|  |  |  |
| --- | --- | --- |
| 南农大  **计算机操作系统课程设计**  **测试分析与评价报告**  XH2 | | |
|  | **题 目:** | 仿真实现操作系统的作业管理及内存管理 |
|  | **姓 名:** | **董自经** |
|  | **班级专业:** | **计科221** |
|  | **学 号:** | **19222126** |
|  | **助 教：**  **指导教师:** | **杨鹏 类型: 研究生**  **姜海燕 职称: 教授** |
| **2025年3月21日**  **南京农业大学人工智能学院** | | |

**注意：**

**1.此文件适用于申请D成绩等级的申请人及成绩评价人。**

**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，否者不认可。本次课设申请人不再自我评分。**

**3.评阅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，评价申请人的每项成绩。**

**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. 此报告字体为5号宋体，黑色**

**6. 3.1-3.4测试项目在学校机房PC机上完成**

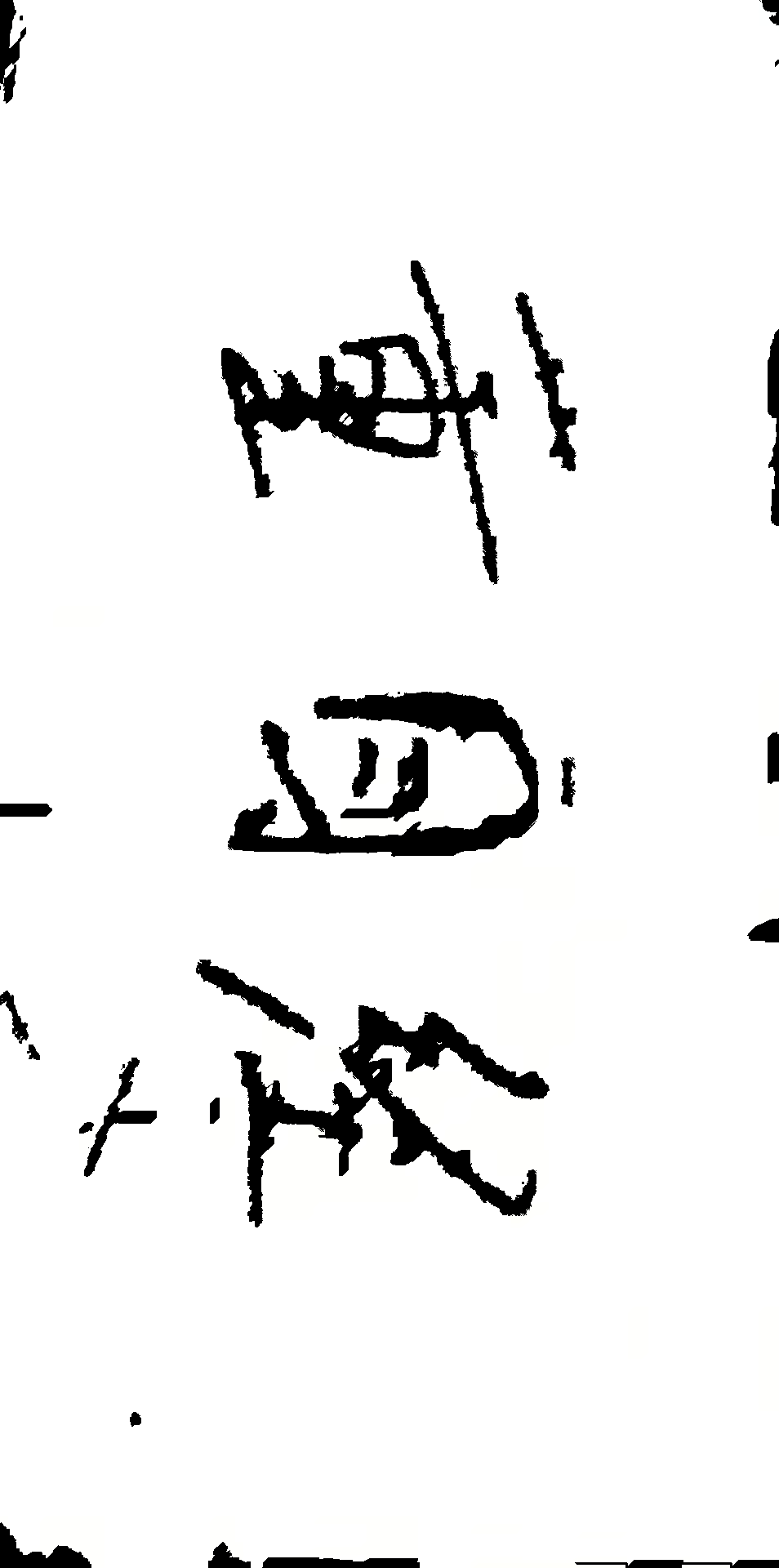
# 申请人诚信声明（申请人填写）

**所提交的课设代码由申请人自己独立完成并调试运行**

**使用申请成绩等级对应的测试数据完成测试**

**所提交的讲解视频由申请人自己独立完成**

**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**

**申请人签字（手写）：**

# 评阅人的评语及成绩(教师填写)

|  |  |  |  |
| --- | --- | --- | --- |
| 1. 题号 | 分值 | 评语 | 成绩 |
| 总成绩 | 100 |  |  |
| 排版 | 2 | 报告字体是否统一？图表及文字排版是否美观 |  |
| 文献 | 3 | 参考文献著录格式是否正确、完整 |  |
| 3.1 | 8 |  |  |
| 3.2 | 6 |  |  |
| 3.3 | 9 |  |  |
| 3.4 | 7 |  |  |
| 4.1 | 5 |  |  |
| 4.2 | 5 |  |  |
| 4.3 | 6 |  |  |
| 4.4 | 9 |  |  |
| 4.5 | 6 |  |  |
| 以下题目申请ABC等级完成 | | | |
| 4.6 | 11 |  |  |
| 以下题目申请AB等级完成 | | | |
| 4.7 | 12 |  |  |
| 以下题目申请A等级完成 | | |  |
| 4.8 | 14 |  |  |

# 3.黑盒测试与分析（共30分）

## 3.1 运行可执行程序并测试结果

**评价标准：共8分**

**程序可独立执行，按测试内容和成绩申请等级要求对结果逐行说明，视频讲解清晰，节过完全正确，得满分。**

**不可运行或者无反应，本题计0分。**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据**

**测试内容：**

1. （2分）在机房PC上安装可执行程序（按照开发者提供的安装说明书）。运行可执行程序。按“执行”按钮，全部完成所有作业的调度运行后，请截下当前界面显示信息等图片，判断程序是否能够脱离开发环境正确运行。
2. （2分）评价界面区域设计是否符合申请成绩等级要求？结合测试数据具体分析界面区域运行结果是否显示正确？

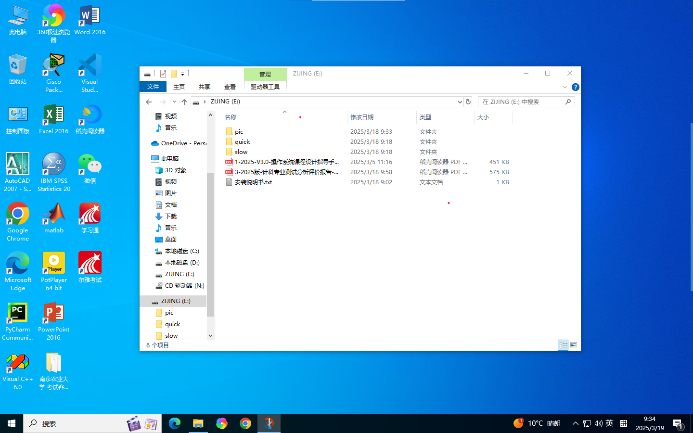
（3）（2分）关闭课设作品程序，请截下所保存文件等图片，并对ProcessResults-???-算法名称代号.txt文件逐条加以说明（根据申请成绩等级），其中???代表所有作业全部运行完成花费的总时间。结合测试数据，逐条说明所保存文件的位置、文件名以及保存内容是否正确？格式是否符合要求？

