****

**实 验 报 告**

**实验课程： 操作系统课程实践**

**学生姓名： 马星**

**学 号： 5418122020**

**专业班级： 计算机科学与技术(卓越)221班**

**2024年11月1日**

** 南昌大学实验报告**

**---（4）进程调度算法的实现**

学生姓名： 马星 学 号： 541812020 专业班级：计算机科学与技术(卓越)221班

实验类型：□ 验证 □ 综合 ■ 设计 □ 创新 实验日期： 11/30 实验成绩：

# 一、实验目的

通过实验加强对进程调度算法的理解和掌握，明确进程和程序的区别，理解常用进程调度算法的具体实现。

# 二、实验内容

编写程序模拟实现处理器系统中进程调度算法，实现对多个进程的模拟调度，具体可以编写程序实现先来先服务算法或优先度高者调度算法或者其它调度算法。

# 三、实验要求

1、需写出设计说明；

2、设计实现代码及说明

3、运行结果；

# 四、主要实验步骤

## 1. 设计说明

### (1) 设计需求

实现一个进程调度模拟器，用户可以通过选择不同的调度算法来模拟进程的调度过程。代码首先读取用户输入的n个进程信息(进程id, 进程到达时间, 进程需要运行的时间, 进程的优先级)，并根据用户选择的调度算法进行调度(仅实现了单队列的各个调度算法)，最后输出调度结果和统计信息。

使用map<int, PCB> processMap存储进程的信息, 进程id->进程PCB

### (2) 时间

为简化程序, 假设时间以分钟计, 且每分钟至多有1个进程到达

进程调度不消耗时间, 进程运行和调度器等待就绪队列消耗时间

(定义太复杂了, 代码写不出来, 只能简化一下了)

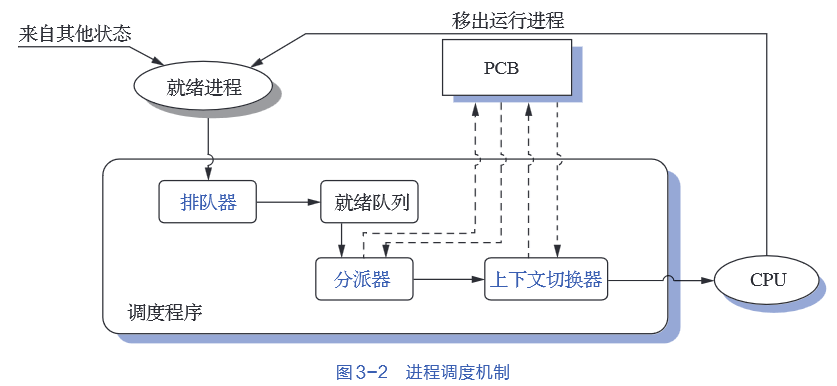
使用map<int, int> arriveTimeMap存储进程的到达信息, 到达时间->进程id

## 2. 实现代码

### (1) PCB定义

typedef struct PCB {  
 int id;// 进程的唯一id  
 int priority;// 优先级  
 int start;// 到达时间  
 int needtime;// 需要运行时间  
 int runtime = 0;// 已运行时间  
 int end = -1;// 结束时间   
 /\*\*  
 \* 进程运行1分钟  
 \*/  
 void run() {runtime++;}  
  
} PCB;  
  
#endif //PCB\_H

### (2) 调度器



调度程序需要做的事情对于外部来说, 其实也就是以下两点:

① 运行进程结束运行时, 从就绪队列调度进程; 如果就绪队列为空, 则等待进程到达

② 运行进程未结束运行时, 让其继续运行; 如果是抢占式的调度算法, 需要检查是否有进程到达, 触发抢占

用伪代码表示如下:

void control(…) {  
 checkProcessArrive();// 检查初始时间是否有进程到达(如果有加入就绪队列)  
 while (now < end) {// 系统未到达设定的结束时间  
 if (isFinish) {// 运行进程结束(调度就绪队列|等待就绪队列)  
 isFinish = processFinished(…);  
 } else {// 运行进程尚未结束(继续运行)  
 isFinish = processRun(…);  
 }  
 }

// 调度结束  
};

所以可以定义一个调度器父类, 然后调度器子类根据算法定义出processFinished和processRun这两个方法的具体实现

调度器父类代码:

内部维护一个就绪队列, 以及正在运行的进程的运行状态, 通过运行状态决定调度

#ifndef OP\_CONTROLLER\_H  
#define OP\_CONTROLLER\_H  
  
#include <bits/stdc++.h>  
#include "../PCB.h"  
  
using namespace std;  
  
class Controller {  
public:  
 vector<int> readyQueue;//就绪队列[进程id]  
 int curRunId = -1;// 当前运行进程id  
 bool isFinish = true;// 当前运行进程状态, 是否继续执行该进程  
 /\*\*  
 \* 进程执行  
 \* @param processMap 进程表  
 \* @param id 正在执行的进程id  
 \* @param arriveTimeMap 进程创建时间表  
 \* @param now 当前时间  
 \* @return 进程是否执行完  
 \*/  
 virtual bool processRun(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) = 0;  
  
 /\*\*  
 \* 当前运行进程结束执行(调度就绪队列|等待就绪队列)  
 \* 调度不耗时now不变, 等待耗时now+1  
 \* @param processMap 进程表  
 \* @return id 如果就绪队列中有进程,则调度并返回调度的进程id  
 \* @param arriveTimeMap 进程创建时间表  
 \* @param now 当前时间  
 \* @return 如果有进程被调度返回false(进程未执行完,可以执行), 否则返回true(进程执行完成,无进程可执行)  
 \*/  
 virtual bool processFinished(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) = 0;  
  
 /\*\*  
 \* 进程调度  
 \* @param processMap 进程列表  
 \* @param arriveTimeMap 进程到达时间->进程id  
 \* @param now 当前时间(初始时间)  
 \* @param end 结束时间  
 \*/  
 virtual void control(map<int, PCB> &processMap, map<int, int> &arriveTimeMap, int now, int end) {  
 checkProcessArrive(arriveTimeMap, now);  
 while (now < end) {  
 if (isFinish) {// 进程运行结束(调度就绪队列|等待就绪队列)  
 isFinish = processFinished(processMap, curRunId, arriveTimeMap, now);  
 } else {// 进程尚未运行结束(继续运行)  
 isFinish = processRun(processMap, curRunId, arriveTimeMap, now);  
 }  
 }  
 };  
  
