### 实验4、银行家算法模拟

1. 实验目的和要求

银行家算法是由Dijkstra设计的最具有代表性的避免死锁的算法。本实验要求用高级语言编写一个银行家的模拟算法。通过本实验可以对预防死锁和银行家算法有更深刻的认识。

三、实验内容：

* 1. 设置数据结构

包括可利用资源向量（Availiable）,最大需求矩阵（Max）,分配矩阵（Allocation）,需求矩阵（Need）

* 1. 设计安全性算法

设置工作向量Work 表示系统可提供进程继续运行可利用资源数目，Finish 表示系统是否有足够的资源分配给进程

四、测试数据：

要求对于书113页给出的资源分配表，可以对输入各种请求进行安全性判断，最后给出安全序列或者不能分配的原因。如输出：

"所需资源全部满足，此进程运行完毕！";

"申请的资源量大于系统可提供的资源量！"

"申请的资源中有某种资源大于其声明的需求量！"

"若满足申请，系统将进入不安全状态，可能导致死锁！"

#include<iostream>

#include<string>

using namespace std;

#define False 0

#define True 1

int Avaliable[100] = { 0 }; //系统可利用资源

int Max[100][100] = { 0 }; //所有进程对资源的最大需求

int Allocation[100][100] = { 0 }; //系统已分配资源

int Need[100][100] = { 0 };//所有进程还需要的资源

int Request[100] = { 0 };//进程的请求资源向量

int temp[100] = { 0 };//存放安全序列

int Work[100] = { 0 };//存放系统可提供给进程继续运行所需的各类资源

int M = 100;//作业的最大数为100

int N = 100;//资源的最大数为100

char name[100] = { 0 }; //资源的名称

void showdata()//显示资源矩阵

{

int i,j;

cout<<"系统目前可利用的资源[Avaliable]:"<<endl;

for (i = 0; i<N; i++)

cout<<name[i]<<endl;

for (j = 0; j<N; j++)

cout<<Avaliable[j]<<endl;//输出分配资源

cout<<"Max Allocation Need "<<endl;

cout<<"进程名";

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

{

for (i = 0; i<N; i++)

cout<<name[i];

cout<<" ";

}

cout<<endl;

for (i = 0; i<M; i++)

{

cout<< i;

for (j = 0; j<N; j++)

cout<<Max[i][j];

cout<<" ";

for (j = 0; j<N; j++)

cout<<Allocation[i][j];

cout<<" ";

for (j = 0; j<N; j++)

cout<<Need[i][j];

cout<<endl;

}

}

int allocatedata(int i)//进行资源分配

{

int j;

for (j = 0; j<M; j++) {

Avaliable[j] = Avaliable[j] - Request[j];

Allocation[i][j] = Allocation[i][j] + Request[j];

Need[i][j] = Need[i][j] - Request[j];

}

return 1;

}

int safe()//安全性算法

{

int i,j,k = 0, m, apply=0, Finish[100] = { 0 };

int flag = 0;

Work[0] = Avaliable[0];

Work[1] = Avaliable[1];

Work[2] = Avaliable[2];

for (i = 0; i<M; i++)

{

// apply = 0;

for (j = 0; j<N; j++)

{

if (Finish[i] == False&&Need[i][j] <= Work[j])

{

apply++;

if (apply == N)

{

for (m = 0; m<N; m++)

Work[m] = Work[m] + Allocation[i][m];

Finish[i] = True;

temp[k] = i;

i = -1;

k++;

flag++;

}

}

}

}

for (i = 0; i<M; i++) {

if (Finish[i] == False) {

printf("系统不安全\n");//不成功系统不安全

return -1;

}

}

printf("系统是安全的!\n");//如果安全，输出成功

printf("分配的序列:");

for (i = 0; i<M; i++) {//输出运行进程数组

printf("%d", temp[i]);

if (i<M - 1) printf("->");

}

cout<<endl;

return 0;

}

void share()//利用银行家算法对申请资源对进行判定

{

char ch;

int i = 0, j = 0;

ch = 'y';

cout<<"请输入要求分配的资源进程号0-:"<< M - 1;

cin>>i;//输入须申请的资源号

cout<<"请输入进程申请的资源:"<<i<<endl;

for (j = 0; j<N; j++)

{

cout<<name[j];

cin>>Request[j];//输入需要申请的资源

}

for (j = 0; j<N; j++) {

if (Request[j]>Need[i][j])//判断申请是否大于需求，若大于则出错

{

printf("进程 %d申请的资源大于它需要的资源, 分配不合理，不予分配！\n", i);

ch = 'n';

break;

