

南京农业大学

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)



题 目: 仿真实现操作系统的作业管理及内存管理

姓 名: 叶俊泽

班级专业: 人工智能 221

学 号: 11522105

助 教:	杨鹏	类型: 研究生
指导教师:	姜海燕	职称: 教授

2024 年 10 月 17 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，每项测试需按要求文字论述，并录制视频讲解文件，否则不认可。本次课设申请人不再自我评分。
- 3.评价人请根据要求及评分标准以及自测人所提供的论述、测试和讲解文件，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。
- 5.评阅人分数统计表

1.作品说明（20分）			2.功能测试（共 60 分）											
1.1	1.2	1.3	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8				
3.问题分析与解决（共 10 分）								手写签名图片						
申请成绩等级					总成绩									

诚信声明：

所提交的课设代码由自测人自己独立完成
 测试数据使用本次课设成绩等级要求对应的测试数据
 所提交的讲解视频由自测人自己独立完成
 没有通过拷贝、购买等手段委托他人代为完成本次课设部分或者全部内容

自测人姓名：叶俊泽
 专业/学号：11522105

签字（手写）：

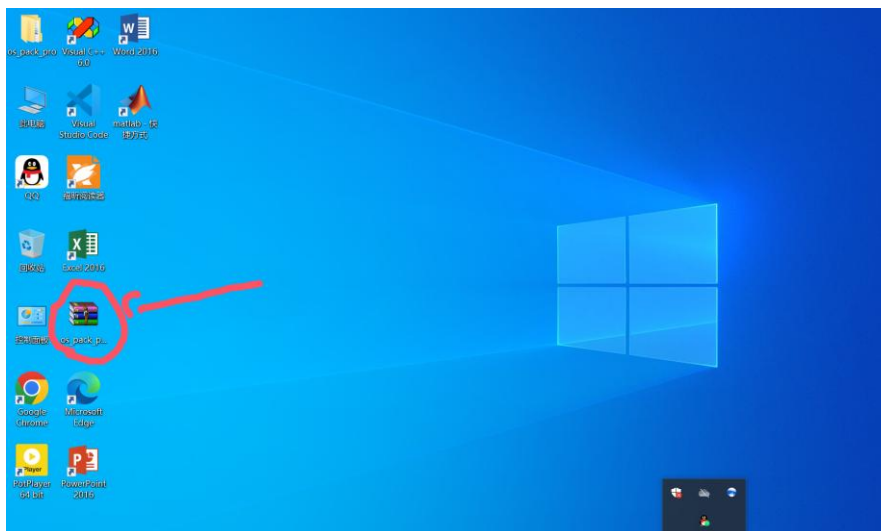
叶俊泽

1.课设作品（共 20 分）

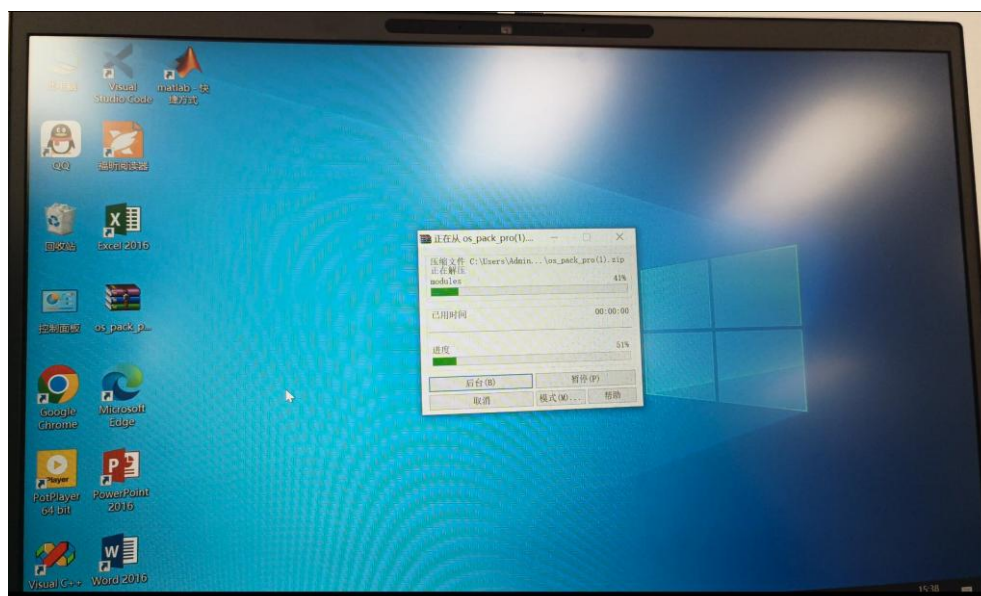
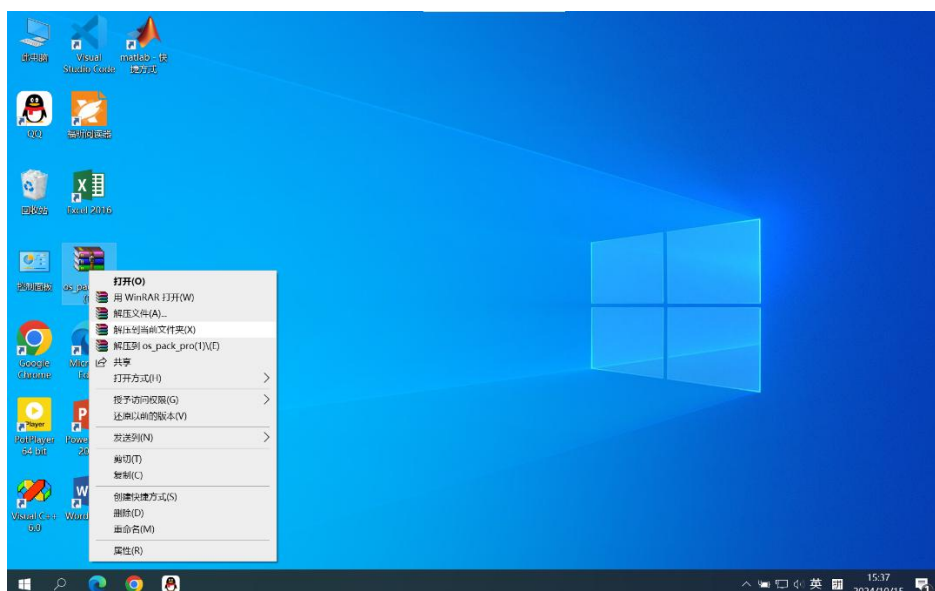
1.1 非开发环境下可执行程序的安装使用说明

（1）请在机房 PC 机电脑上（没安装过 Java 开发环境），面向一般用户提供安装可执行程序的步骤、所需要的相关安装包等内容；

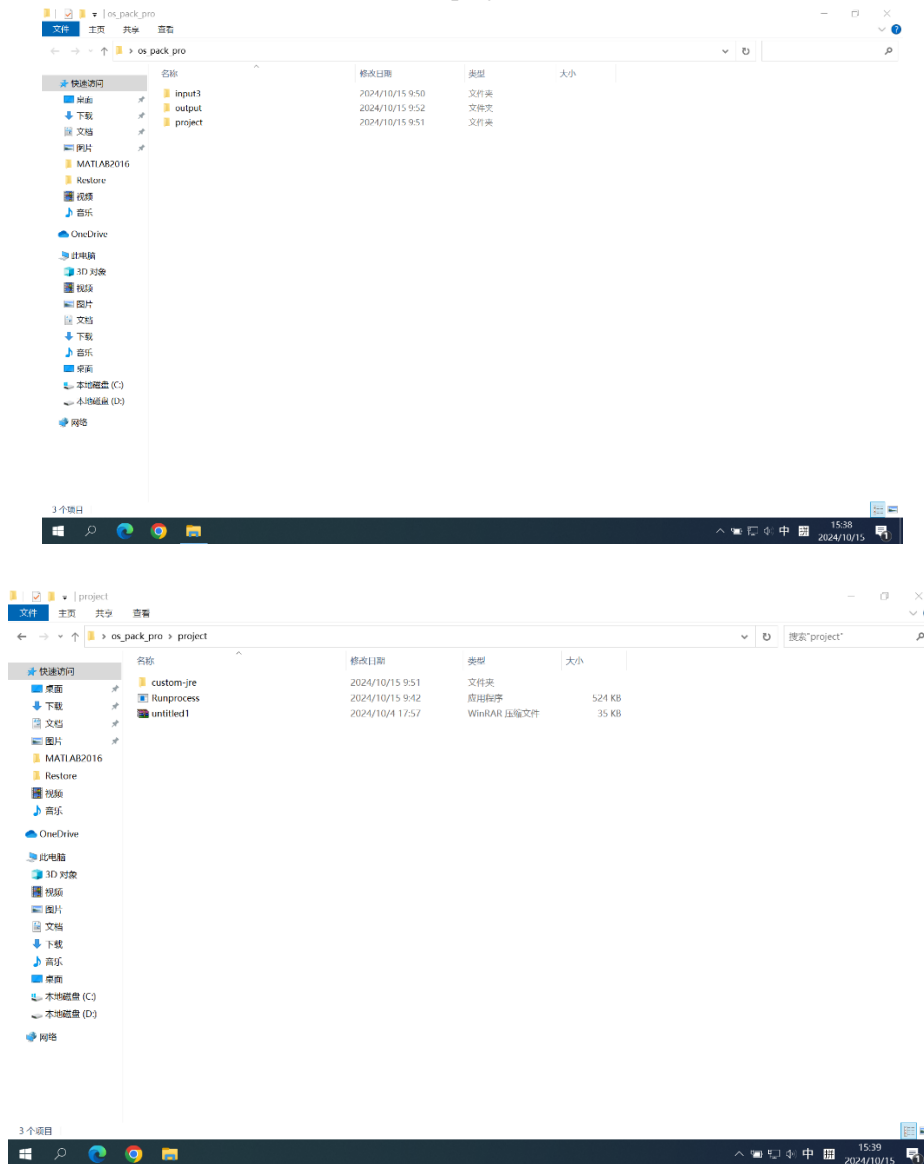
第一步：首先在非开发者的个人电脑上安装程序压缩包



第二步：解压下载压缩包



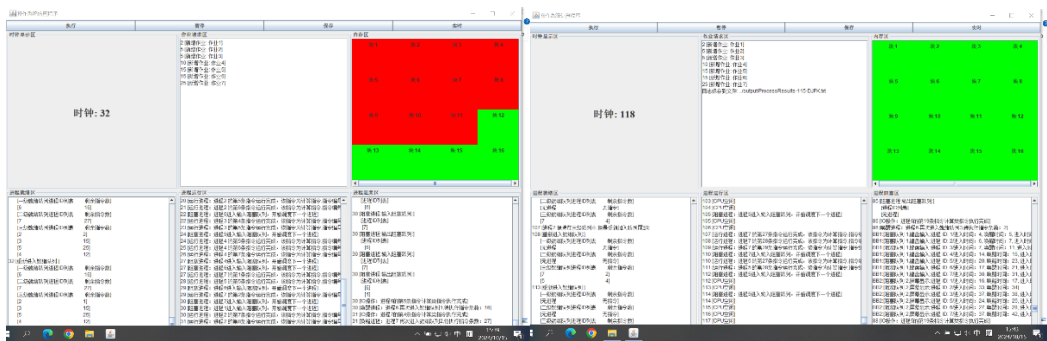
第三步：打开已经解压的文件夹，进入“project”目录：



第四步：点击“Runprocess.exe”文件，运行程序



结果总结：一次测试所有按钮，发现都可以正常运行，并且日志文件保存成功：



(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 6 分；否则计 0 分】

安装步骤总结（配详细步骤图和文字说明）：

评价分数：

理由（文字不少于 10 字）：

1.2 说明开发环境下工程程序及源程序文件的内容

(1) 在 Java 或者 C#开发环境下，说明工程文件、源程序等每个文件功能、内部代码结构

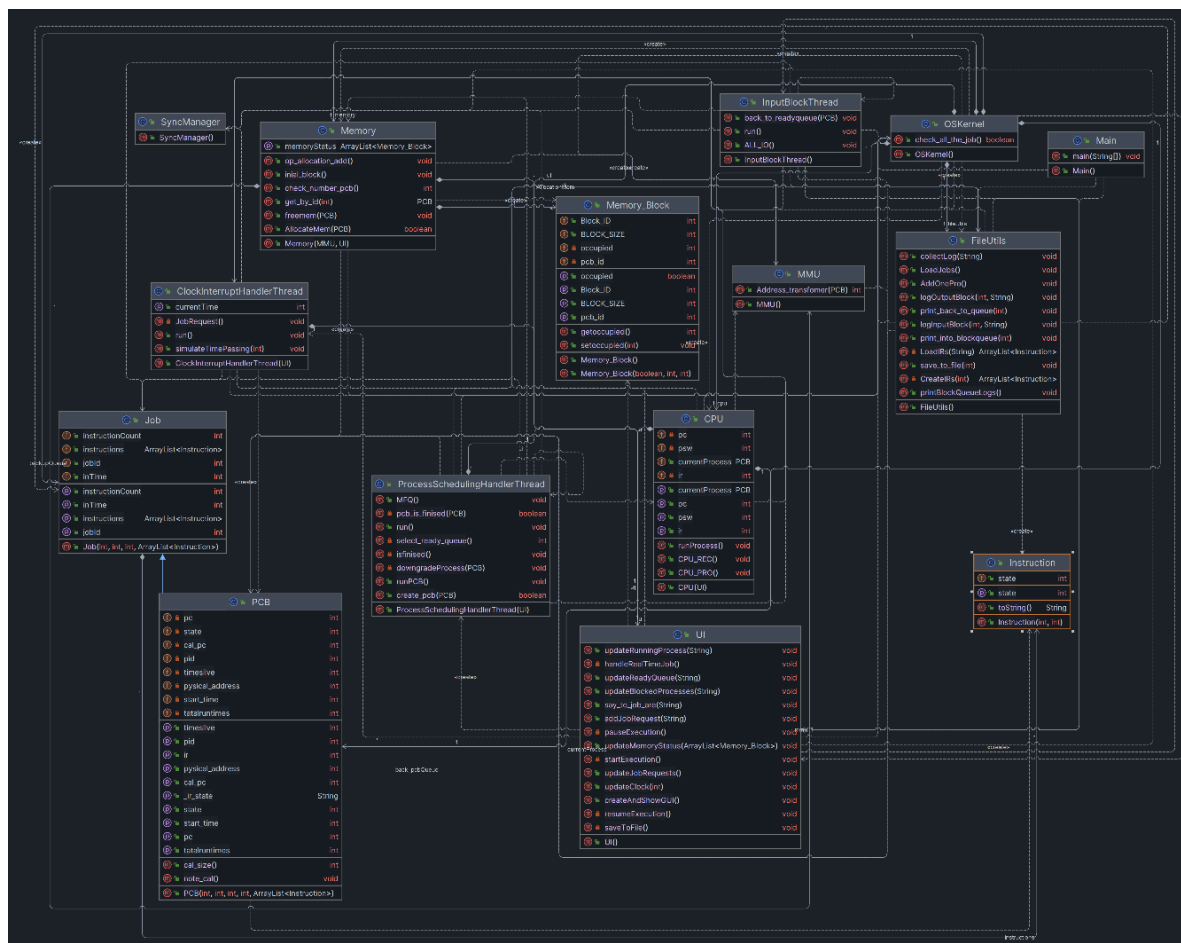
(2) 视频说明文件保存到 test-vidio 子文件夹，文件名：2-工程程序及源程序文件说明

