|  |  |  |
| --- | --- | --- |
| 南农大  **计算机操作系统课程设计**  **测试分析与评价报告**  **（B成绩等级）**  XH2 | | |
|  | **题 目:** | **仿真实现操作系统作业管理、内存管理以及SPOOLing系统** |
|  | **姓 名:** | **李彦墨** |
|  | **班级专业:** | **计科211** |
|  | **学 号:** | **30221228** |
|  | **自评成绩：** | **B** |
|  | **助 教：**  **指导教师:** | **夏烨、杨鹏 类型: 大四本科生**  **姜海燕 职称: 教授** |
| **2024 年3月25 日**  **南京农业大学人工智能学院** | | |

**注意：**

**1.此文件适用于申请B成绩等级的申请人及评价人。**

**2.申请人请根据要求及评分标准，完成每项测试内容并申请分数，每项测试需按要求文字论述，并录制视频讲解文件，否者不认可。**

**3.评价人请根据要求及评分标准以及自测人所提供的申请成绩、理由和讲解文件，结合程序代码，评价自测人的每项成绩。**

**4.此报告保存到申请人提交材料文件夹的根目录，测试讲解视频文件按照要求保存到test-vidio子文件夹.每项申请成绩填写到“申请人申请分数统计表”。**

**5.评阅人分数统计表**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1.作品说明（20分）** | | | | **2.基础功能测试（共63分）** | | | | | | | | | | | | | | | |
| 1.1 | 1.2 | 1.3 | | 2.1 | 2.2 | 2.3 | | 2.4 | 2.5 | 2.6 | | 2.7 | 2.8 |  | | |  |  |  |
|  |  |  | |  |  |  | |  |  |  | |  |  |  | | |  |  |  |
| **3.问题描述与分析（共8分）** | | | | | |  | | | | | **手写签名图片** | | | | |  | | | |
| **申请成绩等级** | | |  | | | | **总成绩** | | | | | | | |  | | | | | |

**6.评价人信息：**

# 1.课设作品（共20分）

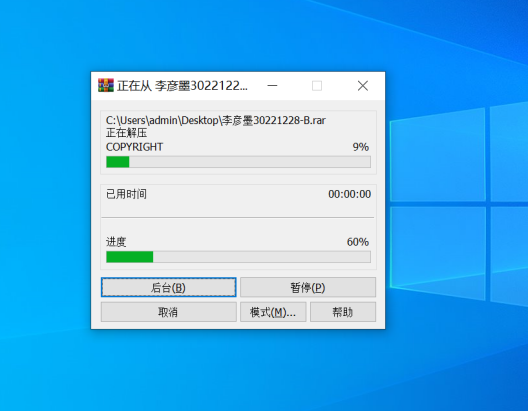
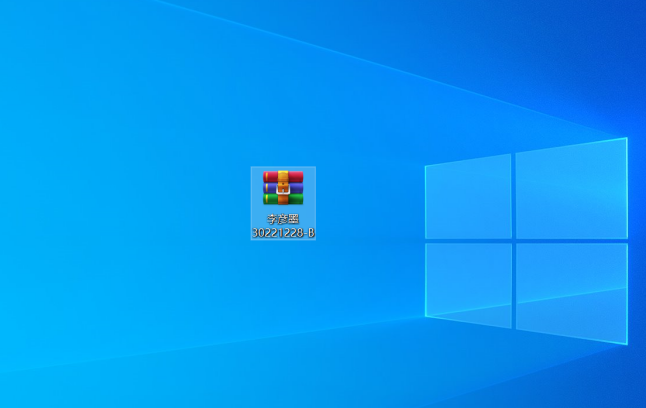
## 1.1非开发环境下可执行程序的安装使用说明（5）

1. **请在学校提供的非开发环境下（没安装过Java开发环境）的机器上，面向普通用户安装可执行程序的步骤，所需要的相关安装包等内容；**
2. **安装使用说明视频文件请保存到test-vidio子文件夹，文件名：1-非开发环境下安装使用说明；**

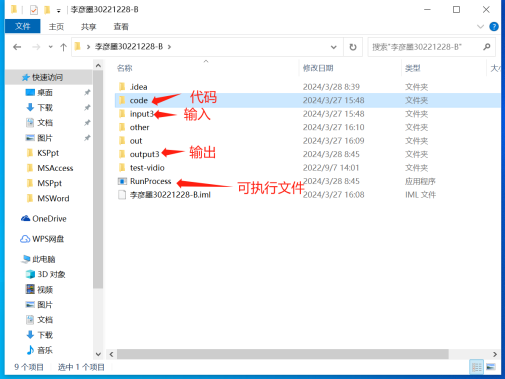
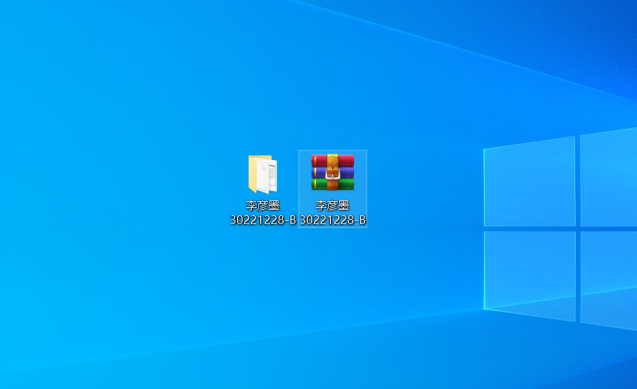
**【评分标准：完整详细，使用者可重复，得5分；否者计0分】**

