2024 年《计算机操作系统》

课程设计指导手册

（面向人工智能及数据科学专业）

第 V1 版

南京农业大学人工智能学院

姜海燕

jianghy@njau.edu.cn

QQ 群号：556067749

2024 年 8 月 30 日

目 录

1 课程设计目的与要求

1.1 课设目的

以计算机操作系统原理为指导，利用多线程并发编程以及面向对象程序设计技术仿真 OS 内核的进程管理、作业管理、连续内存管理等 API 函数功能，可视化显示操作系统内核的工作过程。本次课程设计需要按本手册要求完成仿真程序设计、开发，使用教师发布的基础代码框架扩展编程实现，使用教师提供的测试数据按照测试模版要求测试，撰写自测互测报告，录制代码设计实现的讲解视频、自测、互测过程讲解视频等文件。

2.2 选题及要求

名称：仿真实现操作系统的作业管理及内存管理系统

2.2.1 基础要求（针对所有成绩等级）

◆每位同学独立完成本次课程设计内容，使用教师发布的基础代码框架扩展编程实现，使用教师提供的测试数据，按照测试模版要求撰写测试分析报告，录制代码实现原理及测试过程讲解视频文件。测试分析报告中包括设计论述、代码实现、用例测试、结果分析、技术问题总结、参考文献等内容。

◆课程设计采用的语言为 JAVA 或 C#，需要提交开发环境下运行代码以及脱离开发环境独立测试运行的可执行程序。

◆成绩申请等级分别为：A+、A、A-、B+、B、C、D，不提交课设材料的成绩视为不合格。

◆根据本手册要求和自己的实践能力，选择成绩等级和需要实现的功能，详细设计要求阅读后续小节。跨级实现不得分。

◆每位同学创建“姓名学号-申请成绩”文件夹，拷贝教师发布的课设材料模版中已创建好文件夹作为子文件夹。该文件夹的根目录：保存课设程序被编译为 RunProcess.exe 可执行文件、自测分析报告（模版在停课课设第 1 天发布），给测试人、考评小组阅读提交材料的顺序及注意事项。

◆必须使用 src 文件夹下教师发布的基础代码框架上进行扩展编程，根据设计要求自行添加变量、属性、方法等代码，并补充注释。src 文件夹下的程序需在开发环境下可运行。与申请成绩等级及功能不匹配的代码模块和程序文件请不要出现在该文件夹，否则视为抄袭嫌疑。

◆必须使用 input1 至 input4 文件夹下成绩等级对应的测试数据进行测试分析，调试运行输出结果保存到 output 文件夹中。申请不同成绩等级的同学请认真阅读该手册，对输入测试数据格式要求不同。不使用教师提供测试数据进行测试，或者使用往年测试数据及格式，本课程成绩视为不合格。

◆test-vidio 子文件夹:根据测试分析模板要求，按要求保存多个用例过程的录屏讲解文件（可以多个）。单个文件大小最好不超过 30M；

◆请阅读第 2.2.2 节-第 2.2.5 节所对应成绩等级的功能要求；

2.2.2 及格要求（成绩等级：D）

（1）利用 2 个线程仿真实现时钟中断及作业调度、进程调度请求，实现代码框架分别写在 ClockInterruptHandlerThread、ProcessSchedulingHandlerThread 文件中，请补充完整；

（2）多任务作业并发环境下，实现时钟计数、作业请求判别及调度、进程创建、进程撤销、进程调度等原语、时间片轮转进程调度算法，实现进程就绪态与运行态之间的调度切

换；

（3）不考虑内存分配，创建新进程时不受内存大小限制；

（4）界面上需要设计 4 个按钮，分别为执行、暂停、保存、实时。执行、暂停按钮实现对进程开始调度、暂停调度功能；保存按钮按下时将作业调度、进程调度过程的记录信息保存到 output 文件夹中。实时按钮按下以后模仿当前时刻有实时作业请求，实现实时作业指令生成、作业调度、进程调度、进程撤销等操作；

（5）界面上划分 4 个区域，分别为时钟显示区、作业请求区、进程就绪区、进程运行区。仿真程序一旦运行，时钟显示区开始显示按秒计时。作业请求区用行形式，每行显示后被队列中一个作业 JCB 信息及指令信息。进程就绪区用行形式显示就绪队列进程 PCB 信息；进程运行区显示正处于运行态进程的 PCB 信息；

（6） 作业请求区显示根据当前时钟计时情况和规则，装载已请求的作业进入后备队列。进程就绪区显示通过进程创建进入就绪队列进程信息。当按下“执行”按钮对进程开始调度，进程就绪区、进程运行区动态显示进程调度过程信息；

（7）使用 input1 文件夹读取每个指令文件信息；调度运行过程、JCB、PCB 信息变化过程记录保存到 output 文件夹；

（8）具体详细设计请认真阅读第 2 节内容

2.2.3 中等要求（成绩等级：C）

（1）利用 **2 个线程仿真**实现时钟中断及作业并发调度、进程调度请求，实现代码框架分别写在 ClockInterruptHandlerThread、ProcessSchedulingHandlerThread 文件中，请补充完整；

（2）仿真连续动态内存空间，实现 MMU 地址变换，利用最佳适应算法实现连续动态内存空间分配与回收；

（3）多任务作业并发环境下，仿真实现时钟计数、作业请求判别及调度、系统调用、进程创建、进程撤销、进程调度、内存分配、内存回收等原语；采用静态优先级进程调度算法，实现进程就绪态、运行态之间的调度切换；

（4）界面上需要设计 4 个按钮，分别为执行、暂停、保存、实时。执行、暂停按钮实现对进程开始调度、暂停调度功能；保存按钮按下时将作业调度、进程调度过程的记录信息保存到 output 文件夹中。实时按钮按下以后模仿当前时刻有实时作业请求，实现实时作业指令生成（共 20 条计算类指令）、作业调度、内存分配、进程调度、内存回收、进程撤销；

（5）界面上划分 5 个区域，分别为时钟显示区、作业请求区、内存区、进程就绪区、进程运行区。仿真程序一旦运行，时钟显示区开始显示按秒计时；作业请求区用行形式，根据当前时钟计时情况和规则，装载入后备队列中一个作业 JCB 信息及指令信息。内存区显示进程创建成功并进入就绪队列的进程在内存的布局。进程就绪区用行形式显示就绪队列进程 PCB 信息；当按下“执行”按钮对进程开始调度，进程就绪区、进程运行区动态显示进程调度过程信息；

（6）设计二维表的图形化界面可视化呈现内存分配、撤销时的内存管理过程；

（7）使用 input 2 文件夹读取每个指令文件信息；调度运行过程、JCB、PCB 信息变化过程记录保存到 output 文件夹；

（8）具体详细设计请认真阅读第 2 节内容

2.2.4 良好要求（成绩等级：B 系列，包括 B、B+）

（1）利用 **3 个线程仿真**实现时钟中断及作业并发调度、进程调度、键盘输入及屏幕显示输出中断请求；时钟中断及作业并发调度、进程调度实现代码框架分别写在 ClockInterruptHandlerThread、ProcessSchedulingHandlerThread 文件中，请补充完善；

键盘输入及屏幕显示输出中断请求使用单独的线程程序文件，需要自行设计完成；

（2）仿真连续动态内存空间，实现 MMU 地址变换，利用最佳适应算法实现连续动态内存空间分配与回收；

（3）多任务作业并发环境下，仿真实现时钟计数、作业请求判别及调度、系统调用、进程创建、进程撤销、进程调度、进程阻塞、进程唤醒、内存分配、内存回收等原语；采用多级反馈队列进程调度算法实现进程就绪态、运行态、阻塞态之间进程切换。