【评分标准：完整详细，得 4 分；否则计 0 分】

(1) 说明工程文件、源程序等每个文件功能、内部代码结构（配证据图及说明文字）

(2) 如何组织源程序文件装载运行的步骤？（配证据图及说明文字）

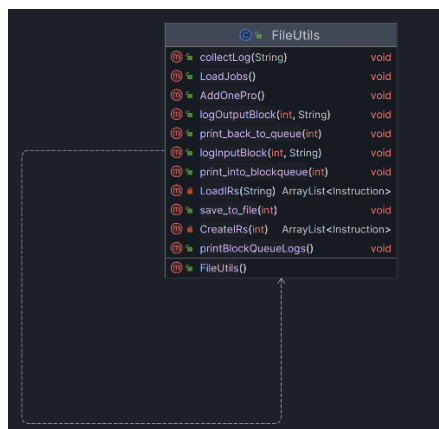
解：第一问（1）



所有的类图

首先在本次课程设计中，我一共创建了 15 个 Java 类完成了本次的操作系统仿真程序。

(1) 对于 **FileUtils** 类，这个 Java 负责处理所有的文件的读入——**LoadJobs()**函数；程序运行日志的保存——**save_to_file(int)**函数；运行日志的打印——**printBlockQueueLogs()**函数，**print_back_to_queue(int)**函数，**print_into_blockqueue(int)**函数等；随机指令的生成——**CreateIRs(int)**函数，**AddOnePro()**函数。简而言之，这个 Java 类全局负责所有的打印、输出、保存、更新信息。相当于现代计算机的输入输出。



FileUtils 内部构成

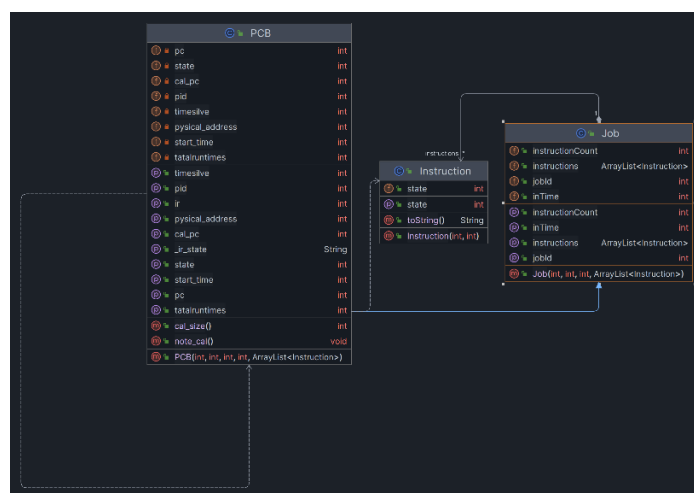
(2) 对于 Instruction、Job、PCB 类和 Memory、Memory_Block、MMU 类以及 CPU、OSKernel 类这些硬件方面的仿真类。

Instruction 类定义并描述了作业中单个指令的指令类型——state 变量和单个指令的 ID——id 变量。这个类非常重要，之后在 Job 类和 PCB 类中都会创建指令的 ArrayList 集合来存储作业的指令集。

Job 类定义并描述了一个作业的到达时间——Intime 变量，作业 ID——jobId 变量，作业的指令数目——instructionCount 变量，作业的指令集——ArrayList<Instruction> instructions。

PCB 类继承了 Job 类，并且还增加了大量的属性描述一个进程，进程的 ID，进程的创建时间，进程的计数器 PC，进程的结束时间，进程的当前运行的时间片，进程的物理起始地址，进程的大小，进程的状态——（就绪态、运行态、阻塞态）。计算 PCB 大小的函数，获取 PCB 某个指令的状态的函数。

下图是这三个类的构成关联图。



对于实现内存管理和 PCB 地址转换的三个类。首先我创建了 Memory_Block 这个类，因为一共有 16 个物理块。在这个类中，创建了描述单个物理块是否被占用的布尔变量——boolean Occupied，物理块的 ID——Block_ID，占用该物理块的 ID——pcb_id，单个物理块的大小——创建了静态变量——BLOCK_SIZE=1000（单位为：B）。

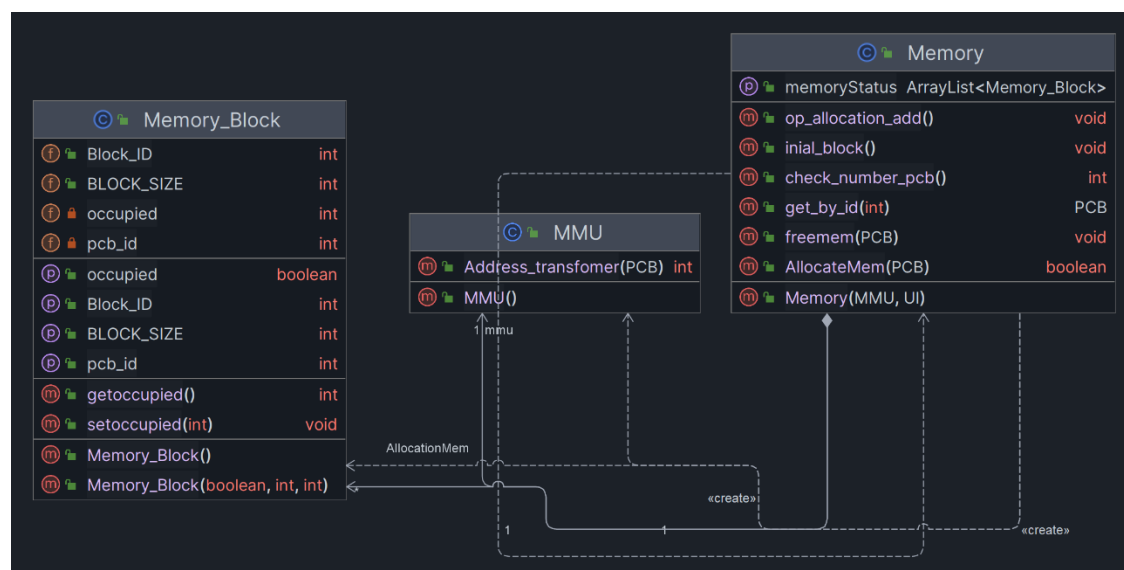
对于仿真实现最佳适应分配算法的 Memory 类，因为指定了物理块的数目和大小，为固

定分区的连续动态内存分配。所以我维护了一个 HashMap 的二维表，用来更新和存放每一个分配、释放内存后的内存的具体信息——HashMap<Integer, Integer> Allocation_add，哈希表中的键 key 代表内存中连续空闲快的起始物理地址，相对应的值 value 代表连续内存块的个数。

同时，为了方便更新每一个物理块 Memory_Block 的信息，我还维护了一个 ArrayList 集合，用来存储 16 个物理块的具体信息——ArrayList<Memory_Block> AllocationMem。Memory 内存类中具体的 op_allocation_add()函数用来更新每一次内存变化是内存分配表。AllocateMem(PCB)和 freemem(PCB)函数，具体实现了最佳适应分配算法和释放内存的逻辑。

对于 MMU 类，地址变化类，实现了地址变化的逻辑，具体在内存分配空间的时候调用，在这个类中，也维护了一张 HashMap 集合表——HashMap<Integer,Integer> addresss，用于存放进程的 pid 和物理地址的映射关系

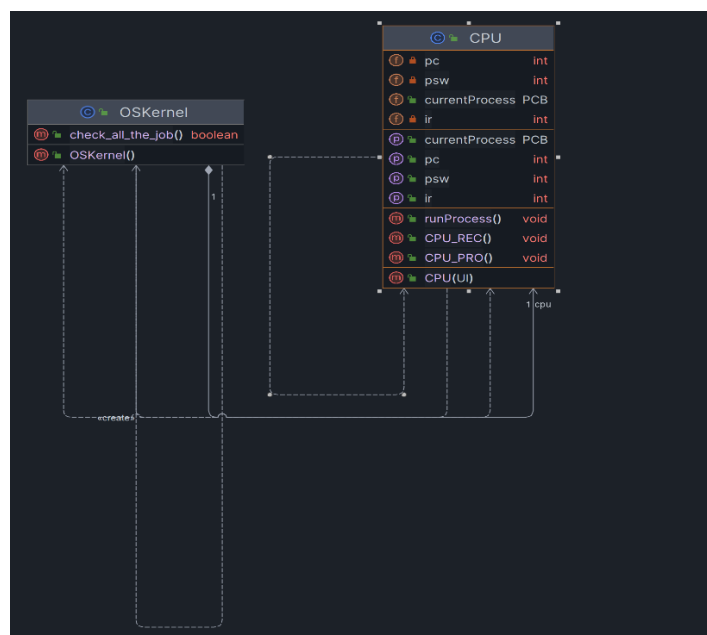
下图是内存类相关的代码构成图



硬件仿真的最后一个部分，CPU 类和 OSkernel 类，其中内核类可以说对于整个仿真程序而言最为关键，在这个类中，定义了静态的全局 CPU、Mmemory、UI 实例。同时，也定义了全局的三级队列、两个 IO 阻塞队列、PCB 缓冲区队列、后备队列、三级时间片。可以说，程序运行的所有关键信息都要调用这个类。

对于 CPU 类，主要定义了 PC 程序计数器、ir 当前正在执行的指令的类型，pswCPU 状态——用于 IO 中断，当前正在运行的进程 PCB——currentProcess。最为重要的一个函数——runProcess()——具体实现了指令运行的逻辑以及如果是 IO 类型的指令的中断处理，加入对应的阻塞队列的操作。还有两个重要的函数，CPU_PRO()和 CPU_REC()用于实现 CPU 中断的现场保护和回复。

下图是 CPU 等内核类的构成图



(3)接下来介绍 ClockInterruptHandlerThread 时钟类、ProcessSchedulingHandlerThread 进程调度类、InputBlockThread 中断 IO 处理类以及 SyncManager 同步并发类。

首先为了同步三个线程，进程调度线程和 IO 中断线程必须先获得对应的锁，进入等待之后，释放自己该有的锁，时钟同时获得两把锁，时钟先走一秒，唤醒之前沉睡的时钟。为此，我在 SyncManager 同步并发类中先后创建了**三把锁**，分别用于时钟和进程同步，时钟和 IO 同步，以及后来的暂停的逻辑。前两把锁每一个都有两个条件变量，最后一个实现暂停逻辑的只有一个条件变量。

需要补充的是：**为了确保每次都是进程调度线程和 IO 线程先拿到锁**，在 SyncManager 同步并发类，我还设置两个标志位用来标识。

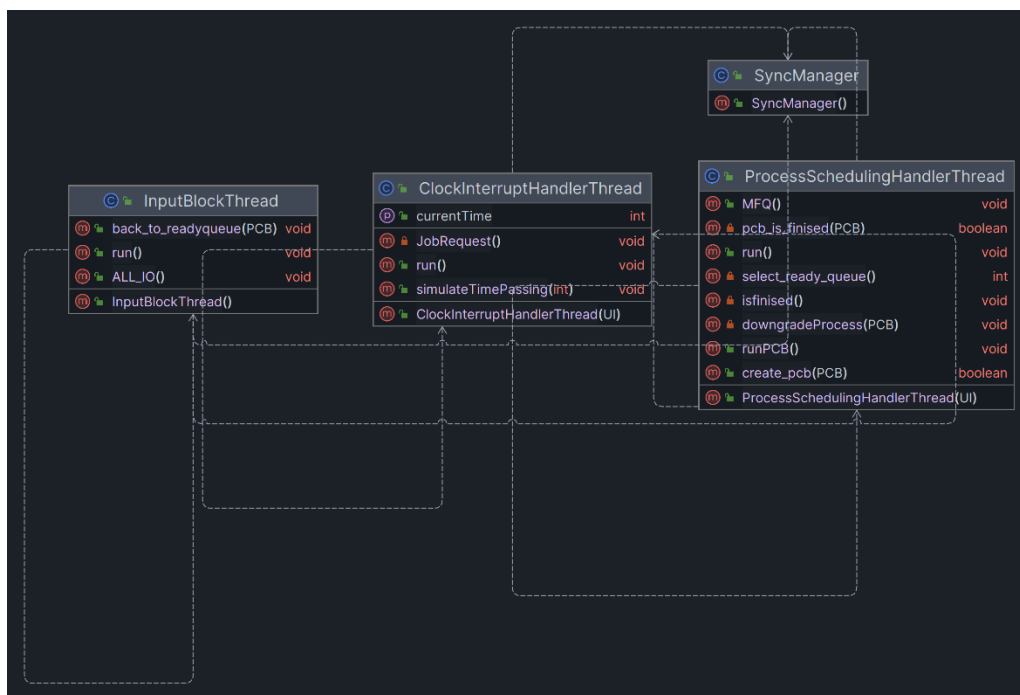
对于时钟线程，**本质上他是驱动整个程序的入口**，只有时钟线程唤醒其他线程。在这个线程中，设定了系统的时钟全局变量——simulationTime，并设置了时钟模拟流逝的函数——simulateTimePassing(int milliseconds)，但是只能在时钟线程中增加系统时钟变量。

在本次课程设计中，作业调度就发生在时钟线程中，在时钟时间每做走一秒之后开始调用 JobRequest()函数处理作业调度，处理查询、新增作业。

在进程调度中 ProcessSchedulingHandlerThread 类中，主要实现了多级反馈队列的算法逻辑，并增加了一个时间片变量——timeSlice，用于保存当前进程所使用的时间片。主要的函数 runPCB()用于创建、选择进程。其中核心代码的逻辑实现在 MFQ()中。

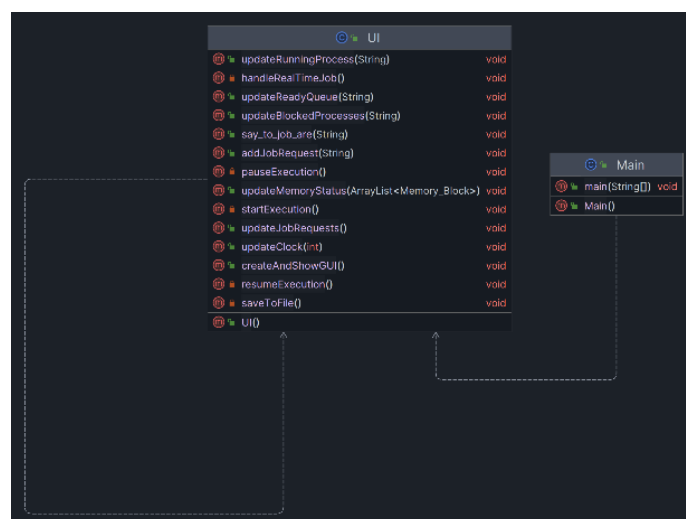
在 IO 处理线程中，定义了 All_IO()函数，用于处理所有的 IO 指令。其中为了确保 IO 指令两秒钟才能做完，定义了两个计数变量——count_i, count_o；用于标识 IO 指令的是否做完。