（4）（2分）对（1）-（3）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：2-可执行程序运行及结果分析。

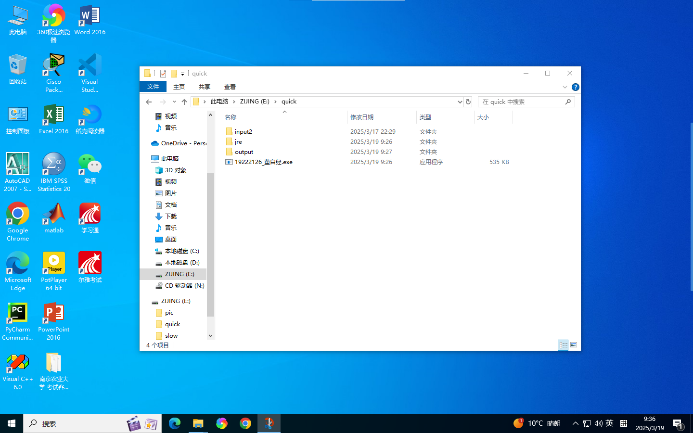
**测试结果与分析：**

安装步骤：

第一步：由于使用exe4j将程序打包为exe可执行程序，直接将文件夹打开即可，如图，打开文件夹后有两个主文件夹“qucik”、“slow”，“slow”文件夹是正常的操作系统仿真程序可执行文件，“quick”文件夹是为机房测试另外生成的一个exe文件，其与“slow”的区别在于“slow”的时钟模拟为1000ms，而“quick”的时钟模拟为100ms，将运行时间减少，以便于快速地在机房电脑上得出运行结果，减少等待时间。



图一：整个文件夹



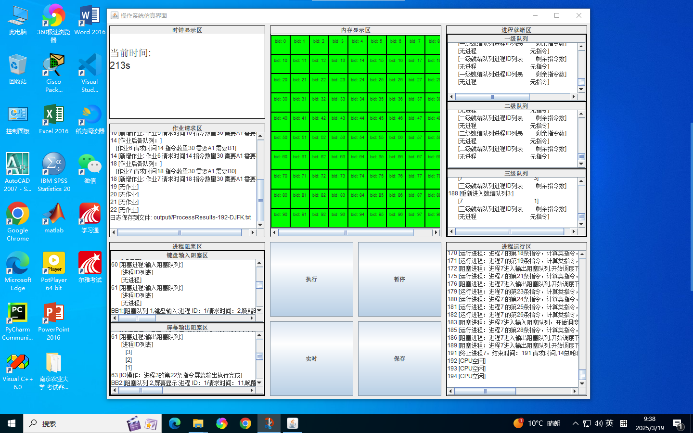
图二：“quick”文件夹

第二步：打开“quick”文件夹中的19222126\_董自经.exe文件，即可出现如下图所示的程序初始化界面。如图所示，程序的左上角为时钟显示区域，会在程序开始运行后显示仿真操作系统的时间；左中部为作业请求区，包含当前时间前系统中的所有作业的请求情况，同时，作业请求区中，每当有作业加入作业后备队列时就显示后备队列的所有内容；左下部进程阻塞区域，包含进入两个阻塞队列的进程信息；中部为内存区，内部为一个10\*10的区域，代表100个内存块，内存块内包含了当前内存块编号，占用内存块的作业id、进程id，如果被占用，会显示为红色；右上角为进程就绪区，分为三级队列，分别输出每一级队列中的内容，包括降级、重新进入、以及就绪队列中的内容；右下部为进程运行区域，包含了当前正在运行的进程以及结束运行的指令、作业信息。

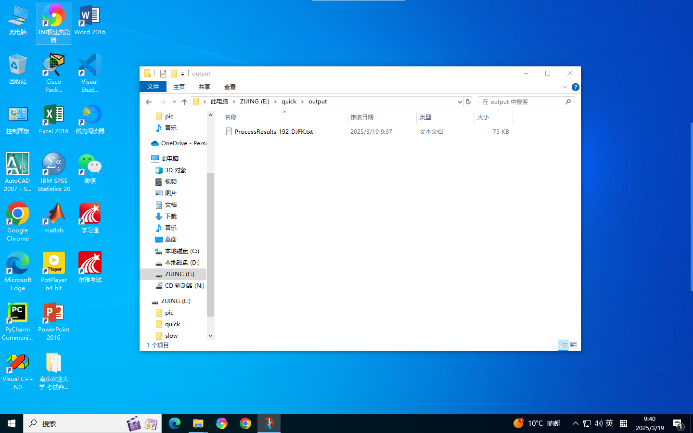


图三：程序初始化界面

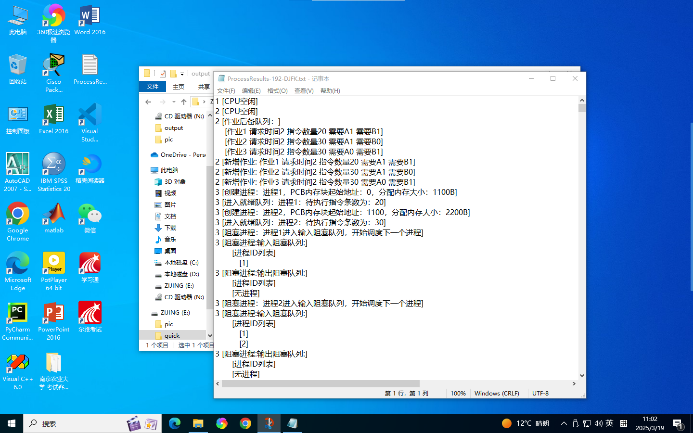
第三步：在按下执行，在程序运行结束后截图保存当前程序运行状态以及文件保存结果。文件信息保存在ProcessResults-???-DJFK.txt文件中，其中???表示保存文件时的系统时间。文件中保存了左右UI界面中的信息。



图四：程序结束时状态



图四：程序运行结束后的文件保存结果



图五：程序运行结束后保存的文件内容

## 3.2 “暂停”功能测试

**评价标准：共6分**

**程序可独立执行，按测试内容和成绩申请等级要求对结果逐行说明，视频讲解清晰，节过完全正确，得满分。**

**不可运行或者无反应，本题计0分。**

**暂停时，作品时钟没有停止，本题成绩得分不超过本题总成绩的一半**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

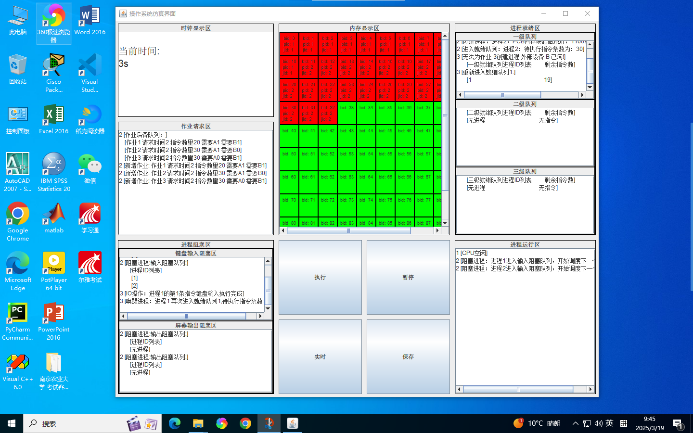
**其他情况，评阅人举证给出得分依据**

**测试内容：**

1. （2分）课设作品运行到第3秒时，按下“暂停”按钮，请截下当前界面图片，对每个区域的显示信息加以分类逐条说明，并判断是否满足申请成绩等级的功能要求？说明显示结果是否正确？
2. （2分）按下“执行”按钮后，继续再运行5秒，再按下“暂停”按钮。请截下当前界面图片，对每个区域的显示信息加以分类逐条说明，并判断是否满足申请成绩等级的功能要求？说明显示结果是否正确？
3. （2分）对（1）-（2）过程进行分步演示录屏讲解。视频文件保存到test-vidio子文件夹，文件名：2-暂停功能测试及结果分析。