}

else

{

if (Request[j]>Avaliable[j])//判断申请是否大于当前资源，若大于则出错

{

printf("进程%d申请的资源大于系统现在可利用的资源\n", i);

printf(" 分配出错，不予分配!\n");

ch = 'n';

break;

}

}

}

if (ch == 'y') {

allocatedata(i);//根据进程需求量变换资源

showdata();//根据进程需求量显示变换后的资源

safe();//根据进程需求量进行银行家算法判断

}

}

int main()//主函数

{

int i, j, q, choice, m, n, flag;

char ming;

printf("请首先输入系统可供资源种类的数量:");

scanf("%d", &n);

N = n;

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

{

printf("资源%d的名称:", i + 1);

scanf("%s", &ming);

name[i] = ming;

printf("资源%d的数量:", i + 1);

scanf("%d", &q);

Avaliable[i] = q;

}

// printf("\n");

printf("请输入作业的数量:");

scanf("%d", &m);

M = m;

printf("请输入各进程的最大需求量%d\*%d矩阵[Max]:\n", m, n);

for (i = 0; i<m; i++)

{

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

scanf("%d", &Max[i][j]);

//printf("\n");

}

do {

flag = 0;

printf("请输入各进程已经申请的资源量(%d\*%d矩阵)[Allocation]:\n", m, n);

for (i = 0; i<m; i++)

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

{

scanf("%d", &Allocation[i][j]);

if (Allocation[i][j]>Max[i][j]) flag = 1;

Need[i][j] = Max[i][j] - Allocation[i][j];

}

if (flag)

printf("申请的资源大于最大需求量，请重新输入!\n");

} while (flag);

showdata();//显示各种资源

safe();//用银行家算法判定系统是否安全

while (choice)

{

printf("\n");

printf(" 0:离开 \n");

printf(" 1:分配资源 \n");

printf("请选择功能号：");

scanf("%d", &choice);

scanf("%d", &choice);

switch (choice)

{

case 1: share(); break;

case 0: break;

default: printf("请正确选择功能号(0-1)!\n"); break;

}

}

return 1;

}

### 实验5、可变分区分配和回收算法模拟

1. 实验目的和要求

模拟实现一个简单的可变分区存储管理系统。通过本实验可以对可变分区的分配和回收算法有更深刻的认识。

三、实验内容：

（1） 建立相关的数据结构

（2） 实现一个分区分配算法，如最先适应分配算法、最优或最坏适应分配算法

（3） 实现一个分区回收算法

（4） 给定一批作业/进程，选择一个分配或回收算法，实现分区存储的模拟管理。

四、测试数据：

空白区号 起始地址 空白区容量 状态

1 5KB 100KB 可用

2 120KB 20KB 可用

3 310KB 256KB 可用

4 1024KB 48KB 可用

现有5个作业J1、J2、J3、J4、J5，它们分别需要内存20KB、42KB、120KB、130KB、18KB的空间请问：

（1）如果采用最先适应分配算法能把这五个作业按Ja～Je的次序全部装入主存吗?

（2）用什么分配算法装入这五个作业可使主存的利用率最高?

（3）并给出装入后的空白区链。当回收顺序为J5、J3、J1、J2、J4，给出每次回收后的空闲区链。

#include<iostream.h>

cout<<"\*\*\*\*\*\*\*\*\*看在你帮我们救灾的份上，帮你搞了一份！\*\*\*\*\*\*\*\*\*"<<endl;

typedef struct

{

int No;

int size;

int start;

int state;//定义分区状态；

}F;

int fun\_allocate(int x,int \*num,F \*arr)//首次适应FF 分区算法，分配成功返回分区数

{

int i=0,p;

F t={0,0,0,0};

for(i=0;i<\*num;i++)

{//找到满足的分区，得到其分区号。

if(arr[i].size>=x&&arr[i].state==0)

break;

}

if(i==\*num)

return 0;

p=i;//记录分区号

t.No=arr[i].No+1;

t.size=arr[i].size-x;

t.start=arr[i].start+x;

t.state=0;

arr[i].size=x;

arr[i].state=1;

if(t.size!=0)

{//将分割后的分区后移

for(i=\*num;i>p;i--)

{

arr[i]=arr[i-1];

arr[i].No=arr[i].No+1;

}

arr[p+1]=t;

\*num=\*num+1;

}

return \*num;//返回分区数

}

void fun\_print(F \*arr,int num)