申请分数：5

安装步骤总结（配证据图）：



首先，将安装压缩包进行解压到当前文件夹（桌面），获得程序总文件夹。



其次，打开解压得到的文件夹，里面分别有源代码和输入输出的文件夹，以及可执行文件RunProcess.exe。



双击可执行文件夹，正常打开运行窗口。点击开始按钮，程序开始正常运行。

**评价分数：**

**理由（文字不少于10字，配证据图）：**

## 1.2说明开发环境下工程程序及源程序文件（5）

**（1）在Java或者C#开发环境下，描述工程文件、源程序等功能说明**

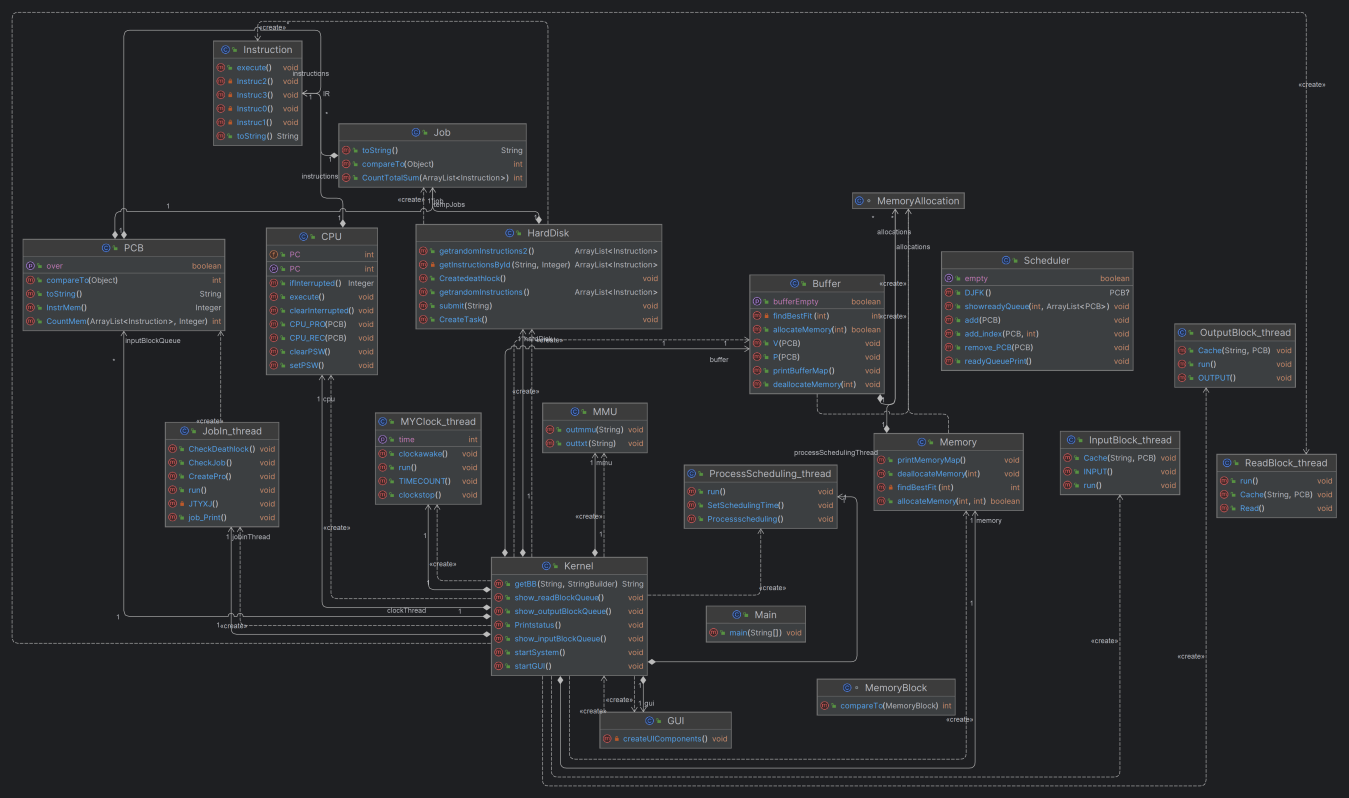
**（2）视频说明文件保存到test-vidio子文件夹，文件名：2-工程程序及源程序文件说明**

**【评分标准：完整详细，得5分；否者计0分】**

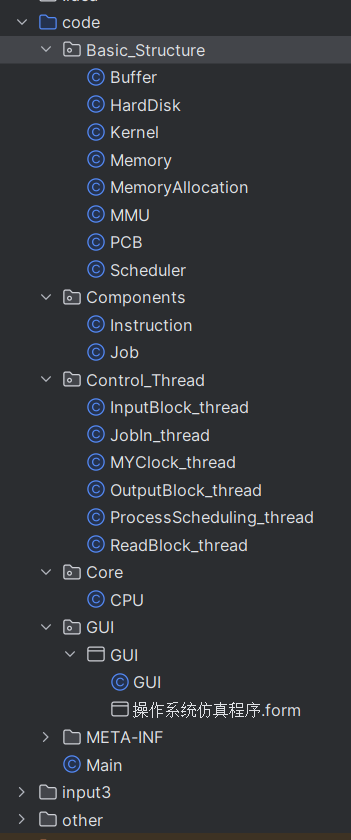
申请分数：5

说明工程文件如何组织其他相关源程序文件的步骤？（配证据图）

说明本课设相关源文件功能、源文件内部组织结构？源文件之间相互关系？（配证据图）：



IDEA生成的类图

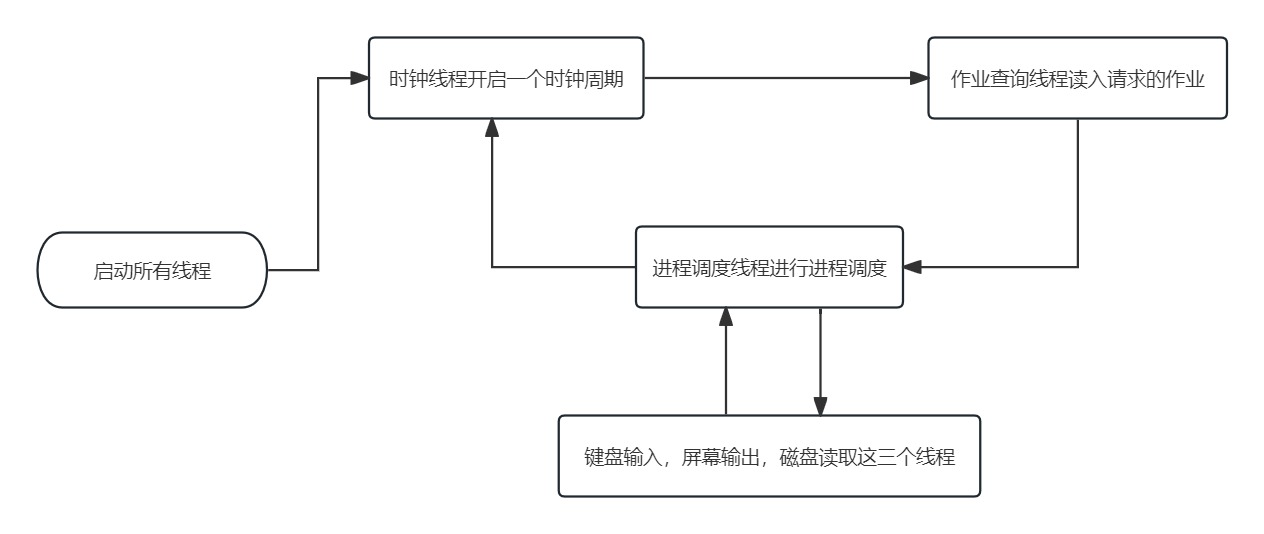


源程序代码文件

在本代码文件下，有20个类组成。其中6个类继承了Runnable接口，为本次操作系统课设体系之核心。又有Job，PCB，Instruction类作为操作的元类型。Kernel，GUI和Main函数提供了开启、暂停、关闭程序的接口。其余几个类均为硬件和方法的仿真。

首先这六个线程的顺序为MYClock\_Thread，JobIn\_Thread和ProcessSchedule\_Thread的互相循环，而在InputBlock\_Thread，OutputBlock\_Thread，ReadBlock\_Thread上也形成了一个中断小循环，但他们的起点和终点都是ProcessSchedule\_Thread。

HardDisk模拟硬盘读入目录下的txt数据，将他先转化为Job类，Job类通过后备队列，就绪队列的操作，将Job类转化为PCB类，而在内存申请时，也需要用到Memory相关的两个类-Memory，MemoryAllocation这两个类进行内存分配。在进程分配时，Schedule类作为三级队列的抽象，具有配合调度和展示的作用。在进程调度到指令后，将采用CPU类进行运行进程。而在中断的三个线程中，进入和读出阻塞态的过程需要用到缓冲区Buffer类和PV操作。MMU类支持将运行的结果输出。