（4）界面上需要设计 4 个按钮，分别为执行、暂停、保存、实时。执行、暂停按钮实现对进程开始调度、暂停调度功能；保存按钮按下时将作业调度、进程调度过程的记录信息保存到 output 文件夹中。实时按钮按下以后模仿当前时刻有实时作业请求，实现实时作业指令生成（共 20 条指令，其中 15 条计算类、5 条键盘输入及屏幕显示输出指令，两类指令出现顺序自行设计规则）、作业调度、内存分配、进程调度、进程阻塞、进程唤醒、内存回收、进程撤销操作；

（5）界面上划分 6 个区域，分别为时钟显示区、作业请求区、内存区、进程就绪区、进程运行区、进程阻塞区。仿真程序一旦运行，时钟显示区开始显示按秒计时；时钟显示区显示仿真时钟，计时到秒。作业请求区用行形式，每行显示后被队列中一个作业 JCB 信息及指令信息，作业请求区显示根据当前时钟计时情况和规则，装载已请求的作业。内存区显示进程创建成功并进入就绪队列的进程在内存的布局。进程就绪区用行形式显示就绪队列进程 PCB 信息；进程运行区显示正处于运行态进程的 PCB 信息；进程阻塞区显示执行键盘输入及屏幕显示指令以后，进入阻塞态，中断处理以后进行唤醒的信息与过程；当按下“执行”按钮对进程开始调度，进程就绪区、进程运行区动态显示进程调度过程信息；当执行到“键盘输入及屏幕显示指令”时，进程阻塞及唤醒状态过程信息。

（6）设计二维表的图形化界面可视化呈现内存分配、撤销时的内存管理过程；

（7）使用 input3 文件夹读取每个指令文件信息；调度运行过程、JCB、PCB 信息变化过程记录保存到 output 文件夹；

（8）具体详细设计请认真阅读第 2 节内容

2.2.5 优秀要求（成绩等级：A 系列，包括 A、A+）

（1）利用 **4 个线程仿真**实现时钟中断及作业并发调度、进程调度、键盘输入及屏幕显示输出中断请求、消息传送请求。时钟中断及作业并发调度、进程调度实现代码框架分别写在 ClockInterruptHandlerThread、ProcessSchedulingHandlerThread 文件中，请补充完善代码；键盘输入及屏幕显示输出中断请求、消息传送请求分别使用两个单独的线程程序文件实现，需要自行设计完成；

（2）仿真连续动态内存空间，实现 MMU 地址变换，利用最佳适应算法实现连续动态内存空间分配与回收；

（3）仿真实现内存缓冲区管理，针对进程执行“消息通信指令”，由消息传送请求线程完成，通过对内存缓冲池管理、PV 同步互斥操作，采用多生产者-多消费者操作；

（4）多任务作业并发环境下，仿真实现时钟计数、作业请求判别及调度、系统调用、进程创建、进程撤销、进程调度、进程阻塞、进程唤醒、内存分配、内存回收、PV 同步互斥操作、多生产者-多消费者操作等原语；采用多级反馈队列进程调度算法，实现进程就绪态、运行态、阻塞态之间进程切换。

（5）界面上需要设计 4 个按钮，分别为执行、暂停、保存、实时。执行、暂停按钮实现对进程开始调度、暂停调度功能；保存按钮按下时将作业调度、进程调度过程的记录信息保存到 output 文件夹中。实时按钮按下以后模仿当前时刻有实时作业请求，实现实时作业指令生成（共 20 条指令，其中 12 条计算类、5 条键盘输入及屏幕显示输出指令、3 条消息通信指令，三类指令出现顺序自行设计规则）、作业调度、内存分配、进程调度、进程阻塞、

进程唤醒、内存回收、进程撤销、PV 操作等；

（5）界面上划分 7 个区域，分别为时钟显示区、作业请求区、内存区、进程就绪区、进程运行区、进程阻塞区、消息通信区。仿真程序一旦运行，时钟显示区开始显示按秒计时；时钟显示区显示仿真时钟，计时到秒。作业请求区用行形式，每行显示后被队列中一个作业 JCB 信息及指令信息，作业请求区显示根据当前时钟计时情况和规则，装载已请求的作业。内存区显示进程创建成功并进入就绪队列的进程在内存的布局。进程就绪区用行形式显示就绪队列进程 PCB 信息；进程运行区显示正处于运行态进程的 PCB 信息；进程阻塞区显示执行键盘输入及屏幕显示指令以后，进入阻塞态，中断处理以后进行唤醒的信息与过程；当按下“执行”按钮对进程开始调度，进程就绪区、进程运行区动态显示进程调度过程信息；当执行到“键盘输入及屏幕显示指令”、“发送消息”时，进程阻塞及唤醒状态过程信息；当执行“发送消息”时，还需要消息通信区显示消息队列、消息缓冲区以及消息发送、接受过程。

（6）设计二维表的图形化界面可视化呈现内存分配、撤销时的内存管理过程；

（7）设计二维表的图形化界面可视化呈现对消息缓冲区采用多生产者-多消费者的同步互斥操作；

（8）使用 input4 文件夹读取每个指令文件信息；调度运行过程、JCB、PCB 信息变化过程记录保存到 output 文件夹；

（9）具体详细设计请认真阅读第 2 节内容

2. 基础的仿真设计

2.1 裸机硬件仿真设计

包括 CPU、内存、系统时钟中断、用户内存区、磁盘数据输入缓冲区、输入输出设备中断、地址变换机构（MMU）等；

（1）CPU 与寄存器的仿真设计

◆CPU 可抽象为一个类，名称：CPU

◆基础框架代码：src\CPU 文件，在此文件上需改完善

◆关键寄存器可抽象为类的属性，至少包括：程序计数器（PC）、指令寄存器（IR）、状态寄存器（PSW）等，寄存器内容的表示方式自行设计，需要在实践报告中说明。

需要实现 CPU 现场保护、现场恢复操作，封装为 CPU 类的方法，供进程切换、CPU 模式切换方法调用。

方法函数的名称须统一，入口出口参数自行定义：

◆CPU-PRO（）：CPU 现场保护函数；

◆CPU-REC（）：CPU 现场恢复函数；

（2）系统时钟中断仿真

设计自己的计算机系统时钟，其他中断使用该时钟统一时钟计时；

◆设计一个时钟中断线程，名称：lockInterruptHandlerThread

◆基础框架代码：src\ClockInterruptHandlerThread 文件，在此文件上需改完善

◆该类中设计一个共享属性变量，名称：simulateTime，整型，单位：秒（s）

◆时钟计时函数，名称统一，入口出口参数自行定义：

◆simulateTimePassing（）：通过该函数对 simulateTime **变量计时操作。此变量为临界资源，需要互斥访问；因此，操作该变量时可以用 JAVA 加锁操作**；

每 1 秒激活该线程计时，simulateTime+1 操作。

此线程与**进程调度线程等其他线程保持同步；**

◆Java 实现系统时钟和多线程交互，实现多线程继承、接口继承、线程同步、线程加

锁，可参考本手册 6.2 节附件 2 及代码框架

（3）作业请求的中断仿真

假设计算机用自己的计时系统，**每 2 秒(s)**查询一次外部是否有新作业的执行请求，在“并发作业请求文件”中判读是否有新进程请求运行。

◆**基础框架代码：src\ClockInterruptHandlerThread 文件，在此文件上需改完善**

◆在**时钟中断线程类：lockInterruptHandlerThread，增加作业请求判读函数**

◆作业请求判读函数，名称：JobRequest（）：

功能：每 2 秒查询一次。利用 simulateTime 变量每计时 2 秒，判断“并发作业请求文件”是否有新进程请求的时间已到达；如果有，进入后备队列；如果没有，空操作；

