥信号量，mutex1 用于进程请求资源，mutex2 用于执行
安全算法
int available[m] = { 3,3,2 };//available 为系统可用资源数
int allocation[n][m] = {//分配资源情况
    { 0,1,0 },
    { 2,0,0 },
    { 3,0,2 },
    { 2,1,1 },
    { 0,0,2 }
};
int maximum[n][m] = {//进程需要最大资源情况
    { 7,5,3 },
    { 3,2,2 },
    { 9,2,2 },
    { 2,2,2 },
    { 4,3,3 }
};
int need[n][m] = { 0 };//进程需要资源情况
bool finished[n] = { false };//安全算法中判断进程是否使用完资源
int request[n][m] = { 0 };//进程请求资源情况

/**
比较进程请求资源和当前可用资源，判断是否可以分配给该进程资源
**/
bool compare(int request_need[][m], int *available_work, int i) {
    for (int j = 0; j < m; j++) {
        if (request_need[i][j] > available_work[j])
            return false;
    }
    return true;
}

/**
比较进程需要资源和当前可用资源，判断是否可以分配给该进程资源
**/
bool compare(int request[][m], int need[][m], int i) {
    for (int j = 0; j < m; j++) {
        if (request[i][j] > need[i][j])
            return false;
    }
    return true;
}

/**
安全算法
**/
bool safe(int need[][m], int *available, int allocation[][m]) {
    int work[m] = { 0 };
    for (int i = 0; i < m; i++) {
        work[i] = available[i];
    }
    bool finish[n] = { false };
    for (int k = 0; k < n; k++)//产生安全序列
    {
        int i;
        for (i = 0; i < n; i++)//寻找所有符合条件的进程
        {
```

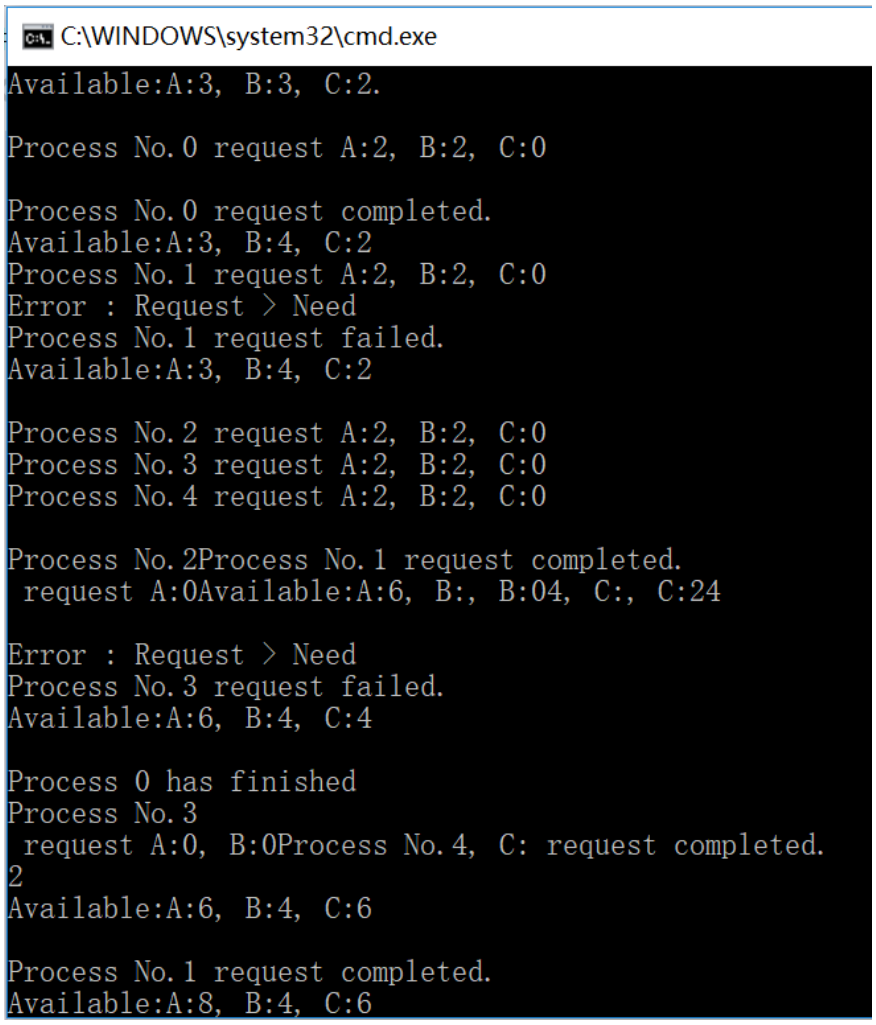
```cpp
            if (finish[i] == false && compare(need, work, i)) {
                for (int j = 0; j < m; j++)//该进程可以得到资源执行，执行完毕后释放资源
                {
                    work[j] = allocation[i][j] + work[j];
                    finish[i] = true;