下图是四个设置线程并发类的构成图



(4)UI 类和 Main 程序启动类。UI 类中初始化了所有的文本框、按钮、事件监听器、内存位示图，并实现了暂停、执行、调用添加作业函数、保存、更新 UI 界面中的信息等功能。其中，UI 类中的函数——createAndShowGUI()用于显示 UI，并在 Main 中调用。

在 Main 类，在所有线程启动之前，必须先读入所有的作业到自己定义的数据结构，相当硬盘，之后启动 UI。下图是 UI 和 Main 类的构成图：



第二问 （2）

仿真程序的运行底层逻辑：

1. 准备数据结构与资源

作业加载：首先，从文件中读取并解析所有的作业，将其存入自定义的数据结构中，模拟硬盘到内存的作业加载。

初始化系统资源：包括 SyncManager 中的三把锁、条件变量，以及进程控制块（PCB）等相关资源。

2. 初始化全局变量与同步机制

初始化全局系统时钟变量 `simulationTime`，并设置 `SyncManager` 中的标志位和锁，确保进程调度线程和 IO 线程在时钟同步下协调运行。

定义多级反馈队列 (MFQ) 所需的队列和参数，如时间片 (`timeSlice`)，以供进程调度使用。

3. 启动 UI (用户界面)

启动 UI，用于展示当前系统状态 (如时钟时间、进程队列、内存使用、IO 状态等)。用户可以通过 UI 提交新作业、暂停仿真、保存调度日志等。

4. 创建并启动线程

时钟线程：首先启动时钟线程，模拟系统时间的流逝，定期唤醒进程调度线程和 IO 线程。通过 `simulateTimePassing()` 函数，时钟线程控制整个仿真进程的执行节奏。

进程调度线程 (`ProcessSchedulingHandlerThread`)：时钟线程启动后，进程调度线程开始工作。这个线程通过 `runPCB()` 函数创建和选择合适的进程，并使用 MFQ 实现多级反馈队列算法。

IO 线程：通过 `All_IO()` 函数处理所有 IO 操作。

5. 主循环与调度执行

时钟驱动主循环：时钟线程不断更新 `simulationTime` 并调用 `simulateTimePassing()`，在每一秒流逝后唤醒进程调度和 IO 线程。

进程调度与多级反馈队列：进程调度线程根据 MFQ 算法，选择适当的进程，调整其时间片并执行。若时间片耗尽，进程可能被降级到较低优先级队列。调度逻辑不断优化系统的响应时间和处理性能。

IO 处理：通过 `All_IO()` 处理所有 IO 指令，`count_i` 和 `count_o` 的计数确保每次 IO 操作严格按照两秒完成，避免过早或过迟执行。

6. 系统监控与交互

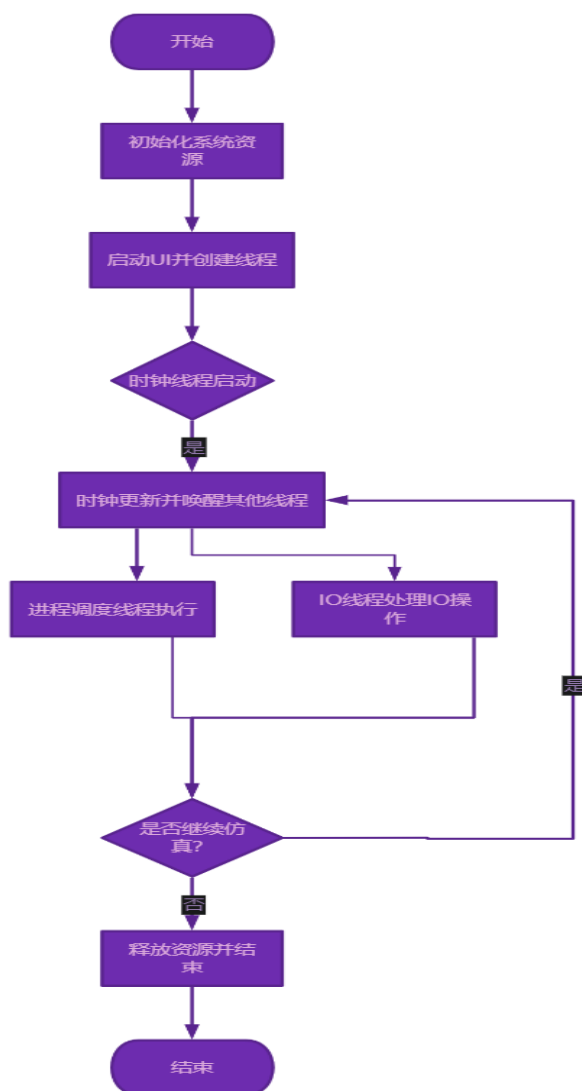
UI 中显示系统状态，包括当前时钟时间、各队列中的进程、正在进行的 IO 操作等。通过用户输入，可以动态修改作业队列、暂停/恢复进程等操作。

提供调度日志保存、仿真退出等功能，确保系统运行可控。

7. 程序结束与资源释放

当所有作业执行完毕或用户选择退出时，终止所有线程，释放资源 (锁、条件变量、内存等)。确保系统仿真正确结束。

需要强调的是：时钟线程是启动程序的关键，时钟是驱动操作系统的“发动机”。CPU 和 IO 处理的函数每次都只能做一秒钟的工作，之后时钟就要中断，重新开始时间流逝，如此循环往复。对此，每次新的中断开始的时候，如果时钟先获得了锁，进程调度和 IO 在这一秒就无法工作。所以，同步的核心就是设置获得锁的顺序并正确唤醒。



仿真程序运行流程图

评价分数：

理由：

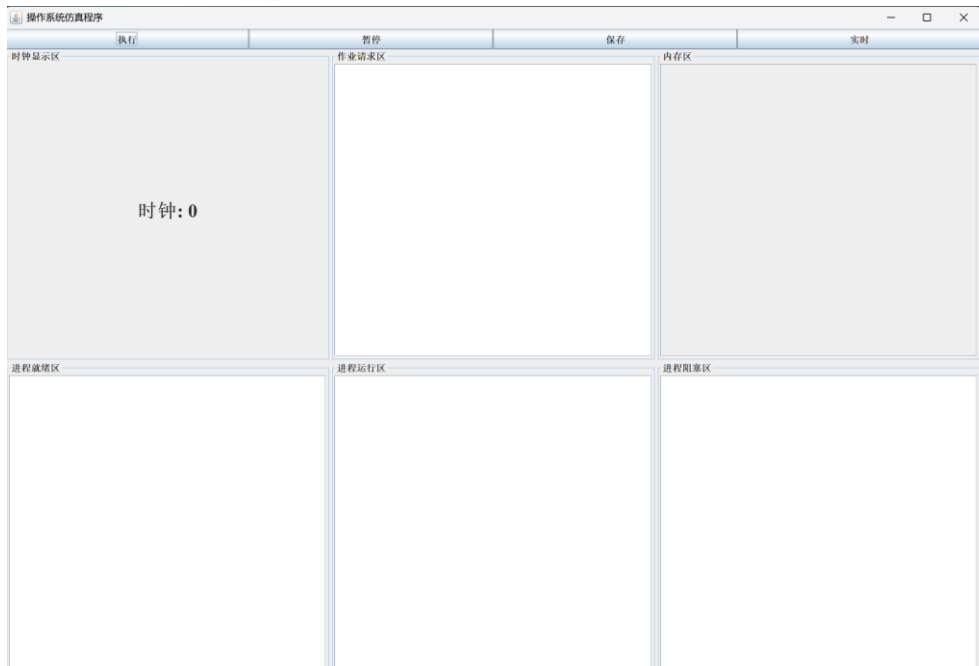
1.3 运行可执行程序测试结果

【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】

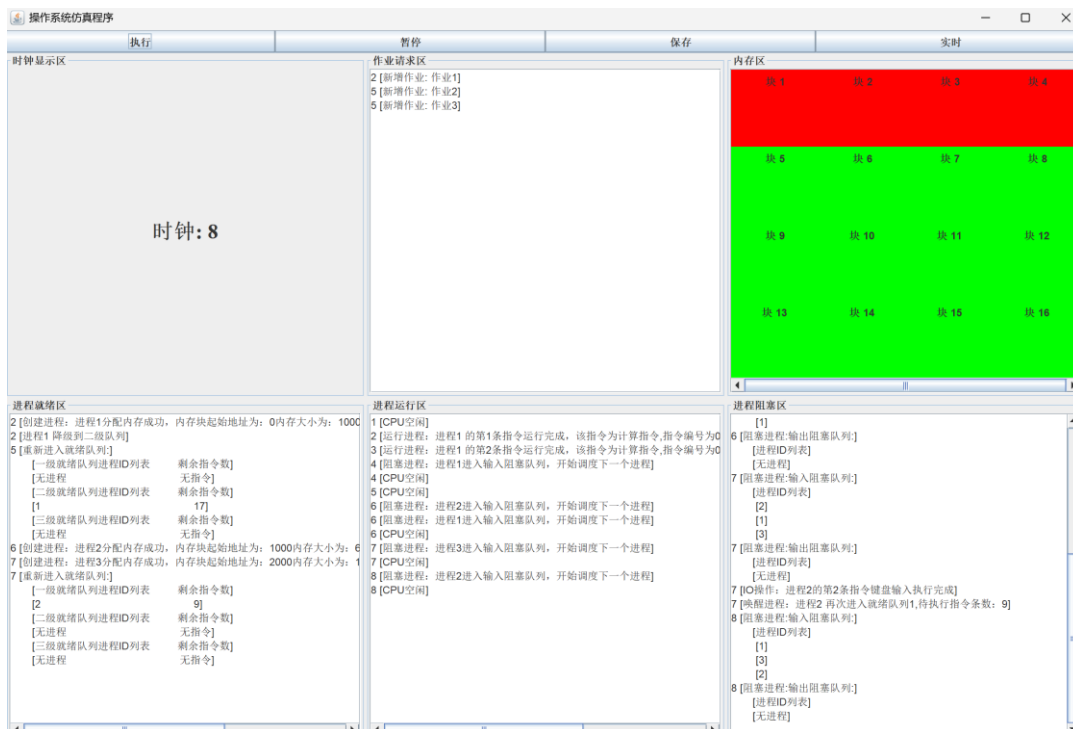
(1) 输入教师提供的测试数据，在机器上分别点击“**执行**、**暂停**、**保存**”按钮，运行可执行程序，生成 ProcessResults-???-算法名称代号.txt 文件。评价界面设计是否完备？界面中每个区域是否能否正确显示？给出每步执行过程截图，并对屏幕每步输出内容具体文字说

明。

第一步，首先启动 Main 类中的 main 函数，启动整个程序，这个时候只是显示一个 UI 界面，时钟显示为“0”，因为没有点击“执行”按钮，仿真程序还不能运行。

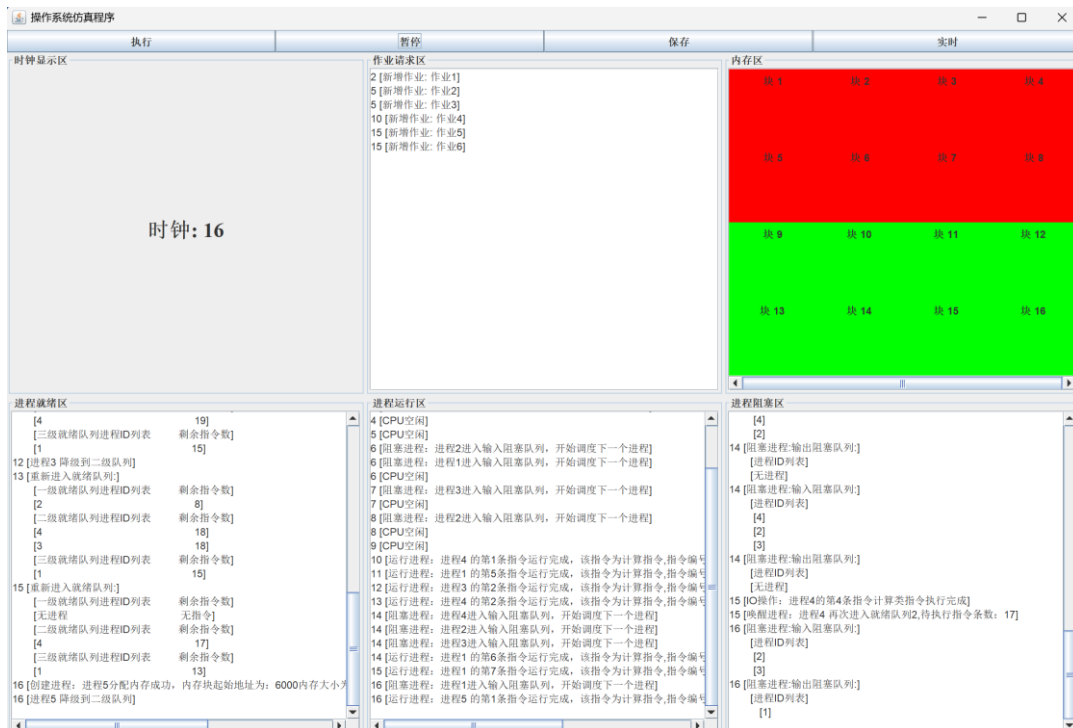


第二步，点击“执行”按钮，程序开始执行。具体如下图所示：



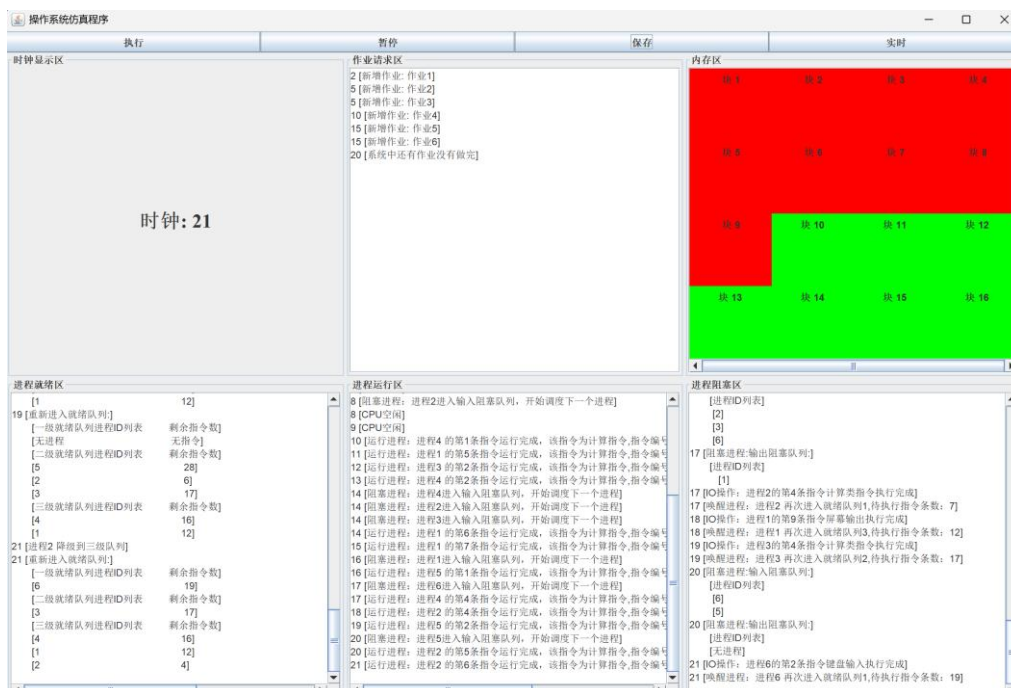
可以观察到，程序一点击“执行”，分别在第 2 和第 5 秒新增了作业 1、2 以及 3。内存显示去显示了内存占用情况；进程就绪区显示了进程 1，2，3 进入相应的就绪队列；进程运行区显示了 CPU 运行进程的执行以及中断阻塞以及可能出现的空闲情况；进程阻塞区显示了因为 IO 中断进入相应的阻塞队列的信息。

