

实 验 报 告

课程名称 操作系统 (B)

开课学期 2023-2024 学年第一学期 指导教师 喻学才

实 验 室 启天 B118

班 级 2021 软件工程 6 班

学 号 20210715B0602

姓 名 张旭君

成绩： (五级)

实验课程 评分表标准

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 全勤、学习态度 端正 、 实验认 真、积极回答问 题、操作过程正 确，结果准确， 实验报告 内容 规范 | 偶有缺勤、实验 认真、回答问题 较积极、操作过 程正确，结果准 确，实验报告内 容规范 | 旷课 2 次以内、 偶有迟到、实验 认真、回答问题 较好、操作过程 基本正确，结果 基本准确，实验 报告 内容较规 范 | 旷课 2 次以上、 学习态度一般、 基本 能回答 出 问题、操作过程 较正确，结果基 本准确，实验报 告 内容基本规 范 | 经常旷课，实验 过程不认真、问 题回答不积极、 实验报告不符 合要求或未交 |
| 优秀 (90-100) | 良好 (80-89) | 中 (70-79) | 及格 (60-69) | 不及格(<59) |
| 实验一 |  |  |  |  |  |
| 实验二 |  |  |  |  |  |
| 实验三 |  |  |  |  |  |
| 实验成绩总评 (五级制) | |  | | | |

说明：1 ．每次实验结束，学生完成一份实验报告，课程结束后汇总,加封面装订成册存档； 2 ．各系 (部) 可在以上五项栏目的基础上，可根据实验课程和实验项目的具体需要，统一 设计和调整项目内容，但封面格式应统一；3 ．对于设计性实验，只要求说明实验的目的要 求、提出可供实验的基本条件和注意事项，实验方案和步骤的设置、仪器的安排等，可由学 生自己设计；4.可根据实验数量自行添加行数。打印到封面背面

桂林航天工业学院学生实验报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 操作系统 (B) | | | 实验项目名称 | | 进程调度 (4 学时) | | |
| 开课教学单位及实验室 | | 计算机科学与工程学院 | | | | 实验日期 | | 2023 年 10 月 25 日 |
| 学生姓名 | 张旭君 | | 学号 | | 20210715B0602 | 专业班级 | | 2020 软件工程 5 班 |
| 指导教师 | 喻学才 | | | | 实验成绩 | |  | |
| 一、实验目的  1.能复述解释进程的相关概念、进程状态及状态之间的转换过程。  2.能复述进程调度的任务，能举例解释进程调度的机制和方式。  3.能实现进程调度算法，并对比分析算法性能。  4. 能与AI对话机器人合作独立完成实验内容。  二、实验内容及要求  编写并调试一个单道处理系统的进程等待模拟程序。  可选择实现下列进程调度算法，对每种调度算法都要求计算并输出每个进程开始运行时刻、完 成时刻、周转时间：  (1) 先来先服务 (FCFS) 与最短作业优先 (SJF)；  先来先服务(FCFS)算法：按照作业/进程进入系统的先后次序进行调度，先进入系统者先调度； 即启动等待时间最长的作业/进程。  短作业优先(SJF)的调度算法：从后备队列中选择一个或若干个估计运行时间最短的作业，将 它们调入内存运行。  (2) 非抢占式与抢占式优先权调度算法  非抢占式优先权调度算法：系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程 便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分 配给另一优先权最高的进程。  抢占式优先权调度算法：只要系统中出现一个新的就绪进程，就进行优先权比较 。若出现优 先权更高的进程，则立即停止当前执行，并将处理机分配给新到的优先权最高的进程。  建议采用静态优先权的方式，赋予各进程固定的优先级。  (3) 基本时间片轮转调度算法  系统将所有的就绪进程按先来先服务的原则，排成一个队列，每次调度时，将 CPU 分配给队 首进程，并令其执行一个时间片。当执行的时间片用完时， 由一个计时器发出时钟中断请求,调度  程序便据此信号停止该进程的执行，并将它送往就绪队列的末尾；然后再把处理机分配给就绪队列  中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程，在给定的  时间内，均能获得一时间片的处理机执行时间。   1. 实验源代码及说明   (1) 先来先服务 (FCFS) 与最短作业优先 (SJF)；  **首先代码如下：**  class Process {  int id;  int arrivalTime;  int serviceTime;  int finishTime;  int startTime;  int turnaroundTime;  public Process(int id, int arrivalTime, int serviceTime) {  this.id = id;  this.arrivalTime = arrivalTime;  this.serviceTime = serviceTime;  }  }  public class ProcessScheduler {  public static void main(String[] args) {  List<Process> processes = new ArrayList<>();  processes.add(new Process(1, 0, 5));  processes.add(new Process(2, 1, 3));  processes.add(new Process(3, 2, 8));  processes.add(new Process(4, 3, 6));  System.