 /\*\*  
 \* 检查当前是否有进程到达, 如果有,则放入就绪队列  
 \* @param arriveTimeMap 进程创建时间->进程id  
 \* @param now 当前时间  
 \* @return 是否有进程到达  
 \*/  
 bool checkProcessArrive(map<int, int> &arriveTimeMap, int now) {  
 if (arriveTimeMap.find(now) == arriveTimeMap.end()) {//无进程到达  
 return false;  
 }  
 readyQueue.push\_back(arriveTimeMap[now]);//放入wait队列  
 return true;  
 }  
};  
  
  
#endif //OP\_CONTROLLER\_H

### (3) 主程序

#include <bits/stdc++.h>  
  
#include "PCB.h"  
#include "Controller/controller.h"  
#include "Controller/FCFS.h"  
#include "Controller/HRRN.h"  
#include "Controller/NPP.h"  
#include "Controller/NPSJF.h"  
#include "Controller/PP.h"  
#include "Controller/PSJF.h"  
#include "Controller/RR.h"  
#include "Util.h"  
  
using namespace std;  
  
int n;//进程数量(用户输入)  
map<int, PCB> processMap;//进程id->进程PCB  
map<int, int> arriveTimeMap;// 到达时间->进程id  
int nowT = toMinute("08:00");//定义初始时间 08:00  
int endT = toMinute("12:00");//定义终止时间 12:00

//① 用户输入数据------------------------------------  
void readProcessList() {  
 cout << "请输入进程数量:";  
 cin >> n;  
 cout << "请输入" << n << "行,每行格式为:进程id 到达时间 需要运行时间 优先级" << endl;  
 for (int i = 0; i < n; i++) {  
 PCB p;  
 cin >> p.id;  
 string startT;  
 cin >> startT;  
 p.start = toMinute(startT);  
 cin >> p.needtime >> p.priority;  
 processMap[p.id] = p;  
 arriveTimeMap[p.start] = p.id;  
 }  
}

//② 打印初始数据------------------------------------

void printInitializedData() {   
 cout << "----------初始数据----------" << endl;  
 cout << "id | 到达时间 | 需要运行时间 | 优先级" << endl;  
 for (int i = 0; i < n; i++) {  
 PCB &p = processMap[i];  
 string hour = pre0(p.start / 60);  
 string minute = pre0(p.start % 60);  
 cout << p.id << " | " << hour << ":" << minute <<  
 " | " << p.needtime << "\t\t | " << p.priority << endl;  
 }   
}

//③ 用户选择调度算法------------------------------------  
Controller \*chooseMethod() {  
  
 int method;  
 cout << "请选择调度算法:" << endl;  
 cout << "1. FCFS(先来先服务)" << endl;  
 cout << "2. HRRN(高响应比)" << endl;  
 cout << "3. NPP(非抢占式优先级)" << endl;  
 cout << "4. NPSJF(非抢占式短作业优先)" << endl;  
 cout << "5. PP(抢占式优先级)" << endl;  
 cout << "6. PSJF(抢占式短作业优先)" << endl;  
 cout << "7. RR(轮转)" << endl;  
 cin >> method;  
 if (method == 1) {  
 return new FCFS();  
 } else if (method == 2) {  
 return new HRRN();  
 } else if (method == 3) {  
 return new NPP();  
 } else if (method == 4) {  
 return new NPSJF();  
 } else if (method == 5) {  
 return new PP();  
 } else if (method == 6) {  
 return new PSJF();  
 } else if (method == 7) {  
 return new RR();  
 } else {  
 cout << "ERROR: 输入错误" << endl;  
 exit(-1);  
 }  
  
}  
//⑤ 执行调度后汇总数据------------------------------------  
void printSummaryData() {  
 cout << "----------结果统计----------" << endl;  
 cout << "id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间" << endl;  
 double s1 = 0, s2 = 0;  
 for (auto &[id, p]: processMap) {  
 pair<string, string> s = timeToString(p.start), e = timeToString(p.end);  
 long turnaroundTime = p.end - p.start;// 周转时间 = 等待 + 运行 = 结束 - 到达  
 cout << p.id << " | " << s.first << ":" << s.second << " | " << p.needtime  
 << "\t\t | " << e.first << ":" << e.second << " | " << turnaroundTime  
 << "\t\t| " << round(turnaroundTime \* 1.0 / p.needtime, 2) << endl;  
 s1 += turnaroundTime;  
 s2 += turnaroundTime \* 1.0 / p.needtime;  
 }  
 cout << "平均周转时间:" << round(s1 / n, 2) << endl;  
 cout << "平均带权周转时间:" << round(s2 / n, 2) << endl;  
}  
//④ 主函数------------------------------------  
int main() {  
 readProcessList();// 数据读入  
 printInitializedData();  
  
 Controller \*controller = chooseMethod();  
 cout << "----------开始调度----------" << endl;  
 controller->control(processMap, arriveTimeMap, nowT, endT); //由调度器子类执行具体调度算法  
  
  
 printSummaryData();// 统计  
 return 0;  
}

### (4) 测试用例

n=5

id 到达时间 需要运行时间 优先级

0 08:00 25 1

1 08:20 10 2

2 08:25 20 3

3 08:30 20 4

4 08:35 15 5

### (5) 非抢占式

非抢占式调度有以下代表: 先来先服务、高响应比优先、非抢占式自定义优先级、非抢占式短作业优先

他们的区别仅在于”当从就绪队列调度进程时, 如何选出最高优先级的进程”

#### FCFS(先来先服务)

##### ①代码

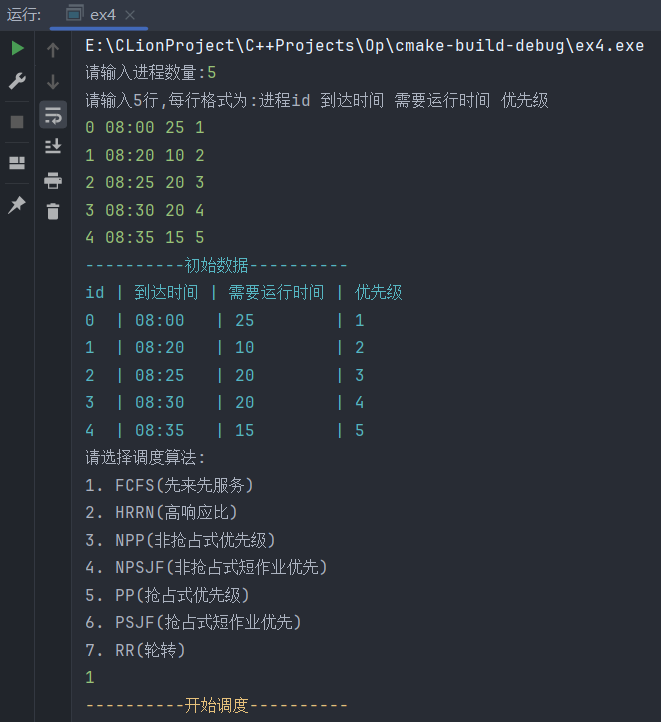
#ifndef OP\_FCFS\_H  
#define OP\_FCFS\_H  
  