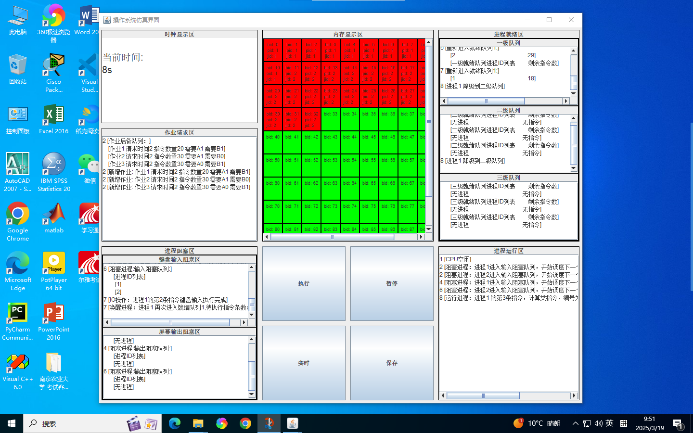
**测试结果与分析：**

在机房电脑上运行程序并在三秒时暂停可以看到有三个作业进入了作业的后备队列，分别为作业1、作业2、作业3、系统为作业1、作业2创建了进程，作业3由于外部设备B条件无法满足，未能成功创建进程，进入等待队列。



图六：三秒暂停

在5秒后按下暂停，即第八秒时，此时没有新的作业进入作业后备队列，所以作业请求区更新；在进程运行区可以看到进程1的第1、2条输入指令已经运行结束，第三条运算类指令也已经运行结束，所以进程就绪区可以看到进程1的一直运行，直到时间片使用完，降级到二级队列；对于作业二其第1、2条指令也已经运行结束，但其因为第三条指令仍然是输入阻塞所以没有调整到就绪队列2；输出阻塞区因暂时没有指令是输出类一直没有更新；输入阻塞区中随着进程1、进程2的前两条指令运行结束，新增了唤醒信息以及相应的IO操作信息。



图六：八秒暂停

## 3.3 “保存”功能测试

**评价标准：共9分。**

**程序可独立执行，按要求对结果逐行说明，并视频演示讲解清晰，得满分。**

**不可运行或者无反应，本题计0分。**

**如果没有保存多个文件，每项成绩得分不超过该项总成绩的一半。**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据**

**测试内容：**

1. （2分）当课设作品运行到第4秒时，按下“保存”按钮，请截下当前界面和所保存文件等图片，并对ProcessResults-???-算法名称代号.txt文件加以说明（根据申请成绩等级），其中???代表当前时间。逐条说明所保存文件的位置、文件名以及保存结果是否正确？格式是否符合要求？

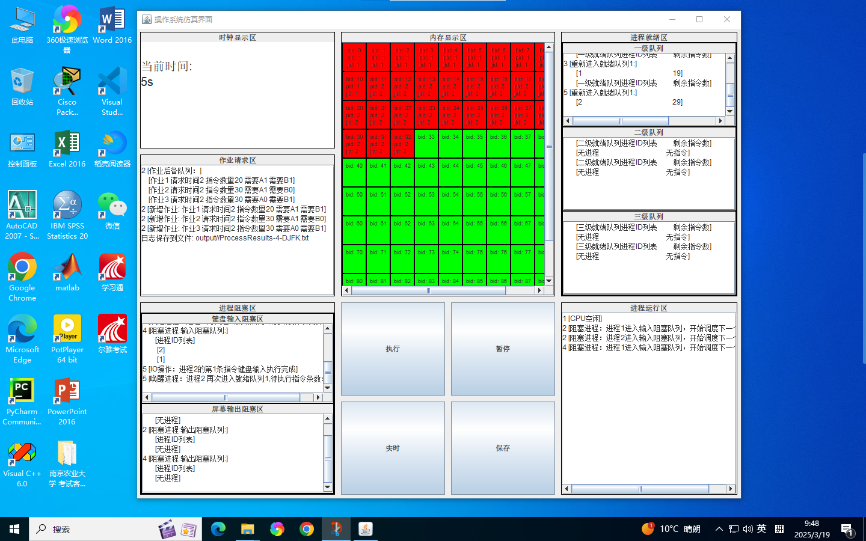
（2）（2分）课设作品运行到第10秒时，再次按下“保存”按钮，请截下当前界面和所保存文件等图片，并对ProcessResults-???-算法名称代号.txt文件加以说明（根据申请成绩等级），其中???代表当前时间。逐条说明所保存文件的位置、文件名以及保存结果是否正确？格式是否符合要求？

（3）（2分）课设作品全部完成所有作业的调度运行后，按下“保存”按钮，请截下当前界面和所保存文件等图片，并对ProcessResults-???-算法名称代号.txt文件加以说明（根据申请成绩等级），其中???代表所有作业全部运行完成花费的总时间。逐条说明所保存文件的位置、文件名以及保存结果是否正确？格式是否符合要求？

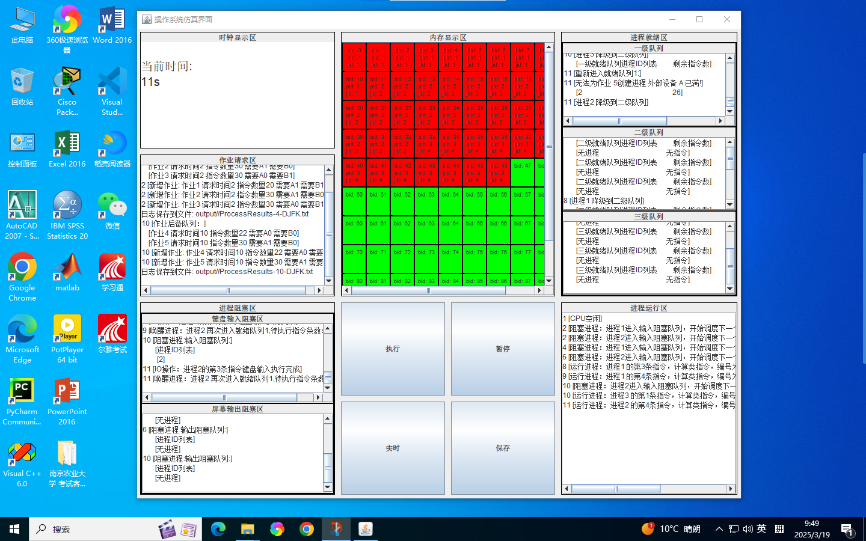
1. （3分）对（1）-（4）过程进行分步演示录屏讲解。视频文件保存到test-vidio子文件夹，文件名：3-保存功能测试及结果分析。

**测试结果与分析：**

在机房电脑上测试时分别在第四秒、第十秒以及程序运行结束后按下保存按钮可以看到在作业请求区出现语句“日志保存到文件：output/ProcessResults-???-DJFK.txt”（???代表按下班按钮是的时钟时间），表示所有面板中的信息都已经成功的保存到了该文件中。