                }
            }
        }
        i = 0;
    }
    cout << endl;
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (finish[i]) {
            count++;
        }
    }
    if (count == n)//所有进程都可以执行,系统安全
        return true;
    else//不安全
        return false;
}


unsigned int __stdcall process(LPVOID pM) {
    int j = *(int *)pM;
    srand(unsigned(time(0)));
    //j 为当前线程序号
    while (!finished[j]) {
        for (int i = 0; i < m; i++) {
            request[j][i] = rand() % 3;
        }
        WaitForSingleObject(mutex1, INFINITE);//进程请求资源
        cout << "Process No." << j << " request A:" << request[j][0]
            << ", B:" << request[j][1] << ", C:"
            << request[j][2] << endl;
        ReleaseSemaphore(mutex1, 1, NULL);

        WaitForSingleObject(mutex2, INFINITE);//更新三个矩阵
        if (compare(request, need, j)) {
            if (compare(request, available, j)) {
                for (int i = 0; i < m; i++) {
                    available[i] -= request[j][i];
                    allocation[j][i] += request[j][i];
                    need[j][i] -= request[j][i];
                }
                if (safe(need, available, allocation)) {//拟分配资源判断是
否安全
                    cout << "Process No." << j << " request completed." <<
endl;
                    finished[j] = true;
                    for (int i = 0; i < m; i++) {
                        available[i] += allocation[j][i];
                        allocation[j][i] = 0;
                        need[j][i] = 0;
                    }
                    cout << "Available:A:" << available[0]
                        << ", B:" << available[1]
                        << ", C:" << available[2] << endl;
                }
                else {//不安全取消拟分配的资源
                    for (int i = 0; i < m; i++) {
                        available[i] += request[j][i];
                        allocation[j][i] -= request[j][i];
                        need[j][i] += request[j][i];
                    }
                    cout << "Error : Not safe." << endl;
                    cout << "Process No." << j << " request failed." << endl;
                    cout << "Available:A:" << available[0]
                        << ", B:" << available[1]
                        << ", C:" << available[2] << endl;
                    cout << endl;
                    Sleep(1000);//1s 后继续发送请求
                }
            }
            else {
                cout << "Error : Request > Available" << endl;
                cout << "Process No." << j << " request failed." << endl;
                cout << "Available:A:" << available[0]
                    << ", B:" << available[1]
                    << ", C:" << available[2] << endl;
                Sleep(1000);//1s 后继续发送请求
            }
        }
        else {
            cout << "Error : Request > Need" << endl;
            cout << "Process No." << j << " request failed." << endl;
            cout << "Available:A:" << available[0]
                << ", B:" << available[1]
                << ", C:" << available[2] << "\n" << endl;
            Sleep(1000);//1s 后继续发送请求
        }
        ReleaseSemaphore(mutex2, 1, NULL);//释放 mutex2
    }
    //该线程 2s 后完成任务，释放资源
    Sleep(2000);
    cout << "Process " << j << " has finished\n";
    WaitForSingleObject(mutex2, INFINITE);//准备释放资源
    for (int i = 0; i < m; i++) {
        available[i] += allocation[j][i];
    }
    ReleaseSemaphore(mutex2, 1, NULL);
    return 0L;
}