**评价分数：**

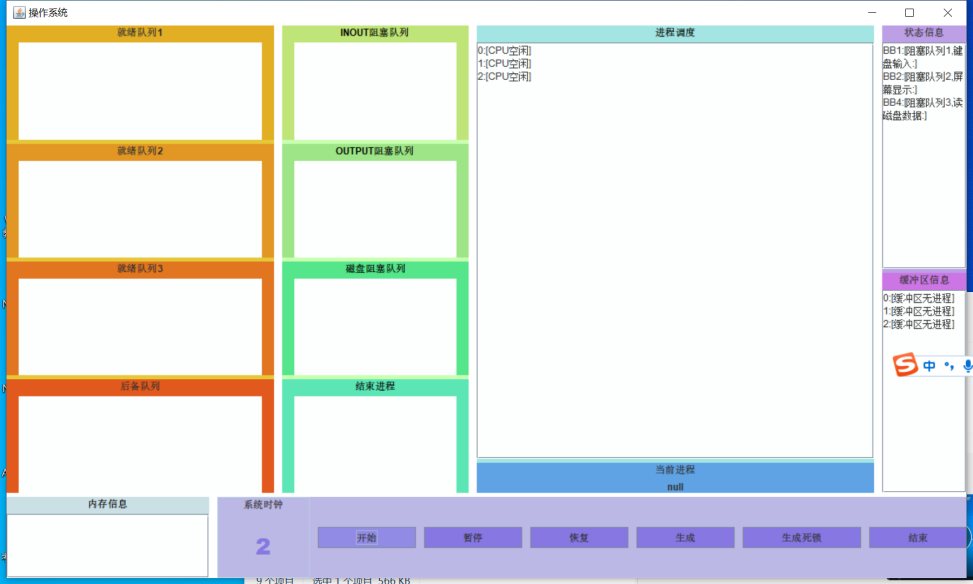
**理由：**

## 1.3 运行可执行程序测试结果（10）

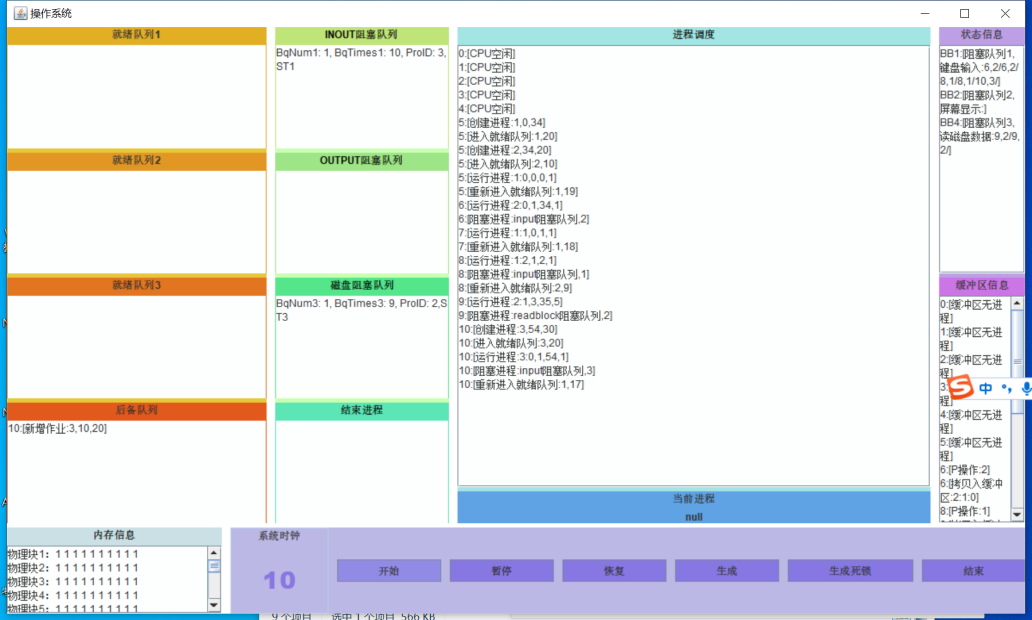
**【评价标准：共10分，程序可独立执行，按要求对结果逐行说明，并视频讲解清晰，得满分。不可运行计0分。其他情况，举证给出得分依据】**

申请分数：10

（1）输入教师提供的测试数据，在机房PC机上运行可执行程序，生成ProcessResults-？？？-算法名称代号.txt文件，给出每步执行过程截图，并对屏幕每步输出内容具体文字说明。



时钟为2，刚开始运行时，进程调度区CPU空闲，此时没有进程读入，缓冲区无操作，状态信息也无操作。



时钟为10时，开始有进程读入，进程调度区域开始出现作业以及作业的运行，进程的阻塞。阻塞队列上开始显现被阻塞的进程，缓冲区开始有了PV操作，内存物理块被占据。



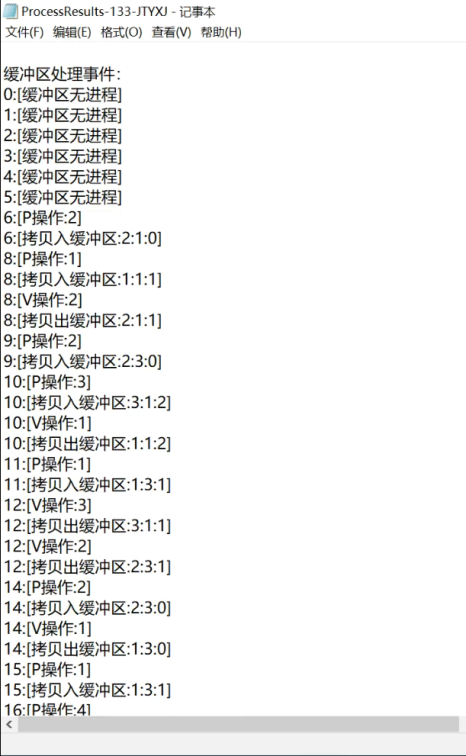
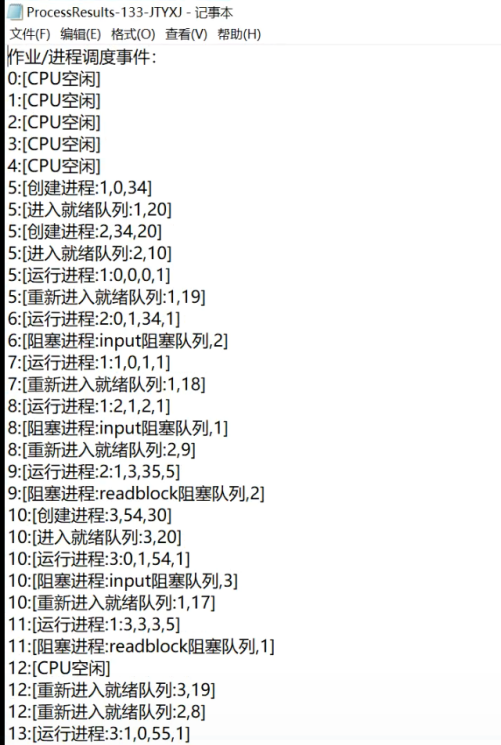
运行到中段时，三级就绪队列开始陆续有进程进入，缓冲区和物理块的进程被占用，此时出现了因内存满了，新进程无法进入就绪队列的情况。