*out*.println("先来先服务（FCFS）调度结果：");  *fcfs*(processes);  System.*out*.println("最短作业优先（SJF）调度结果：");  *sjf*(processes);  }  public static void fcfs(List<Process> processes) {  for (Process process : processes) {  process.startTime = process.arrivalTime;  process.finishTime = process.startTime + process.serviceTime;  process.turnaroundTime = process.finishTime - process.arrivalTime;  System.*out*.println("进程ID: " + process.id + ", 开始运行时刻： " + process.startTime + ", 完成时刻： " + process.finishTime + ", 周转时间： " + process.turnaroundTime);  }  }  public static void sjf(List<Process> processes) {  processes.sort(Comparator.*comparingInt*(p -> p.serviceTime));  int currentTime = 0;  for (Process process : processes) {  if (currentTime < process.arrivalTime) {  currentTime = process.arrivalTime;  }  process.startTime = currentTime;  currentTime += process.serviceTime;  process.finishTime = currentTime;  process.turnaroundTime = process.finishTime - process.arrivalTime;  System.*out*.println("进程ID: " + process.id + ", 开始运行时刻： " + process.startTime + ", 完成时刻： " + process.finishTime + ", 周转时间： " + process.turnaroundTime);  }  }  }  **运行结果如下：**    **代码解析：**  (2) 非抢占式与抢占式优先权调度算法  public class CPUScheduling {  public static void main(String[] args) {  boolean judge2 = true; *//判断循环是否结束*  while (judge2) {  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("\*\*\*\*\*\*\*\*请先输入进程数量：\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  Scanner sc = new Scanner(System.*in*);  int processNum = sc.nextInt(); *// 进程数*  *// 进程的数组，记录进程的所有相关信息。构建多个是为了用于不同算法，避免修改原数组*  Process process[] = new Process[processNum];  Process process1[] = new Process[processNum];  Process process2[] = new Process[processNum];  Random r = new Random(); *// 采用随机数的方式来产生各项参数*  for (int i = 0; i < processNum; i++) {  int ran1 = r.nextInt(156); *// 优先级参数的随机数 100~255*  int ran2 = r.nextInt(9); *// 到达时间参数的随机数 1~9*  int ran3 = r.nextInt(9); *// 服务时间参数的随机数 1~9*  *// 进程相关信息的初始化*  process[i] = new Process(i, ran1 + 100, ran2 + 1, ran3 + 1);  process1[i] = new Process(i, ran1 + 100, ran2 + 1, ran3 + 1);  process2[i] = new Process(i, ran1 + 100, ran2 + 1, ran3 + 1);  }  boolean judge = true;*//判断循环是否结束*  while (judge) {  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("\*\*\*\*\*" + processNum + "个进程已经随机初始化成功！\*\*\*\*\*");  System.*out*.println("\*\*\*\*\*请输入数字，进行相关操作：\*\*\*\*\*\*");  System.*out*.println("\*\*\*\*\*数字1：非抢占短进程优先算法\*\*\*\*\*");  System.*out*.println("\*\*\*\*\*数字2：可抢占优先权调度算法\*\*\*\*\*");  System.*out*.println("\*\*\*\*\*数字3：重置进程相关信息\*\*\*\*\*\*\*\*\*");  System.