图七：四秒保存

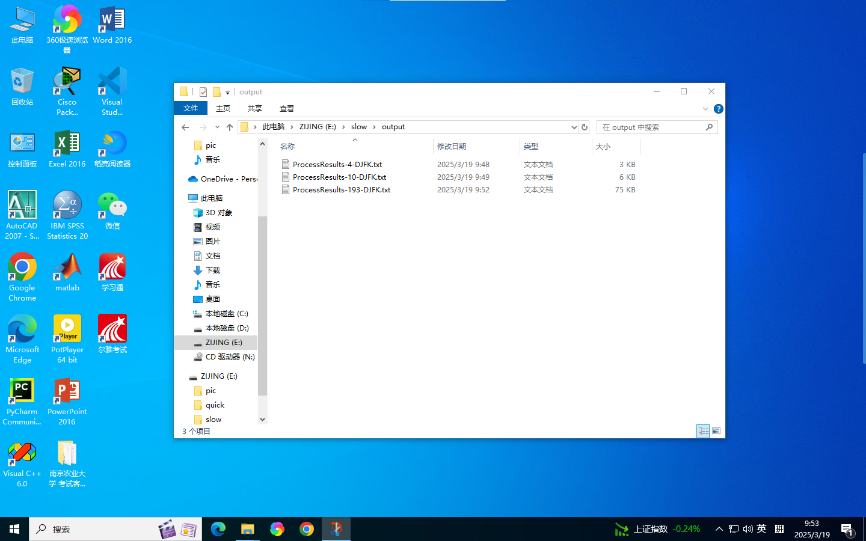


图八：十秒保存

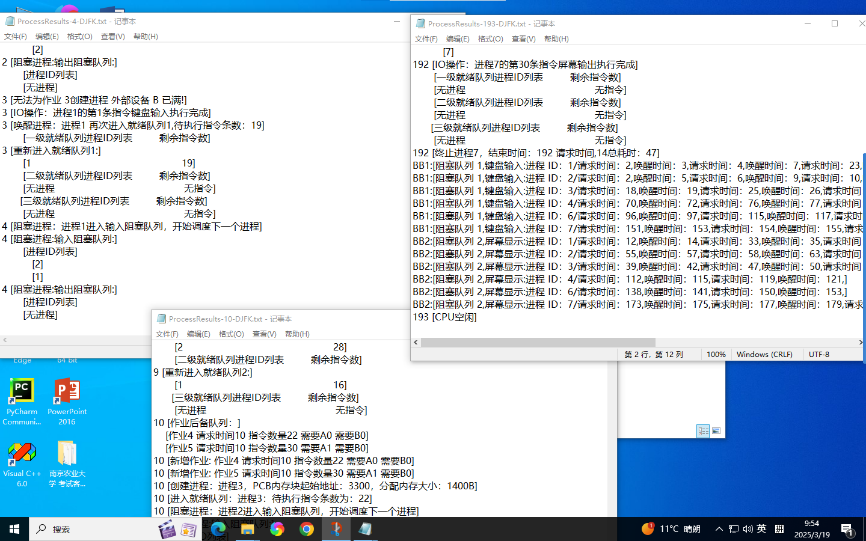


图九：运行结束保存

打开保存的三个文件，可以看到相关的信息，包括作业请求区的作业信息，进程就绪区的进程创建、进程降级、进程重新进入就绪队列的信息，作业运行区的运行信息如阻塞、运行、终止等，以及作业阻塞区的阻塞信息。



图十：保存的文件



图十一：文件内容

## 3.4 “实时”功能测试

**评价标准：共7分。**

**程序可独立执行，按要求对结果逐行说明，并视频演示讲解清晰，得满分。**

**不可运行或者无反应，本题计0分。**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据】**

**测试内容：**

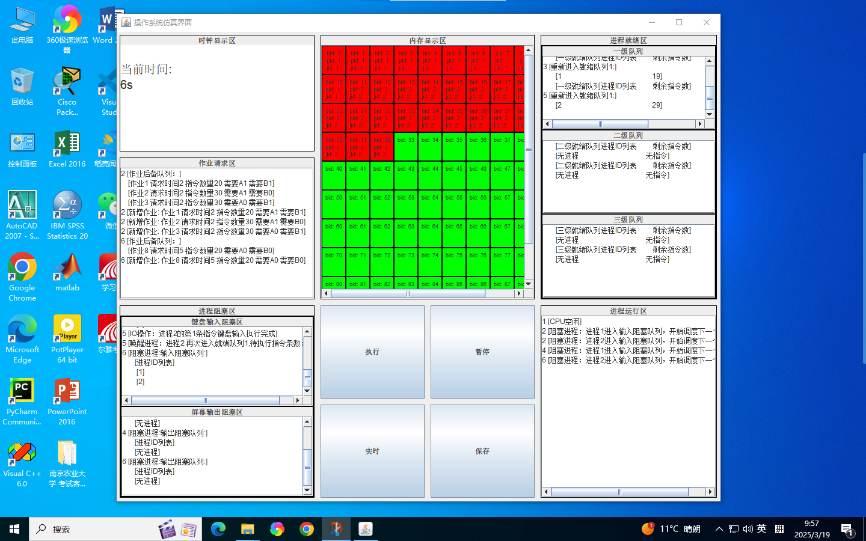
1. **（2分）当课设作品运行到第5秒时，按下“实时”按钮，请按照成绩申请等级要求，给出实时作业的指令集合？给出实时作业在后备队列的情况**

**（2）（2分）实时作业生成以后，继续运行到10秒时，按下“暂停”按钮，**请截下当前界面图片，对每个区域的显示信息加以分类逐条说明？说明实时作业请求在每个区域显示结果是否正确？

（3）（3分）对（1）-（4）过程进行分步演示录屏讲解。视频文件保存到test-vidio子文件夹，文件名：4-实时作业功能测试及结果分析。

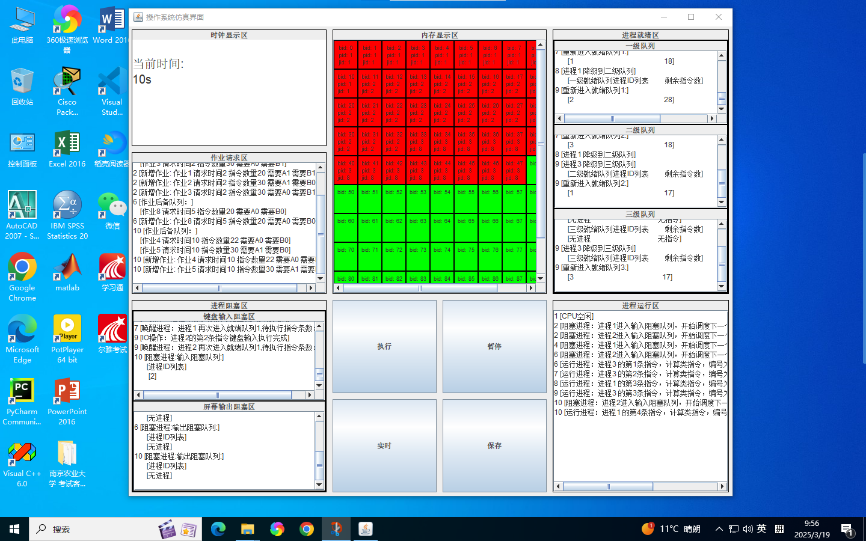
**测试结果与分析：**

在第五秒时点击实时按钮，在作业请求区可以看到作业8添加到了作业后备队列（ID为8的原因是系统预先读入了7个作业所以新的作业ID只能从8开始），添加到后备队列之后在第六秒被系统检测到，同时为作业8分配内存成为进程3（图十二内存区编号）。