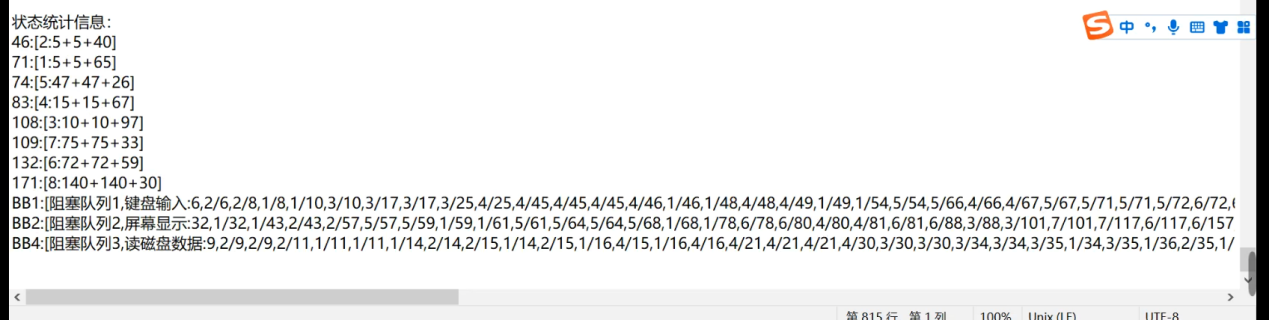


快结束前，结束进程开始在屏幕上显示，屏幕上显示大量运行痕迹。



132时全部进程运行结束，所有队列和缓冲区空闲，结束进程上记录着所有进入且已执行完的进程。





这里是output显示的输出，包括了作业进程调度，缓冲区处理事件以及状态统计信息。

（2）对以上过程进行分步演示讲解。对所生成ProcessResults-？？？-算法名称代号.txt文件的内容，给出逐行讲解。视频说明文件保存到test-vidio子文件夹，文件名：3-可执行程序运行及结果分析。

**评价分数：**

**理由：**

# 2.基础功能测试（共63分）

## 2.1 时钟中断线程运行单步测试（5）

【评价标准：共5分，执行正确，文字论述清晰具体，过程原理讲解清晰，得满分。不可打单步执行或者内容不正确计0分。其他情况，举证给出得分依据】

申请分数：5

（1）给出MYClock\_thread时钟中断线程类TIME-COUNT（）及相关数据结构伪码，并逐行说明实现原理；

MYClock\_thread类实现了一个时钟线程的功能。时钟线程类负责模拟系统的时钟，并通过TIMECOUNT函数更新系统时钟的值。时钟线程类的原理是通过一个静态变量COUNTTIME来记录系统时钟的值，并通过TIMECOUNT函数对COUNTTIME进行更新。TIMECOUNT函数每秒更新一次时钟值，并使用Thread.sleep()来使每次更新时钟时暂停一段时间，表示系统时钟的速度。

1. **public** **class** MYClock\_thread **implements** Runnable { // Clock thread
2. **public** **static** **int** COUNTTIME = 0; // 记录系统时钟
3. **public** **static** **int** simulationSecond = 500; // 模拟时钟更新时间，此处设置为 0.5 秒
4. **public** **void** run() {  // 开始运行线程
5. **while** (**true**){  // 保持线程运行
6. // ... 激活下一个线程
7. // ... 当前进程等待
8. TIMECOUNT();  // 计时函数
9. }
10. }
12. **public** **void** TIMECOUNT() {
13. COUNTTIME++; // 系统时钟+1更新
14. Thread.sleep(simulationSecond); // 休眠特定时间，表示系统时钟更新时间
15. }
16. }

（2）通过在相关程序内设置断点单步运行**MYClock\_thread**时钟中断线程；跟踪显示时钟COUNTTIME变量值的变化情况；录制视频文件讲解该操作过程和结果。视频文件名：4-时钟中断线程单步运行测试

**评价分数：**

**理由：**

## 2.2 作业请求查询单步测试（5）

【评价标准：共5，执行正确，文字论述清晰具体，过程原理讲解清晰，得满分。不可打单步执行或者内容不正确计0分。其他情况，举证给出得分依据】

申请分数：5

（1）给出作业请求查询操作CheckJob（）及相关数据结构伪码，以及每5秒激活该线程的过程代码，并逐行说明实现原理

在此前的读入模块中，系统已经将开始阶段一次性读入的Job放在一个内核的外部读入队列中，CheckJob（）函数将外部读入队列中的最顶端作业（即最早到达的作业）在到达他的作业请求时间时，读入到后备队列中，并通过while循环检查是否有遗漏的作业。

5s激活则采用了if语句进行判断，当时钟时间为5的倍数时，激活一次检查作业。

1. **public** **void** run() {  //启动线程
2. **while** (**true**) {  //保持线程运行
3. // 当前线程等待
4. **if** (MYClock\_thread.getTime() % 5 == 0) {  //每隔5s进行一次激活
5. CheckJob();  //检查新作业函数
6. }
7. // 调取后备队列创建进程
8. // 激活下一线程
9. }
10. }
11. **public** **static** PriorityQueue<Job> HB=**new** PriorityQueue<>();  //定义在Kernel里的后备队列（优先队列为模板，自动实现优先队列调度）
12. **public** **void** CheckJob() {
13. **while** (外部读入队列非空 && 外部读入队列的首元素的InTimes == 当前系统时间) {  //有新进程请求到达，此时后备队列可以读取
14. Kernel.HB.add(HardDisk.jobFallbackQueue.poll());  //将队首作业加入后备队列
15. }
16. }

（2）以2.txt为例，设置断点单步运行跟踪作业请求查询操作CheckJob（），显示每5秒激活操作的过程。并录制视频文件讲解上述操作过程和结果。视频文件名：5-作业请求中断单步测试