（4）进程调度的中断仿真

假设计算机 CPU 在 1 秒(s)发生一次时钟硬件中断；可以执行 1 条指令。

◆**设计一个进程调度线程类，名称：ProcessSchedulingHandlerThread**

◆**基础框架代码：src\ProcessSchedulingHandlerThread 文件，在此文件上修改完善**

◆通过时钟中断线程激活该线程；

◆进程调度算法函数，根据所选题目等级的调度算法

名称：

RR( )：时间片轮转法

SP( )：静态优先级法

MFQ( )：多级反馈队列

◆**时间片变量：timeSlice**

功能：根据选题等级设置时间片大小 timeSlice，当 timeSlice=0 时发生进程调度；

（a）申 C、D 成绩完成：

进程切换时间片大小：timeSlice=2 秒，即每个用户进程获得 2s 的 CPU 执行时间，如果 2s 时间片用完，发生进程切换，切换出的用户进程重新进入就绪队列，就绪队列按照优先级排队。在进程已分配的 2 秒时间内，如果收到其他作业请求，不能中断该进程调度。

（b）申 A、B 成绩完成：

采用多级反馈队列调度算法，设计 3 个级别

第 1 级时间片大小：Times=1 秒

第 2 级时间片大小：Times=2 秒

第 3 级时间片大小：Times=4 秒

每个用户进程按照多级反馈队列调度算法进行调度；如果所执行指令有系统调用，则中断，用户进程进入阻塞队列，由与输入输出中断处理线程唤醒被阻塞进程；调度进程线程与输入输出中断处理线程并发

（5）键盘输入及屏幕显示输出中断仿真

◆申 A、B 成绩实现以下内容：

产生 InputBlockThread 线程类，该线程处理一个 I/O 中断指令需要 2 秒；

ALL_IO（）函数：无论键盘输入、屏幕显示输出中断请求，如键盘给变量赋值，或者将变量结果显示在屏幕上，均利用该线程处理。**假设 OS 内核模块处理 I/O 操作需要 2 秒，之后唤醒进程的阻塞队列。不考虑内存缓冲区操作。I/O 中断处理线程与进程调度、作业调度线程需要并发处理，不能串行处理。**

（6）用户内存区

假设不考虑 OS 内核所占内存空间。

申 A、B、C 成绩需实现以下仿真内容：

◆用户区共 16000B，每个物理块大小 1000B，共 16 个物理块；

◆用位示图实现内存管理。可以假设基础存储单元是 100B，每个物理块占用 10 个基础存储单元。

◆**假设用户进程中每条“用户态计算操作语句”类型的指令占用 100B；在进程创建时根据进程指令中计算类型指令个数计算所创建进程所占用内存大小，即：一个进程占用内存大小=该进程中用户态计算操作指令个数*100B。每个进程指令的逻辑地址、物理地址均连续，物理地址的基地址可以不同；**

◆地址变换过程（MMU）需要在界面上显示并详细记录过程，保存到 ProcessResults-？？？-算法名称代号.txt 文件中。？？？表示数字，为每次运行完成所有进程的总分钟数。

例如：测试输入数据是 input1 中的数据文件，用 JTYXJ（静态优先级）算法完成机器调度所有作业，总运行分钟数是 166 秒，保存进程运行输入输出以及调度过程中状态变化、统计数据等所有信息，保存到 ProcessResults-166-LZ.txt 文件，保存到 output1 子文件夹。

（7）消息通信与消息缓冲区仿真

◆申 A 成绩需要实现以下消息通信功能，自行设计实现代码。

◆实现原理：用户进程 A 申请发送消息给 B 进程，内核的消息通信模块（自行设计消息通信线程）完成消息生成、消息进入消息队列、A 进程通信数据拷贝，然后就直接发送；不管接收的进程 B 是否接收，运行接收原语，完成消息通信以后删除消息队列中 A 进程的申请。消息进程通信模块每 2 秒查询消息队列，如果没有了用户进程 A 的消息申请，就可以从阻塞队列 3 中唤醒被阻塞的用户进程 A，重新回就绪队列

◆消息通信线程类，类名 MessageThread：完成申请消息队列、申请消息缓冲区、消息发送、消息接收等操作，该线程为 OS 的值守线程，程序代码保存到 MessageThread 文件中。

◆申请消息缓冲区：共 10 个缓冲区存储单元，每个缓冲区存储单元 200B，合计 2000B。假设用户进程的一条通信类型指令**提交数据通信申请时，通信数据大小为 200B，消息发送申请进入消息队列。**可支持 10 个用户进程的通信并发；

◆消息发送原语函数：Send（接收消息的用户作业名称），用户进程执行到“发送消息”类型指令时，触发该函数工作，用户进程阻塞。抽取消息队列发送申请，在仿真的 OS 系统 PCB 表中判断接收消息的用户作业名称是否存在，如果存在，将发送消息请求的用户进程通信数据从进程私有用户区复制到消息缓冲区，注意对消息缓冲区的访问需采用多生产者-多消费者同步互斥方法，处理时长为 1 秒，仿真实现教材中消息发送功能。如果没有查到接收消息的用户作业名称，函数终止。

◆消息接收原语函数：Receive（接收消息的进程作业名称），该函数美 1 秒查询一次，抽取消息队列发送申请，在仿真的 OS 系统 PCB 表中判断接收消息的用户作业名称是否存在，如果存在，获取消息缓冲区消息数据拷贝给接受消息的进程的是有用户区，注意对消息缓冲区的访问需采用多生产者-多消费者同步互斥方法，当完成该函数工作以后，将被阻塞的发送消息进程唤醒。如果没有查到接收消息的用户作业名称，函数终止。

◆需要设计实现 PV 操作原语

2.2 并发的作业请求的设计

◆每个作业相当于用户提交给操作系统运行的可执行程序。本实验考虑作业管理、内存分配回收，进程管理、消息缓冲区、消息通信、进程同步与互斥等功能；

◆创建 input1、input2、input3、input4 子文件夹，保存仿真的**用户程序指令文件，每个程序指令文件是一个作业请求。本次实验在此设计上有重要变化，每个用户程序指令**

文件的文件名格式如下：作业名称-作业请求时间-优先级。编号为十进制数字；作业请求时间以自己的计时系统开始计时为准，单位是秒，两位数字。优先级用十进制数字表示，数字越小优先级最高。例如，1-03-0.txt 表示作业名称为 1，作业请求时间是仿真机器开机后第 3 秒，作业优先级为 0。

input1 子文件：申请 D 成绩的同学使用；

input2 子文件：申请 C 成绩的同学使用；

Input3 子文件：申请 B 成绩的同学使用；

Input4 子文件：申请 A 成绩的同学使用；

◆**InTimes：作业达到时间。以自己的计时系统开始计时，单位是秒。例如：InTimes 的值为 8，是指 8 秒后该作业提出请求；**

◆作业调用过程如下：仿真程序开始运行时，开始按秒计时（不要用系统时间，用自己的计时器计时）。读取 input 文件夹中作业文件的文件名，一次性读入一个临时数组。**lockInterruptHandlerThread** 类每 2 秒(s)查询临时数组，根据当前计时器时间，通过判断与作业请求时间（单位为秒）中请求时间的匹配程度，判断该时刻是否有作业请求。如果有，加入作业后备队列，并将该作业文件内容保存到相应数据结构中。