图十二：五秒时点击实时按钮

作业8的指令集合中共20条指令，其中计算类指令15条，3条键盘输入类指令，2条屏幕输出类指令。在图十二中可以看到，十秒时作业1、作业2的前几条指令均为输入类指令一直阻塞，导致进程1、进程2不占用CPU，因此在第十秒时，进程3（作业8）已经进入了第三季就绪队列中，准备进行第四条指令的运行。



图十三：十秒时点击暂停按钮

# 4 代码测试与分析（40分）

## 4.1说明开发环境下工程程序及源程序文件的内容

**评价标准：共5分。**

**文字和图表完整详细，视频演示讲解清晰，得满分。**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据】**

**测试内容：**

（1）（2分）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）（1分）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）

（3）（2分）对（1）-（2）内容视频演示讲解，保存到test-vidio子文件夹，文件名：5-工程程序及源程序文件说明

**测试结果与分析：**

为实现操作系统仿真程序，共设计18个大类，如下图所示，图中箭头表示箭头目的类被箭头源头类使用（如遇交叉线，以直线不转向为准），其中由于OSKernel、PCB类被所有其他类使用，遂未画使用箭头，特在此说明。对于此18个类，可以简要按一下类别分类：线程控制类、CPU核心类、作业加载及显示类、内存类。

线程控制类：包含ClockInterruptHandlerThread、InputBlockThread、JobSchedulingHandlerThread、OutputBlockThread、ProcessSchedulingHandlerThread、SyncManager类。

在系统中，为了保证各个线程之间的同步关系，必须保证其他线程先拿到锁，之后释放锁并等待时钟线程拿到所有的锁对时间进行操作后唤醒所有的线程，未确保其他线程在ClockInterruptHandlerThread线程之前拿到锁，在SyncManager中设计了四把锁，这三把锁分别为进程调度锁、作业调度锁、输入操作锁、输出操作锁，每一把锁都有两个条件变量。对于每一个线程还额外设计了一个标志变量，标志是否在时钟之前拿到锁，8个条件变量配合四个标志变量可大概率降低死锁的出现并保证系统的正常运行。此外，还有一把暂停锁，仅含有一个条件变量，可实现系统暂停功能。

在ClockInterruptHandlerThread类中，包含了当前系统的时间simulationTime以及对时间的操作函数SimulateTimePassing()，同时可以通过设置milliseconds以控制系统的运行速度.

在InputBlockThread、OutputBlockThread类中模拟实现系统输入输出的操作，为了实现处理输入需要两秒，输出需要三秒，在线程内部分别设计了inTime和outTIme来记录进程进入阻塞的时间，由ALL\_IO函数判断时间是否达到。在线程内部还实现了重回就绪队列的函数。

在JobSchedulingHandlerThread类中，定义了JobRequest()实现了从后备作业队列中每隔两秒读取一次的逻辑。

在ProcessSchedulingHandlerThread类中，主要实现了三级反馈就绪队列的算法逻辑，实现线程的调度，包括进入就绪队列MFQ3()，进入等待队列CheckWaitQueue()，进程降级到低级就绪队列ChangeToLowGrade()，释放资源FinishPCB()等函数。

CPU核心类：包含OSKernel、CPU两个类。

OSKernel中包含了系统使用的所有变量如就绪队列、等待队列、阻塞队列、文件加载器等，还定义了一个检查空的函数CheckEmpty()，检测系统是否空闲。

CPU类中包含了pc程序计数器、ir指令寄存器，pswCPU状态，定义了runProcess()函数可根据当前正要执行的指令的状态将其所在的进程pcb加入到相应的队列中，还定义了CPU\_PRO()和CPU\_REC()用于实现CPU中断的现场保护和恢复。

外部类：包含Main、UI、Job、Instruction、JobAndInstructionLoader、PCB六个类。

JobAndInstructionLoader中包含了从外部加载作业、指令LoadAllJobsAndInstructions()，实时创建指令CreateInstructions()，保存日志到指定文件夹SaveResults()的函数等。

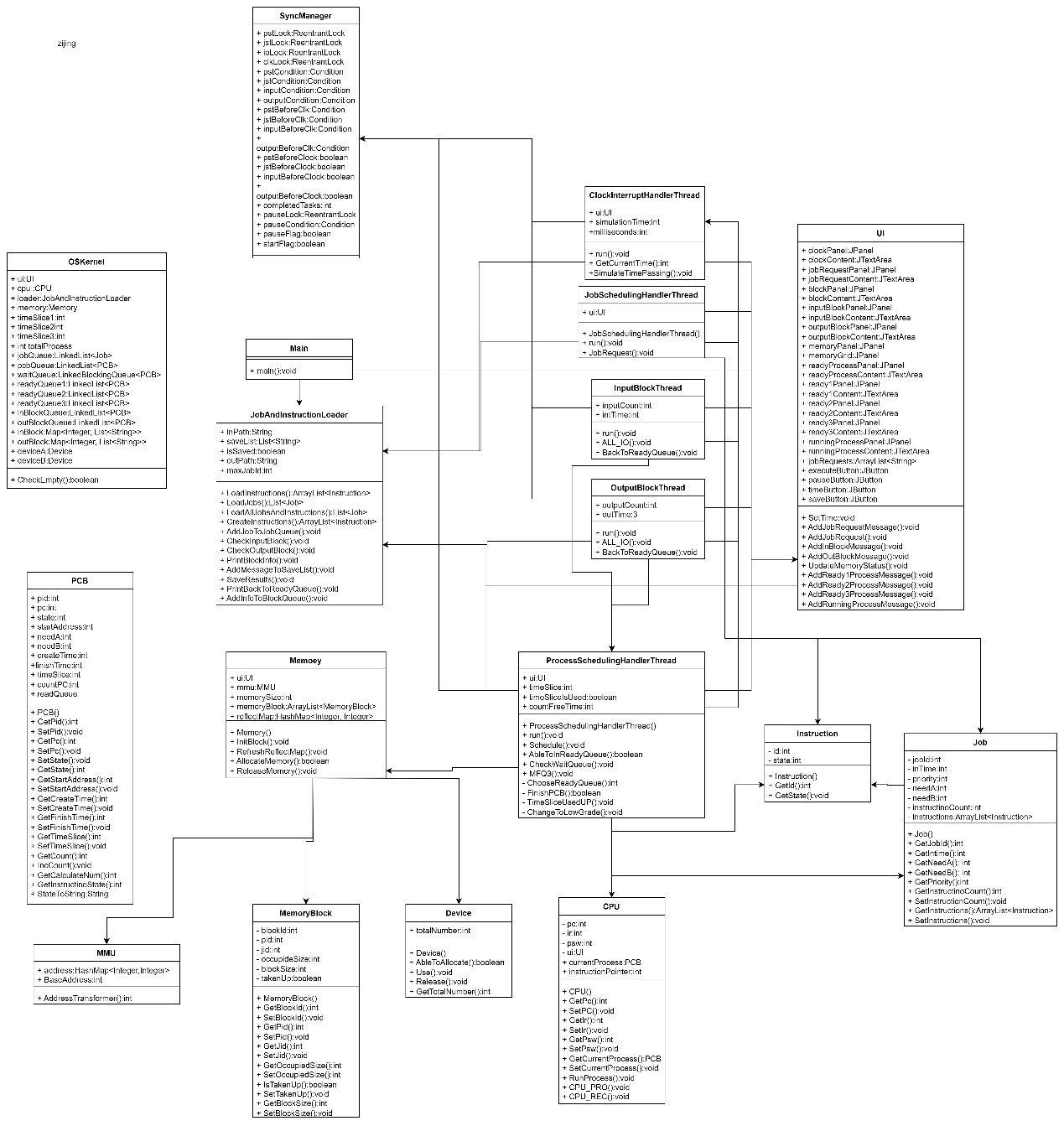
Job、Instruction、PCB则为JobAndInstructionLoader以及其他类提供基础的作业指令、进程信息，如作业ID，指令类型等。其中，PCB类继承自Job类，同时增加了有关于进程的独有信息如时间片、物理地址、进程状态等，还可通过PCB中的函数获取当前pcb中含有的指令及指令状态、指令所需要的空间等。