#include "controller.h"  
  
class FCFS : public Controller {// 先来先服务  
public:  
 bool processRun(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 PCB &p = processMap[id];  
 p.run();  
 now++;  
 checkProcessArrive(arriveTimeMap, now);  
 // 进程结束 -> 结束  
 bool isFinish = p.isFinished();  
 if (isFinish) p.end = now;//运行结束  
 return isFinish;  
 }  
  
 bool processFinished(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 if (!readyQueue.empty()) {  
 id = readyQueue.front();//-----先来先服务-----  
 readyQueue.erase(readyQueue.begin());  
 processMap[id].selectedInfo(now);//打印日志  
 return false;// 可以执行  
 } else {  
 now++;  
 checkProcessArrive(arriveTimeMap, now);  
 return true;// 无进程可执行  
 }  
 }  
};  
  
#endif //OP\_FCFS\_H

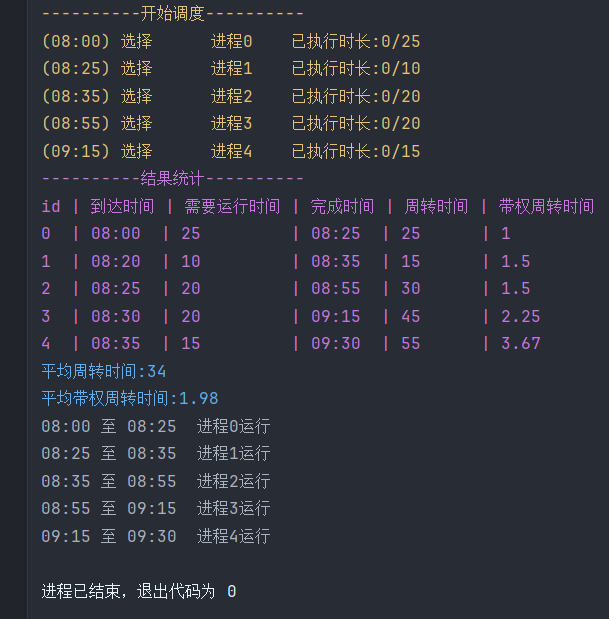
先来先服务:

当运行进程结束时, 从就绪队列的队头取出最先来的进程, 交给cpu运行 (如果就绪队列为空, 则等待就绪队列)

当运行进程未结束时, 则进程运行1分钟, 并检查进程是否结束 (运行过程中如果有进程到达则放入就绪队列)

##### ②运行结果





符合预期结果, 进程到达顺序为01234, 执行顺序也为01234

在控制台打印出具体的占用cpu段:

进程0在8:00到达, 执行25分钟, 期间进程1和进程2到达

进程0结束后, 因为进程1先于进程2到达, 进程1执行10分钟, 期间进程3和到达

进程1结束后, 因为进程2先于进程3和进程4到达, 进程2执行20分钟

…

最后执行进程4, 于9:30结束调度

#### HRRN(高响应比)

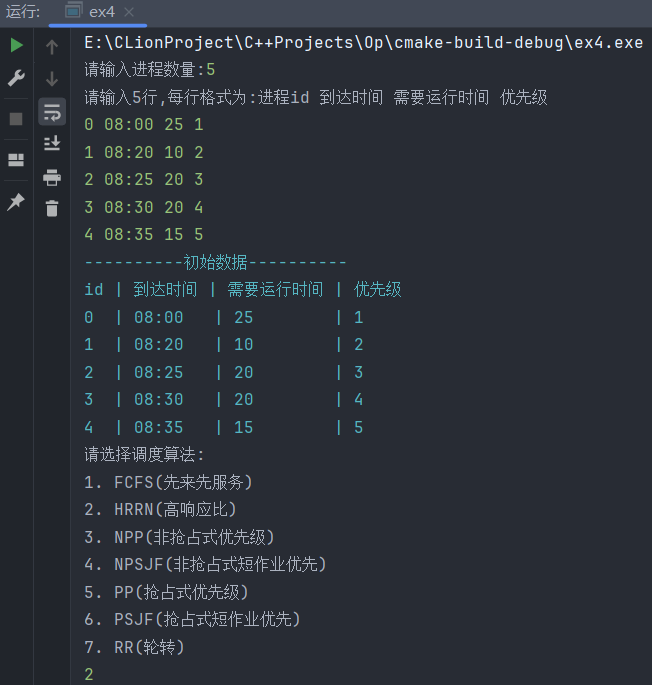
响应比（Response Ratio）是操作系统在进程调度时用来衡量进程优先级的一个指标。它综合考虑了进程的等待时间和运行时间，计算公式为R = (W + T) / T，其中W代表进程在后备状态队列中的等待时间，T代表进程的预计执行时间

通过动态调整优先级，高响应比优先调度算法能够平衡短作业和长作业的需求，避免了长时间作业被无限期推迟的问题。

##### ①代码

#ifndef OP\_HRRN\_H  
#define OP\_HRRN\_H  
  
#include "controller.h"  
  
class HRRN : public Controller {//高响应比优先  
public:  
 bool processRun(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 PCB &p = processMap[id];  
 p.run(now);  
 now++;  
 checkProcessArrive(arriveTimeMap, now);  
 // 进程结束 -> 结束  
 bool isFinish = p.isFinished();  
 if (isFinish) p.end = now;//运行结束  
 return isFinish;  
 }  
  
 /\*\*  
 \* 获取就绪队列中最高响应比的进程所在索引  
 \* @param processMap 进程表  
 \*/  
 int maxRRProcessIdx(map<int, PCB> &processMap, int now) {  
 auto comp = [&processMap, now](int a, int b) {  
 double responseRatioA = processMap.at(a).getResponseRatio(now);// 1 + ((now - start) - runtime) / needtime;  
 double responseRatioB = processMap.at(b).getResponseRatio(now);  
 return responseRatioA < responseRatioB;  
 };  
  
 auto it = std::max\_element(readyQueue.begin(), readyQueue.end(), comp);  
 return distance(readyQueue.begin(), it);  
 }  
  
 bool processFinished(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 if (!readyQueue.empty()) {  
 //----------高响应比优先------------  
 int findIdx = maxRRProcessIdx(processMap, now);  
 id = readyQueue[findIdx];  
 readyQueue.erase(readyQueue.begin() + findIdx);  
  
 processMap[id].selectedInfo(now);  
 return false;  
 } else {  
 now++;  
 checkProcessArrive(arriveTimeMap, now);  
 return true;  
 }  
 }  
  