第三步，我们在第 16 秒点击“暂停”按钮，程序暂停。相应的运行界面情况如下图所示

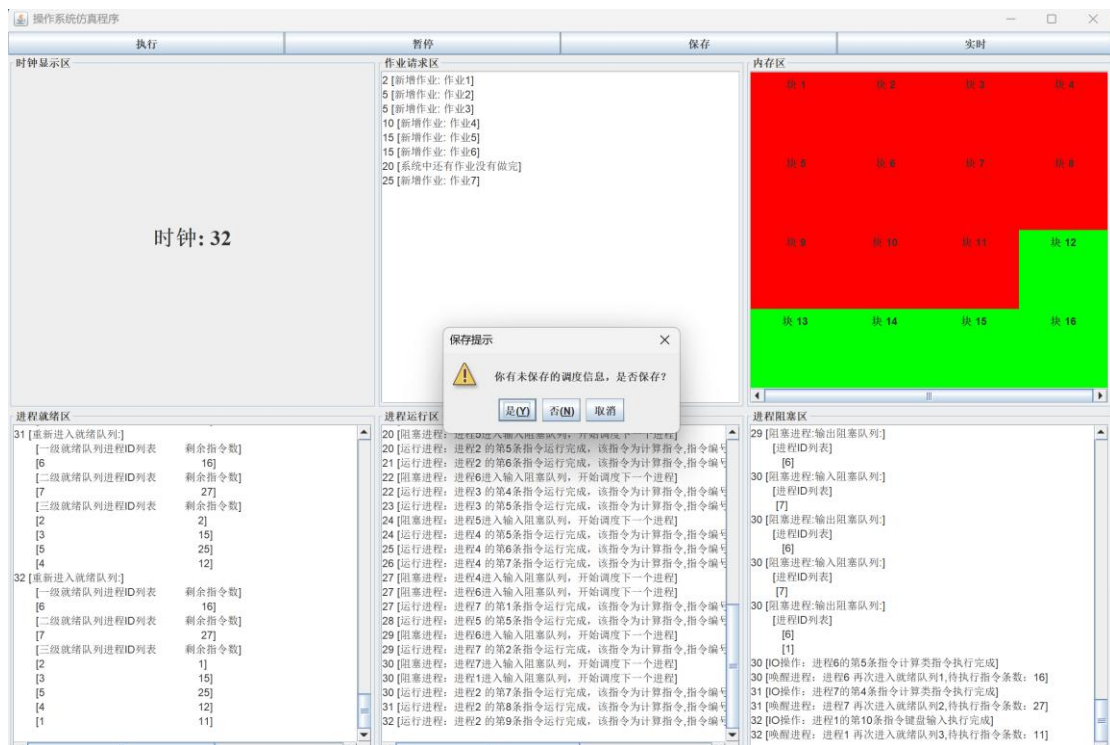


此时刻可以观察到：在第 10 秒以及第 15 秒新增了作业 4, 5, 6.内存区显示了最新的内存占用情况。进程就绪区更新显示了进程的降级以及重入就绪队列的信息，并打印了就绪队列中的进程 ID 情况。进程运行区显示了相应进程的运行和阻塞情况。进程阻塞区打印了阻塞队列中的进程 ID 列表以及做完某个进程的 IO 指令的信息。

第 4 步，我们再次点击“执行”按钮，程序再次接着上一次暂停时刻开始执行。但是如果进程还没有结束，我们贸然点击“保存”按钮，在作业请求区，会打印显示不能保存进程的运行以及阻塞信息，因为还有作业没有做完。具体的 UI 界面如下图所示：

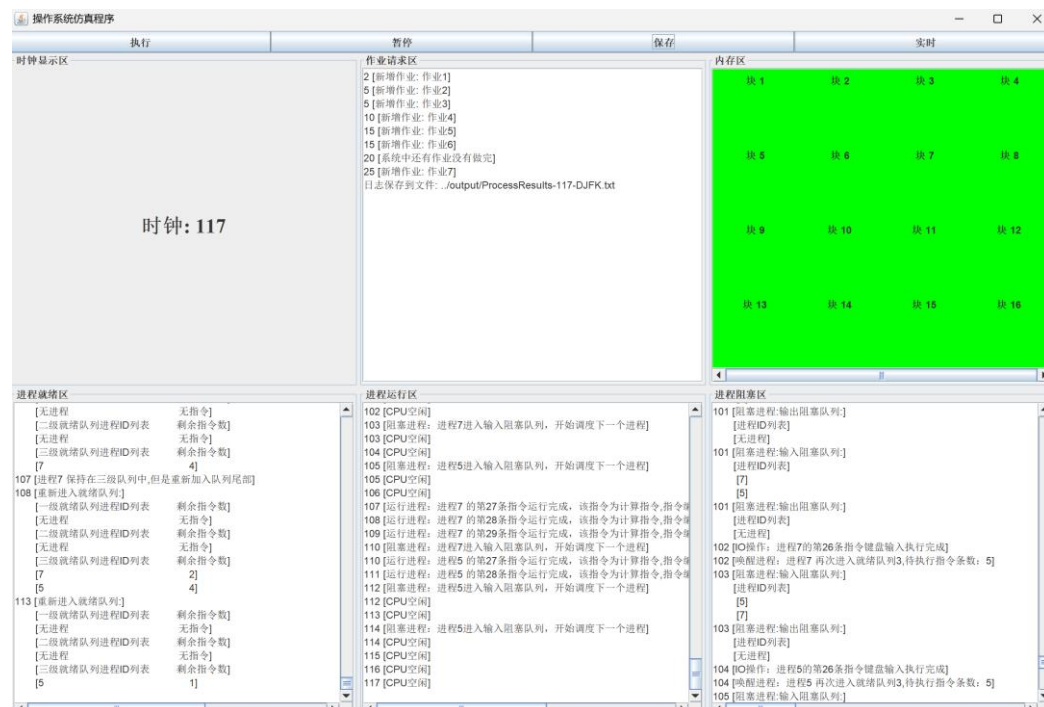


第5步, 如果我们这个时候点击右上角的退出, 会弹出一个小提示我们: 因为进程没有做完没有保存信息, 提示我们是否直接退出还是保存信息, 还是继续执行。相应的 UI 界面如下图所示:



如果我们点击“是”, 会强制保存运行到当前的信息, 如果点击“否”, 我们会直接退出程序, 不保存。如果我们点击“取消”, 会继续执行。这里我们点击“取消”。

第6步, 我们一直等待到所有作业运行结束。在第117秒, 所有进程执行结束, 我们这个时候再次点击保存, 会在作业请求区显示: 保存成功。相应的 UI 界面如下图所示:

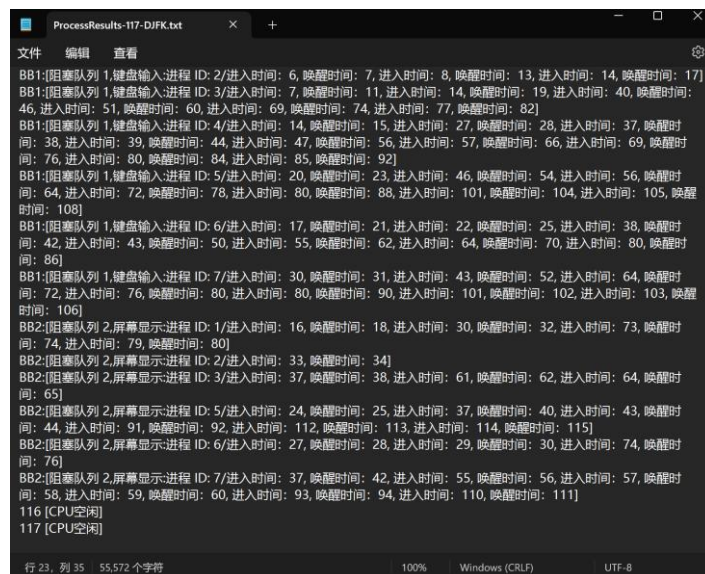


这个时候，内存显示区表明所有物理块空闲，进程运行区显示：CPU 空闲，进程就绪区以及进程阻塞区不在继续打印信息，因为没有新进程了。

最后，生成的 ProcessResults-117-DJFK.txt 文件如下图所示：



```
1 [CPU空闲]
2 [新建作业：作业1]
3 [创建进程：进程1分配内存成功，内存块起始地址为：0内存大小为：1000B,进入就绪队列1,待执行指令条数为：20]
4 [运行进程：进程1的第1条指令运行完成，该指令为计算指令,指令编号为0，数据大小为100B，指令的物理地址为：100]
5 [进程1 等待进入二级队列]
6 [运行进程：进程1的第2条指令运行完成，该指令为计算指令,指令编号为0，数据大小为100B，指令的物理地址为：200]
7 [阻塞进程：进程1进入输入阻塞队列，开始调度下一个进程]
8 [阻塞进程输入阻塞队列]
9 [进程ID列表]
10 [1]
11 [阻塞进程输出阻塞队列]
12 [进程ID列表]
13 [无进程]
14 [CPU空闲]
15 [IO操作：进程1的第3条指令键盘输入执行完成]
16 [唤醒进程：进程1再次进入就绪队列2,待执行指令条数：17]
17 [重新进入就绪队列]
18 [一级就绪队列进程ID列表 剩余指令数]
19 [无进程 无指令]
20 [二级就绪队列进程ID列表 剩余指令数]
21 [1 17]
22 [三级就绪队列进程ID列表 剩余指令数]
23 [无进程 无指令]
24 [新建作业：作业2]
25 [新建作业：作业3]
26 [创建进程：进程2分配内存成功，内存块起始地址为：1000内存大小为：600B,进入就绪队列1,待执行指令条数为：10]
27 [阻塞进程：进程2进入输入阻塞队列，开始调度下一个进程]
```



```
BB1:[阻塞队列 1,键盘输入-进程 ID: 2/进入时间：6, 唤醒时间：7, 进入时间：8, 唤醒时间：13, 进入时间：14, 唤醒时间：17]
BB1:[阻塞队列 1,键盘输入-进程 ID: 3/进入时间：7, 唤醒时间：11, 进入时间：14, 唤醒时间：19, 进入时间：40, 唤醒时间：46, 进入时间：51, 唤醒时间：60, 进入时间：69, 唤醒时间：74, 进入时间：77, 唤醒时间：82]
BB1:[阻塞队列 1,键盘输入-进程 ID: 4/进入时间：14, 唤醒时间：15, 进入时间：27, 唤醒时间：28, 进入时间：37, 唤醒时间：38, 进入时间：39, 唤醒时间：44, 进入时间：47, 唤醒时间：56, 进入时间：57, 唤醒时间：66, 进入时间：69, 唤醒时间：76, 进入时间：80, 唤醒时间：84, 进入时间：85, 唤醒时间：92]
BB1:[阻塞队列 1,键盘输入-进程 ID: 5/进入时间：20, 唤醒时间：23, 进入时间：46, 唤醒时间：54, 进入时间：56, 唤醒时间：64, 进入时间：72, 唤醒时间：78, 进入时间：80, 唤醒时间：88, 进入时间：101, 唤醒时间：104, 进入时间：105, 唤醒时间：108]
BB1:[阻塞队列 1,键盘输入-进程 ID: 6/进入时间：17, 唤醒时间：21, 进入时间：22, 唤醒时间：25, 进入时间：38, 唤醒时间：42, 进入时间：43, 唤醒时间：50, 进入时间：55, 唤醒时间：62, 进入时间：64, 唤醒时间：70, 进入时间：80, 唤醒时间：86]
BB1:[阻塞队列 1,键盘输入-进程 ID: 7/进入时间：30, 唤醒时间：31, 进入时间：43, 唤醒时间：52, 进入时间：64, 唤醒时间：72, 进入时间：76, 唤醒时间：80, 进入时间：80, 唤醒时间：90, 进入时间：101, 唤醒时间：102, 进入时间：103, 唤醒时间：106]
BB2:[阻塞队列 2,屏幕显示-进程 ID: 1/进入时间：16, 唤醒时间：18, 进入时间：30, 唤醒时间：32, 进入时间：73, 唤醒时间：74, 进入时间：79, 唤醒时间：80]
BB2:[阻塞队列 2,屏幕显示-进程 ID: 2/进入时间：33, 唤醒时间：34]
BB2:[阻塞队列 2,屏幕显示-进程 ID: 3/进入时间：37, 唤醒时间：38, 进入时间：61, 唤醒时间：62, 进入时间：64, 唤醒时间：65]
BB2:[阻塞队列 2,屏幕显示-进程 ID: 5/进入时间：24, 唤醒时间：25, 进入时间：37, 唤醒时间：40, 进入时间：43, 唤醒时间：44, 进入时间：91, 唤醒时间：92, 进入时间：112, 唤醒时间：113, 进入时间：114, 唤醒时间：115]
BB2:[阻塞队列 2,屏幕显示-进程 ID: 6/进入时间：27, 唤醒时间：28, 进入时间：29, 唤醒时间：30, 进入时间：74, 唤醒时间：76]
BB2:[阻塞队列 2,屏幕显示-进程 ID: 7/进入时间：37, 唤醒时间：42, 进入时间：55, 唤醒时间：56, 进入时间：57, 唤醒时间：58, 进入时间：59, 唤醒时间：60, 进入时间：93, 唤醒时间：94, 进入时间：110, 唤醒时间：111]
116 [CPU空闲]
117 [CPU空闲]
```

按照要求，.txt 文件按照时钟顺序保存了所有要求的信息，包括运行、阻塞、就绪队列、阻塞队列以及每当有进程执行结束的时候打印进程的状态信息——出入阻塞队列和进程的周转时间。

