## 实验一：实现单处理机下的进程调度程序

**实验内容：**

编写一个单处理机下的进程调度程序，模拟操作系统对进程的调度。

**实验要求：**

1、能够创建指定数量的进程，每个进程由一个进程控制块表示。

2、实现先来先服务调度算法：进程到达时间可由进程创建时间表示。

3、实现短作业优先调度算法：可指定进程要求的运行时间。（说明：对不可剥夺的短作业优先算法，当作业运行时间相等时，优先调度进程号小的进程执行；对可剥夺式的短作业优先算法，即选最短剩余时间的进程进行运行，在剩余时间相同的情况下，选择到达时间早的进程进行运行）

  4、实现时间片轮转调度算法：可指定生成时间片大小。（说明：新进程到来时插入到就绪队列的队尾，当进程P运行完一个时间片时，若同时有进程Q到达，则先在就绪队列队尾插入新到达的进程Q，之后再插入进程P)

5、实现动态优先级调度算法：可指定进程的初始优先级（优先级与优先数成反比，优先级最高为0），优先级改变遵循下列原则：进程在就绪队列中每停留一个时间片，优先级加1，进程每运行一个时间片，优先级减3。（说明：本算法在优先级相同的情况下，选择到达时间早的进程进行运行）

测试用例格式如下：

输入：调度算法

　　　进程号/到达时间/运行时间/优先级/时间片

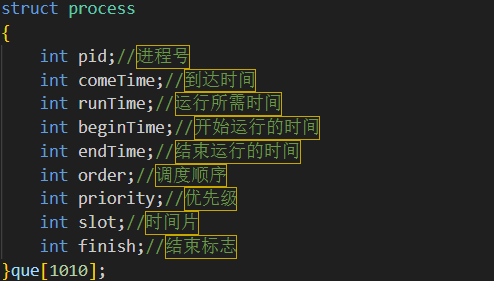
输出：调度顺序/进程号/开始运行时间/结束运行时间/优先级

其中调度算法选项为：1----先来先服务，2----短作业优先，3----最短剩余时间优先，4----时间片轮转，5----动态优先级

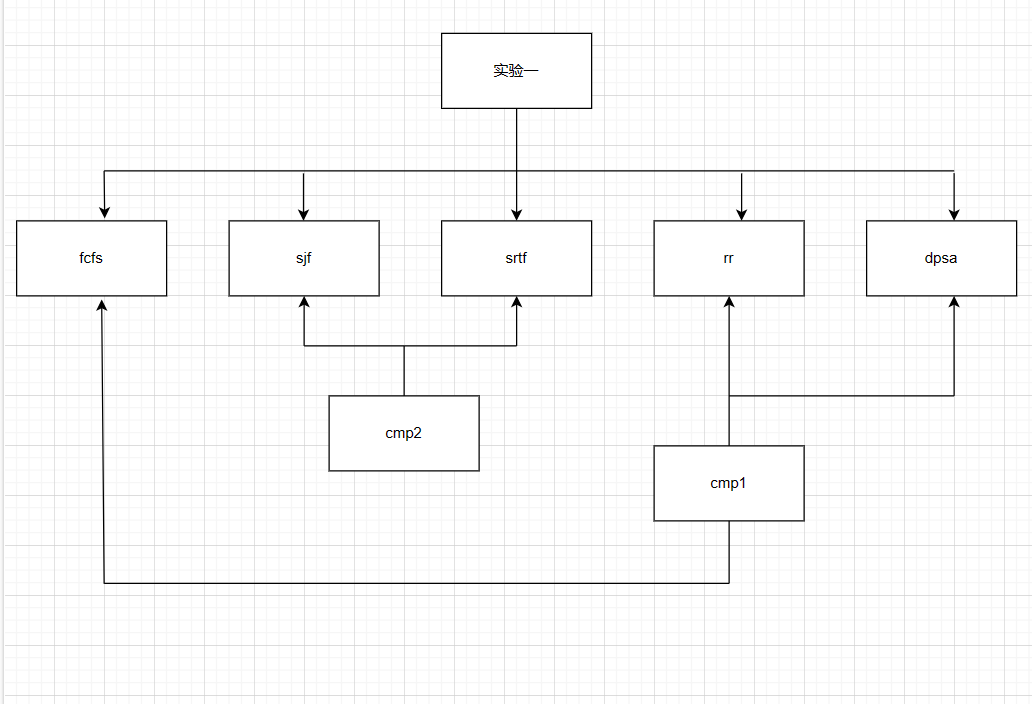
**设计思路：**

因为每一个进程输入的时候需要输入进程号、到达时间、运行时间、优先级、时间片五个属性，想到使用结构体来存储这些属性；同时进程输出的时候又要有调度顺序、开始运行时间、结束运行时间三个属性，以及短作业优先算法中还要判断进程是否运行结束，所以将进程需要用到的全部属性都存储在结构体里面。

