# 实验二 死锁避免的模拟

## 实验目的

1. 深入了解死锁的原因和必要条件。
2. 掌握死锁的处理方式。
3. 实现死锁避免的模拟

## 实验预备知识

1. 系统安全状态
2. 银行家算法

## 实验内容

验证银行家算法避免系统死锁。

假设系统中有三类资源A、B、C和五个进程P0，p1,P2、P3、P4。然后设定每种资源的数量。之后设定每个进程对各类资源的最大需求，最后假定在某一时刻，系统已经给各个进程分配多少资源。要求检查该时刻系统是否处于安全状态。

## 实验要求

给出程序流程和源程序（附有详细注释）

/\*

实验：实验二 死锁避免的模拟

语言：c++

\*/

#include<iostream>

using namespace std;

#include<string>

//进程

struct P{

string name;

char status;//状态

int max[3];

int allocation[3];

int need[3];

int Request[3];

};

//资源

//struct resources{

// string name;

// int available;

//};

//银行家算法中的数据结构

int Available[3]={3,3,2};//可利用资源向量

int Max[5][3]={//最大需求矩阵

{7,5,3},

{3,2,2},

{9,0,2},

{2,2,2},

{4,3,3},

};

int Allocation[5][3]={//分配矩阵

{0,1,0},

{2,0,0},

{3,0,2},

{2,1,1},

{0,0,2},

};

int Need[5][3]={//需求矩阵

{7,4,3},

{1,2,2},

{6,0,0},

{0,1,1},

{4,3,1},

};

int SecurityAlgorithm(P \*p,int n);

//比较

int compare(int\*p,int \*q){

int n=0;

for(int i=0;i<=2;i++){

if(p[i]<=q[i]){

n++;

}

}

if(n==3){

return 1;

}

else{

return 0;

}

}

//资源相加

void addResources(int \*p,int\*q){

for(int i=0;i<3;i++){

q[i]+=p[i];

}

}

//相减

void subtractor(int\*p,int\*q){

for(int i=0;i<3;i++){

q[i]-=p[i];

}

}

//赋值

void func(P \*p,P \*q){

for(int i=0;i<5;i++){

p[i]=q[i];

}

}

//银行家算法

void bankAlgorithm(P \*pn,P \*p,int n){

if(compare((\*p).Request,(\*p).need)){

if(compare((\*p).Request,Available)){

subtractor((\*p).Request,Available);

addResources((\*p).Request,(\*p).allocation);

subtractor((\*p).Request,(\*p).need);

P pn1[5];

func(pn1,pn);

if(SecurityAlgorithm(pn1,n)){

cout<<"完成分配"<<endl;

return;

}

else{

cout<<"试探分配作废"<<endl;

addResources((\*p).Request,Available);

subtractor((\*p).Request,(\*p).allocation);

addResources((\*p).Request,(\*p).need);

return;

}

}

else{

cout<<"没有足够的资源"<<endl;

return;

}

}

else{

cout<<"出错，所需要的资源数超过它所宣布的最大值"<<endl;

return;

}

}

//安全性算法

int SecurityAlgorithm(P p[5],int n){

int work[3];

for(int i=0;i<=2;i++){

work[i]=Available[i];

}

for(int i=0;i<n;i++){

//cout<<"进入";

for(int j=0;j<n;j++){

//cout<<p[j].status<<endl;cout<<compare(p[j].need,work)+1<<endl;

if(p[j].status=='0'&& compare(p[j].need,work)){

addResources(p[j].allocation,work);

p[j].status='1';

cout<<p[j].name<<endl;

cout<<work[0]<<" " <<work[1]<<" "<<work[2]<<endl;

break;

}

}

}

int statusnum=0;

for(int i=0;i<n;i++){

if(p[i].status=='1'){

statusnum++;

}

}

if(statusnum==n){

cout<<"当前系统处于安全状态"<<endl;

return 1;

}

cout<<"不安全"<<endl;

return 0;