int main(int argc, char *argv[])
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            need[i][j] = maximum[i][j] - allocation[i][j];
    cout << "Available:A:" << available[0]
        << ", B:" << available[1]
        << ", C:" << available[2] << ".\n" << endl;
    const int num = 10;
    mutex1 = CreateSemaphore(NULL, 1, 1, NULL);
    mutex2 = CreateSemaphore(NULL, 1, 1, NULL);
    HANDLE hThread[5];
    unsigned int a = 0, b = 1, c = 2, d = 3, e = 4;
    hThread[0] = (HANDLE)_beginthreadex(NULL, 0, process, &a, 0, NULL);
    Sleep(30);
    hThread[1] = (HANDLE)_beginthreadex(NULL, 0, process, &b, 0, NULL);
    Sleep(30);
    hThread[2] = (HANDLE)_beginthreadex(NULL, 0, process, &c, 0, NULL);
    Sleep(30);
    hThread[3] = (HANDLE)_beginthreadex(NULL, 0, process, &d, 0, NULL);
    Sleep(30);
    hThread[4] = (HANDLE)_beginthreadex(NULL, 0, process, &e, 0, NULL);
    Sleep(30);
    WaitForMultipleObjects(5, hThread, TRUE, INFINITE);//主线程等待所有生
产者和消费者运行完
    for (int i = 0; i < 5; i++)
```

```
        CloseHandle(hThread[i]);
    //销毁信号量
    CloseHandle(mutex1);
    CloseHandle(mutex2);
    return 0;
}
```

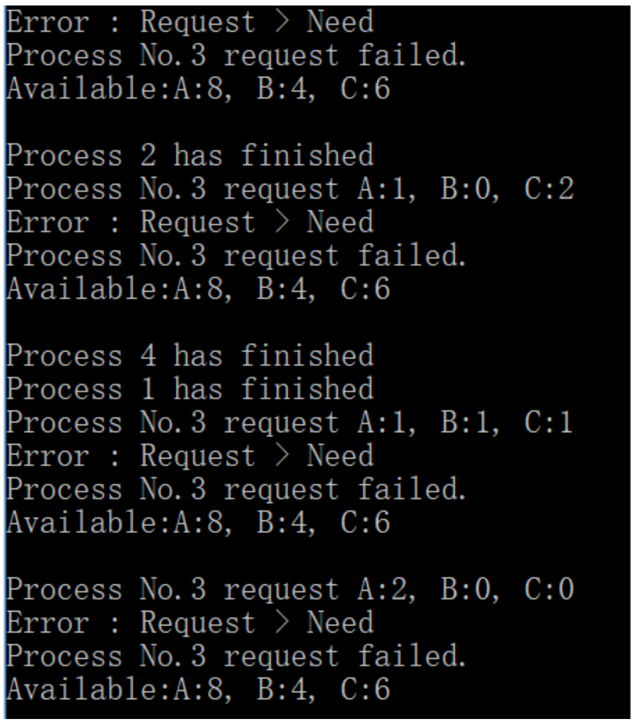
五、主要数据结构及其说明

1. 定义常整型变量 m，代表资源种类数。const int m=3;
2. 定义常整型变量 n，代表进程数。const int n=5;
3. 定义互斥信号量 mutex1,mutex2。其中 mutex1 用于进程请求资源，mutex2 用于执行安全算法
4. 定义可用资源向量 int available[m] = {3,3,2};
5. 定义已分配资源矩阵
   ```
   int allocation[n][m] = {//分配资源情况
       { 0,1,0 },
       { 2,0,0 },
       { 3,0,2 },
       { 2,1,1 },
       { 0,0,2 }
   ```
6. 定义最大值矩阵
   ```
   int maximum[n][m] = {//进程需要最大资源情况
       { 7,5,3 },
       { 3,2,2 },
       { 9,2,2 },
       { 2,2,2 },
       { 4,3,3 }
   ```
7. 定义需求矩阵，可在运行时计算得出
   ```
   int need[n][m] = { 0 };//进程需要资源情况
   ```
8. 定义finish向量，判断该进程是否使用完资源
   ```
   bool finished[n] = { false };//安全算法中判断进程是否使用完资源
   ```
9. 定义request向量，表示程序请求的资源
   ```
   int request[n][m] = { 0 };//进程请求资源情况
   ```

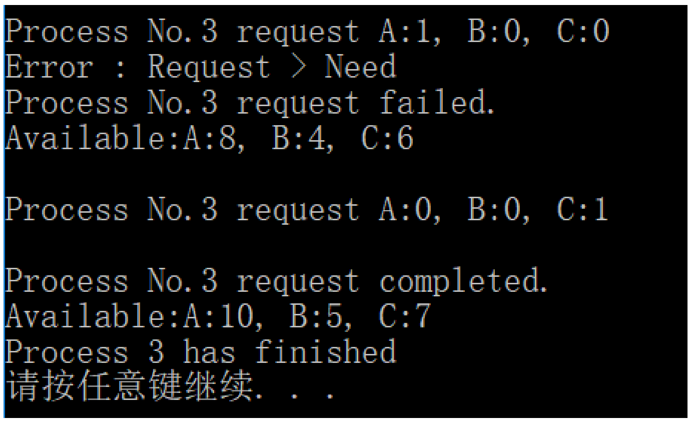
六、程序运行时的初值和运行结果