*out*.println("\*\*\*\*\*数字4：退出程序\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  int num = sc.nextInt(); *// 输入数字*  if (num == 1) {  *// 调用非抢占短进程优先算法*  compute t = new compute(); *// 线程*  t.start();  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*即将调用算法！\*\*\*\*\*\*\*\*\*\*\*");  while(t.isAlive()){}  long starttime = System.*currentTimeMillis*();  *NSP*(process);  System.*out*.println("非抢占短进程优先算法运行时间：");  System.*out*.println(System.*currentTimeMillis*() - starttime +"ms");  compute t1 = new compute(); *// 线程*  t1.start();  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*算法运行结束！\*\*\*\*\*\*\*\*\*\*\*");  while(t1.isAlive()){}  } else if (num == 2) {  *// 调用可抢占优先权调度算法*  compute t = new compute(); *// 线程*  t.start();  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*即将调用算法！\*\*\*\*\*\*\*\*\*\*\*");  while(t.isAlive()){}  long starttime = System.*currentTimeMillis*();  *PPS*(process1);  System.*out*.println("可抢占优先权调度算法运行时间：");  System.*out*.println(System.*currentTimeMillis*() - starttime +"ms");  compute t1 = new compute(); *// 线程*  t1.start();  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*算法运行结束！\*\*\*\*\*\*\*\*\*\*\*");  while(t1.isAlive()){}  } else if (num == 3) {  compute t = new compute(); *// 重置线程*  t.start();  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*即将重置！\*\*\*\*\*\*\*\*\*\*\*\*\*");  while(t.isAlive()){}  break;  } else if (num == 4) {  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*退出成功！\*\*\*\*\*\*\*\*\*\*\*\*\*");  judge2 = false;  judge = false;  break;  } else {  System.*out*.println("输入的数字信息有误，建议重新输入");  }  }  }  }  */\*\**  *\* process：存放进程相关信息的数组*  *\* \*/*  public static void NSP(Process process[]){  int time = 1; *//本次算法当前正在运行的时间片，会随着时间不断增加*  int finish = 0; *//记录进程完成的数量，当完成数量=进程数量，就证明所有进程运行完毕*  int queueFinish = 0; *//记录进程进入队列的数量，当进入数量=进程数量，就证明所有进程进入队列*  Queue queue = new Queue(); *//就绪队列：每次循环按照服务时间由小到大顺序的存放已经到达进程。*  Queue queue2 = new Queue(); *//输出队列：所有已经完成的进程就会被存入到该队列中。*  int index = -1; *//记录这个短进程的下标*  Process firstProcess = new Process(); *//记录就绪队列中的队首进程*  System.*out*.println("首先展示进程初始化后的基本信息");  System.*out*.println("进程ID\t进程优先级\t进程到达时间\t进程服务时间");  for(int i = 0; i < process.length; i++){  System.*out*.println(" "+process[i].id+"\t\t"+process[i].priority+"\t\t\t"+process[i].arriveTime+"\t\t\t"+process[i].serviceTime);  }  *//为了后续算法的方便，直接对所有进程按照服务时间大小排序*  *//排序之后，只要该进程是当前时间片，那么按顺序存放到队列中就必然是从小到大的顺序*  *//采用冒泡排序法*  Process processTemp = new Process(); *//临时变量*  boolean flag = false; *// 标识变量，表示是否进行过交换*  for(int j = 0; j < process.length - 1; j++){  for (int k = 0; k < process.length - 