(2) 对(1)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.功能测试（共 60 分）

2.1 时钟中断线程运行单步测试及原理论述

【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】

（1）给出时钟中断线程类及相关数据结构伪码，并逐行说明实现原理；

以下是时钟线程的伪代码示例：

```
1.  class ClockInterruptHandlerThread extends Thread {
2.      private UI ui                // UI 的引用
3.      public static int simulationTime = 0 // 共享时间变量
4.      public static int milliseconds = 1000 // 计时间隔
5.      method run() {
6.          while (true) {
7.              acquire pauseLock        // 锁定暂停状态
8.              try {
9.                  while (isPaused) {
10.                     wait on pauseCondition // 如果暂停，等待通知
11.                 }
12.             } catch (InterruptedException e) {
13.                 handle exception
14.             } finally {
15.                 release pauseLock
16.             }
17.
18.             acquire lock                // 锁定共享资源
19.             acquire lock1                // 锁定 IO 相关资源
20.             try {
21.                 if (!pst_clk) {
22.                     wait on pst_clk_Condition // 等待进程调度信号
23.                 }
24.                 pst_clk = false // 重置进程调度标志
25.
26.                 if (!io_clk) {
27.                     wait on io_clk_Condition // 等待 IO 信号
28.                 }
29.                 io_clk = false // 重置 IO 标志
30.             }
```

```

31.          simulateTimePassing(milliseconds) // 模拟时钟前进 1 秒
32.          update UI with current simulationTime // 更新 UI 显示当前时间
33.
34.          JobRequest() // 查询是否有作业请求
35.          signal pstCondition // 通知进程调度线程
36.          signal ioCondition // 通知 IO 线程
37.      } catch (Exception e) {
38.          handle exception
39.      } finally {
40.          release lock
41.          release lock1
42.      }
43.  }
44.  }
45.
46.  method getCurrentTime() {
47.      return simulationTime
48.  }
49.
50.  method simulateTimePassing(int milliseconds) {
51.      sleep for milliseconds // 暂停指定毫秒数
52.      simulationTime++ // 时间增加 1 秒
53.  }
54. }

```

时钟中断线程类 (ClockInterruptHandlerThread) 的核心作用是模拟系统时钟的运行，并根据时钟的“中断”机制控制整个系统的运行节奏，尤其是多级反馈队列的进程调度和 I/O 操作。这种机制确保了在每一秒（或特定时间片）内，系统能够有序地处理进程调度和 I/O 操作。

时钟中断线程类的核心原理

1. 时钟模拟：

- 该线程模拟系统时钟的行为，使用 `simulateTimePassing(milliseconds)` 方法每隔 1000 毫秒（即 1 秒）更新一次系统时间 (`simulationTime`)。
- **simulationTime:** 全局共享的时间变量，记录模拟系统的运行时间。每次时钟“滴答”时，时间加 1 秒，系统就会根据新的时间更新所有相关进程和 I/O 操作的状态。

2. 线程的暂停与恢复机制：

- 在主循环的开始，通过 `pauseLock` 和 `pauseCondition` 控制线程的暂停和恢复功能。这部分代码用于实现线程的“暂停-恢复”功能，常见于操作系统中的挂起或继续机制。
- 当 `isPaused` 为真时，线程会进入等待状态，直到收到恢复的通知 (`notify` 或类似操作)，然后继续运行。

3. 锁与同步机制：

- 时钟线程需要与进程调度和 I/O 线程协同工作，因此使用多个锁 (lock 和 lock1) 来确保在操作共享资源时不会产生数据竞争。
- lock 用于同步与进程调度有关的资源，lock1 用于同步 I/O 相关的资源，防止多个线程同时访问和修改这些资源。

4. 进程调度与 I/O 信号同步:

- **进程调度信号:** 通过 pst_clk 标志和 pst_clk_Condition 控制，当时钟中断到来时，如果 pst_clk 为假，时钟线程会等待进程调度信号。收到信号后，继续执行，并将 pst_clk 重置为 false，等待下一次时钟中断。
- **I/O 信号:** 同样，时钟中断还通过 io_clk 和 io_clk_Condition 控制 I/O 操作的同步。当 io_clk 为假时，时钟线程会等待 I/O 信号，直到信号触发后才继续。

5. 时钟滴答的处理:

- 每次时钟滴答（1 秒过去），调用 simulateTimePassing(millisecods)，先暂停指定的时间（1 秒），然后将全局时间 simulationTime 加 1。
- 此时，UI 界面也会被更新，显示当前系统的模拟时间（update UI with current simulationTime）。

6. 作业请求的查询与处理:

- 时钟线程每次中断都会调用 JobRequest()，检查是否有新的作业需要处理，并通过 signal pstCondition 和 signal ioCondition 通知进程调度线程和 I/O 线程，确保它们可以根据最新的时钟时间处理各自的任务

ClockInterruptHandlerThread 线程的核心作用是**模拟系统时钟的运行**，并控制整个系统的进程调度和 I/O 操作。通过时钟中断机制，它能够让进程调度线程和 I/O 线程在合适的时机运行，确保系统能够在每秒的时间片内有效地处理任务。

（2）通过在程序内设置断点单步运行时钟中断线程；跟踪显示时钟变量值的变化情况；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及原理论述

【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】

（1）给出作业请求查询操作及相关数据结构伪码，以及每 2 秒激活该线程的过程代码，并逐行说明实现原理

```
1. public static FileUtils fileUtils=new FileUtils();//全局文件处理,信息保存,输入输出的对象
```

```

2. public static Queue<Job> backupQueue = new LinkedList<>(); //缓冲区，
   硬盘，存放作业
3. public static Queue<PCB> back_pcbQueue=new LinkedList<>();//进程缓冲区
   队列
4.
5.     method JobRequest()
6.         if (currentTime % 2 == 0) // 每 2 秒查询
7.             if (backupQueue is not empty) // 如果作业缓冲区不为空
8.                 for each job in backupQueue // 遍历作业缓冲区
9.                     if (job.inTime <= currentTime) // 判断作业到达时间
10.                        Job job = backupQueue.poll() // 移除作业
11.                        PCB pcb = createPCB(job) // 创建 PCB
12.                        back_pcbQueue.add(pcb) // 添加 PCB 到进程缓冲区
13.                        logEntry = generateLogEntry(job) // 生成日志
14.                        print(logEntry) // 打印日志
15.                        updateUI(logEntry) // 更新 UI 显示
16.                        saveLog(logEntry) // 保存日志到文件
17. method createPCB(job)
18.     return new PCB(job.jobId, job.inTime, job.instructionCount, job.i
   nstructions)
19.
20. method generateLogEntry(job)
21.     return job.inTime + " [新增作业: 作业" + job.jobId + "]"
22.
23. method updateUI(logEntry)
24.     ui.say_to_job_are(logEntry)
25.
26. method saveLog(logEntry)
27.     OSKernel.fileUtils.collectLog(logEntry)

```

原理解释：

作业请求查询方法 (JobRequest):

判断时间: if (ClockInterruptHandlerThread.getCurrentTime() % 2 == 0): 每 2 秒检查一次是否有新的作业请求。

检查缓冲区: if (!OSKernel.backupQueue.isEmpty()): 如果作业缓冲区不为空，进入作业处理逻辑。

遍历作业: for each job in OSKernel.backupQueue: 遍历缓冲区中的所有作业。

检查作业到达时间: if (job.getInTime() <= ClockInterruptHandlerThread.getCurrentTime()): 判断作业是否已到达，如果到达，则处理该作业。

移除作业: Job poll = OSKernel.backupQueue.poll(): 从缓冲区移除到达的作业。

创建 PCB: PCB pcb = new PCB(...): 为新作业创建进程控制块 (PCB)。

添加 PCB 到缓冲区队列: OSKernel.back_pcbQueue.add(pcb): 这里只是创建了 PCB，并不正真意义上的进程，因为还没有检查内存空间是否被足够。

生成日志: logEntry = job.getInTime() + " [新增作业: 作业" + job.getJobId() + "]": 生成日

志信息。

打印日志：print(logEntry)：将日志信息输出到控制台。

更新 UI：ui.say_to_job_are(logEntry)：调用 UI 的方法更新作业请求的显示。

保存日志：OSKernel.fileUtils.collectLog(logEntry)：将日志保存到数据结构中，以便后续保存和分析。

(2) 以 2-05-0.txt 为例，设置断点单步运行跟踪作业请求查询操作，显示每 2 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 进程调度线程单步测试与原理论述

【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】

(1) 给出进程调度线程与时钟中断线程同步操作的伪码，并逐行逐变量说明实现原理；

```

1. // 进程调度线程的 run 方法
2. method run()
3.     while (true) // 无限循环
4.         OSKernel.cpu.CPU_PRO() // 执行 CPU 相关操作
5.         lock1.lock() // 获取锁，确保线程安全
6.         try
7.             pst_clk = true // 设置进程调度标志为真
8.             pst_clk_Condition.signal() // 通知等待的时钟线程
9.             pstCondition.await() // 等待调度信号
10.            runPCB() // 检查能否创建进程并开始调度
11.        catch (Exception e)
12.            print(e) // 处理异常
13.        finally
14.            OSKernel.cpu.CPU_REC() // 恢复 CPU 状态
15.            lock1.unlock() // 释放锁
16.
17.
18. // 时钟线程的调度代码
19. method clockThreadRun()
20.     lock1.lock() // 获取进程调度锁
21.     try

```



```

22.         if (!pst_clk)
23.             pst_clk_Condition.await() // 等待进程调度线程的信号
24.             pst_clk = false // 重置进程调度标志
25.             simulateTimePassing(milliseconds) // 时钟走 1 秒
26.             invokeLater(() -> ui.updateClock(simulationTime)) // 更新 UI 中
                的时钟显示
27.             JobRequest() // 查询作业请求
28.             pstCondition.signal() // 通知进程调度线程
29.         catch (Exception e)
30.             print(e) // 处理异常
31.         finally
32.             lock1.unlock() // 释放进程调度锁

```

原理解释：

进程调度线程

1. while (true): 无限循环，线程持续运行以进行进程调度。
2. OSKernel.cpu.CPU_PRO(): 调用 CPU 处理相关的逻辑，准备进行调度。
3. lock1.lock(): 获取进程调度线程的锁，以确保对共享资源的安全访问。
4. pst_clk = true: 设置 pst_clk 标志为 true，表示进程调度线程已经先获得了锁，并且正在等待 pstCondition 信号，准备进行调度。
5. pst_clk_Condition.signal(): 通知在 pst_clk 条件上等待的时钟线程，进程调度线程准备就绪。
6. pstCondition.await(): 进程调度线程等待，直到被唤醒（由时钟线程或其他线程）。
7. runPCB(): 检查是否可以创建新的进程并开始调度。
8. catch (Exception e): 捕获并处理异常，以避免程序崩溃。
9. finally 部分确保即使发生异常，锁也会被释放，并且 CPU 状态会恢复。

时钟中断线程（Clock Interrupt Thread）

1. lock1.lock(): 获取进程调度的锁，以确保在执行调度操作时的安全。
2. if (!pst_clk): 检查进程调度标志 pst_clk 是否为 false。如果为 false，表示进程调度线程尚未获得锁，并且正在等待调度线程发出的信号。此时，当前线程需在 pst_clk_Condition 上等待，以保持同步。
3. pst_clk_Condition.await(): 进入条件变量的等待状态，时钟线程会暂停执行，直到收到进程调度线程的信号。
4. pst_clk = false: 当被唤醒后，重置 pst_clk 标志为 false，表示时钟线程将继续运行，并且进程调度线程已经成功获得了锁。
5. simulateTimePassing(milliseconds): 模拟时间的流逝，执行时钟逻辑。
6. invokeLater(() -> ui.updateClock(simulationTime)): 在 UI 线程中更新时钟显示，确保 UI 更新是线程安全的。
7. JobRequest(): 查询是否有新的作业请求。
8. pstCondition.signal(): 通知等待在 pstCondition 上的进程调度线程，表示时钟线程已完成时间更新。
9. catch (Exception e): 捕获并处理异常。

10. finally 部分确保即使发生异常，锁也会被释放。

(2) 给出进程调度算法函数伪码，并逐行逐变量说明实现原理；

```

1. public static final int TIME_SLICE_1 = 1; // 一级队列时间片 1 秒
2. public static final int TIME_SLICE_2 = 2; // 二级队列时间片 2 秒
3.     public static final int TIME_SLICE_3 = 4; // 三级队列时间片 4 秒
4. public int timeSlice; // 当前时间片
5.
6. MFQ():
7.     当 (readyQueue1 非空 或 readyQueue2 非空 或 readyQueue3 非空) 时:
8.         根据 选择就绪队列() 的结果:
9.             情况 1:
10.                CPU.当前进程 = readyQueue1.peek()
11.             情况 2:
12.                CPU.当前进程 = readyQueue1.peek()
13.             情况 4:
14.                CPU.当前进程 = readyQueue1.peek()
15.             默认:
16.                CPU.PSW = -1
17.                CPU.runProcess() //运行进程
18.             如果 CPU.PSW == 0:
19.                 如果 pcb_is_finised(CPU.当前进程):
20.                     跳出循环 // 调度线程只能工作 1 秒一次，因为一条计算指令用
                        时一秒
21.             如果 CPU.PSW == 1:
22.                 时间片已用完 = 真 // 因为发生了 IO 阻塞，重新选择新的进程执行
                        非计算类指令
23.
24.
25.
26. select_ready_queue():
27.     如果 is_time_used==true //时间片已用完:
28.         如果 就绪队列 1 非空:
29.             timeSlice = TIME_SLICE_1
30.         否则如果 就绪队列 2 非空:
31.             timeSlice = TIME_SLICE_2
32.         否则:
33.             timeSlice = TIME_SLICE_3
34.         返回 timeSlice
35.     否则:
36.         返回 timeSlice
37.