UI是此次系统仿真的可视化界面，将所有的信息输出到屏幕中。

内存类：包含Memory、MemoryBlock、MMU、Device四个类

MemoryBlock、MMU、Device为内存块的信息以及外部设备信息，供Memory调用。

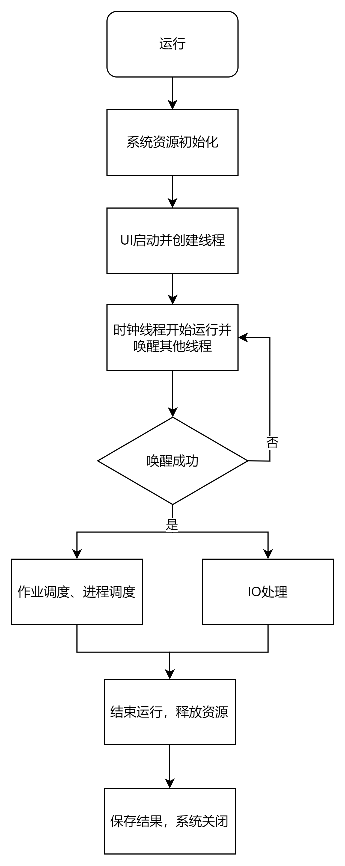
在Memory类中，为了实现最佳分区算法，定义并维护一个二维的HashMap，将空闲块起始地址与从这个地址开始的连续块个数相联系，在AllocateMemory()中只需对此进行查找即可找到是否有一个适合的块去分配。同时在Memory中还定义了ReleaseMemory()方法，用于在进程结束是释放内存以及外部设备。



图十四：项目类图

源程序文件装载运行的步骤步骤：

1. 系统运行自Main开始。点击系统运行后，初始化所有系统中的初始变量，包括OSKernel中的所有变量、SyncManager中的五把锁以及9个条件变量、进程控制块等信息，此时相当于BOIS初级阶段，CUP尚未真正运行，之后从默认路径“input2/”文件夹中读入文件存入到模拟硬盘的后备队列中。
2. 初始化完成，系统启动。此时CUP中的simulationTime初始化完成，同时SyncManager中的锁开始被使用，协调多线程工作。多级反馈队列启动，初始化时间片。
3. UI启动。展示当前系统的状态。
4. 5个线程启动。通过ClockInterruptHandlerThread统一唤醒那个其他四个线程以同步系统线程的工作时间。JobSchedulingHandlerThread从后备队列中读入信息并调入预备pcb队列中。ProcessSchedulingHandlerThread对预备pcb队列中的作业进行检查，如果可以为其创建进程，将之调入就绪队列中并开始执行指令。InputBlockThread、OutputBlockThread线程模拟键盘输入、屏幕输出功能。
5. 持续执行。通过ClockInterruptHandlerThread不断更新系统时间，各线程在系统关闭之前持续保持并行的状态，从各自所需的队列中读取进程（作业）信息，并判断是否满足转移条件，如若满足条件将其转移，不满足则继续在此队列中。
6. 系统的功能。系统主要包含暂停、实时、保存、执行功能。对于每个功能按钮都在UI中实现了监控，如果点击就调用相关线程的相关函数，实现目的功能。
7. 系统关闭。当关闭UI界面时，如果当前系统的模拟时间不为0，且存在为保存的运行日志，系统会自动保存所有的信息到output文件中。



图十五：系统运行步骤图

## 4.2 时钟中断线程跟踪测试及结果分析

**评价标准：共5分。**

**程序断点设置正确，并可以单步或者多步跟踪执行，显示中间过程、变量变化以后最后结果。结果正确，文字论述清晰，视频演示讲解清晰，得满分。**

**不能单步或者多步跟踪执行，显示中间过程、变量变化以后最后结果不正确。本题计0分**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据】**

**测试内容：**

（1）（2分）给出时钟中断线程激活作业调度线程、进程调度线程的代码，并说明实现过程；

（2）（3分）按下“执行”按钮，运行3秒开始设置断点跟踪运行，显示时钟变量值的变化情况；显示并讲解激活作业调度线程、进程调度线程的过程。录制视频文件讲解本题过程和结果。视频文件名：7-时钟中断线程跟踪运行测试

**测试结果与分析：**

时钟中断线程激活作业调度线程、进程调度线程的伪代码：

|  |
| --- |
| method run() |
| while (true) // 无限循环 |
|  |
| lock(pauseLock) // 获取暂停锁 |
| try |
| while (pauseFlag) |
| pauseCondition.await() // 等待解除暂停 |
| catch (Exception e) |
| print(e) // 处理异常 |
| finally |
| unlock(pauseLock) // 释放暂停锁 |
|  |
| // 获取所有相关锁，确保时钟调度的同步 |
| lock(jstLock) |
| lock(pstLock) |
| lock(inLock) |
| lock(outLock) |
| try |
| if (!jstBeforeClock) |
| jstBeforeClk.await() // 等待作业调度线程 |
| jstBeforeClock = false |
|  |
| if (!pstBeforeClock) |
| pstBeforeClk.await() // 等待进程调度线程 |
| pstBeforeClock = false |
|  |
| if (!inputBeforeClock) |
| inputBeforeClk.await() // 等待输入线程 |
| inputBeforeClock = false |
|  |
| if (!outputBeforeClock) |
| outputBeforeClk.await() // 等待输出线程 |
| outputBeforeClock = false |
|  |
| SimulateTimePassing(milliseconds) // 模拟时间流逝 |
| invokeLater(() -> ui.SetTime(simulationTime)) // 更新 UI 时钟显示 |
|  |
| // 唤醒所有相关线程 |
| jstCondition.signal() |
| pstCondition.signal() |
| inputCondition.signal() |
| outputCondition.signal() |
|  |
| catch (Exception e) |
| print(e) // 处理异常 |
| finally |
| unlock(jstLock) // 释放作业调度锁 |
| unlock(pstLock) // 释放进程调度锁 |
| unlock(inLock) // 释放输入锁 |
| unlock(outLock) // 释放输出锁 |
|  |
| method GetCurrentTime() |
| return simulationTime // 返回当前模拟时间 |
|  |
| method SimulateTimePassing(milliseconds) |
| try |
| sleep(milliseconds) // 线程休眠模拟时间流逝 |
| simulationTime++ // 增加模拟时间 |
| catch (Exception e) |
| print(e) // 处理异常 |

ClockInterruptHandlerThread的作用主要是控制仿真操作系统时间的变化，并根据时间同步所有其他线程的运行，特别是在多级反馈就绪队列和输入输出阻塞判断中，需要根据当前系统时间去决定当前的pcb时间片是否用完，是否符合转移的条件。

在时钟线程刚启动时，检查系统的pauseLock以及pauseCondition判断系统是否处于暂停状态，如果pauseFlag是ture则系统进入等待直至用户点击执行按钮释放pauseLock并继续执行。

之后ClockInterruptHandlerThread获取所有的锁并检查是否其他线程都已经在ClockInterruptHandlerThread之前拿到了锁并处于等待状态，如果是则对simulationTime进行加一操作模拟时间流逝，之后对所有等待的线程进行唤醒。如果不是，则时钟线程进入等待，知道其他所有线程获取到锁并在进入等待状态唤醒时钟线程。