**评价分数：**

**理由：**

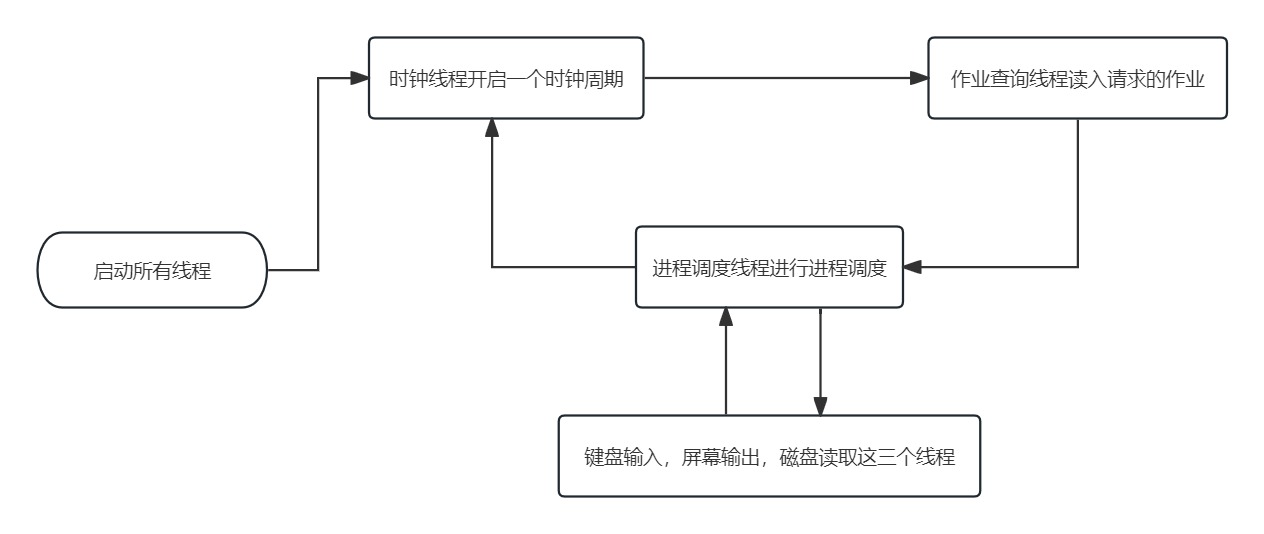
## 2.3 进程调度线程同步控制单步测试（10）

【评价标准：共10，执行正确，文字论述清晰具体，过程原理讲解清晰，得满分。不可打单步执行或者内容不正确计0分。其他情况，举证给出得分依据】

申请分数：10

1. 给出进程调度线程与时钟中断线程同步操作的伪码，并这个线程与时钟中断线程同步过程的代码，并逐行逐变量说明实现原理；

进程调度线程与时钟线程的同步是围绕时钟线程-作业查询线程-进程调度线程这三个线程的同步循环调用产生的，时钟线程作为最开始的计时线程，在一个时钟周期里实现一次线程循环，以实现在这个时钟里同步操作的目的。



在最开始启动Kernel的时候，就创建同时三个线程，让他们同步启动，以实现同步操作。

1. **new** Thread(**new** MYClock\_thread()).start();  // 启动时钟中断线程
2. **new** Thread(**new** JobIn\_thread()).start();  // 启动作业查询线程
3. **new** Thread(**new** ProcessScheduling\_thread()).start();  // 启动进程调度线程

进程调度线程与时钟线程的同步主要通过Condition方法进行的同步。Condition的执行方式，是当在线程中调用await方法后，线程将释放锁，并且将自己沉睡，等待唤醒，线程Producer获取到锁后，开始运行，完毕后调用的signal方法，唤醒沉睡的线程，线程恢复执行。开始的时候，时钟线程在await处激活，操作后唤醒下一线程并让自己休眠；作业查询线程同理，在await处激活并运行完后唤醒下一线程并让自己休眠，进程调度线程在await处被唤醒后运行进程调度，最后唤醒时钟线程让自己休眠，开启新一轮的时钟周期。因此，在同一轮时钟循环中，调用时钟的计时量COUNTTIME都视为在同一时间内操作。

1. **public** **static** Condition clkCondition = lock.newCondition();  // Condition对象用于时钟线程同步
2. **public** **static** Condition jitCondition = lock.newCondition();  // Condition对象用于作业请求线程同步
3. **public** **static** Condition pstCondition = lock.newCondition();  // Condition对象用于进程调度线程同步
5. **public** **class** MYClock\_thread **implements** Runnable { // 时钟线程
6. **public** **void** run() {  //启动线程
7. **while** (**true**) {  //线程自动循环
8. Kernel.lock.lock(); // 当前线程获得锁
9. Kernel.jitCondition.signal(); // 唤醒下一线程：作业请求查询线程
10. Kernel.clkCondition.await(); // 当前线程在此等待被唤醒
11. // 模拟时钟中断
12. }
13. }
14. }
16. **public** **class** JobIn\_thread **implements** Runnable { // 作业请求查询线程
17. **public** **void** run() {  //启动线程
18. **while** (**true**) {  //线程自动循环
19. Kernel.lock.lock(); // 当前线程获得锁
20. Kernel.jitCondition.await(); // 当前线程在此等待被唤醒
21. // 模拟作业请求查询
22. Kernel.pstCondition.signal(); // 唤醒下一线程：进程调度线程
23. }
24. }
25. }
27. **public** **class** ProcessScheduling\_thread **implements** Runnable {  // 进程调度线程
28. **public** **void** run() {  //启动线程
29. **while** (**true**) {  //线程自动循环
30. Kernel.lock.lock(); // 当前线程获得锁
31. Kernel.pstCondition.await(); // 当前线程在此等待被唤醒
32. // 模拟进程调度
33. Kernel.clkCondition.signal(); // 唤醒下一线程：时钟中断线程
34. }
35. }
36. }
37. 给出进程调度算法函数伪码，并逐行逐变量说明实现原理；

进程调度主要采用了多级反馈队列调度算法，它使用3个队列，每个队列具有不同的时间片大小。进程首先进入最高优先级的队列，如果它在一个时间片内没有执行完成，则被移动到下一个较低优先级的队列。这样做的好处是可以在保证高优先级进程及时执行的同时，给予低优先级进程一定的机会，防止低优先级进程长时间饥饿。

此外，系统中的每个进程具有不同的状态，比如就绪、运行、阻塞等。进程调度会根据当前进程的状态和调度算法来决定下一步执行哪个进程。例如，如果当前进程执行完毕，调度器会选择就绪队列中的下一个进程来执行；如果当前进程被阻塞，调度器可能会选择一个处于就绪状态的进程执行，或者等待当前进程解除阻塞后再执行。

1. **public** **class** ProcessScheduling\_thread **implements** Runnable {
2. If (Kernel.curPro != **null**){  // 如果当前有进程在处理机上
3. If (生了中断且当前指令是IO指令){
4. // 不执行任何操作，等待IO指令完成
5. **return**
6. }
7. If (当前进程在二级或三级队列执行，且未用完时间片，且没有发生中断，且当前进程未结束){
8. // 重新加入就绪队列
9. **return**
10. }
11. If (当前进程在一级队列执行，且时间片用完，且当前进程未结束)
12. // 重新加入就绪队列
13. **return**
15. If (如果当前进程指令执行完且未使用时间片){
16. Deallocate memory **for** Kernel.curPro
17. // 释放当前进程内存
18. Remove Kernel.curPro from PCB queue
19. // 从PCB队列中移除当前进程
20. // 递归调用进程调度
21. **return**
22. }
23. If (当前进程指令执行完且使用了时间片){
24. Deallocate memory **for** Kernel.curPro
25. // 释放当前进程内存
26. Remove Kernel.curPro from PCB queue
27. // 从PCB队列中移除当前进程
28. // 递归调用进程调度
29. **return**;
30. }
31. }
32. // 处理机上内容为空或已重新加入就绪队列
33. Kernel.curPro = DJFK()
34. // 从队列中选择一个进程
35. If Kernel.curPro is not **null** and Kernel.curPro is not over{
36. // 如果选择了一个有效进程
37. CPU.CPU\_REC(Kernel.curPro)
38. // 处理机开始执行进程
39. Scheduler.curTimeSlice = Kernel.curPro.curTimeSlice
40. // 设置当前时间片
41. Kernel.curPro.cur\_rest\_time = Kernel.curPro.curTimeSlice
42. // 设置当前进程剩余时间片
43. }
44. }
46. **public** **class** Scheduler { // 就绪队列类
47. **public** **static** PCB DJFK() {
48. **for** (**int** i = 0; i < 3; i++) { // 从第一级队列开始检查是否有进程
49. **if** (!queues[i].isEmpty()) { // 该队列非空
50. **return** queues[i].remove(0); // 从队列头部移除并返回进程
51. }
52. }
53. **return** **null**; // 如果所有队列都为空，则返回null
54. }