◆在界面上需要设计一个作业实时请求接收按钮，随时接收该时刻产生的新作业并加入到作业后备队列；

2.3 用户程序指令类型的设计

◆第 2.2 节用户作业请求文件中的内容为作业程序指令，是每个作业要执行的被编译以后的程序指令集合。每行一条指令，每个作业的指令类型由本文档给出。

（1）作业程序的结构

作业程序语句有 6 类，按照申请成绩等级，分别保存在 input1、input2、input3、input4 子文件夹中。具体包括：

指令编号（Instruc_ID）

程序语句的类型（Instruc_State）

接收消息的进程名称（Mesg_Name）

（2）用户程序指令的类型（Instruc_State）

★本试验**假设 CPU 执行用户态每条指令，在执行一条机器指令时不可以中断。**

★指令编号（Instruc_ID）：作业创建进程以后，进程所执行用户程序段指令序号，从 1 开始计数，该编号为每条指令的逻辑地址，连续编码。

★程序语句的类型（Instruc_State），包括：

0 表示用户态计算操作指令。执行该类型指令需要运行时间 InRunTimes=**1s**。

如果 CPU 执行该类型指令时，有优先级高的抢占进程请求，则执行完该指令以后被抢占；

一个进程占用内存大小=该进程中用户态计算操作指令个数*100B

1 表示键盘输入变量指令。发生系统调用，CPU 进行模式切换，运行进程进入阻塞态；值守的键盘操作输入模块接收到输入变量或输出变量内容，InRunTimes=**2s** 后完成输入，产生硬件终端信息号，阻塞队列 1 的队头节点出队，进入就绪队列；InputBlockThread 类在 **2s 以后自动唤醒该进程；**

2 表示屏幕显示输出指令。发生系统调用，CPU 进行模式切换，运行进程进入阻塞态；值守的屏幕显示模块输出变量内容，InRunTimes=**2s** 后完成显示，产生硬件终端信息号，阻塞队列 2 的队头节点出队，进入就绪队列；InputBlockThread 类在 **2s 以后自动唤醒该进程；**

3 表示发送消息指令。发生系统调用，CPU 进行模式切换，运行进程进入阻塞队

列 3，消息通信线程被唤醒

★接收消息的进程名称（Mesg_Name）

此项不是不选项，申 A 考虑

当 Instruc_State=0 或者 1 或者 2 时，

没有该项数据

当 Instruc_State=3 时，

Mesg_Name=接收消息的进程名称，消息数据大小=200B

2.4 作业运行及调度详细记录文件的格式

◆作业调度与时钟用同一个线程实现，进程调度模块用单独的线程实现。需要重点注意的是：不是每次发生时钟中断都要进行进程调度。

◆**进程调度算法代号**：时间片轮转调度算法**的代号 RR，**静态优先级**调度算法的代号 STYXJ，多级反馈队列调度算法代号 DJFK。**

◆**界面可视化显示作业请求与进程调度过程，需要以文件形式保存。界面显示和文件保存的信息要一致、清晰、完整、可读；**

◆**本次实验与** input1 或者 input2 或者 input3 或者 input4 中**子文件测试数据对应的输出文件夹保存 output 文件夹，文件名称为** ProcessResults-？？？-算法名称代号.txt 文件。

◆文件读写操作可参考附录 1

◆ProcessResults-？？？-算法名称代号.txt 文件的格式如下：

（1）包括 3 段信息，分别为：作业/进程调度事件、消息通信缓冲区处理事件、状态统计信息。其中，作业/进程调度事件、状态统计信息是必须的，其他事件的记录需要根据选题及申请成绩等级来完成。

（2）进程调度事件信息段的输出信息，每行有时间信息，按照时间顺序，记录发生进程调度事件、进程就绪队列、进程阻塞队列等的状态。

（3）每段输出信息的关键信息加半角中括号[]。批改时会对所有带有中括号的内容进行提取，不加中括号都视为无效信息。

每一行信息的具体格式如下：

时间:[关键操作或状态名称：作业或进程信息，]

时间需要用仿真计时器的时间；

输出信息格式如下：

作业/进程调度事件：

0:[新增作业:作业编号,请求时间,指令数量]

1:[创建进程:进程 ID,PCB 内存块始地址,分配内存大小]

2:[进入就绪队列:进程 ID:待执行的指令数]

3:[运行进程:进程 ID:指令编号,指令类型编号,物理地址，数据大小]

4:[运行进程:进程 ID:指令编号,指令类型编号,物理地址，数据大小]

5:[运行进程:进程 ID:指令编号,指令类型编号,物理地址，数据大小]

6:[阻塞进程:阻塞队列编号,进程 ID 列表]

7:[重新进入就绪队列:进程 ID 列表（每个 ID 用/隔开）,对应每个进程对应的剩余没有执行的指令数（每个进程对应的用/隔开）]

8:[阻塞进程:阻塞队列编号,进程 ID 列表]

9:[CPU 空闲]

10:[终止进程 ID]

消息缓冲区处理事件：

3:[P 操作/V 操作：信号量变量名称=具体数值]说明：两个操作选择之一，组合显示
4:[拷贝入缓冲区：进程 ID：指令编号：所操作的缓冲区编号]
5:[拷贝入缓冲区：进程 ID：指令编号：所操作的缓冲区编号]
6:[P 操作/V 操作：信号量变量名称=具体数值]说明：两个操作选择之一，组合显示
7:[P 操作/V 操作：信号量变量名称=具体数值]说明：两个操作选择之一，组合显示
8:[拷贝出缓冲区：进程 ID：指令编号：所操作的缓冲区编号]
9:[P 操作/V 操作：信号量变量名称=具体数值]说明：两个操作选择之一，组合显示
10:[缓冲区无进程]

进程状态统计信息：

结束时间:[进程 ID:作业请求时间+进入时间+总运行时间]
结束时间:[进程 ID:作业请求时间+进入时间+总运行时间]
BB1:[阻塞队列 1,键盘输入:进程 ID/进入时间/唤醒时间,进程 ID/进入时间/唤醒时间]
BB2:[阻塞队列 2,屏幕显示:进程 ID/进入时间/唤醒时间,进程 ID/进入时间/唤醒时间]
BB3:[阻塞队列 3,发送消息:进程 ID/进入时间/唤醒时间,进程 ID/进入时间/唤醒时间]

说明：

在“作业/进程调度事件”中,符号均为半角西文态符号：

时间：每秒记录一条信息，两条记录的时间可以相同，表示并发操作；

关键操作或状态：包括新增作业、创建进程、第一次进入就绪队列、重新进入就绪队列阻塞队列、系统调用、重新进入就绪队列、运行进程、CPU 空闲、终止进程、发送消息等，输出时请写汉字信息；

记录事件的多少根据选题确定；

作业或进程信息：包括进程 ID，指令编号，指令类型编号等，信息用逗号隔开；

在“状态统计信息”中：当每个进程指令全部执行完成以后计算保存。每一行统计一个进程信息；

结束时间：指进程运行结束后的时间；

2.6 进程控制块 PCB 设计

下面是进程相关 PCB 内容，参照 Linux task_struct 的数据结构内容设计；
进程编号（ProID）；值分别为 1,2,3,4,5,6，。。
进程优先数（Priority）；
进程创建时间（InTimes）；
进程结束时间（EndTimes）；
进程状态（PSW）；
进程运行时间列表（RunTimes）；
进程周转时间统计（TurnTimes）；
进程包含的指令数目（InstrucNum）；
程序计数器信息（PC）；
指令寄存器信息（IR）；
在就绪队列信息列表（包括：位置编号（RqNum）、进入就绪队列时间（RqTimes））；
阻塞队列信息列表 1（包括：位置编号（BqNum1）、进程进入键盘输入阻塞队列时间