1 - j; k++) {  *// 如果该进程的服务时间比后面进程的服务时间长，则交换*  if (process[k].serviceTime > process[k + 1].serviceTime) {  flag = true;  processTemp = process[k];  process[k] = process[k + 1];  process[k + 1] = processTemp;  }else if(process[k].serviceTime == process[k + 1].serviceTime){  *// 如果该进程的服务时间和后一个进程的服务时间相同，那么就比较优先级*  if(process[k].priority < process[k + 1].priority){  *//如果该进程优先级比上一个进程低那就交换，否则不需要作任何改动*  flag = true;  processTemp = process[k];  process[k] = process[k + 1];  process[k + 1] = processTemp;  }  }  }  if (!flag) { *// 在一趟排序中，一次交换都没有发生过*  break;  } else {  flag = false; *// 重置flag, 进行下次判断*  }  }  while(true){  *//因为之前已经对进程数组进行过服务时间的冒泡排序*  *//所以对process数组进行循环，寻找的第一个已经到达的进程必然是最短的。*  for(int i = 0; i < process.length; i++){  *//找到一个当前已经到达，且还没进入到队列当中的进程*  if(process[i].arriveTime <= time && process[i].queueNum == 0){  if(index != -1){  *//如果在循环过程中，找到了又一个满足条件的进程*  *//但是很可惜它无法现在进入到就绪队列中，因为index获取到了上一个短进程*  *//因此它的等待时间需要增加*  process[i].waitTime++;  }else{  *//获取它的下标*  index = i;  *//该进程已经进入队列*  process[i].queueNum = 1;  queue.in(process[i]);  *//进入队列数+1*  queueFinish++;  }  }  }  *//如果没有找到进程*  if(index == -1){  *//情况1，所有进程进入队列，直接继续向下运行即可*  if(queueFinish == process.length){  *//向下运行*  }else if(queue.list.size() == 0){  *//情况2，仍有进程没到达，且就绪队列中也没有进程*  System.*out*.println("在当前第"+time+"个时间片，没有进程到达且就绪队列中没有进程！");  time++;  compute t = new compute(); *// 线程*  t.start();  while(t.isAlive()){}  continue;  }  }  *//运行到这里，证明就绪队列的队首元素就是当前时间下的最短进程*  *//获取队首进程*  firstProcess = queue.get(0);  firstProcess.state = "R"; *//设置进程为运行状态*  compute t = new compute(); *// 线程*  t.start();  while(t.isAlive()){}  *//如果队首进程的已经服务时长=进程所需服务时长，证明该进程已经完成任务*  *//在这里提前设置一个，是为了让输出看起来更合理*  if(firstProcess.hasServiceTime == firstProcess.serviceTime){  firstProcess.state = "F"; *//设置进程为完成状态*  firstProcess.finishTime = time; *//记录进程完成的时间*  }  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("在当前第"+time+"个时间片，进入到就绪队列的进程如下：");  System.*out*.println("进程ID\t进程优先级\t进程到达时间\t进程服务时间\t进程状态\t进程已经服务时间\t进程完成时间\t进程等待时间");  queue.outputQueueAll(time);  *//如果队首进程的已经服务时长=进程所需服务时长，证明该进程已经完成任务*  if(firstProcess.hasServiceTime == firstProcess.serviceTime){  finish++;  queue2.in(queue.out()); *//将完成的进程放入到完成队列中*  if(finish == process.length){  *// 如果进程完成数量等于进程数量，就证明所有进程已经完成工作*  break;  }  for(int a = 1; a < queue.list.size(); a++){  *//让除了队首进程以外，所有进入就绪队列的进程，其等待时间+1*  queue.get(a).waitTime++;  }  if(queue.list.size() != 0){  *//如果队列中还存在着下一个进程*  *//上一个进程已经执行完毕，下一个进程开始执行*  queue.get(0).hasServiceTime++; *//已经服务时间增加*  }  time++; *//时间增加*  index = -1; *//初始化*  *//运行到这里证明还有进程没完成操作，因此进入下一次循环，避免进行下面的初始化*  