时间片轮转算法中需要用到队列+结构体数组的形式来实现算法，其余四个算法使用结构体数组的形式实现算法。

如图：

程序概要设计如下图所示：



main（）函数是主程序的入口，接受程序输入，并按照输入的调度信号选择相应的算法模块进行运行。

FCFS（）函数是先来先服务算法，按照到来时间/进程号排序，先来到的进程首先处理，并将处理结果输出。

SJF（）函数是非剥夺的短作业优先算法，按照到来时间/运行时间/进程号排序，每次找出运行时间最短的进程运行，注意要按断进程是否已经运行结束，最后将结果输出。

SRTF（）函数是剥夺式的短作业优先算法，按照到来时间/运行时间/进程号排序，每次找出运行时间最短的进程运行，同时剥夺掉当前在运行的进程，注意要按断进程是否已经运行结束，最后将结果输出。

RR（）函数是时间片轮转算法，按照到来时间/进程号排序，每个进程运行一个时间片时间，注意判断进程是否运行结束，并将处理结果输出。

DPSA（）是动态优先级调度算法，按照到来时间/进程号排序，每个进程根据优先级分配运行时间，同时要动态调整进程的优先级，注意判断进程是否运行结束，并将处理结果输出。

cmp1（）是排序函数 1，按照到来时间/进程号排序。

cmp2（）是排序函数 2，按照到来时间/运行时间/进程号排序。

上机代码：

#include<iostream>

#include<cstdlib>

#include<cstdio>

#include<cmath>

#include<algorithm>

#include<queue>

using namespace std;

struct process

{

    int pid;//进程号

    int comeTime;//到达时间

    int runTime;//运行所需时间

    int beginTime;//开始运行的时间

    int endTime;//结束运行的时间

    int order;//调度顺序

    int priority;//优先级

    int slot;//时间片

    int finish;//结束标志

}que[1010];

void FCFS();//先来先服务

void SJF();//不可剥夺的短作业优先算法

void SRTF();//可剥夺式的短作业优先算法

void RR();//时间片轮转

void DPSA();//动态优先级

bool cmp1(process, process);//FCFS、RR、DPSA

bool cmp2(process, process);//SJF、SRTF

int num;//输入的总进程数

const int IsEnd = 1;//进程已经结束

const int NoEnd = 0;//进程还未结束

void readData() {

    num = 0;

    while (~scanf("%d/%d/%d/%d/%d", &que[num].pid, &que[num].comeTime, &que[num].runTime, &que[num].priority, &que[num].slot)) {

        que[num].beginTime = que[num].endTime = que[num].order = 0;

        que[num].finish = NoEnd;

        num++;

    }

}

void chooseAlgorithm(int sig) {

    switch (sig) {

        case 1: FCFS(); break;

        case 2: SJF(); break;

        case 3: SRTF(); break;

        case 4: RR(); break;

        case 5: DPSA(); break;

    }

}

int main() {

    //freopen("osin.txt", "r", stdin);

    //freopen("osout.txt", "w", stdout);

    int sig = 0;

    cin >> sig;   //算法选择标志

    readData();   //读入数据

    chooseAlgorithm(sig);   //选择算法

    return 0;

}

void FCFS()

{

    sort(que, que + num, cmp1); //先来后到排好序

    int i = 0;

    while (i < num)

    {

        que[i].order = i + 1;

        if (i == 0) //第一个进程特判一下

        {

            que[i].beginTime = que[i].comeTime;

        }

        else

        {

            que[i].beginTime = max(que[i].comeTime, que[i - 1].endTime);

        }

        que[i].endTime = que[i].beginTime + que[i].runTime;

        //输出

        printf("%d/%d/%d/%d/%d\n", que[i].order, que[i].pid, que[i].beginTime, que[i].endTime, que[i].priority);

        i++;

    }

}

void SJF()

{

    sort(que, que + num, cmp2);

    int lastTime = que[0].comeTime; //当前时间

    int tmp = 0; //每次选中的短作业

    int i = 0; //筛选num次，每次选出最佳的短作业

    while (i < num)

    {

        bool isBool = false; //判断当前时间是否有短作业在就绪队列中

        while (1)

        {

            if (i == 0)

            {

                //排序后的第一个肯定满足条件，特判

                break;

            }

            else

            {

                int j = 1; //for一遍的复杂度为O(n),比排序O(nlogn)快

                while (j < num)