};  
  
#endif //OP\_HRRN\_H

##### ②运行结果





进程0于8:00到达, 调度执行25分钟, 期间进程1和进程2到达

进程0结束后, 进程1响应比=(5+10)/10 = 1.5, 进程2响应比=(0+20)/20=1, 所以选择进程1执行10分钟, 期间进程3和进程4到达

进程1结束后, 进程2响应比=1.5, 进程3响应比=1.25, 进程4响应比=1, 所以选择进程2执行20分钟

进程2结束后, 进程3响应比= 2.25, 进程4响应比=2.33, 所以选择进程4执行15分钟

进程4结束后执行进程3于9:30结束调度

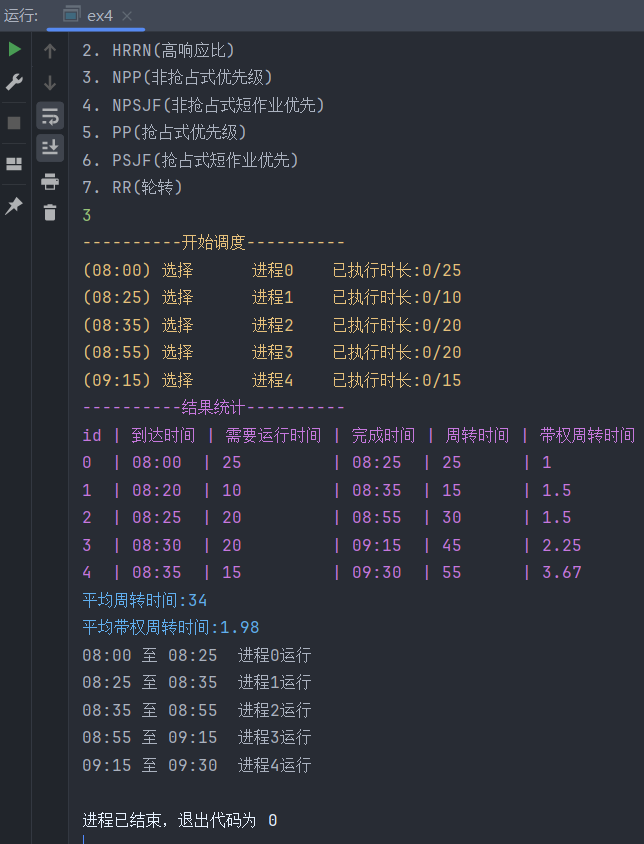
可以从控制台输出看到, 进程模拟调度情况与计算一致

#### NPP(非抢占式优先级)

##### ①区别部分代码

/\*\*  
 \* 获取就绪队列中剩余需要运行时间最少的进程所在索引  
 \* @param processMap 进程表  
 \*/  
 int minPriorityProcessIdx(map<int, PCB> &processMap) {  
 auto comp = [&processMap](int a, int b) {  
 return processMap.at(a).priority < processMap.at(b).priority;  
 };  
 auto it = std::min\_element(readyQueue.begin(), readyQueue.end(), comp);  
 return distance(readyQueue.begin(), it);  
 }  
  
 bool processFinished(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 if (!readyQueue.empty()) {  
 //--------高优先级-------------  
 int findIdx = minPriorityProcessIdx(processMap);  
 id = readyQueue[findIdx];  
 readyQueue.erase(readyQueue.begin() + findIdx);  
  
 processMap[id].selectedInfo(now);//打印日志  
 return false;  
 } else {  
 now++;  
 checkProcessArrive(arriveTimeMap, now);  
 return true;  
 }  
 }

##### ②运行结果



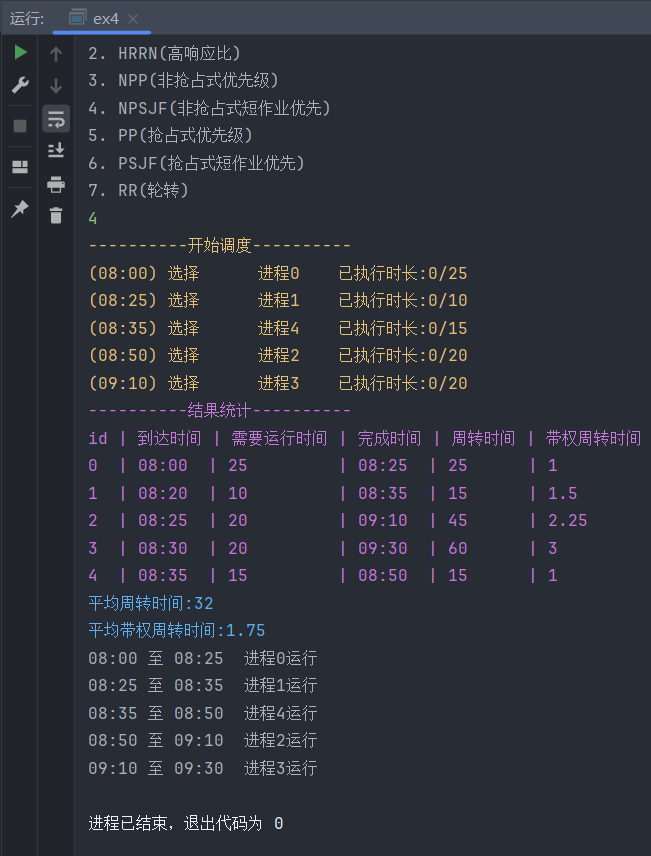
由于输入的优先级与时间顺序一致, 所以直接比对FCFS的测试输出, 输出一致

#### NPSJF(非抢占式短作业优先)

##### ①区别部分代码

/\*\*  
 \* 获取就绪队列中剩余需要运行时间最少的进程所在索引  
 \* @param processMap 进程表  
 \*/  
int minTimeProcessIdx(map<int, PCB> &processMap) {  
 auto comp = [&processMap](int a, int b) {  
 return processMap.at(a).getRemainTime() < processMap.at(b).getRemainTime();  
 };  
 auto it = std::min\_element(readyQueue.begin(), readyQueue.end(), comp);  
 return distance(readyQueue.begin(), it);  
}  
  
bool processFinished(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 if (!readyQueue.empty()) {  
 //找到最短的进程  
 int findIdx = minTimeProcessIdx(processMap);  
 id = readyQueue[findIdx];  
 readyQueue.erase(readyQueue.begin() + findIdx);  
  
 processMap[id].selectedInfo(now);//打印日志  
 return false;  
 } else {  
 now++;  
 checkProcessArrive(arriveTimeMap, now);  
 return true;  
 }  
}