```

```
38.
39.
40. isfinised(): //判断时间片是否用完
41.     如果 CPU.当前进程.已执行计算指令数 == 1:
42.         is_time_used = true
43.         降级进程(CPU.当前进程)
44.     否则如果 CPU.当前进程.已执行计算指令数 == 3:
45.         is_time_used = true
46.         降级进程(CPU.当前进程)
47.     否则如果 CPU.当前进程.已执行计算指令数 > 3:
48.         count = CPU.当前进程.已执行指令数 - 3
49.         is_time_used = (count % 时间片 3 == 0)
50.         如果 is_time_used==true: //如果时间片已经使用完了
51.             downgradeProcess (CPU.当前进程)
52.     否则:
53.         is_time_used = false
54.
55.
56.
57. pcb_is_finised (pcb):
58.     如果 pcb.PC >= pcb.指令数:
59.         pcb.总运行时间 = 当前时间
60.         内存.释放内存(pcb)
61.
62.     根据 pcb.时间片:
63.         情况 1: readyQueue1.移除(pcb)
64.         情况 2: readyQueue2.移除(pcb)
65.         情况 4: readyQueue3.移除(pcb)
66.     返回 true
67. isfinised()
68. 返回 false
69.
70.
71. downgradeProcess(pcb):
72.     如果 readyQueue1.包含(pcb):
73.         readyQueue1.移除(pcb)
74.         readyQueue2.添加(pcb)
75.         pcb.时间片 = TIME_SLICE_2
76.     否则如果 readyQueue2.包含(进程):
77.         readyQueue2.移除(进程)
78.         readyQueue3.添加(进程)
79.         pcb.时间片 = TIME_SLICE_3
80.     否则:
81.         readyQueue3.移除(进程)
```

```

82.         readyQueue3.添加(进程)  // 重新加入到队尾
83.         pcb.时间片 = TIME_SLICE_3

```

原理解释：

整个调度算法的核心思想是根据进程的时间片和运行情况，动态调整进程的队列位置，从而实现不同时间片的任务调度。算法主要由以下几个功能模块组成：

1. 时间片的初始化和就绪队列的设置：

- 你的代码中有三个不同的就绪队列 `readyQueue1`、`readyQueue2` 和 `readyQueue3`，每个队列对应不同的时间片，分别为 1 秒、2 秒和 4 秒（`TIME_SLICE_1`、`TIME_SLICE_2`、`TIME_SLICE_3`）。
- 进程进入 CPU 调度时，最初被放入优先级最高的队列（`readyQueue1`），如果它在时间片内未完成任务或发生 I/O 中断，则被降级到低优先级队列，直至完成任务。

2. 调度器的主循环（MFQ）：

- **MFQ** 是调度算法的核心控制器。它不断检查三个就绪队列（`readyQueue1`，`readyQueue2`，`readyQueue3`）是否有进程待处理。如果任意队列不为空，它会调用 `select_ready_queue()` 来选择最高优先级的非空队列。
- 调度器会将最高优先级队列中的进程（`readyQueue.peek()`）分配给 CPU 运行，通过 `CPU.runProcess()` 执行。
- 如果执行过程中，CPU 碰到 I/O 阻塞或时间片用完，会重新选择新的进程进行调度；否则进程继续执行，直至完成。

3. 选择就绪队列（`select_ready_queue()`）：

- `select_ready_queue()` 用于根据进程当前的状态和是否已用完时间片，选择适当的队列执行。优先选择时间片小的队列，保证优先级高的进程得到及时处理。
- 如果时间片已用完（`is_time_used == true`），则根据队列状态重新分配时间片，并返回下一个队列中等待调度的进程。

4. 判断时间片是否用完（`isfinised()`）：

- 每个进程都有一个记录其执行时间的属性。`isfinised()` 函数通过检查进程已执行的指令数，判断其是否已经用完时间片。
- 如果时间片用完，`is_time_used` 被设置为 `true`，同时调用 `downgradeProcess()` 将进程降级到更低优先级队列继续执行。
- 对于优先级高的进程（在一级队列或二级队列中的），如果它们没有在规定时间内完成任务，会被降级到下一队列，延长时间片处理。

5. 进程的降级机制（`downgradeProcess(pcb)`）：

- 该函数用于处理时间片用完后的进程降级。通过检查进程当前所在的队列，决定是否将其从 `readyQueue1` 移到 `readyQueue2`，或者从 `readyQueue2` 移到 `readyQueue3`。
- 每次降级，进程的时间片也会相应调整。例如，从一级队列降级到二级队列，时间片会从 1 秒增加到 2 秒。

6. 进程完成的处理（`pcb_is_finised(pcb)`）：

- 该函数检查当前进程是否已经完成所有指令。当 `pcb.PC >= pcb.指令数`（程序计数器达到或超过总指令数）时，说明进程已经执行完毕。

- 完成的进程会从对应的就绪队列中移除，并调用内存管理模块释放内存资源。

核心变量与逻辑说明

- **readyQueue1、readyQueue2、readyQueue3:** 三个就绪队列，按照优先级高低存放进程。一级队列优先处理紧急任务，三级队列处理长时间运行任务。
- **timeSlice:** 当前进程的时间片，根据所在队列动态调整（1 秒、2 秒或 4 秒）。
- **is_time_used:** 布尔变量，标识时间片是否已经用完。如果用完，进程会被降级。
- **CPU.PSW:** 进程状态字，用于判断进程是正常运行、被中断还是完成任务。
- **pcb.PC 和 pcb.指令数:** PC 是程序计数器，记录进程已经执行的指令数；指令数 是进程的总指令数，用于判断进程是否完成。

整体流程总结

整个调度算法的核心机制是**多级队列调度**，并辅以**时间片管理与进程降级机制**。完整流程可以概括如下：

1. **进程进入调度:** 新进程先进入最高优先级的 readyQueue1，并分配最短时间片（1 秒）。
2. **运行过程:**
 - 调度器通过 MFQ() 主循环检查队列，选择最高优先级的进程，并调用 CPU.runProcess() 开始执行。
 - 每次调度线程工作时，会检测 CPU 的状态字（PSW），判断是继续执行还是发生了 I/O 中断。
3. **时间片管理:**
 - 如果时间片已用完，isfinished() 函数会触发降级逻辑，将进程从高优先级队列移到低优先级队列。
 - 进程在不同队列中经历越来越长的时间片，直到其在最低优先级的 readyQueue3 中完成所有工作。
4. **进程完成:** 当进程执行完毕，pcb_is_finised(pcb) 函数负责将其从队列中移除，释放内存并结束调度。

（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 2-02-0.txt 和 3.txt 进程调度的完整过程，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

评价分数：

理由：

2.4 CPU、PCB 及进程操作原语实现讲解

【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】

(1) 给出 CPU 现场保护操作的实现伪码，并逐行逐变量说明实现原理；

1. CPU_PRO() 方法

```
2. public void CPU_PRO() {
3.     setPsw(0);
4. }
```

setPsw(0): 这行代码将程序状态字 (PSW) 设置为 0，表示 CPU 进入用户态。在进程运行前，CPU 应该从内核态切换到用户态，这确保用户进程只能访问受限资源，不会破坏系统安全。

2. runProcess() 中的 CPU 相关操作

在 runProcess() 方法中，实际存在更多与 CPU 状态相关的逻辑，包括对 PSW、PC (程序计数器) 等的操作。

```
1. int current_pc = this.getCurrentProcess().getPc(); // 获取当前进程的程
   序计数器
2. int current_ir_state = this.getCurrentProcess().getIr(); // 获取当前
   指令的类型
3.
4. pc = current_pc; // 将 CPU 的 PC 设为当前进程的 PC
5.
6. if (current_ir_state == 0) { // 处理计算类指令
7.     this.setPsw(0); // CPU 切换到用户态
8.     this.getCurrentProcess().note_cal(); // 执行计算指令
9.     current_pc++;
10.    this.getCurrentProcess().setPc(current_pc); // 更新进程的 PC
11.    psw = 0; // CPU 仍然在用户态
12. }
13.
14. else if (current_ir_state == 1) { // 处理 IO 类型的指令
15.     this.setPsw(1); // CPU 切换到内核态
16.     // ...
17.     psw = 1; // CPU 仍然处于内核态
18. }
```

current_pc = this.getCurrentProcess().getPc(); 获取当前进程的程序计数器 PC，这是 CPU 状态中的重要一部分。它指向下一条将被执行的指令地址。

current_ir_state = this.getCurrentProcess().getIr(); 获取当前的指令状态，决定接下来执行的是计算类指令还是 IO 指令。

pc = current_pc; 将当前进程的 PC 赋值给 CPU，表示 CPU 准备好继续执行该进程的指令。

setPsw(0): CPU 切换到用户态，表示此时进程正在执行用户级别的指令。

current_pc++: 执行完一条指令后，程序计数器递增，指向下一条即将执行的指令。

setPsw(1): 当处理 IO 指令时，CPU 切换到内核态，这使 CPU 能够执行特

权操作。

3. ProcessSchedulingHandlerThread 中的 CPU 保护操作

```

1.  @Override
2.  public void run() {
3.      while (true) { // 无限循环, 调度进程
4.          OSKernel.cpu.CPU_PRO(); // 现场保护
5.          SyncManager.lock1.lock(); // 获取锁, 保证线程安全
6.          try {
7.              SyncManager.pst_clk = true;
8.              SyncManager.pst_clk_Condition.signal(); // 唤醒等待的线程
9.              SyncManager.pstCondition.await(); // 等待调度信号
10.             runPCB(); // 调度新的进程
11.         } catch (Exception e) {
12.             e.printStackTrace(); // 错误处理
13.         } finally {
14.             OSKernel.cpu.CPU_REC(); // 恢复现场
15.             SyncManager.lock1.unlock(); // 释放锁
16.         }
17.     }
18. }

```

OSKernel.cpu.CPU_PRO(); 在每次调度新的进程之前, 调用 CPU_PRO(), 执行 CPU 现场保护。虽然 CPU_PRO() 目前只做了简单的 PSW 切换, 但这是进程切换的关键步骤之一。

OSKernel.cpu.CPU_REC(); 这是现场恢复操作, 表示在新进程被调度后, 将之前保存的 CPU 状态重新恢复, 以便进程继续运行。

(2) 给出 PCB 表、进程创建、撤销、进程切换的伪码, 并逐行逐变量说明实现原理;

1. 1. 进程创建的伪代码

```

function create_pcb(pcb):
2.     if (系统并发进程数 > 12):
3.         打印 "无法为作业" + pcb.getPid() + "创建进程, 系统最大并发进程数
           已达到 12"
4.         更新 UI 界面
5.         记录日志
6.         return false
7.     else:
8.         if (成功分配内存(pcb)):
9.             将 pcb 加入一级就绪队列
10.            设置 pcb 状态为 "就绪"

```

```

11.          设置 pcb 的时间片为 1
12.          记录当前时间作为进入就绪队列时间
13.          记录当前时间作为进程创建时间
14.          return true
15.      else:
16.          打印 "内存已满, 无法为作业" + pcb.getPid() + "分配"
17.          更新 UI 界面
18.          记录日志
19.          return false

```

1. 检查系统的并发进程数:

- 在创建进程之前, 首先检查当前系统中是否已经存在超过 12 个并发进程(即系统设置的最大并发限制)。
- 如果超过限制, 函数会直接返回 false, 表示无法创建新的进程, 并通过 UI 和日志系统更新相应信息。

2. 内存分配尝试:

- 如果并发进程数在允许范围内, 则系统尝试为新的进程分配内存块。
- 如果内存分配成功, 进程控制块(PCB)将被添加到 **一级就绪队列**, 并且进程的状态被设置为 "就绪态"。
- 同时, 记录该进程的时间片(在此设为 1 秒), 并记录进程进入队列的时间和创建时间。

3. 内存分配失败:

- 如果系统内存不足, 无法为进程找到足够的连续内存块, 创建进程将失败, 函数返回 false, 并在 UI 和日志中反映该情况。

此逻辑背后主要涉及三个关键问题: **资源限制检查(并发限制)**、**内存资源分配**、**系统状态更新**

2.进程切换相关伪代码:

```

1. MFQ():
2.     当 (readyQueue1 非空 或 readyQueue2 非空 或 readyQueue3 非空) 时:
3.         根据 select_ready_queue() 的结果:
4.             情况 1:
5.                 CPU.当前进程 = readyQueue1.peek()
6.             情况 2:
7.                 CPU.当前进程 = readyQueue2.peek()
8.             情况 4:
9.                 CPU.当前进程 = readyQueue3.peek()
10.            默认:
11.                CPU.PSW = -1
12.            CPU.runProcess() //运行进程
13.            如果 CPU.PSW == 0:
14.                如果 pcb_is_finised(CPU.当前进程):

```



```

15.          跳出循环  // 调度线程只能工作 1 秒一次，因为一条计算指令用
           时一秒
16.          如果 CPU.PSW == 1:
17.          时间片已用完 = 真  // 因为发生了 IO 阻塞，重新选择新的进程执行
           非计算类指令

```

MFQQ 表示多级反馈队列调度算法。它从三个就绪队列中选择进程，并根据优先级依次分配给 CPU。

具体的调度过程通过**select_ready_queue()** 来选择适合的时间片，并在每个时间片到达时切换到新的进程。

如果当前的进程执行完成，或者因为发生 IO 阻塞而导致时间片用完，会重新调度下一个进程。

3.选择就绪队列伪代码：

```

1. select_ready_queue():
2.     如果 is_time_used == true  // 时间片已用完:
3.         如果 就绪队列 1 非空:
4.             timeSlice = TIME_SLICE_1
5.         否则如果 就绪队列 2 非空:
6.             timeSlice = TIME_SLICE_2
7.         否则:
8.             timeSlice = TIME_SLICE_3
9.         返回 timeSlice
10.    否则:
11.        返回 timeSlice

```

- select_ready_queue() 用来选择哪个就绪队列中的进程要被调度。
- 优先选择一级队列，其次二级，最后是三级，并根据队列情况动态设置时间片。

4. 进程完成/撤销相关伪代码：

```

5. pcb_is_finised (pcb):
6.     如果 pcb.PC >= pcb.指令数:
7.         pcb.总运行时间 = 当前时间
8.         内存.释放内存(pcb)  // 释放 PCB 占用的内存
9.         根据 pcb.时间片:
10.            情况 1: readyQueue1.移除(pcb)
11.            情况 2: readyQueue2.移除(pcb)
12.            情况 4: readyQueue3.移除(pcb)
13.         返回 true
14.     返回 false

```

• pcb_is_finised() 判断进程是否完成，若已完成则释放该进程占用的内存资源，并从相应的就绪队列中移除 PCB。

- 然后记录进程的总运行时间并返回 true 表示进程已完成，否则返回 false。

总结：

- **PCB 表**相关的操作如进程创建、进程撤销都与内存管理密切相关，创建进程时首先要确保内存充足，若进程完成则释放其占用的内存。

- **进程切换**主要依赖多级反馈队列调度算法，每次从不同优先级的队列中选择进程执行，并根据进程类型（计算型、IO 型）进行相应处理。

- **进程降级**则是在多级反馈队列调度中的重要操作，通过逐步降低优先级，保证长时间运行的进程不会占用高优先级的队列资源。

（3）设置断点单步跟踪测试进程切换过程；并通过 1-02-0.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-进程操作原语实现过程

评价分数：

理由：

2.5 实时操作单步执行测试

【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】

（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；

```

1.     private int TempIndex = new Random().nextInt(8 + 1) + 24; // 随机
      生成一个作业索引
2.
3.     // 随机生成指令
4.     // 每一行代表一个指令，第一列代表指令 ID，下标从 1 开始到 InstrucNum，
5.     // 第二列代表指令状态，有 0,1,2 三种取值，其中 0 取值概率为 0.7，1 取值概率
      为 0.2，2 取值概率为 0.1
6.     private ArrayList<Instruction> CreateIRs(int InstrucNum) {
7.         //InstrucNum 代表生成指令的数目
8.         double[] probabilities = {0.7, 0.9, 1.0}; // 各状态的概率
9.         ArrayList<Instruction> instructions = new ArrayList<>();
10.        Random random = new Random();
11.
12.        for (int i = 0; i < InstrucNum; i++) {
13.            int instructionID = i + 1;
14.            double randomValue = random.nextDouble();
15.            int instructionStatus = 0; // 默认为 0（用户态计算操作指令）
16.
17.            for (int j = 0; j < probabilities.length; j++) {
18.                if (randomValue < probabilities[j]) {
19.                    instructionStatus = j;
20.                    break;
21.                }
            }
        }
    }