```
Available:A:3, B:3, C:2.

Process No.0 request A:2, B:2, C:0

Process No.0 request completed.
Available:A:3, B:4, C:2
Process No.1 request A:2, B:2, C:0
Error : Request > Need
Process No.1 request failed.
Available:A:3, B:4, C:2

Process No.2 request A:2, B:2, C:0
Process No.3 request A:2, B:2, C:0
Process No.4 request A:2, B:2, C:0

Process No.2Process No.1 request completed.
 request A:0Available:A:6, B:, B:04, C:, C:24

Error : Request > Need
Process No.3 request failed.
Available:A:6, B:4, C:4

Process 0 has finished
Process No.3
 request A:0, B:0Process No.4, C: request completed.
2
Available:A:6, B:4, C:6

Process No.1 request completed.
Available:A:8, B:4, C:6
```



```
Error : Request > Need
Process No.3 request failed.
Available:A:8, B:4, C:6

Process 2 has finished
Process No.3 request A:1, B:0, C:2
Error : Request > Need
Process No.3 request failed.
Available:A:8, B:4, C:6

Process 4 has finished
Process 1 has finished
Process No.3 request A:1, B:1, C:1
Error : Request > Need
Process No.3 request failed.
Available:A:8, B:4, C:6

Process No.3 request A:2, B:0, C:0
Error : Request > Need
Process No.3 request failed.
Available:A:8, B:4, C:6
```


```
Process No.3 request A:1, B:0, C:0
Error : Request > Need
Process No.3 request failed.
Available:A:8, B:4, C:6