##### ②运行结果



进程0于8:00到达, 执行25分钟, 期间进程1和进程2到达

进程0结束, 进程1时间短于进程2, 所以进程1执行10分钟, 期间进程3和进程4到达

进程1结束, 进程4时间最短, 所以进程4执行15分钟

进程4结束, 进程2和进程3时间一致, 先执行进程2

进程2结束, 执行进程3, 于9:30结束

### (6) 抢占式

抢占式与非抢占式区别在于”当有进程运行时有新的进程到达, 则触发抢占”

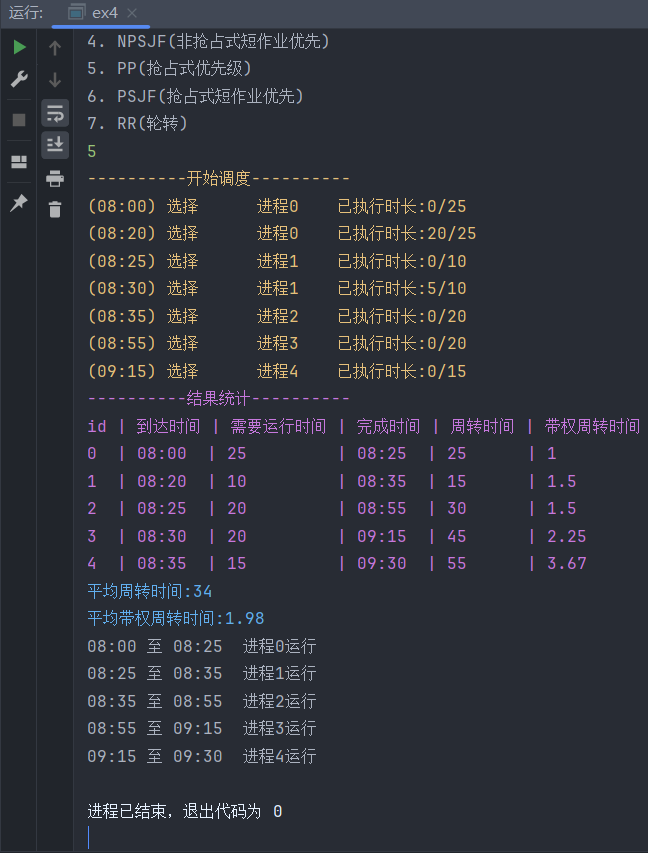
我的设计是在触发抢占时, 直接将运行进程退出cpu, 重新从就绪队列进行调度

#### PP(抢占式优先级)

##### ①代码

#ifndef OP\_PP\_H  
#define OP\_PP\_H  
  
#include "controller.h"  
  
class PP : public Controller {//抢占式优先级调度  
public:  
 bool processRun(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 PCB &p = processMap[id];  
 p.run(now);  
 now++;  
 // 触发抢占 或 进程结束 -> 结束  
 bool isFinish = false;  
 if (checkProcessArrive(arriveTimeMap, now)) {//有进程到达时抢占  
 isFinish = true;  
 if (!p.isFinished()) readyQueue.push\_back(id);//当前进程放入wait队列  
 }  
  
 isFinish = isFinish || p.isFinished();  
 if (p.isFinished()) p.end = now;//运行结束  
 return isFinish;  
 }  
  
 /\*\*  
 \* 获取就绪队列中剩余需要运行时间最少的进程所在索引  
 \* @param processMap 进程表  
 \*/  
 int minPriorityProcessIdx(map<int, PCB> &processMap) {  
 auto comp = [&processMap](int a, int b) {  
 return processMap.at(a).priority < processMap.at(b).priority;  
 };  
 auto it = std::min\_element(readyQueue.begin(), readyQueue.end(), comp);  
 return distance(readyQueue.begin(), it);  
 }  
  
 bool processFinished(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 if (!readyQueue.empty()) {  
 //找到最高优先级的进程  
 int findIdx = minPriorityProcessIdx(processMap);  
 id = readyQueue[findIdx];  
 readyQueue.erase(readyQueue.begin() + findIdx);  
  
 processMap[id].selectedInfo(now);//打印日志  
 return false;  
 } else {  
 now++;  
 checkProcessArrive(arriveTimeMap, now);  
 return true;  
 }  
 }  
  
};  
  
#endif //OP\_PP\_H

##### ②运行结果



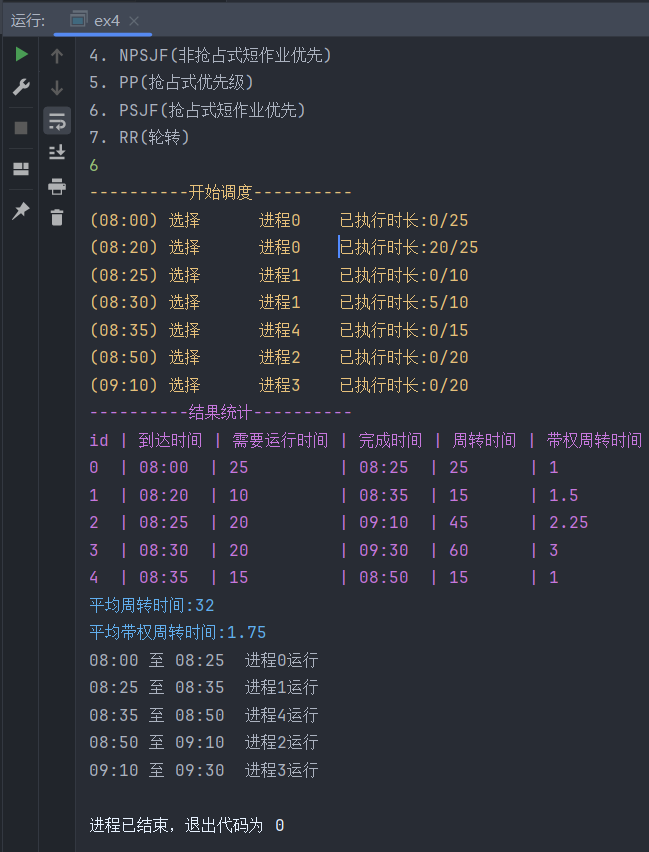
优先级顺序为01234, 与到达顺序一致, 不存在抢占, 所以输出与FCFS一致

