实验一、单处理机下进程调度

程序中使用的数据结构及符号说明

1. 进程类Proc，包含进程的属性，进程赋值函数和进程比较函数

```
class Proc
{
public:
    int pid, ariveTime, runTime, priority, timeSlice;
    void setProc(int tpid, int tar, int trun, int tpri, int ttime);
    Proc()
    {
        pid = ariveTime = runTime = priority = timeSlice = 0;
    }
    bool equl(Proc b);//判断两个进程是否相同
};
```

2. 控制器类Manage，全局使用唯一一个控制器实例，用于实现进程调度功能。

   控制器类中包含指定数量的最大数量的Proc数组，单次用例最大运行时间mtime，当前时间nowtime，每一批中进程数cont以及五种进程调度算法。

```
class Manage
{
public:
    Proc pro[m4x];//暂定的最大的进程数
    int nowtime, cont, mtime;
    void FIFS();                //先到先服务算法
    void shortFirst();          //不可剥夺式短作业优先算法
    void leastTime();           //可剥夺式最短剩余时间优先算法
    void RR();                  //时间片轮转法
    void changePri();           //动态优先级调度算法
    int chooseMethod(int choice, int cont, int maxtime);
};
```

3. 宏定义：
   - 指定支持的最大并行进程数量m4x
   - 设定无限大整数unlimit

```
#define m4x 1000
#define unlimit (int)1e9
```

各调度算法的流程及重要模块功能接口说明

1. 先来先服务调度算法

FIFS调度算法最为简单直接，按照到达时间排序之后顺序输出即可。

```
void Manage::FIFS()
{
    stable_sort(pro, pro + cont, compArr);
    nowtime = pro[0].ariveTime;
    for (int i = 0; i < cont; i++)
    {
        nowtime = nowtime < pro[i].ariveTime ? pro[i].ariveTime : nowtime;
        cout << i + 1 << "/" << pro[i].pid << "/" << nowtime << "/";
        nowtime += pro[i].runTime;
        cout << nowtime << "/" << pro[i].priority<<endl;
    }
}
```

2. 短作业优先调度算法

先将所有进程按照到达时间排序之后，创建Proc runlist[m4x]进程数组,表示当下在等待中的进程。

*运行时间*小于总运行时间mtime时，进行循环查找到达时间小于等于当前时间的进程，加入runlist中等待调度。

对runlist中的进程按照运行时长排序后取最短的进程，即runlist[0]进行调度，进程调度结束之后将其运行时间置为unlimit，以保证在runlist中排序时永远处于未被调度的进程之后， 并及时各时间参数。

```
while (nowtime>= pro[i].ariveTime&&i<cont)
{
    runlist[i] = pro[i++];
    if (i >= cont)
        break;
}
stable_sort(runlist, runlist + i, compShort);
//取排最前面的一个运行，结束之后将其运行时间置位unlimited
if (runlist[0].runTime == unlimit)
{
    nowtime++;
    continue;
}
cout << ++runNum << "/" << runlist[0].pid << "/" << nowtime << "/";
```

3. 最短剩余时间优先调度算法

在短作业优先的基础上增加了剥夺机制。在总运行时间未完成之前一直进行**每一秒**的循环。

使用Proc runing表示当前正在进行的任务，每一个秒循环中都检测是否有新的进程加入runlist数组中，并按照剩余时间进行排序，取runlist[0]作为被调度进程，需要判断本次被调度进程是否为上轮调度正在运行的进程。当本次被调度进程结束时，将该进程剩余时间置为unlimit，runing.pid置为0，代表当前没有正在运行的程序。

```
stable_sort(runlist, runlist + i, compLeast);
if (runing.pid)//有正在运行的程序
{
    if (runing.equl(runlist[0]))
    {//继续运行上一秒的程序
        nowtime++;
        runing.runTime--;
        runlist[0].runTime--;
    }
    else//要运行新来的程序
    {
        cout << nowtime << "/" << runing.priority << endl;
        runing = runlist[0];
        runing.runTime--;
        runlist[0].runTime--;
        cout << ++runNum << "/" << runing.pid << "/" << nowtime << "/";
        nowtime++;
    }
    if (runing.runTime == 0)
    {
        cout << nowtime << "/" << runing.priority << endl;
        runing.pid = 0;
        runlist[0].runTime = unlimit;
```