57. **public** **static** **void** add\_index(PCB pcb, **int** timeslice) { // 分配新的时间片
58. **int** x = 0; // 队列号（从0开始）
59. **int** y = 0; // 时间片大小
60. **if** (timeslice == 1) { // 从1级队列里来
61. x = 1; // 进入2级队列
62. y = 2; // 时间片大小为2
63. } **else** **if** (timeslice >= 2) { // 从2级队列里来
64. x = 2; // 进入3级队列
65. y = 4; // 时间片大小为4
66. }
67. // 分配pcb所在时间片y
68. queues[x].add(pcb); // 放入三级队列
69. }
71. **public** **static** **void** remove\_PCB(PCB pcb) { // 进程从队列中移除
72. **for** (**int** i = 0; i < 3; i++) { // 循环三级队列
73. **for** (PCB p : queues[i]) { // 查找匹配进程
74. **if** (pcb.ProID == p.ProID) { // 进程匹配
75. queues[i].remove(p); // 取出进程
76. **return**; // 返回
77. }
78. }
79. }
80. }
81. }

（3）设置断点单步运行跟踪进程调度线程，显示这个线程与时钟中断线程同步的过程；录制视频文件讲解上述操作过程和结果。视频文件名：6-进程调度线程激活单步测试

（4）设置断点单步运行跟踪2.txt和3.txt进程调度过程，具体讲解说明显示在屏幕上的输出信息；录制视频文件讲解上述操作过程和结果。视频文件名：7-进程调度单步测试

## 2.4 进程阻塞唤醒线程单步测试（10）

【评价标准：共10，执行正确，文字论述清晰具体，过程原理讲解清晰，得满分。不可打单步执行或者内容不正确计0分。其他情况，举证给出得分依据】

申请分数：10

1. 给出进程**阻塞**操作、调用阻塞过程伪码，并逐行逐变量说明实现原理；

进程阻塞是在执行完CPU后，当前进程的指令类型符合进入条件，且在当前时间片内没有多次进入，则进入该阻塞队列，并移除出就绪队列和当前进程。

1. **public** **static** ConcurrentLinkedQueue<PCB> BlockQueue = **new** ConcurrentLinkedQueue<>();  //阻塞队列
2. **public** **void** PUT() {
3. **if** (Kernel.curPro的当前指令类型为阻塞指令&&阻塞队列进入锁为**true**即此前这个时间片内未进入阻塞队列) {
4. //记录进入阻塞队列时间
5. //缓冲区P操作
6. Scheduler.remove\_PCB(Kernel.curPro);  //就绪队列中移除当前进程
7. //设置阻塞队列进入锁防止多次进入
8. Kernel.BlockQueue.add(Kernel.curPro); // 当前进程阻塞
9. Kernel.curPro = **null**;  //当前进程置空
10. }
11. }
12. 给出输入、显示等阻塞**唤醒**操作线程伪码，并逐行逐变量说明实现原理；

当阻塞队列中的等待进程完成设定的等待时间后，进程就从阻塞队列中释放。与此同时，需要完成PC+1，即取得下一条指令后再将当前指令放回就绪队列。在此期间需要保持CPU的同步来模拟内核态重现转变为用户态的过程。

1. **if** (Kernel.BlockQueue.peek() != **null** && Kernel.BlockQueue.peek().BlockQueue\_time + 2 <= MYClock\_thread.COUNTTIME) {  //阻塞队列非空且到达阻塞时间
2. PCB outpcb = Kernel.BlockQueue.poll();  //进程从队列中释放
3. //缓冲区V操作
4. Kernel.cpu.CPU\_REC(outpcb);  //将取出进程的指令同步到CPU
5. Kernel.cpu.PC+=1;  //CPU的指令+1，准备取下一条指令
6. **if**(Kernel.cpu.PC<outpcb.instructions.size()) {  //指令未达到上限时更新IR
7. Kernel.cpu.IR = outpcb.instructions.get(Kernel.cpu.PC);  //取新的指令
8. }
9. Kernel.cpu.CPU\_PRO(outpcb);  //CPU的PC和IR同步到新的进程
10. Scheduler.add(outpcb);  //取出进程返回就绪队列
11. }
12. 设置断点单步跟踪测试唤醒过程是否与进程调度线程并发；并通过1.txt测试讲解。录制视频文件讲解上述操作过程和结果。视频文件名：8-阻塞唤醒过程

## 2.5 实时操作单步执行测试（10）

【评价标准：共10，执行正确，文字论述清晰具体，过程原理讲解清晰，得满分。不可打单步执行或者内容不正确计0分。其他情况，举证给出得分依据】

申请分数：9

（1）给出按界面实时按钮产生操作**指令文件生成代码**，并逐行逐变量说明实现原理；

指令文件生成代码主要功能是模拟系统中指令的生成和处理过程。每当按钮被触发，会生成一个新的指令任务，并将其添加到作业队列中，以便后续的处理。这里模仿了读取txt并解析的代码过程，创建Job并用随机数生成类Random()生成一串指令序列。

1. **public** **void** actionPerformed(ActionEvent e) {  //按钮触发函数
2. Kernel.hardDisk.CreateTask();  //指令生成函数
3. }
4. **public** **static** ArrayList<Instruction> getrandomInstructions()
5. {
6. **int** i=1; //指令从1开始
7. ArrayList<Instruction> instructions = **new** ArrayList<>();  // 创建一个指令集合
8. **while**(i<=10) {   // 循环读取文件
9. Integer Instruc\_ID = **null**, Instruc\_State = **null**, Data\_Size = 0;  //初始化指令
10. Instruc\_ID=i;  //第i条指令
11. Random random = **new** Random();
12. Instruc\_State = random.nextInt(4);  // 生成 0-3 之间的随机数，为指令类型
13. // 3号指令随机生成新的Data\_Size
14. instructions.add(**new** Instruction(Instruc\_ID, Instruc\_State, Data\_Size));  // 将指令添加到集合中
15. i++;
16. }
17. **return** instructions;  // 返回指令集合
18. }
19. **public** **static** **void** CreateTask(){  //创建新指令函数
20. Job job = **new** Job(createtimes, MYClock\_thread.COUNTTIME, 0, 10, getrandomInstructions());  //生成新的随机指令作业
21. createtimes+=1;  //指令计数+1
22. jobFallbackQueue.add(job);  // 将新建的作业添加到作业后备队列
23. Kernel.Job\_count+=1;  //作业总数+1
24. }

（2）给出就绪队列**插入实时进程**的伪码，并逐行逐变量说明实现原理；

直接将未计算大小，未判断读入所需内存大小的进程插入就绪队列会产生死锁的现象，因此考虑将作业先放入后备队列，等到所分配空间和并发度都合适时再创建PCB，生成进程进入就绪队列。

1. **public** **void** CreatePro() {  //进程创建函数+添加到就绪队列
2. **while** (Kernel.currency < Kernel.MAX\_CURRENCY && !Kernel.HB.isEmpty()) {  //当前系统并发度小于最大并发度,作业后备队列非空/就绪队列中的进程数目一定小于最大并发度
3. **if**(Kernel.memory.allocateMemory(Kernel.HB.peek().JobsID,Kernel.HB.peek().TotalSum)){  //当前剩余空间可以为后备队列的作业分配空间
4. PCB pcb = **new** PCB(Kernel.HB.poll());  //取出后备队列的作业，生成新的PCB
5. Scheduler.add(pcb);  //加入就绪队列
6. }
7. **else**{
8. **break**;  //空间不足，直接退出
9. }
10. }
11. }