#### PSJF(抢占式短作业优先)

##### ①代码

#ifndef OP\_PSJF\_H  
#define OP\_PSJF\_H  
  
#include "controller.h"  
  
class PSJF : public Controller {//抢占式短作业优先  
public:  
 bool processRun(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 PCB &p = processMap[id];  
 p.run(now);  
 now++;  
 // 触发抢占 或 进程结束 -> 结束  
 bool isFinish = false;   
 if (checkProcessArrive(arriveTimeMap, now)) {//有进程到达时抢占  
 isFinish = true;  
 if (!p.isFinished()) readyQueue.push\_back(id);//当前进程放入wait队列  
 }  
  
 isFinish = isFinish || p.isFinished();  
 if (p.isFinished()) p.end = now;//运行结束  
 return isFinish;  
 }  
 /\*\*  
 \* 获取就绪队列中剩余需要运行时间最少的进程所在索引  
 \* @param processMap 进程表  
 \*/  
 int minTimeProcessIdx(map<int, PCB> &processMap) {  
 auto comp = [&processMap](int a, int b) {  
 return processMap.at(a).getRemainTime() < processMap.at(b).getRemainTime();  
 };  
 auto it = std::min\_element(readyQueue.begin(), readyQueue.end(), comp);  
 return distance(readyQueue.begin(), it);  
 }  
  
 bool processFinished(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 if (!readyQueue.empty()) {  
 //找到最短的进程  
 int findIdx = minTimeProcessIdx(processMap);  
 id = readyQueue[findIdx];  
 readyQueue.erase(readyQueue.begin() + findIdx);  
  
 processMap[id].selectedInfo(now);//打印日志  
 return false;  
 } else {  
 now++;  
 checkProcessArrive(arriveTimeMap, now);  
 return true;  
 }  
 }  
  
};  
  