4. 时间片轮转调度算法

使用一个队列表示在等待状态的进程，按照到达时间排序后，每个时间片循环都检测是否有新的进程加入队列。

每次取队头进程进行调度，若在时间片内执行完毕将running.pid置为0，并直接结束本次时间片。下一个时间片开始检测新到达进程加入队列后，若runing未运行完毕，则加入队列，否则抛弃。

```
while (nowtime >= pro[i].ariveTime&&i < cont)
    waitList.push(pro[i++]);
if (runing.pid)//有正在运行的程序;
{
    waitList.push(runing);
    runing = waitList.front();
    waitList.pop();
}
else
{
    if (waitList.empty())//当前队列空
    {
        nowtime++;
        continue;
    }
    //没有正在运行的程序，且队列非空，取队头元素运行
    runing = waitList.front();
    waitList.pop();
}
```

5. 动态优先级调度算法

每次进入新的时间片前，检测到达进程加入runlist，并根据其到达时间，对已经等待的进程优先级减一，随后根据优先级进行排序，取runlist[0]作为被调度程序，运行一个时间片。

```
sliceInTime = 0;//记录当前时间片已经运行的时间
while (nowtime >= pro[i].ariveTime&&i < cont)
    runList[i] = pro[i++];
for (j = 1; j < i; j++)
    if (runList[j].priority&&runList[j].priority != unlimit&&runList[j].ariveTime<nowtime)
        runList[j].priority--;
stable_sort(runList, runList + i, compPri);
if (runList[0].priority == unlimit)
{
    nowtime++;
    continue;
```

若进程在时间片内结束时，将runing.pid置0，runlist[0]的优先级置为unlimit，确保不会再被调度。否则将其优先级+3。

```
if (runing.runTime == 0)
{
    runing.pid = 0;
    runList[0].priority = unlimit;
    break;
}
```

源代码及注释

源代码行数较多，见文末附录

测试方法及分析

因为将整个算法调度分成了5个主要模块，在测试时可以分模块进行，分别将题目公开的测试用例测试相应的功能。

采用了**等价类划分**和**边界值法**对程序进行测试。

- 等价类划分：

  分别使用题目给出的测试用例和自己构造简单的测试用例，对程序进行测试，测试程序是否能够实现按照选择进行不同调度方案的基本功能。

  经过测试，程序能够根据选择基本正确的实现不同方案的进程调度。

- 边界值法：
  - 手动构造与支持的最大进程数相等数量的测试用例，检测临界数量能否正常执行。
  - 构造时间差别较大的用例，测试能够正常完成期望输出。
  - 经过测试，在进程间时间差别较大的时候，出现未执行完毕即退出的情况，经过调试，发现使用了nowtime与最大运行时间进行比较，当进程间出现空挡时，程序判断出错。

实验经验及体会

经验：

我是属于最早看到题目的一批人，所以提前很久开始了编码，整体来说算法并不难，基本没有思维障碍，可以直接进行模拟实现，但是因为一个没有注意到的小bug被卡了两天，即可能进程到来不是连续的，中间有可能出现空挡，nowtime需要推进而runtime不动，我想到了这个问题，加入了空挡处理的部分，但是忘了还有时间循环限制的问题，导致第九个用例一直无法通过。

体会：

在纸面上学的进程调度算法，只能算是了解，只有当真正的去编码实现这些过程，直接面对需要处理的各个细节问题，调试自己的设计中的一个个隐藏bug，才能真正体会到这些调度算法的巧妙，理解在计算机发展路上做出贡献的先辈的伟大。

附：程序源文件及注释

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <algorithm>
#include <queue>
#define m4x 1000
#define unlimit (int)1e9
#define ture true
using namespace std;