Process No.3 request A:0, B:0, C:1

Process No.3 request completed.
Available:A:10, B:5, C:7
Process 3 has finished
请按任意键继续. . .
```

七、实验体会（实验中遇到的问题及解决方法、实验中产生的错误及原因分析、实验的体会及收获、对做好今后实验提出建设性建议等）

1. 问题：实验的的重难点是编写安全算法判断操作系统分配资源后是否能够处于安全状态。实现方法为：
   (1) 设置两个向量： 在执行安全算法开始时，Work ∶ =Available; Finish [i] := False;
   (2) 从进程集合中找到一个能满足下述条件的进程： ① Finish ［i］ = False; ② Need ［i,j］ ≤Work ［j］；
   若找到， 执行步骤(3)， 否则，执行步骤(4)。
   (3) (3) 当进程 Pi 获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行： Work ［j］ ∶ = Work ［i］ + Allocation ［i, j］；Finish ［i］ ∶ = true; go to step 2;
   (4) (4) 如果所有进程的 Finish ［i］ =true 都满足， 则表示系统处于安全状态；否则，系统处于不安全状态。
2. 问题：win32 向子线程传递参数时，传递的是参数的引用，而不是参数的值，所以每个线程的编号都要申请一个新的变量，而不能使用 for 循环直接向子线程传值。

3. 收获：通过本次实验学会了银行家算法的代码实现，对银行家算法有了更加清楚的认识。同时学会了使用 win32 多线程的方法以及向子线程传递参数的方法

八、源程序并附上注释

```
#include <iostream>
#include <stdio.h>
#include <Windows.h>
#include <process.h>
```

```
#include<time.h>
using namespace std;
const int m = 3;//m 为每个进程需要的资源种类，3 个
const int n = 5;//n 为进程数量，5 个
HANDLE mutex1, mutex2;//互斥信号量，mutex1 用于进程请求资源，mutex2 用于执行
安全算法
int available[m] = { 3,3,2 };//available 为系统可用资源数
int allocation[n][m] = {//分配资源情况
    { 0,1,0 },
    { 2,0,0 },
    { 3,0,2 },
    { 2,1,1 },
    { 0,0,2 }
};
int maximum[n][m] = {//进程需要最大资源情况
    { 7,5,3 },
    { 3,2,2 },
    { 9,2,2 },
    { 2,2,2 },
    { 4,3,3 }
};
int need[n][m] = { 0 };//进程需要资源情况
bool finished[n] = { false };//安全算法中判断进程是否使用完资源
int request[n][m] = { 0 };//进程请求资源情况

/**
比较进程请求资源和当前可用资源，判断是否可以分配给该进程资源
**/
bool compare(int request_need[][m], int *available_work, int i) {
    for (int j = 0; j < m; j++) {
        if (request_need[i][j] > available_work[j])
            return false;
    }
    return true;
}

/**
比较进程需要资源和当前可用资源，判断是否可以分配给该进程资源
**/
bool compare(int request[][m], int need[][m], int i) {
    for (int j = 0; j < m; j++) {
        if (request[i][j] > need[i][j])
            return false;
    }
```

```cpp
    return true;
}

/**
安全算法
**/
bool safe(int need[][m], int *available, int allocation[][m]) {
    int work[m] = { 0 };
    for (int i = 0; i < m; i++) {
        work[i] = available[i];
    }
    bool finish[n] = { false };
    for (int k = 0; k < n; k++)//产生安全序列
    {
        int i;
        for (i = 0; i < n; i++)//寻找所有符合条件的进程
        {
            if (finish[i] == false && compare(need, work, i)) {
                for (int j = 0; j < m; j++)//该进程可以得到资源执行，执行完毕后释放资源
                {
                    work[j] = allocation[i][j] + work[j];
                    finish[i] = true;

                }
            }
        }
        i = 0;
    }
    cout << endl;
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (finish[i]) {
            count++;
        }
    }
    if (count == n)//所有进程都可以执行,系统安全
        return true;
    else//不安全
        return false;
}


unsigned int __stdcall process(LPVOID pM) {
    int j = *(int *)pM;
    srand(unsigned(time(0)));
    //j 为当前线程序号
    while (!finished[j]) {
        for (int i = 0; i < m; i++) {
            request[j][i] = rand() % 3;
        }
        WaitForSingleObject(mutex1, INFINITE);//进程请求资源
        cout << "Process No." << j << " request A:" << request[j][0]
            << ", B:" << request[j][1] << ", C:"
            << request[j][2] << endl;
        ReleaseSemaphore(mutex1, 1, NULL);