（ BqTimes1））；

阻塞队列信息列表 2（包括：位置编号（BqNum2）、进程进入显示器输出阻塞队列时间（ BqTimes2））；

说明：

★系统请求运行的并发进程个数最小值为 5，可以自行设计最大值。

★进程编号（ProID）：整数

★进程优先级（Priority）：整数

★进程创建时间（InTimes）：由仿真时钟开始计时，整数，假设每条指令执行时间 1s；

★进程结束时间（EndTimes）：显示仿真时钟的时间，整数；

★进程运行时间列表（RunTime）：统计记录进程开始运行时间、时长，时间由仿真时钟提供；

★进程包含的指令数目（InstrucNum）：可自行扩展；

★PSW：保存该进程当前状态：运行、就绪、阻塞；

★指令寄存器信息（IR）：正在执行的指令编号；

★程序计数器信息（PC）；下一条将执行的指令编号；

◆需要设计系统空白 PCB 表，假设系统中进程最大并发数为 12 个，可用数组、链表、队列实现。

3 2024-2025 年第一学期课程设计时间节点安排

3.1 第 1 周：周三晚上 7:00-8:00 线上辅导，讲解课程设计指导手册，多线程并发编程（地点另行通知），学生申请成绩等级

3.2 第 2 周：周三晚上 7:00-8:00，课程设计线上辅导讲解，学生编写课设代码

3.4 第 3 周：周三晚上课设线上答疑，学生编写课设代码

3.5 第 4-5 周：学生修改课设代码并测试

3.6 第 6 周，10 月 14 日-18 日，全周停课课设,地点：滨江校区实验室（地点另行通知）

3.7 正式提交全部材料时间：

10 月 17 日（周四）提交自己的课设完整材料；

10 月 18 日（周五）提交互测完整材料

3.8 停课 5 天时间安排：

停课课设期间每日点名，提交每日工作报告，作为平时成绩。具体安排以后续通知为准：

（1）第 1-2 天：发测试报告模板文件，学生完善程序功能、修改代码测试等；

（2）第 3 天：完成程序代码测试，撰写自测分析报告，录制测试视频文件；

（3）第 4 天：完成程序代码测试，完成自测分析报告、录制带讲解的视屏文件等；提交自己参加课设所有正式材料；

（4）第 5 天：学生相互测试，填写互测报告；提交互测所有正式材料。

（5）因特殊原因不能到校上课的学生，需要提前一周提供学工办、教务办签字的证明，按照课设时间节点提交材料。

（6）第 7-12 周，完成课设内容进一步测试与检查

（7）第 13-16 周，完成抽查答辩及成绩统计

（8）本学期课设采用新的自测分析报告模板、互测评价报告模板

4 自测材料提交要求

（1）自测材料提交时间：2024 年 10 月 17 日（周四）16:00 前，只提交一次，过期不收。

（2）要求学生提交完整全面的可执行程序、全套工程文件及代码、自测分析报告、自我演示说明等材料。

◆提交材料要求格式正确性、完整性、讲解清晰，程序能够利用设置断点分段演示

（3）创建“姓名学号-申请成绩”文件夹，根据申请成绩等级拷贝教师提供的、所发布的课设材料中已创建好文件夹。

input1、input2、input3、input4 子文件夹中分别是申 D、申 C、申 B、申 A 成绩等级的输入测试数据：

output 子文件夹中分别保存程序运行结果。程序每次运行结果保存到一个新的 ProcessResults-？？？-算法名称代号.txt 文件中，该文件不覆盖。

例如，申请 B 以上成绩的同学，使用 input3 子文件夹中测试输入数据，程序运行结果保存到 output 子文件夹中.

（4）“姓名学号-申请成绩”文件夹的根目录：保存课设程序被编译为 RunProcess.exe 可执行文件、自测分析报告，并提交一份“材料阅读使用顺序说明文档”，给测试人、考评小组阅读提交材料的顺序及注意事项。

（5）src 子文件夹：工程文件、源程序等，要求在教师所提供代码框架基础上扩展。扩展的变量、函数、函数输入输出变量有详细的注释，并提供本文件夹每个文件用途的说明视频文件；

（6）test-vidio 子文件夹:根据测试分析模板要求，保存多个用例代码、测试过程的录屏讲解文件（可以多个）。单个文件大小最好不超过 30M；

（7）west 文件夹：保存使用了第三方程序或框架的库文件、安装使用手册等

5 课程设计课程联系方式

设计问题联系任课教师或助教，方式可通过 QQ 群或当面交流；

办公地点：滨江校区 A12 栋 A721 室

6 基础代码框架的说明

本次课设需在 src 文件夹下基础代码框架上扩展编程，代码框架说明如下：

控制线程

1. ClockInterruptHandlerThread：时钟中断线程

变量：

simulateTime：模拟时间

方法：

simulateTimePassing：模拟时间流逝

方法：

JobRequest：检查是否有作业进入，5s 检查一次

2. ProcessSchedulingHandlerThread：进程调度线程

方法：

RR：时间片轮转法

SP：静态优先级法

MFQ：多级反馈队列

基础数据结构：

3. Instruction：指令类

变量：

id：指令序号

state：指令类型

4. Job：作业类

变量：

jobId：作业编号

inTime：作业进入时间

instructionCount：作业指令数

5. PCB：进程控制块（进程类）

变量：

pid：进程号

pc：指令计数器

state：进程状态

仿真硬件

6. CPU：仿真 CPU

变量：

pc：指令计数器

ir：指令寄存器

psw：程序状态字

registerBackup：寄存器备份，用于进程切换时保存寄存器状态

方法：

runProcess：运行当前进程，打印相关进程信息

辅助实现程序

7. OSKernel：操作系统内核，用于存放队列等相关信息

backupQueue：后备队列，用于存放备份的作业

readyQueue：就绪队列，用于存放已准备好执行的进程

8. FileUtlis.java：文件处理类，根据需求自主设计。

9. SyncManger.java：同步管理类，根据选用的进程同步方法，自主设计。

7 基础实验参考

以下三个基础实验，实验步骤及代码说明供参考。供参考代码，分别在 Lab1、Lab2、Lab3 子文件夹中，程序不需要自行修改调试。

7.1 Lab 1 文件读取

【实验目标】

掌握文件读取的基本操作

理解作业请求的处理流程

实现从文件读取作业请求并加载到系统中

【实验内容】

编写代码实现作业加载的模拟
设计一个简单的作业队列系统
实现作业从模拟外部存储设备加载到模拟操作系统中

【实验步骤】

Step1 文件基本结构

编写代码从文件读取作业请求，文件格式为 txt，参考代码见 Lab 1 文件读取文件夹。

作业输入文件

jobs_input.txt：

1,8,20
2,10,30
3,15,30
4,22,22
5,24,30

作业请求文件：每行表示一个作业请求，包含作业 ID、作业的到达时间和作业包含的指令条数。以 1,8,20 为例，代表 1 号作业在 8 秒时进行请求，作业中的指令条数为 20。

Job 类

```
public class Job {
    private int jobId;                    // 作业的唯一标识符
    private int inTime;                   // 作业进入系统的时间
    private int instructionCount;         // 作业包含的指令数量
    private List<Instruction> instructions; // 与作业关联的指令列表

    public Job(int jobId, int inTime, int instructionCount) {

    }
}
```

作业输入文件

1.txt
1,0
2,0
3,0
...
20,0

作业文件：每行代表一个指令，包含指令 ID、指令类型。以 1,0 为例，代表 1 号指令指令类型为 0.