                {

                    if (que[j].finish == IsEnd) //当前短作业已经结束就跳过

                    {

                        j++;

                        continue;

                    }

                    if (que[j].comeTime <= lastTime) //当前短作业在就绪队列中

                    {

                        if (isBool == false) //没有标记最优短作业

                        {

                            isBool = true;

                            tmp = j;

                        }

                        else //已经标记了最优短作业

                        {

                            //比较，更新最优短作业

                            //当短作业运行时间相等时，优先调度进程号小的短作业执行

                            if (que[j].runTime < que[tmp].runTime)

                                tmp = j;

                            else if (que[j].runTime == que[tmp].runTime && que[j].pid < que[tmp].pid)

                                tmp = j;

                        }

                    }

                    j++;

                }

                if (isBool) //如果存在短进程满足条件就输出

                    break;

                else //不存在就把时间后移再寻找

                    lastTime++;

            }

        }

        que[tmp].order = i + 1;

        que[tmp].beginTime = max(que[tmp].beginTime, lastTime);

        que[tmp].endTime = que[tmp].beginTime + que[tmp].runTime;

        printf("%d/%d/%d/%d/%d\n", que[tmp].order, que[tmp].pid, que[tmp].beginTime, que[tmp].endTime, que[tmp].priority);

        lastTime = que[tmp].endTime; //更新当前时间

        que[tmp].finish = IsEnd; //标记短作业结束

        i++;

    }

}

void SRTF()

{

    sort(que, que + num, cmp2);

    int lastTime = que[0].comeTime;

    int ID = 1;//输出顺序

    int tmp = 0, counts = 0;//当前进程，输出次数

    int isRun = -1, start = 0;//当前是否有进程运行,运行开始时间

    while (counts < num)

    {

        while (que[tmp].comeTime <= lastTime && tmp < num)//当前时间内的进程

        {

            tmp++;

        }

        int minx = 0x3f3f3f, minId = -1;//最短时间和下标

        for (int i = 0; i < tmp; i++)//寻找当前进程中剩余运行时间最短的进程

        {

            if (que[i].runTime < minx && que[i].finish == NoEnd)

            {

                minx = que[i].runTime;

                minId = i;

            }

        }

        if (minId == -1)//如果当前时间进程都结束了就等待下一个进程

        {

            lastTime = que[tmp].comeTime;

            continue;

        }

        if (isRun == -1)//当前没有进程在运行

        {

            isRun = minId;

            start = max(que[isRun].comeTime, lastTime);//运行刚找到的进程

        }

        //如果找到进程的剩余运行时间小于当前进程的剩余运行时间

        if (que[minId].runTime < que[isRun].runTime)

        {

            que[isRun].order = ID++;

            que[isRun].beginTime = start;

            que[isRun].endTime = lastTime;

            printf("%d/%d/%d/%d/%d\n", que[isRun].order, que[isRun].pid, que[isRun].beginTime, que[isRun].endTime, que[isRun].priority);

            isRun = minId;

            start = lastTime;

        }

        //运行进程

        int run = que[tmp].comeTime - lastTime;

        //进程能运行完就运行完

        if (run >= que[isRun].runTime || run <= 0)

        {

            lastTime += que[isRun].runTime;

            que[isRun].order = ID++;

            que[isRun].beginTime = start;

            que[isRun].endTime = lastTime;

            printf("%d/%d/%d/%d/%d\n", que[isRun].order, que[isRun].pid, que[isRun].beginTime, que[isRun].endTime, que[isRun].priority);

            que[isRun].runTime = 0;

            que[isRun].finish = IsEnd;

            counts++;

            isRun = -1;

            start = lastTime;

        }

        //进程不能运行完就运行一个时间片

        else

        {

            lastTime += run;

            que[isRun].runTime -= run;

        }

    }

}

void RR()

{

    sort(que, que + num, cmp1);

    queue<process>q;//就绪队列

    int lastTime = 0;//当前时间

    int ID = 1;//输出顺序

    int tmp = 0, counts = 0;//当前进程，输出次数

    while (counts < num)

    {

        if (q.empty())//队列为空则等待进程到来

        {

            lastTime = que[tmp].comeTime;

            while (que[tmp].comeTime <= lastTime && tmp < num)//当前时间内的进程

            {

                if (que[tmp].finish == NoEnd)//还没完成的进程入队

                {

                    q.push(que[tmp]);

                }

                tmp++;

            }

        }

        else

        {

            process ans = q.front();//取出处于队列头的进程

            q.pop();

            //进程可以运行完

            if (ans.runTime <= ans.slot)