## 4.3 作业请求线程跟踪测试及结果分析

**评价标准：共6分。**

**程序断点设置正确，可以单步或者多步跟踪执行，显示中间过程、变量变化以后最后结果。结果正确，文字论述清晰，视频演示讲解清晰，得满分。**

**不能单步或者多步跟踪执行，不显示中间过程、变量变化，最后结果不正确。本题计0分**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据】**

**测试内容：**

（1）（3分）给出作业调度线程的每项操作函数代码及相关数据结构伪码，逐行说明实现过程

（2）（3分）按下“执行”按钮，运行3秒后，设置断点单步运行作业调度程序，并跟踪2.txt和3.txt作业调度过程。录制视频文件讲解上述操作过程和结果分析。视频文件名：8-作业调度跟踪测试

**测试结果与分析：**

作业调度线程的每项操作函数代码及相关数据结构伪码

|  |
| --- |
| method run() |
| while (true) // 无限循环 |
|  |
| lock(jstLock) // 获取作业调度锁 |
| try |
| jstBeforeClock = true // 设置作业调度标志 |
| jstBeforeClk.signal() // 通知时钟线程 |
| jstCondition.await() // 等待时钟线程信号 |
|  |
| print("完成作业调度") |
|  |
| JobRequest() // 处理作业请求 |
| catch (Exception e) |
| print(e) // 处理异常 |
| finally |
| unlock(jstLock) // 释放作业调度锁 |
|  |
| method JobRequest() |
| if (!jobQueue.isEmpty()) // 如果作业队列不为空 |
| countFree = 0 |
| sortedJobQueue = sort(jobQueue, by GetInTime) // 按进入时间排序 |
| tmpjob = peek(sortedJobQueue) // 获取最早进入的作业 |
|  |
| if (tmpjob.GetInTime() <= simulationTime) // 如果作业已到达 |
| info = simulationTime + " [作业后备队列：]" |
| print(info) |
| invokeLater(() -> ui.AddJobRequestInfo(info)) |
| AddInfoToSaveList(info) |
|  |
| for tempJob in sortedJobQueue // 遍历作业队列 |
| if (tempJob.GetInTime() > simulationTime) |
| break |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddJobRequestInfo(info)) |
| AddInfoToSaveList(info) |
|  |
| else if (jobQueue.isEmpty() and countFree <= 3) // 如果作业队列为空 |
| countFree++ |
| info = simulationTime + " [无作业]" |
| print(info) |
| invokeLater(() -> ui.AddJobRequestInfo(info)) |
| AddInfoToSaveList(info) |
|  |
| if (GetCurrentTime() % 2 == 0) // 每2秒检查作业请求 |
| if (!jobQueue.isEmpty()) |
| sortedJobQueue = sort(jobQueue, by GetInTime) |
| while (!sortedJobQueue.isEmpty() and peek(sortedJobQueue).GetInTime() <= GetCurrentTime()) |
| tempJob = poll(sortedJobQueue) // 取出最早的作业 |
|  |
| if (jobQueue.contains(tempJob)) |
| jobQueue.remove(tempJob) // 从作业队列移除该作业 |
|  |
| pcb = createPCB(tempJob.GetJobId(), tempJob.GetInTime(), tempJob.GetNeedA(), tempJob.GetNeedB(), tempJob.GetInstructionCount(), tempJob.GetInstructions()) |
| pcbQueue.add(pcb) // 将作业加入PCB队列 |
|  |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddJobRequestInfo(info)) |
| AddInfoToSaveList(info) |

在run()方法中，JobSchedulingHandlerThread先获取相关的作业锁，并将jstBeforeClock设置为true，唤醒在条件变量jstBeforeClk中等待的线程clock，之后释放锁并开始等待时钟线程的调度，当时钟线程唤醒JobSchedulingHandlerThread时，打印相关信息并调用JobRequest()方法开始作业调度，最后释放锁。

在JobRequest()中，每秒扫描一次后备作业队列，先判断当前的作业队列是否为空，如果不空，对其中的内容按照进入时间进行升序排序，如果作业队列中的第一个作业的进入时间小于当前系统的时间，开始输出后备队列的信息，直到作业队列中的某一个作业的进入时间大于当前的系统时间，停止输出后备队列的信息。

同时，每两秒扫描一次后备队列，先判断当前的作业队列是否为空，如果不空，对其中的内容按照进入时间进行升序排序，同时将其中所有进入时间小于当前时间的作业加入到预备pcb队列中，并从作业队列中删除，输出相关信息。

## 4.4 进程调度线程跟踪测试与结果分析

**评价标准：共9分。**

**程序断点设置正确，可以单步或者多步跟踪执行，显示中间过程、变量变化以后最后结果。结果正确，文字论述清晰，视频演示讲解清晰，得满分。**

**不能单步或者多步跟踪执行，不显示中间过程、变量变化，最后结果不正确。本题计0分**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据】**

**测试内容：**

（1）（3分）给出作业调度线程的变量、每个操作函数代码及相关数据结构伪码，逐行说明实现过程

（2）（3分）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：9-进程调度线程激活单步测试

（4）（3分）设置断点跟踪运行2.txt和3.txt进程调度过程，一直跟踪到这两个作业运行完。具体讲解说明中间过程、变量变化情况，解释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：10-进程调度单步测试