```

```

22.         }
23.         Instruction instruction = new Instruction(instructionID, i
           nstructionStatus);
24.         instructions.add(instruction);
25.     }
26.     return instructions;
27. }
28.
29. public void AddOnePro() throws IOException//添加一个作业直接到后备队
    列中
30. {
31.     //随机生成一个作业，然后添加到后备队列中
32.     TempIndex++;
33.     int arriveTime=ClockInterruptHandlerThread.simulationTime;//作
        业的到达时间为当前的时钟时间
34.     //随机生成指令数目在 20-30 之间
35.     int InstrucNum = (int)(Math.random()*10)+20;
36.     // 创建作业对象并传递优先级
37.     Job job = new Job(TempIndex, arriveTime, InstrucNum, CreateIRs
        (InstrucNum));
38.     OSKernel.backupQueue.add(job);
39. }
40.
41. private void handleRealTimeJob() {
42.     try {
43.         OSKernel.fileUtils.AddOnePro();
44.     } catch (IOException e) {
45.         System.out.println(ClockInterruptHandlerThread.getCurrentT
            ime()+" [随机添加作业失败]");
46.         say_to_job_are(ClockInterruptHandlerThread.getCurrentTime(
            )+" [随机添加作业失败]");
47.     }
48. }
49.
50. realTimeButton.addActionListener(e -> handleRealTimeJob());

```

原理解释：

随机生成指令集 CreateIRs(int InstrucNum)

- **InstrucNum:** 参数，表示需要生成的指令数量。
- **probabilities:** 数组，定义了三种指令状态的累积概率，其中：
 - 0 的概率为 0.7
 - 1 的概率为 0.2
 - 2 的概率为 1.0（表示剩余的概率）

- **instructions:** 存储生成的指令列表。
 - **Random random:** 用于生成随机数的随机对象。
- 循环生成指令:
- **for (int i = 0; i < InstrucNum; i++):** 循环 InstrucNum 次, 生成指定数量的指令。
 - **instructionID:** 指令的唯一标识, 从 1 开始递增。
 - **randomValue:** 生成一个 0.0 到 1.0 的随机小数, 用于决定指令状态。
 - **instructionStatus:** 指令状态, 初始为 0, 表示计算操作指令。

确定指令状态:

- **内部循环 for (int j = 0; j < probabilities.length; j++):**
 - 检查 randomValue 是否小于 probabilities[j], 一旦满足条件, 就将 instructionStatus 设置为对应的 j 值, 并跳出循环。
 - **问题:** probabilities 数组表示的是累积概率, 需要确保各概率区间正确。例如, 应该是 {0.7, 0.9, 1.0}, 以表示状态 0 的概率为 0.7, 状态 1 的概率为 0.2, 状态 2 的概率为 0.1。

创建指令对象并添加到列表:

- **Instruction instruction = new Instruction(instructionID, instructionStatus);** 创建新的指令对象, 包含指令 ID 和状态。
- **instructions.add(instruction);** 将新指令添加到指令列表中。

返回指令列表:

- **return instructions;** 函数返回生成的指令列表。

handleRealTimeJob()

handleRealTimeJob(): 这是一个私有方法, 用于处理实时作业的添加。

try 块:

- **OSKernel.fileUtils.AddOnePro();** 调用 AddOnePro() 方法, 添加一个新的作业到后备队列中。
- **catch (IOException e):**
- 如果发生 IOException 异常, 表示作业添加失败。
- **System.out.println(...):** 在控制台打印错误信息, 包括当前模拟时间和错误提示。
- **say_to_job_are(...):** 可能是一个用于更新 UI 界面的函数, 显示错误信息给用户。

添加一个作业到后备队列 AddOnePro()

TempIndex++: 作业索引自增, 确保每个新作业都有唯一的 ID。

arriveTime: 作业的到达时间, 设置为当前的模拟时间 (simulationTime)。

InstrucNum: 随机生成的指令数量, 范围在 20 到 29 之间。

- **Math.random() * 10:** 生成 0.0 到 10.0 之间的随机小数。
- **(int)(...) + 20:** 取整并加上 20, 使得指令数目在 20 到 30 之间。

创建作业对象:

- **Job job = new Job(TempIndex, arriveTime, InstrucNum, CreateIRs(InstrucNum));**
 - **TempIndex:** 作业 ID。
 - **arriveTime:** 到达时间。
 - **InstrucNum:** 指令数量。

- **CreateIRs(InstrucNum)**: 调用之前的函数生成指令列表。

添加作业到后备队列:

- **OSKernel.backupQueue.add(job);**: 将新创建的作业添加到操作系统内核的后备队列中, 等待调度。

(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;

```

1.  函数 AddRealTimeJob():
2.      TempIndex++
3.      arriveTime = 当前的时钟时
        间 (ClockInterruptHandlerThread.getCurrentTime())
4.
5.      // 随机生成 20-30 条指令
6.      InstrucNum = 随机数(20 到 30)
7.
8.      // 创建作业对象 Job, 包含作业 ID, 到达时间, 指令数目, 以及生成的指令列表
9.      job = Job(TempIndex, arriveTime, InstrucNum, CreateIRs(InstrucNum
        ))
10.
11.     // 将作业添加到后备队列中
12.     OSKernel.backupQueue.add(job)
13.
14.
15. 函数 JobRequest():
16.     // 每两秒检查一次是否有作业可以从后备队列移入进程缓冲区
17.     如果 当前时钟时间 % 2 == 0:
18.         如果 后备队列不为空:
19.             遍历 后备队列:
20.                 如果 作业的到达时间 <= 当前时钟时间:
21.                     从后备队列取出作业
22.                     创建相应的 PCB (进程控制块)
23.                     将 PCB 添加到进程缓冲区队列
24.                     记录日志 "新增作业: 作业 ID"
25.                     更新 UI, 显示作业进入进程缓冲区
26.                     保存日志到文件
27.
28.
29. 函数 HandlePCBQueue():
30.     遍历进程缓冲区队列:
31.         如果 成功为进程分配内存:
32.             从进程缓冲区移除进程
33.             进程插入到就绪队列 1
34.             记录日志 "创建进程: 进程 ID, 内存分配成功"
35.             更新 UI, 显示进程进入就绪队列
36.             保存日志到文件

```

- 实时作业的生成并插入后备队列 (AddRealTimeJob())
- 定时查询后备队列, 将满足条件的作业转移到进程缓冲区 (JobRequest())
- 处理进程缓冲区, 将作业插入到就绪队列并开始执行 (HandlePCBQueue())

1. 实时作业的生成并插入后备队列

- 实时作业是在调用 AddRealTimeJob() 方法时生成的。该方法通过随机生成的作业 ID 和指令数目, 构造一个新的作业对象, 并将该作业放入后备队列。
- 作业的到达时间记录的是系统当前的时钟时间, 反映了作业进入系统的时刻。
- 每个作业包含一定数量的指令, 这些指令具有不同的执行类型 (如用户态计算、IO 操作等), 模拟了真实操作系统中的各种任务。
- 生成的作业被放入后备队列, 后备队列是系统中所有还未处理的作业的集合。

2. 定时查询后备队列, 将满足条件的作业转移到进程缓冲区

- 这个函数通过定时机制, 每 2 秒查询一次是否有作业可以从后备队列中转移到进程缓冲区。
- 作业到达后备队列的时间并不意味着它立即被处理, 而是要等到作业的到达时间与系统当前时钟匹配时, 才会进入进程缓冲区。
- 一旦作业满足条件, 它就会从后备队列中移出, 并生成一个对应的 PCB (进程控制块), 这是进程管理中的核心数据结构, 记录了作业的 ID、指令数、内存分配情况等。
- 这些作业的 PCB 会进入到 进程缓冲区, 等待进一步的内存分配和调度。
- 每当有作业进入缓冲区, 系统会记录日志并通过 UI 更新显示, 确保用户可以直观地看到系统内的作业状态

3. 处理进程缓冲区, 将作业插入到就绪队列并开始执行

- 该函数处理进程缓冲区中的 PCB, 核心任务是为作业分配内存。如果分配成功, 作业就会被转移到就绪队列。
- 就绪队列是等待 CPU 调度的进程集合, 尤其是 多级反馈队列 中的第一级队列。刚分配好内存的进程优先进入第一级队列等待执行。
- 为进程分配内存成功后, 系统记录相关的进程信息, 显示分配的内存起始地址和大小, 并通过日志记录下来, 同时更新 UI, 让用户可以看到进程进入就绪队列的情况。
- 这些进程将按调度算法 (多级反馈队列) 在 CPU 上执行。

通过这三个函数, 系统实现了一个完整的作业生命周期管理, 包括作业的生成、从后备队列到进程缓冲区的转移、内存分配以及进入就绪队列的全过程。整个流程是通过定时器和系统时钟来协调完成的, 确保作业在正确的时间被调度并执行。

(3) 在 3-05-0.txt 执行到第 5 条指令时按下实时作业请求按钮, 设置断点单步跟踪执行, 生成与调度完整过程; 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

评价分数：

理由：

2.6 连续动态内存分配与回收单步执行测试

【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：

(1) 给出本等级要求的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；

```

1.     public void op_allocation_add() {
2.         //每一次分配为进程分配内存之后就查找一次，更新 Allocation_add 这个
        map 集合，
3.         //每一个键。key 代表起始的物理地址
4.         //每一个 key 对应的值 value 代表从 key 开始连续空闲空间的物理块
5.         //每一次释放内存之后也要更新
6.         //总的来说，就是每一次内存空间变化了就要运行这个函数
7.         Allocation_add.clear(); // 先清空原来的映射
8.         int startAddress = -1; // 用于记录连续空闲块的起始物理地址
9.         int freeBlockCount = 0; // 连续空闲块的计数
10.
11.        for (int i = 0; i < AllocationMem.size(); i++) {
12.            Memory_Block block = AllocationMem.get(i);
13.
14.            if (!block.isOccupied()) { // 如果该块未被占用
15.                if (startAddress == -1) {
16.                    startAddress = block.Block_ID * Memory_Block.BLO
                    CK_SIZE; // 记录起始地址
17.                }
18.                freeBlockCount++; // 计数器增加
19.            }
20.            else {
21.                if (freeBlockCount > 0) {
22.                    // 如果之前记录了空闲块，更新映射
23.                    Allocation_add.put(startAddress, freeBlockCount);
24.                }
25.                // 重置计数器
26.                startAddress = -1;
27.                freeBlockCount = 0;
28.            }
29.        }
30.        // 处理最后一段连续空闲块
31.        if (freeBlockCount > 0) {

```



```
32.         Allocation_add.put(startAddress, freeBlockCount);
33.     }
34.         //当最后一段内存块是空闲的，并且遍历到最后一块时，
35.         // 我们没有机会通过占用块来触发前面的 if (freeBlockCount > 0) 来
        更新映射。
36.         // 因此，这个逻辑是为了解决 遍历结束时，最后一段连续空闲块没有被记
        录 的问题。
37.     }
38.
39.     public boolean AllocateMem(PCB pcb)
40.     {
41.         int ob_size = pcb.cal_size();
42.         int block_number = ob_size / Memory_Block.BLOCK_SIZE;
43.
44.         // 如果大小不是整块，向上取整
45.         if (ob_size % Memory_Block.BLOCK_SIZE > 0) {
46.             block_number++;
47.         }
48.
49.         int bestFitStartAddress = -1;
50.         int bestFitBlockSize = Integer.MAX_VALUE; // 选择最小的可用
        块
51.
52.         // 遍历 Allocation_add 找到适合的块
53.         for (Map.Entry<Integer, Integer> entry : Allocation_add.entrySet()) {
54.             int startAddress = entry.getKey();
55.             int freeBlocks = entry.getValue();
56.
57.             // 如果空闲块大于或等于需要的块数，并且是最小的可用块
58.             if (freeBlocks >= block_number && freeBlocks < bestFitBlockSize) {
59.                 bestFitStartAddress = startAddress;
60.                 bestFitBlockSize = freeBlocks;
61.             }
62.         }
63.         // 如果找到了适合的块，进行分配
64.         if (bestFitStartAddress != -1) {
65.             // 将内存块从 bestFitStartAddress 开始分配给 PCB
66.             int block_id = bestFitStartAddress / Memory_Block.BLOCK_SIZE; // 起始的物理块的 id
67.             if (bestFitStartAddress % Memory_Block.BLOCK_SIZE > 0) block_id++;
68.
```

```

69.         int remainingSize = pcb.cal_size(); // 进程的大小
70.
71.         for (int i = block_id; i < block_id + block_number; i++)
72.         {
73.             AllocationMem.get(i).setOccupied(true);
74.             AllocationMem.get(i).setPcb_id(pcb.getPcb_id()); // 设置被
内存块占用的 pcb 的 id
75.
76.             if (remainingSize >= Memory_Block.BLOCK_SIZE) {
77.                 // 如果剩余大小大于等于块大小，完全占用该块
78.                 AllocationMem.get(i).setoccupied(Memory_Block.BLO
CK_SIZE);
79.                 remainingSize -= Memory_Block.BLOCK_SIZE; // 减少
剩余大小
80.             }
81.             else {
82.                 // 否则，部分占用该块
83.                 AllocationMem.get(i).setoccupied(remainingSize);
// 记录实际占用的大小
84.                 remainingSize = 0; // 设置为 0，分配完成
85.             }
86.             if (remainingSize == 0) {
87.                 pcb.setPysical_address(block_id * Memory_Block.BL
OCK_SIZE);
88.                 mmu.addressss.put(pcb.getPcb_id(), pcb.getPysical_addr
ess());
89.                 // 为新创建的进程添加物理地址
90.             }
91.         }
92.         // 分配成功后，更新 Allocation_add
93.         op_allocation_add();
94.         // 在内存分配完成后更新 UI
95.         SwingUtilities.invokeLater(() -> {
96.             // 这里可以调用 UI 更新方法
97.             ui.updateMemoryStatus(AllocationMem); // 更新内存状
态
98.         });
99.         return true; // 分配成功
100.    }
101.    return false;
102. }
103.
104.