Instruction 类

```
public class Instruction {
    private int id;                    // 指令的唯一标识符
    private int state;                 // 指令的状态

    public Instruction(int id, int state) {

    }
```

```
    }
```

Step2 文件读取和写入

方式 1：使用 BufferedReader 和 BufferedWriter

读取文件：

```
    import java.io.*;
    import java.util.ArrayList;
    import java.util.List;

    public class FileReadWriteExample {

        public static List<String> readLines(String filePath) throws IOException
{
            List<String> lines = new ArrayList<>();
            try    (BufferedReader    reader    =    new    BufferedReader(new
FileReader(filePath))) {
                String line;
                while ((line = reader.readLine()) != null) {
                    lines.add(line);
                }
            }
            return lines;
        }

        public static void main(String[] args) {
            try {
                List<String> lines = readLines("example.txt");
                for (String line : lines) {
                    System.out.println(line);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

写入文件：

```
    import java.io.*;
    import java.util.List;

    public class FileReadWriteExample {

        public static void writeLines(String filePath, List<String> lines) throws
IOException {
            try    (BufferedWriter    writer    =    new    BufferedWriter(new
FileWriter(filePath))) {
```

```
                for (String line : lines) {
                    writer.write(line);
                    writer.newLine();
                }
            }
        }

        public static void main(String[] args) {
            List<String> lines = List.of("First line", "Second line", "Third
line");
            try {
                writeLines("output.txt", lines);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

方式 2：使用 Files 类

读取文件

```
    import java.nio.file.*;
    import java.io.IOException;
    import java.util.List;

    public class FileReadWriteExample {

        public static List<String> readAllLines(String filePath) throws
IOException {
            return Files.readAllLines(Paths.get(filePath));
        }

        public static void main(String[] args) {
            try {
                List<String> lines = readAllLines("example.txt");
                for (String line : lines) {
                    System.out.println(line);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

写入文件

```
    import java.nio.file.*;
```

```
    import java.io.IOException;
    import java.util.List;

    public class FileReadWriteExample {

        public static void writeLines(String filePath, List<String> lines) throws
IOException {
            Files.write(Paths.get(filePath), lines);
        }

        public static void main(String[] args) {
            List<String> lines = List.of("First line", "Second line", "Third
line");
            try {
                writeLines("output.txt", lines);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Step3 作业请求线程的实现

JobRequestHandlerThread:假设计算机每 5 秒(s)查询一次外部是否有新作业的执行请求，在“并发作业请求文件”中判读是否有新进程请求运行。

类伪代码

```
    类 JobRequestHandlerThread 继承自 Thread {

        方法 run() {
            无限循环 {
                调用 handleJobRequests()
            }
        }

        私有方法 handleJobRequests() {
            获取 SyncManager 的锁
            尝试 {
                // 等待作业请求信号
                等待 SyncManager 的 jrtCondition 信号
                调用 processIncomingJob()
                如果当前时间是 5 的倍数 {
                    调用 transferJobsToReadyQueue()
                }
                // 通知线程调度线程
                通知所有等待在 pstCondition 上的线程
            } 捕获中断异常 {
```

```
                恢复中断状态
                打印异常堆栈
            } 最终 {
                释放 SyncManager 的锁
            }
        }

        私有方法 processIncomingJob() {
            如果 OSKernel 的缓冲区不为空 {
                获取缓冲区队列的第一个作业
                如果作业的进入时间等于当前时间 {
                    从缓冲区移除作业
                    将作业加入到后备队列
                    打印 "当前时间:[新增作业:作业信息]"
                }
            }
        }

        私有方法 transferJobsToReadyQueue() {
            当后备队列不为空 {
                获取后备队列的第一个作业
                如果作业的进入时间小于等于当前时间 {
                    创建一个新的进程，参数为作业和下一个进程 ID
                    从后备队列移除该作业
                    将进程加入到就绪队列
                    打印 "当前时间:[进入就绪队列:进程信息]"
                } 否则 {
                    跳出循环 // 后备队列按到达时间排序，后面的作业还没到时间
                }
            }
        }
    }
```

伪代码解释

1. run() 方法：该方法是线程的入口，线程启动后会在无限循环中不断调用 handleJobRequests 方法处理作业请求。

2. handleJobRequests() 方法：此方法首先获取 SyncManager 的锁，然后等待作业请求信号。在接收到信号后，处理新到的作业，并根据当前时间判断是否将作业从后备队列转移到就绪队列。最后，通知所有等待在 pstCondition 上的线程。

3. processIncomingJob() 方法：此方法检查 OSKernel 的缓冲区是否有作业请求。如果有并且请求的进入时间与当前时间相同，则将该作业从缓冲区移到后备队列，并打印作业信息。

4. transferJobsToReadyQueue() 方法：此方法遍历后备队列，将所有进入时间小于或等于当前时间的作业移到就绪队列，并创建相应的进程。后备队列按作业到达时间排序，如果某个作业还未到时间，则停止转移作业。

【代码结构说明】

1. FileUtlis.java: 文件读取的基本操作，主程序入口。
2. Instruction.java: 指令类的基本操作内容。
3. Job.java: 作业类的基本操作内容。
4.InstructionLoader.java: 指令读取，将作业和指令进行关联，

7.2 Lab2 系统时钟和多线程交互

【实验目标】

理解系统时钟的设计与实现
掌握多线程之间的交互与同步机制
实现一个能够模拟系统时钟和作业请求处理的多线程程序

【实验内容】

设计并实现一个系统时钟模块
实现多线程交互，模拟作业请求处理
实现线程间的同步与通信

【实验步骤】

Step1 系统时钟设计

在计算机操作系统中,统一时钟计时是通过系统时钟来实现的。系统时钟是一个定时器，定期触发中断，以便操作系统可以执行一些定时任务，例如进程调度、资源管理和系统监控。在本实验中，我们将设计一个时钟线程类来模拟系统时钟的功能，每秒计时并增加一个计数器。

ClockInterruptThread 类

```
类 ClockInterruptHandlerThread 继承自 Thread:

    静态整型 simulationTime = 0
    静态整型 milliseconds = 100

    方法 run():
        while true:
            SyncManager.lock.lock()
            try:
                SyncManager.jrtCondition.signal()
                SyncManager.clkCondition.await()
                simulateTimePassing(milliseconds)
            捕获 InterruptedException 异常 as e:
                e.printStackTrace()
            finally:
                SyncManager.lock.unlock()

    静态方法 getCurrentTime():
        返回 simulationTime

    静态方法 simulateTimePassing(整型 milliseconds):
        try:
```

```
            Thread.sleep(milliseconds)
            simulationTime++
        捕获 InterruptedException 异常 as e:
            e.printStackTrace()
```

伪代码解释

1.ClockInterruptHandlerThread 类继承自 Thread 类。它包含两个静态变量 simulationTime 和 milliseconds，分别用于跟踪模拟时间和表示时间模拟的间隔。run 方法是线程的主要方法，在无限循环中持续执行，首先获取锁，然后通知作业请求处理线程，接着等待时钟中断信号，并模拟时间流逝。如果捕获到 InterruptedException 异常，打印异常堆栈跟踪。finally 块确保锁被释放。

2.getCurrentTime 方法返回当前模拟时间。

3.simulateTimePassing 方法模拟指定毫秒数的时间流逝。如果捕获到 InterruptedException 异常，打印异常堆栈跟踪。

Step2 Java 中多线程同步的策略和方法

1. Synchronized 关键字

常见使用场景：

方法同步：适用于需要确保整个方法在同一时间只能被一个线程访问的场景，例如修改共享资源的方法。

代码块同步：适用于只需要同步部分代码块的场景，通过减少锁的粒度来提高性能，例如只在访问共享资源时进行同步。

同步方法

使用 synchronized 关键字可以声明同步方法,这样在同一时间内只有一个线程可以执行该方法。

```
public class SynchronizedExample {
    public synchronized void synchronizedMethod() {
        // 线程安全的代码
    }
}
```