#endif //OP\_PSJF\_H

##### ②运行结果



进程0于8:00到达, 执行20分钟后进程1到达, 触发抢占, 但此时进程0剩余5分钟<进程1剩余10分钟, 仍由进程0执行, 进程0再执行5分钟

进程0执行结束, 此时进程2也到达, 进程1需要10分钟,而进程2需要20分钟, 所以由进程1执行

进程1执行5分钟时进程3到达, 此时进程1剩余5分钟,进程2剩余20分钟,进程3剩余20分钟, 所以进程1再执行5分钟

进程1执行结束, 此时进程4也到达, 进程2和进程3都需要20分钟, 进程4需要15分钟, 所以进程4执行

然后进程2执行, 最后进程3执行, 于9:30结束调度

### (7) RR(轮转)

轮转式与前面的抢占式和非抢占式不太一样, 所以单拎出来了

#### ①代码

轮转式调度算法, 首先需要设置一个时间片长度, 当时间片用完后, 运行进程需要退出cpu, 所以需要保留当前已使用的时间片长度

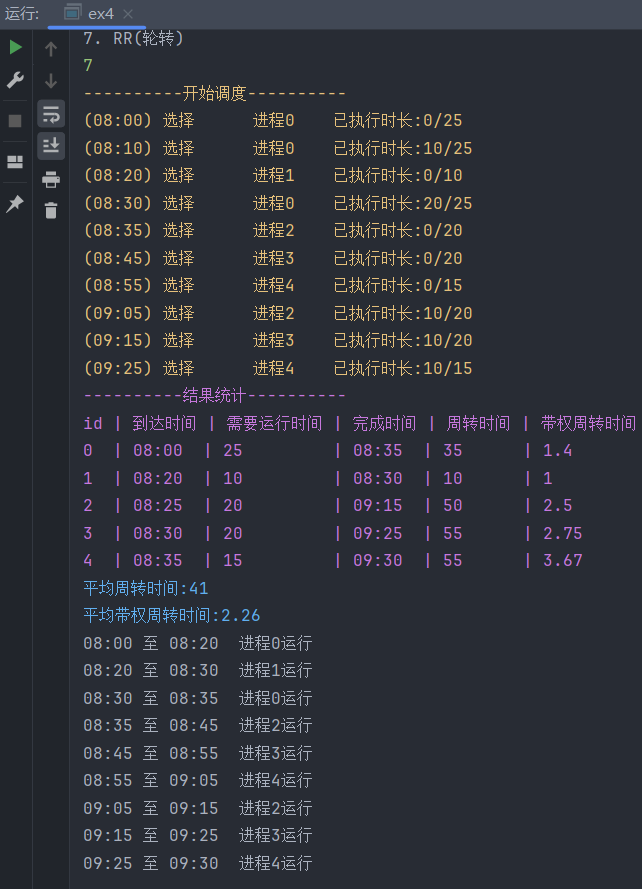
进程执行时已使用时间片长度也需要同步增加

时间片用完 或者 进程运行结束时 进程都需要退出cpu, 然后重新记录已使用时间片长度

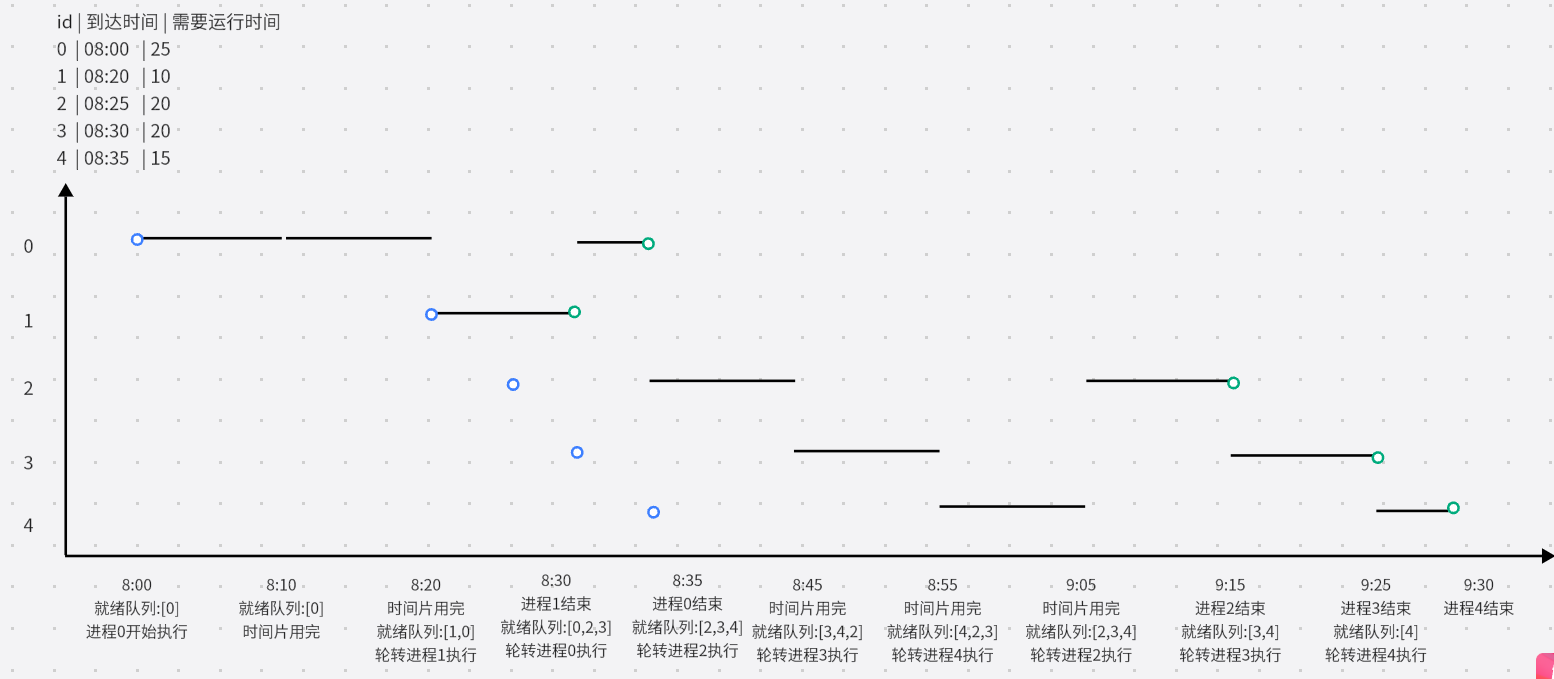
从就绪队列调度进程采用先来先服务的轮转方式

#ifndef OP\_RR\_H  
#define OP\_RR\_H  
#include "controller.h"  
  
class RR : public Controller {//轮转   
public:  
 int TimeSlice = 10;//时间片长度  
 int curUseTimeSplice = 0;//当前已使用时间片长度  
 /\*\*  
 \* @return 当前时间片是否用完  
 \*/  
 bool isRunOut() const {  
 return curUseTimeSplice == TimeSlice;  
 }  
  
 bool processRun(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 PCB &p = processMap[id];  
 p.run(now);  
 now++;  
 checkProcessArrive(arriveTimeMap, now);  
 curUseTimeSplice++;  
 // 进程结束 或 时间片用完 -> 结束  
 bool isFinish = p.isFinished()||isRunOut();  
  
 if (p.isFinished()) {// 进程结束  
 p.end = now;  
 isFinish = true;  
 } else if (isRunOut()) {// 进程未结束但时间片用完, 重新放回就绪队列  
 readyQueue.push\_back(id);  
 isFinish = true;  
 }  
 return isFinish;  
 }  
  
 bool processFinished(map<int, PCB> &processMap, int &id, map<int, int> &arriveTimeMap, int &now) override {  
 curUseTimeSplice = 0;  
 if (!readyQueue.empty()) {  
 id = readyQueue.front();//先来先服务  
 readyQueue.erase(readyQueue.begin());  
 processMap[id].selectedInfo(now);//打印日志  
 return false;// 可以执行  
 } else {  
 now++;  
 checkProcessArrive(arriveTimeMap, now);  
 return true;// 无进程可执行  
 }  
 }  
   
};  
  
#endif //OP\_RR\_H

#### ②运行结果



输出与计算一致:



# 五、实验数据及处理结果

## (1)测试数据:

|  |  |  |  |
| --- | --- | --- | --- |
| 进程id | 到达时间 | 需要运行时间 | 优先级 |
| 0 | 08:00 | 25 | 1 |
| 1 | 08:20 | 10 | 2 |
| 2 | 08:25 | 20 | 3 |
| 3 | 08:30 | 20 | 4 |
| 4 | 08:35 | 15 | 5 |

### 使用FCFS(先来先服务)调度算法模拟结果:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 08:25 | 25 | 1 |
| 1 | 08:20 | 10 | 08:35 | 15 | 1.5 |
| 2 | 08:25 | 20 | 08:55 | 30 | 1.5 |
| 3 | 08:30 | 20 | 09:15 | 45 | 2.25 |
| 4 | 08:35 | 15 | 09:30 | 55 | 3.67 |

平均周转时间: (25+15+30+45+55)/5 = 34

平均带权周转时间: (1+1.5+1.5+2.25+3.67)/5=1.98

### 使用HRRN(高响应比)调度算法模拟结果:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 08:25 | 25 | 1 |
| 1 | 08:20 | 10 | 08:35 | 15 | 1.5 |
| 2 | 08:25 | 20 | 08:55 | 30 | 1.5 |
| 3 | 08:30 | 20 | 09:30 | 60 | 3 |
| 4 | 08:35 | 15 | 09:10 | 35 | 2.33 |

平均周转时间: (25+15+30+60+35)/5 = 33

平均带权周转时间: (1+1.5+1.5+3+2.33)/5=1.87

### 使用NPP(非抢占式优先级)调度算法模拟结果:

输入优先级与时间顺序一致, 所以和FCFS结果相同

### 使用NPSJF(非抢占式短作业优先) 调度算法模拟结果:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 08:25 | 25 | 1 |
| 1 | 08:20 | 10 | 08:35 | 15 | 1.5 |
| 2 | 08:25 | 20 | 09:10 | 45 | 2.25 |
| 3 | 08:30 | 20 | 09:30 | 60 | 3 |
| 4 | 08:35 | 15 | 08:50 | 15 | 1 |

平均周转时间: (25+15+45+60+15)/5 = 32

平均带权周转时间: (1+1.5+2.25+3+1)/5 = 1.75

### 使用PP(抢占式优先级) 调度算法模拟结果:

输入优先级与时间顺序一致, 所以和FCFS结果相同

### 使用PSJF(抢占式短作业优先)调度算法模拟结果:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 08:25 | 25 | 1 |
| 1 | 08:20 | 10 | 08:35 | 15 | 1.5 |
| 2 | 08:25 | 20 | 09:10 | 45 | 2.25 |
| 3 | 08:30 | 20 | 09:30 | 60 | 3 |
| 4 | 08:35 | 15 | 08:50 | 15 | 1 |

平均周转时间: (25+15+45+60+15)/5 = 32

平均带权周转时间: (1+1.5+2.25+3+1)/5 = 1.75

### 使用RR(轮转)调度算法模拟结果:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 08:35 | 35 | 1.4 |
| 1 | 08:20 | 10 | 08:30 | 10 | 1 |
| 2 | 08:25 | 20 | 09:15 | 50 | 2.5 |
| 3 | 08:30 | 20 | 09:25 | 55 | 2.75 |
| 4 | 08:35 | 15 | 09:30 | 55 | 3.67 |