continue;  }  for(int a = 1; a < queue.list.size(); a++){  if(firstProcess.hasServiceTime + 1 == firstProcess.serviceTime && a == 1){  *//如果下一回合队列首部的进程完成任务，那么队首后的第一个进程等待时间就不需要增加了*  continue;  }  *//让除了队首进程以外，所有进入就绪队列的进程，其等待时间+1*  queue.get(a).waitTime++;  }  firstProcess.hasServiceTime++; *//已经服务时间增加*  time++; *//时间增加*  index = -1; *//初始化*  }  compute t = new compute(); *// 线程*  t.start();  while(t.isAlive()){}  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("在当前第"+time+"个时间片，所有进程都执行完毕：");  System.*out*.println("进程ID\t进程优先级\t进程到达时间\t进程服务时间\t进程状态\t进程已经服务时间\t进程完成时间\t进程等待时间");  queue2.outputQueueAll(time);  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("算法结束，开始计算平均周转时间和平均等待时间");  compute t1 = new compute(); *// 线程*  t1.start();  while(t1.isAlive()){}  *outputTime*(process);  }  */\*\**  *\* process：存放进程相关信息的数组*  *\* \*/*  public static void PPS(Process process[]){  int time = 1; *//记录当前正在运行的时间片*  int finish = 0; *//记录进程完成的数量*  int priorityT = 0; *//记录哪个优先级最高*  int index = -1; *//记录这个优先级最高进程的下标*  System.*out*.println("首先展示进程初始化后的基本信息");  System.*out*.println("进程ID\t进程优先级\t进程到达时间\t进程服务时间");  for(int i = 0; i < process.length; i++){  System.*out*.println(" "+process[i].id+"\t\t"+process[i].priority+"\t\t\t"+process[i].arriveTime+"\t\t\t"+process[i].serviceTime);  }  while(true){  *// 因为是可抢占优先权调度算法，所以每次循环都是一个时间片内做的操作*  *// 每次循环都需要找到当前优先级最高的进程*  for(int i = 0; i < process.length; i++){  *//找到一个当前已经到达，且还没有运行完成的进程*  if(process[i].arriveTime <= time && process[i].state != "F"){  *// 如果该进程的优先级比上一个进程的优先级大*  if(process[i].priority > priorityT){  index = i; *//记录这个下标*  priorityT = process[i].priority; *//记录这个优先级*  }else if(process[i].priority == priorityT){  *// 如果该进程的优先级和上一个进程的优先级相同，那么就比较哪个进程服务时间短*  if(process[i].serviceTime < process[index].serviceTime){  *//如果该进程服务时间比上一个进程小那就修改，否则不需要作任何改动*  index = i; *//记录这个下标*  }  }  }  }  if(index == -1){  compute t = new compute(); *// 线程*  t.start();  while(t.isAlive()){}  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  *//如果index为-1，也就意味着当前没有进程到达*  System.*out*.println("当前时间片"+time+"没有进程到达！");  time++;  continue;  }  compute t = new compute(); *// 线程*  t.start();  while(t.isAlive()){}  process[index].state = "R"; *//当前进程开始运行*  *outputContent*(process, time);  process[index].hasServiceTime++; *//服务时间增加*  if(process[index].hasServiceTime == process[index].serviceTime){  *//如果相等，也就意味着该进程已经运行完毕*  finish++;  process[index].state = "F"; *//当前进程运行完成*  *//动态优先级：让本次时间片内除运行进程外的所有进程优先级上调，以便给其他进程机会*  for(int a = 0; a < process.length; a++){  *//只要该进程未完成，且还没到达，并且该进程不是正在运行的进程*  if(process[a].state != "F" && a != index && process[a].arriveTime <= time){  *//那么 优先级上调*  process[a].priority += 3;  }  }  time++; *//时间增加*  process[index].finishTime = time; *//当前进程的完成时间就是当前时间片*  priorityT = 0; *//初始化*  index = -1; *//初始化*  if(finish == process.length){  compute t1 = new compute(); *// 线程*  t1.start();  while(t1.isAlive()){}  *// 如果完成数量等于进程数量，就证明所有进程已经完成工作*  *outputContent*(process, time);  break;  }  continue;  }  *//动态优先级：让本次时间片内除运行进程外的所有进程优先级上调，以便给其他进程机会*  for(int a = 0; a < process.length; a++){  *//只要该进程未完成，且还没到达，并且该进程不是正在运行的进程*  if(process[a].state != "F" && a != index && process[a].arriveTime <= time){  *//那么 优先级上调*  process[a].priority += 3;  }  }  time++;  priorityT = 0; *//初始化*  process[index].state = "W"; *//初始化，因为下一次循环不一定还是该进程运行*  index = -1; *//初始化*  }  compute t2 = new compute(); *// 线程*  t2.start();  while(t2.isAlive()){}  System.*out*.println("算法结束，开始计算平均周转时间和平均等待时间");  *outputTime*(process);  }  */\*\**  *\* outputContent：在运行过程中每进行一次所需要输出的内容，并作出相关修改操作。专门用于PPS算法。*  *\* 使用到的相关参数如下：*  *\* process：存放进程相关信息的数组*  *\* time：当前经历了多少时间片*  *\* \*/*  public static void outputContent(Process process[], int time) {  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("经过了第"+time+"个时间片后，各进程的状态如下：");  System.*out*.println("进程ID\t进程优先级\t进程到达时间\t进程服务时间\t进程状态\t进程已经服务时间\t进程等待时间\t进程完成时间");  for(int i = 0; i < process.length; i++){  *//当某进程处于等待阶段，且已经到达，等待时间才会增加*  if(process[i].state=="W" && process[i].arriveTime <= time){  process[i].waitTime++;  }  System.*out*.println(" "+process[i].id+"\t\t\t"+process[i].priority+"\t\t\t"+process[i].arriveTime+"\t\t\t"+process[i].serviceTime+  "\t\t\t"+process[i].state+"\t\t\t"+process[i].hasServiceTime+"\t\t\t"+process[i].waitTime+"\t\t\t"+process[i].finishTime);  }  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  }  */\*\**  *\* outputTime：用于计算平均周转时间*  *\* 周转时间 = 完成时间 - 到达时间*  *\* 平均周转时间 = 周转时间 / 进程个数*  *\* 使用到的相关参数如下：*  *\* process：存放进程相关信息的数组*  *\* \*/*  public static void outputTime(Process process[]) {  double time = 0; *//周转时间*  double waitTime = 0; *//等待时间*  for(int i = 0; i < process.length; i++){  time += process[i].finishTime - process[i].arriveTime;  waitTime += process[i].waitTime;  }  double averageTime = (time / process.length); *//平均周转时间*  double averageWait = (waitTime / process.length); *//平均等待时间*  System.*out*.println("算法的平均周转时间如下："+averageTime);  System.