}

int main(void){

cout<<"------------------------"<<endl;

cout<<"所有权属于：聂良鹏"<<endl;

cout<<"------------------------"<<endl;

P p[5];

for(int i=0;i<5;i++){

if(i==0){

p[i].name="p0";

}

if(i==1){

p[i].name="p1";

}

if(i==2){

p[i].name="p2";

}

if(i==3){

p[i].name="p3";

}

if(i==4){

p[i].name="p4";

}

p[i].status='0';

for(int j=0;j<3;j++){

p[i].max[j]=Max[i][j];

p[i].allocation[j]=Allocation[i][j];

p[i].need[j]=Need[i][j];

}

}

P q[5];

func(q,p);

SecurityAlgorithm(q,5);

//cout<<Available[2]<<endl;

cout<<"输入p1的请求向量：";

cin>>p[1].Request[0];

cin>>p[1].Request[1];

cin>>p[1].Request[2];

bankAlgorithm(p,&p[1],5);

cout<<"输入p4的请求向量：";

cin>>p[4].Request[0];

cin>>p[4].Request[1];

cin>>p[4].Request[2];

bankAlgorithm(p,&p[4],5);

cout<<"输入p0的请求向量：";

cin>>p[0].Request[0];

cin>>p[0].Request[1];

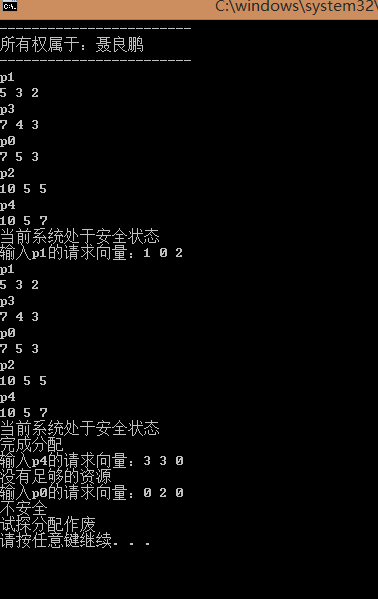
cin>>p[0].Request[2];

bankAlgorithm(p,&p[0],5);

return 0;

}

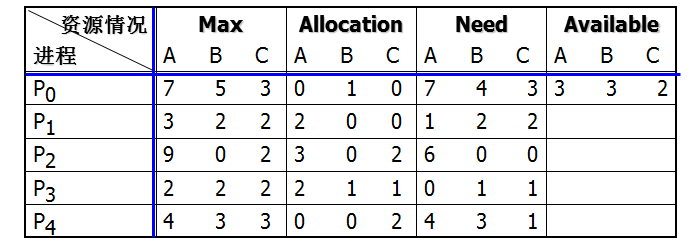
程序运行截图

1. 
2. 收获、体会及对该实验的改进意见和见解

从代码层面更加对银行家算法有了深刻的领悟，银行家算法虽然理解起来很简单，但是写到代码中还是出现了一些逻辑上的问题，比如，在执行安全性算法的时候不应该去真正修改状态属性，如果直接修改，下一次执行安全性算法就直接都是为TRUE了，导致错误；花了好长的时间写出来代码，银行家算法算是可以说掌握的比较好了；但是这里的程序还可以再优化，可以设置循环，设置多个进程的不同的请求向量。。。。；

## 实验提示

由于是实验，没有真正的避免死锁。所以在实验中首先假定一种系统状态，假设T0时刻资源的分配情况如下所示：



要求实现：

（1）确定系统在T0时刻的安全性。

（2）P1发出资源请求向量Request1(1,0,2)，按照银行家算法确定能否将资源分配给P1。

（3）在（2）的基础上， P4发出请求向量Request4(3,3,0)，按照银行家算法确定能否将资源分配给P1。

（4）再（3）的基础上，P0发出请求向量Request0(0,2,0)，按照银行家算法确定能否将资源分配给P1。