平均周转时间: (35+10+50+55+55)/5 = 41

平均带权周转时间: (1.4+1+2.5+2.75+3.67)/5 = 2.26

### 结果对比:

|  |  |  |
| --- | --- | --- |
| 算法 | 平均周转时间 | 带权周转时间 |
| FCFS | 34 | 1.98 |
| HRRN | 33 | 1.87 |
| NPP | 34 | 1.98 |
| NPSJF | 32 | 1.75 |
| PP | 34 | 1.98 |
| PSJF | 32 | 1.75 |
| RR | 41 | 2.26 |

FCFS简单易行, 但进程的堆积可能导致后面的进程长时间等待

HRRN可以较好地平衡响应时间和周转时间

采用SJF短作业优先算法能够得到较低的平均周转时间, 短作业先完成, 整体的等待时间就会减少

优先级调度则能快速响应高优先级任务，却可能引发饥饿问题

而对于RR(轮转)调度算法, 它的平均周转时间明显高于其他算法, 因为尽管它均匀地照顾到了每一个进程, 但是整体的等待时间会变长

## (2) 数据补充:

优先级没有设置好, 重新测试一份数据

|  |  |  |  |
| --- | --- | --- | --- |
| 进程id | 到达时间 | 需要运行时间 | 优先级 |
| 0 | 08:00 | 25 | 4 |
| 1 | 08:10 | 20 | 2 |
| 2 | 08:25 | 10 | 1 |
| 3 | 08:30 | 20 | 5 |
| 4 | 08:35 | 15 | 3 |

### FCFS:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 08:25 | 25 | 1 |
| 1 | 08:10 | 20 | 08:45 | 35 | 1.75 |
| 2 | 08:25 | 10 | 08:55 | 30 | 3 |
| 3 | 08:30 | 20 | 09:15 | 45 | 2.25 |
| 4 | 08:35 | 15 | 09:30 | 55 | 3.67 |

平均周转时间: (25+35+30+45+55)/5 = 38

平均带权周转时间: (1+1.75+3+2.25+3.67)/5 = 2.33

### HRNN:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 08:25 | 25 | 1 |
| 1 | 08:10 | 20 | 08:45 | 35 | 1.75 |
| 2 | 08:25 | 10 | 08:55 | 30 | 3 |
| 3 | 08:30 | 20 | 09:30 | 60 | 3 |
| 4 | 08:35 | 15 | 09:10 | 35 | 2.33 |

平均周转时间: (25+35+30+60+35)/5 = 37

平均带权周转时间: (1+1.75+3+3+2.33)/5 = 2.22

### NPP:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 08:25 | 25 | 1 |
| 1 | 08:10 | 20 | 08:55 | 45 | 2.25 |
| 2 | 08:25 | 10 | 08:35 | 10 | 1 |
| 3 | 08:30 | 20 | 09:30 | 60 | 3 |
| 4 | 08:35 | 15 | 09:10 | 35 | 2.33 |

平均周转时间: (25+45+10+60+35)/5 = 35

平均带权周转时间: (1+2.25+1+3+2.33)/5 = 1.92

### NPSJF

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 08:25 | 25 | 1 |
| 1 | 08:10 | 20 | 09:10 | 60 | 3 |
| 2 | 08:25 | 10 | 08:35 | 10 | 1 |
| 3 | 08:30 | 20 | 09:30 | 60 | 3 |
| 4 | 08:35 | 15 | 08:50 | 15 | 1 |

平均周转时间: (25+60+10+60+15)/5 = 34

平均带权周转时间: (1+3+1+3+1)/5 = 1.8

### PP:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 09:10 | 70 | 2.8 |
| 1 | 08:10 | 20 | 08:40 | 30 | 1.5 |
| 2 | 08:25 | 10 | 08:35 | 10 | 1 |
| 3 | 08:30 | 20 | 09:30 | 60 | 3 |
| 4 | 08:35 | 15 | 08:55 | 20 | 1.33 |

平均周转时间: (70+30+10+60+20)/5 = 38

平均带权周转时间: (2.8+1.5+1+3+1.33)/5 = 1.93

### PSJF:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 08:25 | 25 | 1 |
| 1 | 08:10 | 20 | 09:10 | 60 | 3 |
| 2 | 08:25 | 10 | 08:35 | 10 | 1 |
| 3 | 08:30 | 20 | 09:30 | 60 | 3 |
| 4 | 08:35 | 15 | 08:50 | 15 | 1 |

平均周转时间: (25+60+10+60+15)/5 = 34

平均带权周转时间: (1+3+1+3+1)/5 = 1.8

### RR:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | 到达时间 | 需要运行时间 | 完成时间 | 周转时间 | 带权周转时间 |
| 0 | 08:00 | 25 | 09:05 | 65 | 2.6 |
| 1 | 08:10 | 20 | 08:40 | 30 | 1.5 |
| 2 | 08:25 | 10 | 08:50 | 25 | 2.5 |
| 3 | 08:30 | 20 | 09:25 | 55 | 2.75 |
| 4 | 08:35 | 15 | 09:30 | 55 | 3.67 |

平均周转时间: (65+30+25+55+55)/5 = 46

平均带权周转时间: (2.6+1.5+2.5+2.75+3.67)/5 = 2.6

### 结果对比:

|  |  |  |
| --- | --- | --- |
| 算法 | 平均周转时间 | 带权周转时间 |
| FCFS | 38 | 2.33 |
| HRRN | 37 | 2.22 |
| NPP | 35 | 1.92 |
| NPSJF | 34 | 1.8 |
| PP | 38 | 1.93 |
| PSJF | 34 | 1.8 |
| RR | 46 | 2.6 |

与之前结论一致

FCFS简单易行, 但进程的堆积可能导致后面的进程长时间等待

HRRN可以较好地平衡响应时间和周转时间

采用SJF短作业优先算法能够得到较低的平均周转时间, 短作业先完成, 整体的等待时间就会减少

优先级调度则能快速响应高优先级任务，却可能引发饥饿问题

而对于RR(轮转)调度算法, 它的平均周转时间明显高于其他算法, 因为尽管它均匀地照顾到了每一个进程, 但是整体的等待时间会变长

# 六、实验体会或对改进实验的建议

通过实现一个进程调度模拟器，我深入理解了不同调度算法的工作原理和优缺点。在实验过程中，我首先设计了一个数据结构来存储每个进程的信息，包括进程ID、到达时间、运行时间和优先级等。然后，根据用户选择的调度算法，实现了相应的调度逻辑，并输出了调度结果和统计信息。

通过这个实验，我对进程调度有了更深入的理解，也提高了自己的编程能力和解决问题的能力。

# 七、参考资料

《计算机操作系统实验指导》