*out*.println("算法的平均等待时间如下："+averageWait);  }  }  *// 进程类*  class Process {  public int id; *// 进程ID*  public String state = "W"; *// 进程状态：F完成，W等待，R运行*  public int priority; *// 优先级*  public int arriveTime; *// 进程到达时间*  public int serviceTime; *// 进程服务时间*  public int finishTime; *//进程的完成时间*  public int waitTime; *//进程的等待时间*  public int hasServiceTime; *//进程已经被服务的时间 ->主要用于PPS算法，去判断该进程是否结束*  public int queueNum = 0; *//进程目前是否进入队列 -> 主要用于MFQS(多级反馈队列调度算法)和NSP算法，去判断该进程是否进入队列：0为没进入，1为进入*  public Process() {  }  public Process(int id, int priority, int arriveTime, int serviceTime) {  this.id = id;  this.priority = priority;  this.arriveTime = arriveTime;  this.serviceTime = serviceTime;  }  }  *//线程类：用于实现停顿效果*  class compute extends Thread {  public void run() {  try {  *sleep*(2500);  } catch (Exception e) {  }  }  }  *//使用集合定义一个队列，用于实现先进先出效果*  class Queue {  List<Process> list = new ArrayList<Process>();  int index = 0; *//下标*  int num = 0; *//队列编号*  int q; *//时间片长度*  *//入队*  public void in(Process n){  list.add(n);  index++;  }  *//出队*  public Process out(){  if(!list.isEmpty()){  index--;  return list.remove(0);  } else{  Process n = new Process();  n.id = -1; *// 如果id为-1就证明，队列为空*  return n;  }  }  *//获取当前队列中的相应位置的进程*  public Process get(int n){  return list.get(n);  }  *//输出当前队列全部信息*  public void outputQueueAll(int time){  for(int i = 0; i < list.size(); i++){  Process process = list.get(i);  System.*out*.println(" "+process.id+"\t\t\t"+process.priority+"\t\t\t"+process.arriveTime+"\t\t\t"+  process.serviceTime+"\t\t\t"+process.state+"\t\t\t"+process.hasServiceTime+"\t\t\t"+  process.finishTime+"\t\t\t"+process.waitTime);  }  }  }  **运行结果如下：**  ①非抢占式的      ②抢占式的        (3) 基本时间片轮转调度算法  public void RR() {*//时间片轮转调度算法*  ProcessQueue pq = new ProcessQueue();  pq.EnqueueLast();  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*时间片轮转调度算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  while (!link.isEmpty()) {  pq.DisplayQueue();*//打印当前队列中的进程*  pq.Dequeue();*//出队，一次一个，因为上一轮出的得让刚到达的进程先进队列，所以没办法，进队操作只能也放在这个函数里了*  }  }  class ProcessQueue {  int k = 0;*// jcb中的进程遍历时的下标*  int nowTime = 0;*// 当前时间*  double sliceTime;*//轮转调度时间片*  int i = 0;*//记录当前出入队列的次数*  public void EnqueueLast() {*//进程首次入队，可一次进多个,从队尾进入*  while (k < jcb.size()) {*//当遍历完jcb中的所有进程时结束*  if (jcb.get(k).arriveTime <= nowTime) {*//已经到达的进程按到达时间先后进入队列*  link.addLast(jcb.get(k));  k++;  } else {  break;*//如果该进程还未入队，即先结束遍历，保留当前下标k值，注意：此处不要k--；*  }  }  }  public void EnqueueFirst() {*//进程首次入队，可一次进多个,从队首进入*  while (k < jcb.size()) {*//当遍历完jcb中的所有进程时结束*  if (jcb.get(k).arriveTime <= nowTime) {*//已经到达的进程按到达时间先后进入队列*  link.addFirst(jcb.get(k));  k++;  } else {  break;*//如果该进程还未入队，即先结束遍历，保留当前下标k值，注意：此处不要k--；*  }  }  }  public void Dequeue() {*//进程出队，一次只出一个*  nowProess = link.removeFirst();*//移除队列的队首元素并且返回该对象元素*  