class Proc
{
public:
    int pid, ariveTime, runTime, priority, timeSlice;
    void setProc(int tpid, int tar, int trun, int tpri, int ttime);
    Proc()
    {
        pid = ariveTime = runTime = priority = timeSlice = 0;
    }
    bool equl(Proc b);//判断两个进程是否相同
};
bool Proc::equl(Proc b)
{//
    return pid == b.pid;
    //&& ariveTime == b.ariveTime&&runTime == b.runTime
    //  &&priority == b.priority&& timeSlice == b.timeSlice;
}
class Manage
{
public:
    Proc pro[m4x];//暂定的最大的进程数
    int nowtime, cont, mtime;
    void FIFS();              //先到先服务算法
    void shortFirst();        //不可剥夺式短作业优先算法
    void leastTime();         //可剥夺式最短剩余时间优先算法
    void RR();                //时间片轮转法
    void changePri();         //动态优先级调度算法
    int chooseMethod(int choice, int cont, int maxtime);
};
//根据到达时间排序
bool compArr(const Proc &a, const Proc &b)
{
    if (a.ariveTime != b.ariveTime)
        return a.ariveTime < b.ariveTime ? true : false;
    return a.pid < b.pid ? true : false;
    //return a.ariveTime < b.ariveTime ? true : false;
}
//根据剩余时间进行排序
```

```
bool compShort(const Proc &a, const Proc &b)
{
    if (a.runTime != b.runTime)
        return a.runTime < b.runTime ? true : false;
    return a.pid < b.pid ? true : false;
}
bool compLeast(const Proc &a, const Proc &b)
{
    if (a.runTime != b.runTime)
        return a.runTime < b.runTime ? true : false;
    if (a.ariveTime != b.ariveTime)
        return a.ariveTime < b.ariveTime ? true : false;
    return a.pid < b.pid ? true : false;

}
bool compPri(const Proc &a, const Proc &b)
{
    if (a.priority != b.priority)
        return a.priority<b.priority?true:false;
    if (a.ariveTime != b.ariveTime)
        return a.ariveTime < b.ariveTime ? true : false;
    return a.pid < b.pid ? ture : false;
}
void Proc::setProc(int tpid, int tar, int trun, int tpri, int ttime)
{
    pid = tpid;
    ariveTime = tar;
    runTime = trun;
    priority = tpri;
    timeSlice = ttime;
}

int Manage::chooseMethod(int choice, int contOfPro, int maxtime)
{
    mtime = maxtime;
    int result = 1;
    nowtime = 0;
    cont = contOfPro;
    switch (choice)
    {
    case 1:FIFS(); break;
    case 2:shortFirst(); break;
    case 3:leastTime(); break;
    case 4:RR(); break;
    case 5:changePri(); break;
    default:
        cout << "wrong input!" << endl;
        result = 0;
        break;
    }
    return result;
}
void Manage::FIFS()
```

```
{
    stable_sort(pro, pro + cont, compArr);
    nowtime = pro[0].ariveTime;
    for (int i = 0; i < cont; i++)
    {
        nowtime = nowtime < pro[i].ariveTime ? pro[i].ariveTime : nowtime;
        cout << i + 1 << "/" << pro[i].pid << "/" << nowtime << "/";
        nowtime += pro[i].runTime;
        cout << nowtime << "/" << pro[i].priority<<endl;
    }
}
void Manage::shortFirst()
{//不可抢占式短作业优先
    int i = 0, runNum = 0;
    stable_sort(pro, pro + cont, compArr);
    nowtime = pro[0].ariveTime;
    Proc runlist[m4x];
    while(nowtime<mtime)
    {
        while (nowtime>= pro[i].ariveTime&&i<cont)
        {
            runlist[i] = pro[i++];
            if (i >= cont)
                break;
        }
        stable_sort(runlist, runlist + i, compShort);
        //取排最前面的一个运行，结束之后将其运行时间置位unlimited
        if (runlist[0].runTime == unlimit)
        {
            nowtime++;
            continue;
        }
        cout << ++runNum << "/" << runlist[0].pid << "/" << nowtime << "/";
        nowtime += runlist[0].runTime;
        cout << nowtime << "/" << runlist[0].priority << endl;
        runlist[0].runTime = unlimit;
    }
}
void Manage::leastTime()
{
    int i = 0 , j, runNum = 0;
    stable_sort(pro, pro + cont, compArr);
    nowtime = pro[0].ariveTime;
    Proc runlist[m4x];
    Proc runing;
    while (nowtime < mtime)
    {
        j = 0;
        while (nowtime >= pro[i].ariveTime&&i<cont)
        {
            runlist[i] = pro[i++];
            if (i >= cont)
                break;
```