            {

                //输出当前进程调度

                ans.order = ID++;

                ans.beginTime = lastTime;

                ans.endTime = ans.beginTime + ans.runTime;

                printf("%d/%d/%d/%d/%d\n", ans.order, ans.pid, ans.beginTime, ans.endTime, ans.priority);

                lastTime = ans.endTime;//更新当前时间

                ans.finish = IsEnd;//标记短作业结束

                counts++;

                while (que[tmp].comeTime <= lastTime && tmp < num)//当前时间内的进程

                {

                    if (que[tmp].finish == NoEnd)//还没完成的进程入队

                    {

                        q.push(que[tmp]);

                    }

                    tmp++;

                }

            }

            //进程不能运行完

            else

            {

                //输出当前进程调度

                ans.order = ID++;

                ans.beginTime = lastTime;

                ans.endTime = ans.beginTime + ans.slot;

                printf("%d/%d/%d/%d/%d\n", ans.order, ans.pid, ans.beginTime, ans.endTime, ans.priority);

                lastTime = ans.endTime;

                ans.runTime -= ans.slot;

                while (que[tmp].comeTime <= lastTime && tmp < num)//当前时间内的进程

                {

                    if (que[tmp].finish == NoEnd)//还没完成的进程入队

                    {

                        q.push(que[tmp]);

                    }

                    tmp++;

                }

                q.push(ans);//先让新进程入队再让当前进程入队

            }

        }

    }

}

void DPSA()

{

    sort(que, que + num, cmp1);

    int lastTime = que[0].comeTime;//当前时间

    int ID = 1;//输出顺序

    int tmp = 0, counts = 0;//当前进程，输出次数

    int start = lastTime;//当前进程开始时间

    int minId, Smin;

    while (counts < num)

    {

        while (que[tmp].comeTime <= lastTime && tmp < num)//当前时间内的进程

        {

            if (que[tmp].comeTime > start && que[tmp].comeTime < lastTime)

            {

                que[tmp].priority = max(que[tmp].priority - 1, 0);//等待时优先级加1

            }

            tmp++;

        }

        //寻找最高优先级进程

        minId = -1;

        Smin = 0x3f3f3f;

        for (int i = 0; i < tmp; i++)

        {

            if (que[i].priority < Smin && que[i].finish == NoEnd)

            {

                Smin = que[i].priority;

                minId = i;

            }

        }

        if (minId == -1)//如果当前时间进程都结束了就等待下一个进程

        {

            lastTime = que[tmp].comeTime;

            continue;

        }

        //运行进程

        start = lastTime;

        //进程能运行完

        if (que[minId].runTime <= que[minId].slot)

        {

            que[minId].order = ID++;

            que[minId].beginTime = lastTime;

            que[minId].endTime = lastTime + que[minId].runTime;

            printf("%d/%d/%d/%d/%d\n", que[minId].order, que[minId].pid, que[minId].beginTime, que[minId].endTime, que[minId].priority + 3);

            counts++;

            lastTime += que[minId].runTime;

            que[minId].finish = IsEnd;

        }

        //进程不能运行完

        else

        {

            que[minId].order = ID++;

            que[minId].beginTime = lastTime;

            que[minId].endTime = lastTime + que[minId].slot;

            printf("%d/%d/%d/%d/%d\n", que[minId].order, que[minId].pid, que[minId].beginTime, que[minId].endTime, que[minId].priority + 3);

            lastTime += que[minId].slot;

            que[minId].runTime -= que[minId].slot;

        }

        //改变进程优先级

        for (int i = 0; i < tmp; i++)

        {

            if (que[i].finish == NoEnd)

            {

                if (i == minId)//运行优先级减3

                {

                    que[i].priority += 3;

                }

                else//等待优先级加1

                {

                    que[i].priority = max(que[i].priority - 1, 0);

                }

            }

        }

    }

}

bool cmp1(process a, process b)

{

    if (a.comeTime != b.comeTime)

        return a.comeTime < b.comeTime;

    else

        return a.pid < b.pid;

}

bool cmp2(process a, process b)

{

    if (a.comeTime != b.comeTime)

        return a.comeTime < b.comeTime;

    else if (a.runTime != b.runTime)

        return a.runTime > b.runTime;

    else

        return a.pid < b.pid;

}

运行结果：



### **心得体会**

通过本次实验实现了五种进程调度算法，令我对操作系统内部的进程调度方式有了更深刻的认识和感受。对于非剥夺的短作业优先算法而言，因为每次都要检测当前时刻的短作业，排序其实可有可无；对于其余四种算法而言，排序则显得尤为重要。