        WaitForSingleObject(mutex2, INFINITE);//更新三个矩阵
        if (compare(request, need, j)) {
            if (compare(request, available, j)) {
                for (int i = 0; i < m; i++) {
                    available[i] -= request[j][i];
                    allocation[j][i] += request[j][i];
                    need[j][i] -= request[j][i];
                }
                if (safe(need, available, allocation)) {//拟分配资源判断是
否安全
                    cout << "Process No." << j << " request completed." <<
endl;
                    finished[j] = true;
                    for (int i = 0; i < m; i++) {
                        available[i] += allocation[j][i];
                        allocation[j][i] = 0;
                        need[j][i] = 0;
                    }
                    cout << "Available:A:" << available[0]
                        << ", B:" << available[1]
                        << ", C:" << available[2] << endl;
                }
                else {//不安全取消拟分配的资源
                    for (int i = 0; i < m; i++) {
                        available[i] += request[j][i];
                        allocation[j][i] -= request[j][i];
                        need[j][i] += request[j][i];
                    }
                    cout << "Error : Not safe." << endl;
                    cout << "Process No." << j << " request failed." << endl;
                    cout << "Available:A:" << available[0]
                        << ", B:" << available[1]
                        << ", C:" << available[2] << endl;
                    cout << endl;
                    Sleep(1000);//1s 后继续发送请求
                }
            }
            else {
                cout << "Error : Request > Available" << endl;
                cout << "Process No." << j << " request failed." << endl;
                cout << "Available:A:" << available[0]
                    << ", B:" << available[1]
                    << ", C:" << available[2] << endl;
                Sleep(1000);//1s 后继续发送请求
            }
        }
        else {
            cout << "Error : Request > Need" << endl;
            cout << "Process No." << j << " request failed." << endl;
            cout << "Available:A:" << available[0]
                << ", B:" << available[1]
                << ", C:" << available[2] << "\n" << endl;
            Sleep(1000);//1s 后继续发送请求
        }
        ReleaseSemaphore(mutex2, 1, NULL);//释放 mutex2
    }
    //该线程 2s 后完成任务，释放资源
    Sleep(2000);
    cout << "Process " << j << " has finished\n";
    WaitForSingleObject(mutex2, INFINITE);//准备释放资源
    for (int i = 0; i < m; i++) {
        available[i] += allocation[j][i];
    }
    ReleaseSemaphore(mutex2, 1, NULL);
    return 0L;
}

int main(int argc, char *argv[])
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            need[i][j] = maximum[i][j] - allocation[i][j];
    cout << "Available:A:" << available[0]
        << ", B:" << available[1]
        << ", C:" << available[2] << ".\n" << endl;
    const int num = 10;
    mutex1 = CreateSemaphore(NULL, 1, 1, NULL);
    mutex2 = CreateSemaphore(NULL, 1, 1, NULL);
    HANDLE hThread[5];
    unsigned int a = 0, b = 1, c = 2, d = 3, e = 4;
    hThread[0] = (HANDLE)_beginthreadex(NULL, 0, process, &a, 0, NULL);
    Sleep(30);
    hThread[1] = (HANDLE)_beginthreadex(NULL, 0, process, &b, 0, NULL);
    Sleep(30);
    hThread[2] = (HANDLE)_beginthreadex(NULL, 0, process, &c, 0, NULL);
    Sleep(30);
    hThread[3] = (HANDLE)_beginthreadex(NULL, 0, process, &d, 0, NULL);
    Sleep(30);
    hThread[4] = (HANDLE)_beginthreadex(NULL, 0, process, &e, 0, NULL);
    Sleep(30);
    WaitForMultipleObjects(5, hThread, TRUE, INFINITE);//主线程等待所有生
产者和消费者运行完
    for (int i = 0; i < 5; i++)
        CloseHandle(hThread[i]);
    //销毁信号量
    CloseHandle(mutex1);
    CloseHandle(mutex2);
    return 0;
}
```