（3）在3.txt执行到第5条指令时按下实时作业请求按钮，设置断点单步跟踪生成与调度过程；录制视频文件讲解上述操作过程和结果。视频文件名：9-实时作业请求

## 2.6 连续动态内存分配与回收单步执行测试（10）

【评价标准：共10，执行正确，文字论述清晰具体，过程原理讲解清晰，得满分。不可打单步执行或者内容不正确计0分。其他情况，举证给出得分依据】

申请分数：10

（1）给出**连续动态内存分配**可视化过程代码，并逐行逐变量说明实现原理；

在可视化UI上，都是通过直接打印映射内存的BitSet类型的memoryMap来显示内存情况。

内存分配过程通过 allocateMemory() 方法实现。先计算需要分配的内存单元数。调用 findBestFit() 方法找到最佳适应的内存索引，即找到一个空闲块，其大小大于等于所需内存单元数。如果找到了符合条件的空闲块，则将该空闲块标记为已占用，并记录内存分配情况。如果未找到符合条件的空闲块，则分配失败，输出相应提示信息。

1. **public** **void** printMemoryMap() {
2. Kernel.gui.memfield.setText("");  //每次刷新内存显示区
3. **for** (**int** i = 0; i < MEMORY\_SIZE-20; i++) {  //遍历内存空间
4. **if**(i%10==0)  //10个为一块物理块
5. Kernel.gui.memfield.append("物理块"+(i/10+1)+"：");  //提示物理快信息
6. Kernel.gui.memfield.append(memoryMap.get(i) ? "1 " : "0 ");  //获取内存单元信息并展示
7. //换行操作
8. }
9. }
10. **public** **boolean** allocateMemory(**int** jobId, **int** size) {
11. **int** unitsNeeded = size;  // 计算需要分配的内存单元数
12. **int** startIndex = findBestFit(unitsNeeded);  // 找到最佳适应的内存索引
13. **if** (startIndex != -1) {  // 如果找到了最佳适应的内存索引
14. memoryMap.set(startIndex, startIndex + unitsNeeded);  // 将内存映射中的起始索引设置为起始索引和单元数
15. allocations.put(jobId, **new** MemoryAllocation(startIndex, size));  // 将分配的内存添加到分配映射中
16. **return** **true**;  //分配成功
17. } **else** {
18. **return** **false**;  //分配失败
19. }
20. }
21. **private** **int** findBestFit(**int** unitsNeeded) {
22. // 遍历内存映射，找到第一个空闲块，其大小大于等于unitsNeeded
23. **int** startIndex = 0;  // 当前正在查找的空闲块的起始地址
24. **while** (startIndex + unitsNeeded <= MEMORY\_SIZE) {  //内存空间可分配
25. startIndex = memoryMap.nextClearBit(startIndex);  // 找到下一个空闲块的起始地址
26. **if** (startIndex + unitsNeeded > MEMORY\_SIZE) { **break**;}  //无法找到匹配大小的块
27. **int** endIndex = memoryMap.nextSetBit(startIndex);  // 找到下一个占用块的起始地址
28. **int** blockSize = endIndex - startIndex;  // 计算空闲块的大小
29. **if** (blockSize >= unitsNeeded && blockSize < minBlockSize) {  // 如果空闲块的大小大于等于unitsNeeded，并且小于最小块大小
30. minBlockSize = blockSize;  //更新最小块大小
31. minStartIndex = startIndex;  //更新起始地址
32. }
33. // 更新startIndex为下一个占用块的起始地址+1
34. }
35. **return** minStartIndex;  //输出最小块
36. }

（2）给出**连续动态内存回收**可视化过程代码，并逐行逐变量说明实现原理；

内存释放过程通过 deallocateMemory() 方法实现。根据作业 ID 获取对应的内存分配情况。

如果存在该作业的内存分配情况，则将对应的内存空间标记为未占用，并将其从内存分配记录中移除。如果不存在该作业的内存分配情况，则输出相应的错误提示信息。

1. **public** **void** deallocateMemory(**int** jobId) {  // 释放内存函数
2. MemoryAllocation allocation = allocations.get(jobId);  // 获取指定jobId的内存分配
3. **if** (allocation != **null**) {  // 如果存在内存分配
4. memoryMap.clear(allocation.startIndex, allocation.startIndex + allocation.size);  // 从内存中清除指定的分配
5. allocations.remove(jobId);  // 移除指定的内存分配
6. } **else** {
7. //释放失败
8. }
9. }



内存区的显示分配

（3）以1.txt、2.txt、3.txt作业执行为例，设置断点单步跟踪内存分配与回收过程；录制视频文件讲解上述操作过程和结果。视频文件名：10-连续动态内存分配与回收

## 2.7 用户缓冲区PV操作访问的单步执行测试（8）

【评价标准：共８分，执行正确，文字论述清晰具体，过程原理讲解清晰，得满分。不可打单步执行或者内容不正确计0分。其他情况，举证给出得分依据】

申请分数：7

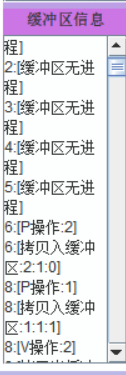
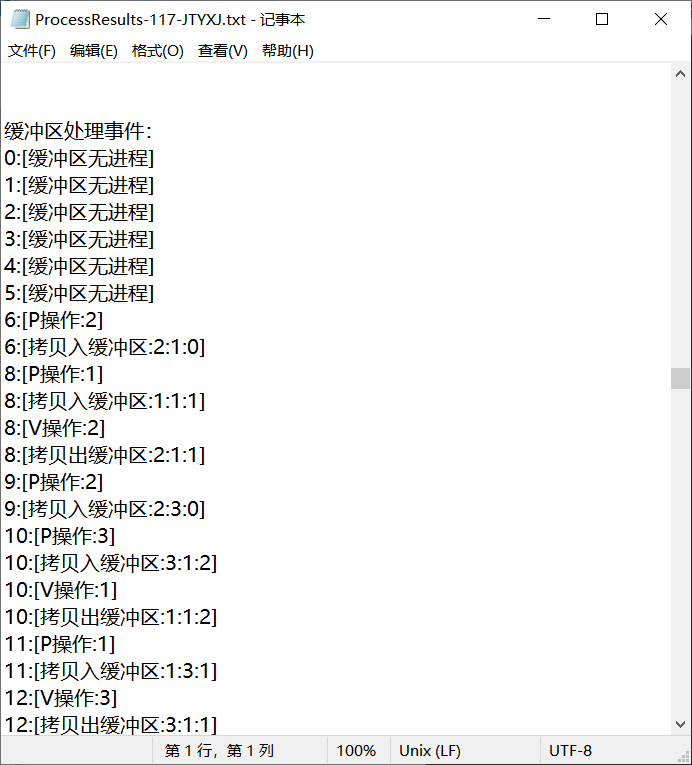
（1）给出内存**缓冲区**同步与互斥操作过程代码，并逐行逐变量说明实现原理；