nowProess.beginTime = nowTime;*//计算开始时间，即为上一个进程的结束时间*  nowProess.finshTime = nowProess.beginTime + nowProess.serveTime;*//计算结束时间，该进程开始时间+服务时间*  nowProess.roundTime = nowProess.finshTime - nowProess.arriveTime;*//计算周转时间*  nowProess.aveRoundTime = (double) nowProess.roundTime / nowProess.serveTime;*//计算平均周转时间*  nowTime = nowProess.finshTime;*//获得结束时间，即当前时间，方便判断剩下的进程是否已到达*  new\_jcb.add(nowProess);*//经处理过数据后加入new\_jcb容器*  for (int i = 0; i < link.size(); ++i) {  link.get(i).waitTime++;*//所有进入等待队列的进程等待时间+1,此处只为最高响应比算法所用*  }  }  public void Dequeue(double sliceTime) {*//重载Dequeue方法，实现轮转调度算法的出队*  nowProess = link.removeFirst();*//移除队列的队首元素并且返回该对象元素*  if (nowProess.firstTimeTag == false) {  */\*轮转调度进程可能会多次反复进出队列，不像FCFS和SJF的进程只会进出一次，所以计算开始时间可以设个标志位，让每个进程在*  *\* 第一次执行时记录一遍即可\*/*  nowProess.beginTime = nowTime;*//进程开始执行的时间*  nowProess.firstTimeTag = true;*//计算第一次即可，下次无需更新计算*  }  nowTime += sliceTime;*//每次出队，用时一个时间片，更新当前时间*  nowProess.clock += sliceTime;*//更新当前出队列的进程已服务时间*  if (nowProess.clock >= nowProess.serveTime) {  nowProess.finshTime = nowTime;*//计算该进程完成时间*  nowProess.roundTime = nowProess.finshTime - nowProess.arriveTime;*//计算周转时间*  nowProess.aveRoundTime = (double) nowProess.roundTime / nowProess.serveTime;*//计算平均周转时间*  new\_jcb.add(nowProess);*//经处理过数据后加入new\_jcb容器*  EnqueueFirst();*//已到达的进程先入队*  } else {  EnqueueFirst();*//已到达的进程先入队*  link.addLast(nowProess);*//上一轮出的再紧接着进入队尾*  }  }  public void DisplayQueue() {*//队列中等候的进程*  i++;  System.*out*.println("第" + i + "次队列中排队的进程：" + link);  }  }  **运行结果如下:**     1. 实验结果及分析   (1) 先来先服务 (FCFS) 与最短作业优先 (SJF)；  首先创建了一个Process类，用于表示进程。这个类包含了进程ID、到达时间、服务时间和完成时间等属性。然后实现了两种调度算法：先来先服务（FCFS）和最短作业优先（SJF）。**在FCFS算法中**，我们按照作业/进程进入系统的先后次序进行调度，即启动等待时间最长的作业/进程。**在SJF算法中**，我们从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。  最后，编写了一个主程序来模拟进程等待过程，并输出每个进程的开始运行时刻、完成时刻和周转时间（此代码的开始时间和服务时间是在主方法中给定的，运行结果与老师上课所理解的内容一致）  (2) 先来先服务 (FCFS) 与最短作业优先 (SJF)；  此实验的代码过长，详细的实验解析以及注解已经在上述代码中演示，具体总结两种方式的实验的分析以及个人总结：  非抢占式优先权调度算法：在非抢占式优先权调度算法下，高优先级的进程能够快速得到执行，但低优先级的进程可能会长时间得不到执行机会，导致饥饿现象。平均周转时间较短，但吞吐量较低。  抢占式优先权调度算法：在抢占式优先权调度算法下，高优先级的进程可以抢占低优先级进程的CPU资源，避免了饥饿现象。平均周转时间较长，但吞吐量较高。  综合分析：非抢占式优先权调度算法简单易实现，适用于系统对实时性要求较高的场景；而抢占式优先权调度算法能够更好地利用CPU资源，适用于系统对吞吐量要求较高的场景。  (3) 先来先服务 (FCFS) 与最短作业优先 (SJF)；  代码分析：  在主函数中，创建了一个ProcessQueue对象pq，并调用了EnqueueLast()方法将进程加入队列。然后进入循环，不断调用Dequeue()方法进行进程调度，直到队列为空。在每次调度后，调用DisplayQueue()方法打印当前队列中的进程。  从运行结果来看，基本时间片轮转调度算法可以有效地管理进程的执行。每个进程都有一个服务时间，当进程到达时，将其加入队列。在调度过程中，每次从队列中取出一个进程进行处理，处理完后，根据剩余的服务时间判断是否需要将进程放回队列。如果剩余的服务时间小于时间片，则将进程放回队列，等待下一次调度；否则，继续处理下一个进程。这样可以保证每个进程都能得到一定的执行时间，从而实现进程调度的目的。 | | | | | | | | |