同步代码块

使用 synchronized 关键字可以声明同步代码块，通过指定一个对象作为锁。

```
public class SynchronizedBlockExample {
    private final Object lock = new Object();

    public void synchronizedBlockMethod() {
        synchronized (lock) {
            // 线程安全的代码
        }
    }
}
```

2. ReentrantLock 关键字和 Condition

ReentrantLock 是一个可重入锁，提供了比 synchronized 更加灵活的锁机制。

ReentrantLock 常见使用场景：

复杂的同步需求：需要更细粒度的锁控制，例如可中断的锁获取、超时的锁获取和非

块结构的锁释放。

实现公平锁：控制线程获取锁的顺序，避免线程饥饿。

尝试获取锁：希望尝试获取锁，但不会无限期等待锁释放的场景。

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockExample {
    private final Lock lock = new ReentrantLock();

    public void lockMethod() {
        lock.lock();
        try {
            // 线程安全的代码
        } finally {
            lock.unlock();
        }
    }
}
```

与 ReentrantLock 一起使用,Condition 提供了类似 Object 的 wait 和 notify 的功能，但更加灵活。

Condition 常见使用场景：

复杂的线程通信：需要更复杂的等待/通知机制，而不仅仅是简单的 wait/notify，例如多条件变量。

实现生产者-消费者模式：控制生产者和消费者之间的同步，通过条件变量协调生产消费的速度。

等待特定条件：在获取锁的同时等待特定条件的场景，例如某个队列不为空时才能取出元素。

示例：

1. 使用 Condition 实现生产者-消费者模式：

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.LinkedList;
import java.util.Queue;

public class ConditionExample {
    private final Lock lock = new ReentrantLock();
    private final Condition notEmpty = lock.newCondition();
    private final Condition notFull = lock.newCondition();
    private final Queue<Integer> queue = new LinkedList<>();
    private final int MAX_SIZE = 10;
```

```
    public void produce(int value) throws InterruptedException {
        lock.lock();
        try {
            while (queue.size() == MAX_SIZE) {
                notFull.await();
            }
            queue.add(value);
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public int consume() throws InterruptedException {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                notEmpty.await();
            }
            int value = queue.poll();
            notFull.signal();
            return value;
        } finally {
            lock.unlock();
        }
    }
}
```

2. 等待特定条件

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ConditionWaitExample {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private boolean conditionMet = false;

    public void awaitCondition() throws InterruptedException {
        lock.lock();
        try {
            while (!conditionMet) {
                condition.await();
            }
            // 执行条件满足后的操作
```

```
            } finally {
                lock.unlock();
            }
        }

        public void signalCondition() {
            lock.lock();
            try {
                conditionMet = true;
                condition.signal();
            } finally {
                lock.unlock();
            }
        }
    }
```

3. Semaphore 信号量

常见使用场景：

限制访问资源的线程数量：控制同时访问共享资源的线程数量，例如数据库连接池。实现多资源的管理：协调多个线程对多个共享资源的访问，例如限制某一时刻只能有固定数量的线程执行某个任务。

实现某些类型的生产者-消费者问题：允许指定数量的生产者和消费者进行同步工作。

示例：

1. **限制访问资源的线程数量：**

```
    import java.util.concurrent.Semaphore;

    public class SemaphoreExample {
        private final Semaphore semaphore = new Semaphore(3); // 允许 3 个线程同时访问
        private int count = 0;

        public void accessResource() {
            try {
                semaphore.acquire(); // 获取许可
                try {
                    // 访问共享资源的代码
                    count++;
                    System.out.println("Resource accessed by thread: " + Thread.currentThread().getName() + " | Count: " + count);
                    Thread.sleep(1000); // 模拟资源访问的耗时操作
                } finally {
                    semaphore.release(); // 释放许可
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }
        }

        public static void main(String[] args) {
            SemaphoreExample example = new SemaphoreExample();
            for (int i = 0; i < 10; i++) {
                new Thread(example::accessResource).start();
            }
        }
    }
```

2. 实现多资源的管理：

```
    import java.util.concurrent.Semaphore;

    public class MultiResourceSemaphoreExample {
        private final Semaphore semaphore;

        public MultiResourceSemaphoreExample(int resourceCount) {
            this.semaphore = new Semaphore(resourceCount);
        }

        public void useResource(int resourceId) {
            try {
                semaphore.acquire(); // 获取许可
                try {
                    // 使用资源的代码
                    System.out.println("Resource " + resourceId + " used by thread:
" + Thread.currentThread().getName());
                    Thread.sleep(1000); // 模拟资源使用的耗时操作
                } finally {
                    semaphore.release(); // 释放许可
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        public static void main(String[] args) {
            MultiResourceSemaphoreExample example = new
MultiResourceSemaphoreExample(5); // 允许同时使用的资源数
            for (int i = 0; i < 10; i++) {
                final int resourceId = i % 5;
                new Thread(() -> example.useResource(resourceId)).start();
            }
        }
```

```
}
```

3. 实现生产者-消费者模式：

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.concurrent.Semaphore;

public class ProducerConsumerSemaphoreExample {
    private final Queue<Integer> queue = new LinkedList<>();
    private final Semaphore notFull;
    private final Semaphore notEmpty = new Semaphore(0);
    private final Semaphore mutex = new Semaphore(1);
    private final int MAX_SIZE = 5;

    public ProducerConsumerSemaphoreExample() {
        this.notFull = new Semaphore(MAX_SIZE);
    }

    public void produce(int item) throws InterruptedException {
        notFull.acquire(); // 等待有空余空间
        mutex.acquire(); // 获取互斥锁
        try {
            queue.add(item);
            System.out.println("Produced: " + item);
        } finally {
            mutex.release(); // 释放互斥锁
            notEmpty.release(); // 通知有新元素可消费
        }
    }

    public int consume() throws InterruptedException {
        notEmpty.acquire(); // 等待有可消费的元素
        mutex.acquire(); // 获取互斥锁
        try {
            int item = queue.poll();
            System.out.println("Consumed: " + item);
            return item;
        } finally {
            mutex.release(); // 释放互斥锁
            notFull.release(); // 通知有空余空间
        }
    }

    public static void main(String[] args) {
        ProducerConsumerSemaphoreExample example = new
```

```
ProducerConsumerSemaphoreExample();

            // 生产者线程
            Thread producer = new Thread(() -> {
                for (int i = 0; i < 10; i++) {
                    try {
                        example.produce(i);
                        Thread.sleep(500); // 模拟生产耗时
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });

            // 消费者线程
            Thread consumer = new Thread(() -> {
                for (int i = 0; i < 10; i++) {
                    try {
                        example.consume();
                        Thread.sleep(1000); // 模拟消费耗时
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });

            producer.start();
            consumer.start();
        }
    }
```

【代码结构】

1. ClockInterruptHandlerThread.java: 模拟时钟线程，时间流逝。
2. JobThread.java: 模拟作业请求线程，每五秒钟请求一次。
3. SyncManager.java: 用于存储同步变量。
4. Test.java: 用于启动线程，主程序入口。

7.3 Lab 3 进程调度

【实验目标】

理解进程调度的基本概念与算法。

学习如何实现常见的进程调度算法，如先来先服务（FCFS）、最短作业优先（SJF）和轮转调度（Round Robin）。

实现一个简单的进程调度模拟系统。

【实验内容】

设计一个进程类用于模拟进程

调度算法实现：先来先服务（FCFS）：按照进程到达的顺序进行调度、轮转调度（Round Robin）：

【实验步骤】

Step1 Process 类的设计

包含属性：进程 ID、到达时间、执行时间、剩余时间等。

实现构造函数、获取属性的方法和 toString() 方法。

Process 类

```
package jobmanager;

类 Process 继承自 Job:
    属性:
        整型 pid // 进程 ID
        整型 pc // 程序计数器（Program Counter）
        整型 state // 进程状态

    方法:
        构造方法 Process(jobId, inTime, instructionCount, pid):
            调用父类的构造方法初始化 jobId, inTime, instructionCount
            初始化 pid 为参数 pid
            初始化 pc 为 0
            初始化 state 为 0 // 假设 0 表示就绪状态

        构造方法 Process(job, pid):
            调用父类的构造方法初始化 jobId, inTime, instructionCount
            初始化 pid 为参数 pid
            初始化 pc 为 0
            初始化 state 为 0 // 默认状态
            调用 setInstructions(job.getInstructions()) 复制指令列表
```