```
        }
        stable_sort(runlist, runlist + i, compLeast);
        if (runing.pid)//有正在运行的程序
        {
            if (runing.equl(runlist[0]))
            {//继续运行上一秒的程序
                nowtime++;
                runing.runTime--;
                runlist[0].runTime--;
            }
            else//要运行新来的程序
            {
                cout << nowtime << "/" << runing.priority << endl;
                runing = runlist[0];
                runing.runTime--;
                runlist[0].runTime--;
                cout << ++runNum << "/" << runing.pid << "/" << nowtime << "/";
                nowtime++;
            }
            if (runing.runTime == 0)
            {
                cout << nowtime << "/" << runing.priority << endl;
                runing.pid = 0;
                runlist[0].runTime = unlimit;
            }
        }
        else
        {
            if (runlist[0].runTime == unlimit)
            {
                nowtime++;
                continue;
            }
            runing = runlist[0];
            cout << ++runNum << "/" << runing.pid << "/" << nowtime<<"/";
            runing.runTime--;
            runlist[0].runTime--;
            nowtime++;
            if (runing.runTime == 0)//运行结束
            {
                cout << nowtime << "/" << runing.priority << endl;
                runing.pid = 0;//标记当前没有运行的程序。
                runlist[0].runTime = unlimit;
            }
        }

    }
}
void Manage::RR()
{
    queue <Proc> waitList;
    nowtime = 0;
    int sunRunTime = 0;
```

```
    int i = 0, runNum = 0, sliceInTime = 0;
    Proc runing;
    stable_sort(pro, pro + cont, compArr);
    nowtime = pro[0].ariveTime;
    while (sunRunTime < mtime)
    {
        sliceInTime = 0;//当前轮花费的时间
        while (nowtime >= pro[i].ariveTime&&i < cont)
            waitList.push(pro[i++]);
        if (runing.pid)//有正在运行的程序;
        {
            waitList.push(runing);
            runing = waitList.front();
            waitList.pop();
        }
        else
        {
            if (waitList.empty())//当前队列空
            {
                nowtime++;
                continue;
            }
            //没有正在运行的程序，且队列非空，取队头元素运行
            runing = waitList.front();
            waitList.pop();
        }
        cout << ++runNum << "/" << runing.pid << "/" << nowtime << "/";
        while (sliceInTime < runing.timeSlice)
        {
            sliceInTime++;
            nowtime++;
            sunRunTime++;
            runing.runTime--;
            if (runing.runTime == 0)
            {
                runing.pid = 0;
                break;
            }
        }
        cout << nowtime << "/" << runing.priority << endl;
    }
}
void Manage::changePri()
{
    nowtime = 0;
    int i = 0, j = 0, runNum = 0,sliceInTime=0;
    Proc runList[m4x],runing;
    stable_sort(pro, pro + cont, compArr);
    while (nowtime < mtime)
    {
        sliceInTime = 0;//记录当前时间片已经运行的时间
        while (nowtime >= pro[i].ariveTime&&i < cont)
            runList[i] = pro[i++];
```

```
        for (j = 1; j < i; j++)
            if (runList[j].priority&&runList[j].priority !=
unlimit&&runList[j].ariveTime<nowtime)
                runList[j].priority--;
        stable_sort(runList, runList + i, compPri);
        if (runList[0].priority == unlimit)
        {
            nowtime++;
            continue;
        }
        runing = runList[0];
        cout << ++runNum << "/" << runing.pid << "/" << nowtime << "/";
        while (sliceInTime < runing.timeSlice)
        {
            sliceInTime++;
            nowtime++;
            runList[0].runTime--;
            runing.runTime--;
            if (runing.runTime == 0)
            {
                runing.pid = 0;
                runList[0].priority = unlimit;
                break;
            }
        }
        runList[0].priority += 3;
        cout << nowtime << "/" << runing.priority+3<<endl;
    }
}

int main()
{
    int choice, cont = 0, maxtime = 0;;
    Manage pn;//创建调度器
    scanf("%d", &choice);
    int tmpId, tmpaArr, tmpRun, tmpPir, tmpSlice;
    while (~scanf("%d/%d/%d/%d/%d", &tmpId, &tmpaArr, &tmpRun, &tmpPir, &tmpSlice))
    {
        if (tmpId == 0)
            break;
        maxtime += tmpRun;
        pn.pro[cont++].setProc(tmpId, tmpaArr, tmpRun, tmpPir, tmpSlice);
    }
    pn.chooseMethod(choice,cont,maxtime);
    return 0;
}
```