```

```

105.     public void updateMemoryStatus(ArrayList<Memory_Block> memoryBlocks) {
106.         // 遍历当前的内存块，更新显示
107.         for (int i = 0; i < memoryBlocks.size(); i++) {
108.             Memory_Block block = memoryBlocks.get(i);
109.             JPanel blockPanel;
110.
111.             // 检查该块是否已存在
112.             if (memoryArea.getComponentCount() > i) {
113.                 blockPanel = (JPanel) memoryArea.getComponent(i);
114.             } else {
115.                 blockPanel = new JPanel();
116.                 blockPanel.setPreferredSize(new Dimension(100, 30)); // 设置每个块的大小
117.                 memoryArea.add(blockPanel); // 添加到内存区域
118.             }
119.
120.             // 根据块的占用状态更新颜色
121.             if (block.isOccupied()) {
122.                 blockPanel.setBackground(Color.RED); // 被占用，设置为红色
123.             } else {
124.                 blockPanel.setBackground(Color.GREEN); // 空闲，设置为绿色
125.             }
126.
127.             blockPanel.removeAll(); // 清空之前的标签
128.             blockPanel.add(new JLabel("块 " + (block.getBlock_ID() + 1))); // 显示块编号
129.         }
130.         memoryArea.revalidate(); // 重新验证面板
131.         memoryArea.repaint(); // 刷新面板
132.     }

```

1. 内存块的表示

在这段代码中，内存被划分为多个块，每个块的状态（是否被占用）通过 `Memory_Block` 类来表示。`Memory_Block` 类通常会包含以下几个属性：

- `Block_ID`：块的标识符。
- `isOccupied()`：方法，用于判断块是否被占用。
- `setOccupied(boolean)`：设置块的占用状态。
- `setPcb_id(int)`：设置占用该块的进程控制块（PCB）ID。
- `setoccupied(int)`：记录实际占用的大小。

2. `op_allocation_add()` 方法

这个方法的主要功能是更新内存的状态映射 `Allocation_add`，它是一个哈希表（或字典），用于记录空闲内存块的起始物理地址和连续空闲块的数量。

- ****清空映射****: 在每次内存状态变化时，首先清空 `Allocation_add` 映射。
- ****遍历内存块****: 遍历所有内存块，查找空闲块。
 - 当发现空闲块时，记录其起始地址并计数。
 - 一旦遇到占用块，若之前有记录的空闲块，则将其起始地址和数量加入映射。
- ****处理最后一段****: 在遍历结束后，如果最后一段是空闲的，也要更新映射。

这个逻辑确保了在每次内存状态变化时，映射能够及时反映当前的内存使用情况。

3. `AllocateMem(PCB pcb)` 方法

这个方法负责为给定的进程控制块（PCB）分配内存。

- ****计算所需内存****: 首先计算 PCB 所需的内存大小，并确定需要的内存块数量。
- ****寻找最佳适应块****: 遍历 `Allocation_add` 映射，查找满足条件的最小可用块：
 - 如果空闲块数量大于或等于所需块数，且小于当前找到的最小块，则更新最佳适应块的起始地址和大小。
- ****进行分配****: 如果找到了适合的块，则从最佳适应块的起始地址开始分配内存：
 - 更新每个块的占用状态和占用的 PCB ID。
 - 根据实际需要分配完整或部分块。

最后，调用 `op_allocation_add()` 更新映射，并通过 Swing 的事件调度线程更新 UI。

4. `updateMemoryStatus(ArrayList<Memory_Block> memoryBlocks)` 方法

这个方法负责更新内存状态的可视化展示：

- ****遍历内存块****: 遍历当前的内存块，更新每个块的显示。
- ****创建或更新面板****: 对于每个内存块，检查其是否已在显示区域中，若不存在则创建新面板。
- ****设置颜色****: 根据块的占用状态设置颜色（红色表示占用，绿色表示空闲）。

5. 总结

整体来说，这段代码通过最佳适应算法来优化内存的分配，使得在内存中能够更有效地利用空闲块。映射 `Allocation_add` 的维护是动态的，每当内存状态变化时都会更新，确保了内存使用情况的实时性。通过可视化更新，用户可以直观地了解内存的使用情况。

背后原理

- ****最佳适应算法****: 这种算法通过找到最小的适合块来减少内存碎片，尽量保持内存的连续性。虽然这种方法在某些情况下可能导致更多的内存碎片（即，更多的小块空闲），但它在内存使用效率上表现良好。
- ****哈希表****: 使用哈希表来存储空闲内存块的状态可以快速查找和更新，使得内存分配和释放操作变得高效。

(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；

```

1.  public void freemem(PCB pcb) {
2.      // 进程已经运行结束
3.      if (pcb.getState() == -1)
4.      {
5.          // 获取进程的物理地址
6.          int startPhysicalAddress = pcb.getPhysical_address();
7.          // 计算进程占用的块数
8.          int blockNumber = pcb.cal_size() / Memory_Block.BLOCK_SIZE
          ;
9.          if (pcb.cal_size() % Memory_Block.BLOCK_SIZE > 0) {
10.              blockNumber++; // 如果还有剩余的大小，加一块
11.          }
12.          // 释放内存块
13.          for (int i = 0; i < blockNumber; i++) {
14.              int block_id = (startPhysicalAddress / Memory_Block.BLOCK_SIZE) + i; // 计算当前块的 ID
15.              Memory_Block block = AllocationMem.get(block_id); // 获取内存块
16.              // 标记块为未占用
17.              block.setOccupied(false);
18.              block.setoccupied(0); // 重置占用大小
19.              block.setPcb_id(0);
20.          }
21.          // 更新内存映射
22.          op_allocation_add(); // 重新计算空闲内存块映射
23.          // 在内存分配完成后更新 UI
24.          SwingUtilities.invokeLater(() -> {
25.              // 这里可以调用 UI 更新方法
26.              ui.updateMemoryStatus(AllocationMem); // 更新内存状态
27.          });
28.      }
29.  } // 进程运行结束，释放内存
30.
31.
32. public void updateMemoryStatus(ArrayList<Memory_Block> memoryBlocks)
    {
33.      // 遍历当前的内存块，更新显示
34.      for (int i = 0; i < memoryBlocks.size(); i++) {
35.          Memory_Block block = memoryBlocks.get(i);
36.          JPanel blockPanel;
37.

```

```

38.          // 检查该块是否已存在
39.          if (memoryArea.getComponentCount() > i) {
40.              blockPanel = (JPanel) memoryArea.getComponent(i);
41.          } else {
42.              blockPanel = new JPanel();
43.              blockPanel.setPreferredSize(new Dimension(100, 30)); /
          // 设置每个块的大小
44.              memoryArea.add(blockPanel); // 添加到内存区域
45.          }
46.
47.          // 根据块的占用状态更新颜色
48.          if (block.isOccupied()) {
49.              blockPanel.setBackground(Color.RED); // 被占用, 设置为红
          色
50.          } else {
51.              blockPanel.setBackground(Color.GREEN); // 空闲, 设置为绿
          色
52.          }
53.
54.          blockPanel.removeAll(); // 清空之前的标签
55.          blockPanel.add(new JLabel("
          块 " + (block.getBlock_ID() + 1))); // 显示块编号
56.      }
57.      memoryArea.revalidate(); // 重新验证面板
58.      memoryArea.repaint(); // 刷新面板
59.  }

```

内存释放 (freemem 方法)

1. **进程结束:** 当进程结束时, 调用 freemem 方法释放其占用的内存。
 2. **获取信息:** 通过 PCB 获取物理地址和所占用的块数。
 3. **标记为空闲:**
 - 遍历占用的内存块, 将每个块的状态标记为未占用, 重置占用大小和关联的 PCB ID。
 4. **更新映射:** 再次调用 op_allocation_add() 更新空闲内存块的映射。
 5. **UI 更新:** 通过 updateMemoryStatus 方法更新可视化界面, 确保用户能够看到最新的内存状态。
- 当进程结束时, 释放其占用的内存块, 将内存状态更新为可用。这样可以避免内存泄漏, 并保证系统能充分利用每一个内存块。
- 通过在释放后更新映射表, 可以即时反映系统的内存状态, 便于后续的内存分配。

UI 更新 (updateMemoryStatus 方法)

1. **遍历内存块:** 遍历当前所有内存块, 检查每个块的状态。
2. **更新显示:** 根据每个块的占用状态更新其显示颜色 (如红色表示占用, 绿色表示空闲)。

3. **刷新界面**: 重新验证和刷新用户界面, 以确保内存状态变化能够在屏幕上实时展示。采用 Swing 进行可视化展示, 能够清晰地显示内存的使用情况。用户可以直观地看到哪些内存块被占用, 哪些是空闲的, 有助于理解内存管理的动态过程。在用户界面更新过程中, 使用事件调度线程 (SwingUtilities.invokeLater) 来保证 UI 的流畅性和响应性。

(3) 以 1-02-0.txt、2-05-0.txt、3-05-0.txt 作业执行为例, 设置断点单步跟踪运行内存分配与回收完整过程; 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](#)

评价分数:

理由:

2.7 进程阻塞唤醒线程单步测试与原理论述 (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】

(1) 给出进程阻塞线程唤醒、调用阻塞过程伪码, 并逐行逐变量说明实现原理;

```

1.    FUNCTION ALL_IO()
2.    IF I_block_queue NOT EMPTY THEN
3.        pcb = I_block_queue PEEK()
4.        count_i INCREMENT BY 1
5.
6.        IF count_i EQUAL TO op_IO_time THEN
7.            I_block_queue REMOVE pcb // 从 IO 阻塞队列中移除 pcb
8.            pc = pcb.getPc()
9.            pc INCREMENT BY 1
10.
11.           IF pc GREATER THAN OR EQUAL TO pcb.getInstructionCount()
               THEN
12.               pcb.setState(-1)
13.               CALL freemem(pcb) // 释放内存
14.           ELSE
15.               pcb.setPc(pc) // 更新程序计数器
16.               CALL back_to_readyqueue(pcb) // 重回就绪队列
17.           ENDIF
18.
19.           count_i SET TO 0 // 重置计数器
20.       ENDIF
21.   ENDIF
22.
23.   IF O_block_queue NOT EMPTY THEN

```



```

24.      pcb = O_block_queue PEEK()
25.      count_o INCREMENT BY 1
26.
27.      IF count_o EQUAL TO op_IO_time THEN
28.          O_block_queue REMOVE pcb // 从 IO 阻塞队列中移除 pcb
29.          pc = pcb.getPc()
30.          pc INCREMENT BY 1
31.
32.          IF pc GREATER THAN OR EQUAL TO pcb.getInstructionCount()
            THEN
33.              pcb.setState(-1)
34.              CALL freemem(pcb) // 释放内存
35.          ELSE
36.              pcb.setPc(pc) // 更新程序计数器
37.              CALL back_to_readyqueue(pcb) // 重回就绪队列
38.          ENDIF
39.
40.          count_o SET TO 0 // 重置计数器
41.      ENDIF
42.  ENDIF
43. END FUNCTION
44.
45. FUNCTION back_to_readyqueue(pcb)
46.     SWITCH pcb.getTimesilve() DO
47.         CASE 1:
48.             ADD pcb TO readyQueue1
49.             BREAK
50.         CASE 2:
51.             ADD pcb TO readyQueue2
52.             BREAK
53.         CASE 4:
54.             ADD pcb TO readyQueue3
55.             BREAK
56.     END SWITCH
57. END FUNCTION

```

代码整体结构

代码分为两个主要部分：

1. **ALL_IO()** 方法：处理 I/O 阻塞队列中的进程。
2. **back_to_readyqueue(PCB pcb)** 方法：将完成 I/O 操作的进程重新放入相应的就绪队列。

代码运行逻辑

1. I/O 操作处理 (ALL_IO 方法)

- 输入阻塞队列处理:

- 检查 `I_block_queue`（输入阻塞队列）是否为空。如果队列不为空，获取队列头部的 PCB（进程控制块）。
- 计数器 `count_i` 加一，表示已经经历了一个时间片。
- 如果 `count_i` 等于预设的 `op_IO_time`，表示 I/O 操作已经完成：
 - 从队列中移除该 PCB。
 - 读取并更新程序计数器 `pc`，表示进程要执行的下一条指令。
 - 如果 `pc` 大于或等于指令数量，表示该进程执行完毕：
 - 更新进程状态为 -1（结束状态）。
 - 调用 `freemem(pcb)` 释放其占用的内存。
 - 如果 `pc` 小于指令数量，表示进程尚未结束：
 - 更新 PCB 的程序计数器 `pc`。
 - 调用 `back_to_readyqueue(pcb)` 将进程放回就绪队列。
 - 最后，重置计数器 `count_i`。

• 输出阻塞队列处理:

- 逻辑与输入阻塞队列相似，只是处理的是 `O_block_queue`（输出阻塞队列）。
- 从队列中移除 PCB，更新程序计数器，判断进程是否完成，释放内存或重回就绪队列。

2. 进程重回就绪队列 (`back_to_readyqueue` 方法)

- 根据 PCB 的时间片类型 (`timesilve`)，将进程添加到相应的就绪队列 (`readyQueue1`, `readyQueue2`, 或 `readyQueue3`)。
- 每个队列可能代表不同优先级的进程。

(2) 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：

```

1.  class InputBlockThread extends Thread:
2.      variable count_i = 0
3.      variable count_o = 0
4.      constant op_IO_time = 2
5.
6.      method run():
7.          while true:
8.              acquire lock from SyncManager.lock // 获取锁，确保线程安全
9.              try:
10.                 set SyncManager.io_clk to true // 标记 I/O 时钟为活动状态
11.                 signal SyncManager.io_clk_Condition // 通知等待 I/O 时钟的线程
12.                 wait for SyncManager.ioCondition // 等待 I/O 操作的条件
13.                 call ALL_IO() // 处理所有类型的 I/O 逻辑
14.             catch InterruptedException:
15.                 print error message // 处理异常
16.         finally:

```

17.

release lock from SyncManager.lock // 释放锁

`InputBlockThread` 类是一个线程类，负责处理与 I/O 操作相关的任务，包括输入和输出的阻塞与唤醒。它使用了 Java 的线程和锁机制，以确保在多线程环境下对共享资源的安全访问。

`InputBlockThread` 继承自 `Thread`，表示这个类将作为一个线程运行。

`count_i` 和 `count_o` 是用来跟踪输入和输出 I/O 操作的计数器。

`op_IO_time` 是一个常量，表示每次 I/O 操作所需的时间。

IO 线程同步逻辑

无限循环：`while (true)` 表示线程将一直运行，直到程序终止。

获取锁：`SyncManager.lock.lock()` 用于获取一个锁，确保在多线程环境下对共享资源的安全访问。

设置 I/O 时钟：`SyncManager.io_clk = true` 表示 I/O 先获得了锁，

通知其他线程：`SyncManager.io_clk_Condition.signal()` 唤醒可能因为时钟先获得了锁而引起得等待

等待条件：`SyncManager.ioCondition.await()` 将当前线程挂起，等待时钟获得锁后唤醒

处理 I/O 操作：`ALL_IO()` 方法调用，处理所有相关的 I/O 逻辑。

异常处理：如果在等待条件时被中断，捕获 `InterruptedException` 异常并打印堆栈信息。

释放锁：在 `finally` 块中确保释放锁，避免潜在的死锁。

需要特别指出的是：IO 线程和进程调度是并发的——CPU 和 IO 设备可以同时工作，和进程调度一样，IO 线程也必须先获得锁，进入等待，之后时钟才获得锁，唤醒 IO 线程。

(3) 设置断点单步跟踪测试阻塞线程唤醒过程，与进程调度线程并发过程；并通过 1-05-0.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：11-阻塞唤醒过程

评价分数：

理由：

3. 问题描述与解决方案（10 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】

评价分数：

理由:

4 参考文献

【不少于 5 篇，著录格式使用南京农业大学学报（自然科学版）格式】

- [1] Research on the Framework of Smart City Operating System Based on New ICTs[J]. Wu Jun.American Journal of Artificial Intelligence,2020(1)
- [2] Parallel vision for perception and understanding of complex scenes: methods, framework, and perspectives[J]. Kunfeng Wang;;Chao Gou;;Nanning Zheng;;James M. Rehg;;Fei-Yue Wang.Artificial Intelligence Review.2017
- [3] Perceived Benefits of Implementing and Using Hospital Information Systems and Electronic Medical Records.[J]. Khalifa Mohamed.Studies in health technology and informatics.2017
- [4] Parallel Control and Management for Intelligent Transportation Systems: Concepts, Architectures, and Applications.[J]. Fei-Yue Wang 0001.IEEE Trans. Intelligent Transportation Systems.2010
- [5] The Emergence of Intelligent Enterprises: From CPS to CPSS[J]. Wang, Fei-Yue.IEEE intelligent systems.2010
- [6] Analyzing open-source software systems as complex networks[J]. Xiaolong Zheng;;Daniel Zeng;;Huiqian Li;;Feiyue Wang.Physica A: Statistical Mechanics and its Applications.2008