{//打印分区结果

int i=0;

cout<<"编号"<<" "<<"起始"<<" "<<"大小"<<" "<<"状态"<<endl;

for(i=0;i<num;i++)

{

cout<<arr[i].No<<" "<<arr[i].start<<" "<<arr[i].size<<" "<<arr[i].state<<endl;

}

}

void fun\_huishou(int No,int \*num ,F \*arr)

{

int i=0,p=0;

for(i=0;i<\*num;i++)

{//查找回收位置

if(arr[i].size==No)

{

arr[i].state=0;

}

}

for(i=0;i<\*num;i++)

{

if(arr[i].start+arr[i].size==arr[i+1].start&&arr[i].state==0&&arr[i+1].state==0)

{

p=i;

arr[p+1].size=arr[p+1].size+arr[p].size;

for(i=p;i<\*num;i++)

{

arr[i]=arr[i+1];

arr[i].No=arr[i].No-1;

}

\*num-=1;

}

}

}

int main()

{

F arr[50]={{1,100,5,0},{2,20,120,0},{3,256,310,0},{4,48,1024,0}};

int num=4;//统计分区数；

int i=0,x,t;

cout<<"初始分区情况,0表示分区可用，1表示分区已经被分配"<<endl;

cout<<"编号"<<" "<<"起始"<<" "<<"大小"<<" "<<"状态"<<endl;

for(i=0;i<4;i++)

{

cout<<arr[i].No<<" "<<arr[i].start<<" "<<arr[i].size<<" "<<arr[i].state<<endl;

}

cout<<"输入请求大小以（0）结束"<<endl;

while(cin>>x&&x!=0)

{

t=fun\_allocate(x,&num,arr);

if(t==0)

cout<<"分配失败"<<endl;

else

{

num=t;

fun\_print(arr,num);

}

}

cout<<"回收资源"<<endl;

cout<<"输入回收大小以（0）结束"<<endl;

while(cin>>x&&x!=0)

{

fun\_huishou(x,&num,arr);

fun\_print(arr,num);

}

return 1;

}

### 实验6、时间片轮转调度算法模拟

1. 实验目的

设计一个按时间片调度算法实现处理器调度的进程。通过本实验可以对时间片轮转算法有更深刻的认识。

二、实验内容：

1. 假定系统有五个进程，每一个进程用一个进程控制块PCB来代表。进程控制块的格式为：

|  |
| --- |
| 进程名 |
| 指针 |
| 要求运行时间 |
| 已运行时间 |
| 状态 |

其中，假设五个进程的进程名分别是Q1，Q2，Q3，Q4，Q5。

进程按顺序排成循环队列，用指针指出下一个进程的进程控制块首地址，最后一个进程中的指针指出第一个进程的进程控制块首地址。

要求运行时间----假设进程需要运行的单位时间数。

已运行时间----假设进程已经运行的单位时间数，初始值为“0”。

状态----有两种状态，“就绪”状态和“结束”状态，初始状态都为“就绪”，用“R”表示，当一个进程运行结束后，它的状态变为“结束”，用“E”表示。

1. 每次运行你所设计的处理器调度程序之前，为每个进程任意确定它的“要求运行时间”。
2. 用一标志单元记录轮到运行的进程。处理器调度总是选择标志单元指示的进程运行。由于本实验是模拟处理器调度的功能，所以，对被选中的进程并不实际启动运行，而是执行：已运行时间－1。来模拟进程的一次运行，表示进程已经运行过一个单位的时间。

请注意：在实际的系统中，当一个进程被选中运行时，必须置上该进程可以运行的时间片值，以及恢复进程的现场，让它占有处理器运行，直到出现等待事件或运行满一个时间片。在这里省去了这些工作，仅用“已运行时间+1”来表示进程已经运行满一个时间片。

1. 进程运行一次后，应把该进程的进程控制块中的指针值送到标志单元，以指示下一个轮到运行的进程。同时，应判断该进程的要求运行时间与已运行时间，若该进程要求运行时间≠已运行时间，则表示它尚未执行结束，应待到下一轮时再运行。若该进程的要求运行时间=已运行时间，则表示它已经执行结束，应把它的状态修改为“结束”（E）且退出队列。此时，应把该进程的进程控制块中的指针值送到前面一个进程的指针位置。
2. 若“就绪”状态的进程队列不为空，则重复上面（4）和（5）的步骤，直到所有进程都成为“结束”状态。

三、测试：

输入：为五个进程确定名称和任意一组“要求运行时间”

输出：每次被选中进程的进程名以及运行一次后进程队列的变化。

测试数据：课本p94例