伪代码解释

Process 类继承自 Job 类，包含三个属性：进程 ID (pid)，程序计数器 (pc)，和进程状态 (state)。它有两个构造方法。

Process 构造方法使用 jobId、inTime、instructionCount 和 pid 参数初始化 Process 对象，调用父类 Job 的构造方法来初始化作业 ID、进入时间和指令数量，然后将 pid 初始化为传入的参数 pid，将 pc 初始化为 0，将 state 初始化为 0，假设 0 表示就绪状态。

Process 构造方法使用 Job 对象和 pid 参数来初始化 Process 对象，调用父类 Job 的构造方法来初始化作业 ID、进入时间和指令数量，然后将 pid 初始化为传入的参数 pid，将 pc 初始化为 0，将 state 初始化为 0 作为默认状态，并调用 setInstructions(job.getInstructions()) 方法复制指令列表。

Step2 调度算法实现

先来先服务（FCFS）调度算法伪代码

```
类 FCFS:
    方法 schedule(processList):
        按照到达时间排序 processList
```

```
        currentTime = 0
        对于每个 process 在 processList 中:
            如果 currentTime 小于 process 的到达时间:
                currentTime = process 的到达时间
            打印 "进程 process 的 ID 在 currentTime 开始"
            currentTime += process 的执行时间
            打印 "进程 process 的 ID 在 currentTime 结束"
```

FCFS 类的 schedule 方法接受一个进程列表 processList 作为参数，首先根据进程的到达时间对列表进行排序。然后初始化当前时间 currentTime 为 0。对于 processList 中的每个进程，检查当前时间是否小于进程的到达时间，如果是，则将当前时间更新为进程的到达时间。然后打印进程的开始时间和结束时间，并将当前时间增加进程的执行时间。

4. 先来先服务（FCFS）调度算法伪代码

```
类 RoundRobin:
    属性:
        整型 timeQuantum // 时间片长度

    构造方法 RoundRobin(timeQuantum):
        初始化 timeQuantum 为参数 timeQuantum

    方法 schedule(processList):
        使用队列 queue 存储 processList 中的进程
        currentTime = 0
        当 queue 不为空:
            process = queue 的第一个元素
            如果 process 的到达时间小于等于 currentTime:
                executeTime = 最小值(timeQuantum, process 的剩余时间)
                打印 "进程 process 的 ID 在 currentTime 开始"
                currentTime += executeTime
                process 的剩余时间 减少 executeTime
                如果 process 的剩余时间 > 0:
                    将 process 添加到 queue 的尾部
                否则:
                    打印 "进程 process 的 ID 在 currentTime 结束"
                将 process 从 queue 中移除
            否则:
                currentTime++
```

① 初始化和准备阶段：

RoundRobin 类有一个属性 timeQuantum，表示时间片的长度。在 RoundRobin 类的构造方法中，使用传入的 timeQuantum 参数初始化该属性。

schedule 方法接受一个进程列表 processList 作为参数，并使用队列 queue 存储 processList 中的进程。初始化当前时间 currentTime 为 0。

② 进程调度循环：

当 queue 不为空时，获取队列中的第一个进程 process。

检查该进程的到达时间是否小于等于当前时间。如果是，则计算该进程的执行时间

executeTime，其值为时间片长度 timeQuantum 和进程剩余时间中的最小值。

打印进程的开始时间，然后将当前时间增加执行时间，并减少进程的剩余时间。

③ 处理剩余时间和时间片轮转

如果进程的剩余时间大于 0，则将进程重新添加到队列的尾部；否则，打印进程的结束时间，并将进程从队列中移除。

如果进程的到达时间大于当前时间，则当前时间增加 1。这一步确保系统时间的推进，即使当前没有进程可以执行。

5. FCFS 多级反馈队列（Multilevel Feedback Queue）调度算法伪代码

```
类 MultilevelFeedbackQueue:
    属性:
        整型 timeQuantum // 基础时间片长度
        列表 queueList // 多级队列列表
        整型 levels // 多级队列的数量

    构造方法 MultilevelFeedbackQueue(timeQuantum, levels):
        初始化 timeQuantum 为参数 timeQuantum
        初始化 levels 为参数 levels
        初始化 queueList 为长度为 levels 的空队列列表

    方法 schedule(processList):
        currentTime = 0
        将 processList 中的所有进程按照到达时间排序并添加到 queueList[0]

        当 queueList 中的任一队列不为空:
            对于每个 level 从 0 到 levels-1:
                如果 queueList[level] 不为空:
                    process = queueList[level] 的第一个元素
                    如果 process 的到达时间小于等于 currentTime:
                        executeTime = 最小值(timeQuantum * (level + 1), process 剩余时间)
                        打印 "进程 process 的 ID 在 currentTime 开始于等级 level"
                        currentTime += executeTime
                        process 的剩余时间 减少 executeTime
                        如果 process 的剩余时间 > 0:
                            如果 level < levels - 1:
                                将 process 添加到 queueList[level + 1] 的尾部
                            否则:
                                将 process 重新添加到 queueList[level] 的尾部
                        否则:
                            打印 "进程 process 的 ID 在 currentTime 结束"
                        将 process 从 queueList[level] 中移除
                    否则:
                        currentTime++
```

跳出内层循环

① 初始化和准备阶段：

MultilevelFeedbackQueue 类有三个属性：timeQuantum 表示基础时间片长度，`queueList levels 是多级队列的数量。在 MultilevelFeedbackQueue 类的构造方法中，使用传入的 timeQuantum 和 levels 参数初始化这些属性，并将 queueList 初始化为长度为 levels 的空队列列表。

② 进程调度准备：

schedule 方法接受一个进程列表 processList 作为参数。

初始化当前时间 currentTime 为 0。

将 processList 中的所有进程按照到达时间排序，并添加到 queueList[0] 中，即最高优先级队列。

③ 进程调度循环：

当 queueList 中的任一队列不为空时，循环处理进程。

对于每个等级 level 从 0 到 levels - 1，检查对应的队列 queueList[level] 是否为空。

如果 queueList[level] 不为空，获取队列中的第一个进程 process。

检查该进程的到达时间是否小于等于当前时间。如果是，则计算该进程的执行时间 executeTime，其值为 timeQuantum * (level + 1) 和进程剩余时间中的最小值。

打印进程的开始时间和所在等级，然后将当前时间增加执行时间，并减少进程的剩余时间。

如果进程的剩余时间大于 0，则根据当前等级决定将其重新添加到队列中。如果当前等级小于 levels - 1，则将进程添加到下一等级的队列 queueList[level + 1]；否则，将进程重新添加到当前等级的队列 queueList[level] 的尾部。

如果进程的剩余时间等于 0，则打印进程的结束时间，并将其从当前队列中移除。

如果 process 的到达时间大于当前时间，则增加当前时间 currentTime++。

处理完一个进程后，跳出内层循环，继续处理下一个进程。

【使用说明】

根据 Lab1 和 Lab2 的内容，将 Lab3 的代码进行适当调整，即可组成可运行的基础程序，最终实现：文件读取，模拟时钟，作业请求和进程调度的基本功能。