缓冲区的同步和互斥是通过两个方法来实现的：P和V方法。这两个方法是用于控制缓冲区的写入和读取操作的，其中P方法是用于写入操作，而V方法是用于读取操作。

同步体现在对缓冲区状态的管理和控制上。当一个进程要往缓冲区写入数据时，需要先检查缓冲区是否已满，如果已满则需要等待；而当一个进程要从缓冲区读取数据时，需要确保缓冲区中有数据可供读取。这种操作的同步性保证了对缓冲区的有效利用，避免了写入和读取操作之间的冲突。

互斥体现在对缓冲区的互斥访问上。在写入操作时，只允许一个进程写入数据，其他进程必须等待；而在读取操作时，也只允许一个进程读取数据，其他进程必须等待。但由于串行指令的影响，互斥的设定是默认的。

1. **public** **class** Buffer {  //缓冲区类
2. **public** **void** P(PCB pcb) { // 写入缓冲区
3. Buffer\_full=Buffer\_full-1;
4. **if**(Buffer\_full>=0) {
5. Kernel.buffer.allocate(pcb.ProID);  //申请缓冲区空间
6. }
7. **else**
8. {
9. //Buffer已满
10. }
11. }
12. **public** **void** V(PCB pcb) {  //读取/释放缓冲区
13. Kernel.buffer.deallocate(pcb.ProID);  //释放缓冲区空间
14. }
15. }
16. **public** **boolean** allocate(**int** jobId) {
17. //寻找缓冲区合适插入点
18. **if** (可插入) {
19. //在缓冲区中分配空间
20. **return** **true**;  //缓冲区未满
21. } **else** {
22. **return** **false**;  //缓冲区已满
23. }
24. }
26. **public** **void** deallocate(**int** jobId) {
27. //通过jobID寻找占用的缓冲区位置
28. **if** (allocation != **null**) {
29. //找到缓冲区可清除位置并清除占用缓冲区
30. } **else** {
31. //失败的V操作
32. }
33. }

缓冲区操作的GUI与output输出

（2）以测试数据为例，设置断点单步跟踪3.txt内执行指令3的利用用户缓冲区传送数据的过程过程；录制视频文件讲解上述操作过程和结果。视频文件名：11-用户缓冲区PV操作访问

## 2.8 死锁仿真（5）

【评价标准：共5分，按要求逐项描述及讲解完备，得满分。其他情况，自己举证，给出得分及依据】

文字配套图片描述如何扩展设计，使其能够仿真实现进程死锁。给出该模块较为详细的设计和实现伪码，给出死锁生成、死锁检测的测试用例及过程。并对以上内容录屏讲解，文件保存：12-死锁仿真

申请分数：3

观察发现，由于读磁盘指令需要占据额外的内存空间，因此，如果设计一个指令序列，让它需求的内存足够大，在进入后备队列后，由于迟迟无法给这个作业分配足够的内存空间，将会产生持续的等待现象，并将该进程与后续进入的后备队列进程“饿死”。

1.生成死锁作业

此处生成了一个类似于这样的死锁进程序列：

1,3,6

2,3,6

3,3,6

4,3,6

5,3,6

6,3,6

7,3,6

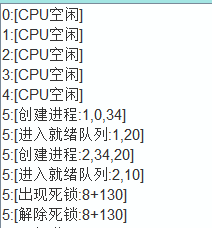
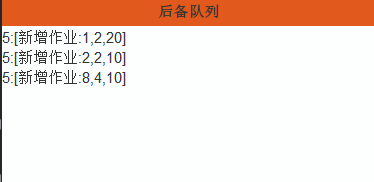
8,3,6

9,3,6

10,3,6

2.经过CountTotalSum()函数的计算，该作业需要13000B的内存，而内存去掉缓冲区就只有12000B，无法满足需求，在进入后备队列中，面临自己以及后续作业“饿死”的现象。

3.去除不合理占用的作业，可以解决死锁。



死锁的产生和解决

4.但是，这个“死锁”并不体现互斥条件和循环等待条件。在通常的情况下，我们可以改变指令的类型，但具体实现步骤较为困难。如设计新型指令5,此类指令能同时申请键盘输入、屏幕输出或是磁盘读取中的两个以上的资源。这时候，如一个进程的指令在申请键盘输入，正在屏幕输出，而下一进程在申请屏幕输出，正在键盘输入。两者互相循环等待，这就出现了同步互斥的死锁现象。解决该类死锁可以释放其中的一个进程以恢复正常。

# 3.问题描述与解决方案（8分）

【评价标准：共10，记录课设过程中遇到的技术问题及解决方案，问题描述不少于5个，需要有具体截图呈现问题场景与解决方案。文字论述清晰具体，过程分析论述清晰，得满分。其他情况，举证给出得分依据】

申请分数：8

理由：

1.在UI的制作过程中，我通过Jetbrains IDEA中自带的Swing框架设计器进行了框架的设计，但出现了界面的混乱和空间挤占，我咨询了同学后改边了GUI上文本的输出顺序，同时因为之前采用了JTextField来展示窗体中的提示信息，但无法进行换行操作，最后查阅资料使用JTextArea来输出较长成串的字符信息。



2.接上一回，继我设计了新的文本输出框架后，我发现我在文本量过大时，多行文本框依然无法满足需求。这时候，我查阅资料发现了平时浏览器里常见的滚空框。这个组件在Java的swing框架中也得到了广泛的应用。因此我采用了先在JTextArea下方铺上JScrollPane组件来进一步扩展文本输出空间。

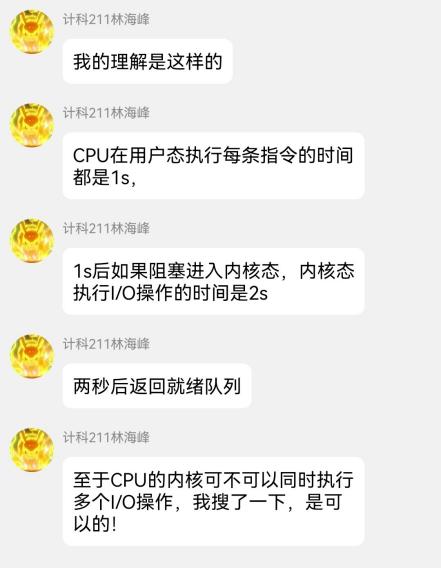




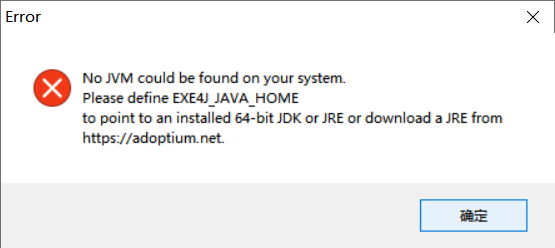
3.在视频的录制过程中，我发现多断点调试并不简单，打上断点之后不知道怎么跳到下一个进程。但是在咨询了同学后，我将断点的类型和位置都做了一些修改，成功了解决了多断点问题。



4.在我比较早编写代码的时候，出现了严重的理解问题。我误以为IO等阻塞是在等待资源的读入，而在阻塞队列出来以后，会继续执行2s的运行流程。但后来经过和同学的讨论，我及时的修改了调度算法和指令执行的代码。



5.在打包的过程中，我一开始采用广泛的Jdk1.8版本（亚马逊版），但后来发现在生成exe文件后跳出了错误的命令弹窗：



于是经过查阅资料和询问同学，我发现可能是打完包后Jdk版本为32位，因此无法正确打开。于是我又换成了64位的Jdk21。经过调试后成功可运行。