**测试结果与分析：**

进程调度线程的变量、每个操作函数代码及相关数据结构伪码

|  |
| --- |
| class ProcessSchedulingHandlerThread extends Thread |
| property ui |
| property timeSlice = 2 |
| property timeSliceIsUsed = true |
| property countFreeTime = 0 |
|  |
| method constructor(ui) |
| this.ui = ui |
|  |
| method run() |
| while (true) |
| lock(pstLock) |
| try |
| pstBeforeClock = true |
| signal(pstBeforeClk) |
| await(pstCondition) |
| Schedule() |
| print("完成进程调度") |
| completedTasks++ |
| catch (Exception e) |
| print(e) |
| finally |
| unlock(pstLock) |
|  |
| method Schedule() |
| CheckWaitQueue() |
| if (!pcbQueue.isEmpty()) |
| for i = 0 to size(pcbQueue) |
| if (AbleToInReadyQueue(peek(pcbQueue))) |
| pcb = poll(pcbQueue) |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddReady1ProcessMessage(info)) |
| AddMessageToSaveList(info) |
| else |
| pcb = poll(pcbQueue) |
| waitQueue.add(pcb) |
|  |
| MFQ3() |
|  |
| if (simulationTime != 0 and readyQueue1.isEmpty() and readyQueue2.isEmpty() and readyQueue3.isEmpty() |
| and inBlockQueue.isEmpty() and outBlockQueue.isEmpty()) |
| info = "info" |
| countFreeTime++ |
| if (countFreeTime <= 3) |
| print(info) |
| invokeLater(() -> ui.AddRunningProcessMessage(info)) |
| AddMessageToSaveList(info) |
| if (countFreeTime == 1 and simulationTime != 1) |
| SaveResults(GetCurrentTime()) |
| else |
| countFreeTime = 0 |
|  |
| method AbleToInReadyQueue(pcb) |
| if (size(outBlockQueue) + size(readyQueue1) + size(readyQueue2) + size(readyQueue3) > 12) |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddReady1ProcessMessage(info)) |
| AddMessageToSaveList(info) |
| return false |
|  |
| if (deviceA.AbleToAllocate(pcb.GetNeedA())) |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddReady1ProcessMessage(info)) |
| AddMessageToSaveList(info) |
| return false |
|  |
| if (deviceB.AbleToAllocate(pcb.GetNeedB())) |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddReady1ProcessMessage(info)) |
| AddMessageToSaveList(info) |
| return false |
|  |
| if (memory.AllocateMemory(pcb)) |
| readyQueue1.add(pcb) |
| pcb.SetState(0) |
| pcb.SetTimeSlice(timeSlice1) |
| pcb.readQueue.add(GetCurrentTime()) |
| pcb.SetCreateTime(GetCurrentTime()) |
| return true |
|  |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddReady1ProcessMessage(info)) |
| AddMessageToSaveList(info) |
| return false |
|  |
| method CheckWaitQueue() |
| totalA = deviceA.GetTotalNumber() |
| totalB = deviceB.GetTotalNumber() |
| if (waitQueue.isEmpty()) |
| return |
| for pcb in waitQueue |
| if (totalA >= pcb.GetNeedA() and totalB >= pcb.GetNeedB()) |
| totalA -= pcb.GetNeedA() |
| totalB -= pcb.GetNeedB() |
| addFirst(pcbQueue, pcb) |
| remove(waitQueue, pcb) |
|  |
| method MFQ3() |
| while (!readyQueue1.isEmpty() or !readyQueue2.isEmpty() or !readyQueue3.isEmpty()) |
| if (ChooseReadyQueue() == -1) |
| print("所有队列都为空，无法调度进程") |
| return |
|  |
| switch (ChooseReadyQueue()) |
| case 1: |
| cpu.SetCurrentProcess(peek(readyQueue1)) |
| break |
| case 2: |
| cpu.SetCurrentProcess(peek(readyQueue2)) |
| break |
| case 4: |
| cpu.SetCurrentProcess(peek(readyQueue3)) |
| break |
| default: |
| cpu.SetPsw(-1) |
| break |
|  |
| if (cpu.GetCurrentProcess() == null) |
| print("当前没有进程在运行") |
| return |
|  |
| cpu.RunProcess() |
|  |
| if (cpu.GetPsw() == 0) |
| FinishPCB(cpu.GetCurrentProcess()) |
| CheckWaitQueue() |
| break |
|  |
| if (cpu.GetPsw() == 1) |
| timeSliceIsUsed = true |
|  |
| method FinishPCB(pcb) |
| if (pcb.GetPc() >= pcb.GetInstructionCount()) |
| pcb.SetFinishTime(GetCurrentTime()) |
| pcb.SetState(-1) |
| memory.ReleaseMemory(pcb) |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddRunningProcessMessage(info)) |
| AddMessageToSaveList(info) |
|  |
| switch (pcb.GetTimeSlice()) |
| case 1: |
| remove(readyQueue1, pcb) |
| break |
| case 2: |
| remove(readyQueue2, pcb) |
| break |
| case 4: |
| remove(readyQueue3, pcb) |
| break |
|  |
| PrintBlockInfo() |
| return true |
|  |
| TimeSliceUsedUP() |
| return false |
|  |
| method TimeSliceUsedUP() |
| if (cpu.GetCurrentProcess().GetCount() == 1) |
| timeSliceIsUsed = true |
| ChangeToLowGrade(cpu.GetCurrentProcess()) |
| else if (cpu.GetCurrentProcess().GetCount() == 3) |
| timeSliceIsUsed = true |
| ChangeToLowGrade(cpu.GetCurrentProcess()) |
| else if (cpu.GetCurrentProcess().GetCount() > 3) |
| count = cpu.GetCurrentProcess().GetCount() - 3 |
| timeSliceIsUsed = count % timeSlice3 == 0 |
| if (timeSliceIsUsed) |
| ChangeToLowGrade(cpu.GetCurrentProcess()) |
| else |
| timeSliceIsUsed = false |
|  |
| method ChangeToLowGrade(pcb) |
| if (contains(readyQueue1, pcb)) |
| remove(readyQueue1, pcb) |
| add(readyQueue2, pcb) |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddReady1ProcessMessage(info)) |
| invokeLater(() -> ui.AddReady2ProcessMessage(info)) |
| pcb.SetTimeSlice(timeSlice2) |
| AddMessageToSaveList(info) |
|  |
| else if (contains(readyQueue2, pcb)) |
| remove(readyQueue2, pcb) |
| add(readyQueue3, pcb) |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddReady2ProcessMessage(info)) |
| invokeLater(() -> ui.AddReady3ProcessMessage(info)) |
| pcb.SetTimeSlice(timeSlice3) |
| AddMessageToSaveList(info) |
|  |
| else |
| remove(readyQueue3, pcb) |
| add(readyQueue3, pcb) |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddReady3ProcessMessage(info)) |
| pcb.SetTimeSlice(timeSlice3) |
| AddMessageToSaveList(info) |

ui为系统显示界面，timeSlice为当前pcb的时间片，timeSliceIsUsed标志当前时间片是否使用完，countFreeTime表示截止至当前时间CPU连续空闲的时间。

ProcessSchedulingHandlerThread()方法从外部初始化ui界面，以便于向ui界面中加入信息。run()方法为线程运行方法，先获取进程调度锁，保证在时钟线程之前获取锁并进入等待同时释放调度锁，直到时钟线程唤醒本线程。

Schedule()方法为调度方法，在其中先判断作业后备队列是否为空，不为空则遍历作业后备队列，调用AbleToInReadyQueue()方法判断其是否符合调度条件，如果符合条件，则为其分配内存使其进入就绪队列同时在界面上输出相关信息，不能就将其加入作业等待队列，等待条件满足再加入后备作业队列进行调度。加入就绪队列后运行MFQ3()方法，三级反馈就绪队列开始工作，工作结束后判断当前cpu是否空闲，如果空闲则累计计数，当累计计数达到一定次数后停止计数。

AbleToInReadyQueue()检测能否为进程分配内存以及外部设备，依次判断系统中的并发进程数目、外部设备A、外部设备B以及内存情况，如果满足则在此函数内调用内存分配函数AbleToAllocate()为作业分配内存，否则返回false，

CheckWaitQueue()函数检查等待队列中的作业，如果外部设备满足条件，降级加入到作业后备队列中，不在此处检查内存是因为内存在此次设计中是在AbleToAllocate()中检查，后续才注意到有外部设备A、B。

MFQ3()三级反馈就绪队列主函数，根据ChooseReadyQueue()返回结果对进程进行调度判断应该加入哪一级就绪队列中，之后调用CPU.RunProcess()对就绪队列中的进程进行处理，同时修改CPU的状态字。

ChooseReadyQueue()函数，根据当前的时间片判断应该返回哪一级就绪队列的代号。

FinishPCB()函数先检测pcb的pc（已运行指令的数量）数量，将之pcb的指令数量进行对比，如果前者大于后者，则判断进程已经运行结束，释放进程所占用的相关内存资源和外部设备，同时将进程从相应的就绪队列中移除。

TimeSliceUsedUP()判断当前进程在当前就绪队列中的时间片是否使用完，如果使用完，则调用ChangeToLowGrade()函数，将当前pcb移入到下一级就绪队列中。

ChangeToLowGrade()函数，将当前pcb一如到下一级就绪队列中，如果当前就绪队列在三级就绪队列中则重新加入三级就绪队列的末尾。

## 4.5 实时操作跟踪测试与分析

**评价标准：共6分。**

**程序断点设置正确，可以单步或者多步跟踪执行，显示中间过程、变量变化以后最后结果。结果正确，文字论述清晰，视频演示讲解清晰，得满分。**

**不能单步或者多步跟踪执行，不显示中间过程、变量变化，最后结果不正确。本题计0分**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据】**

**测试内容：**

（1）（3分）给出作业后备队列插入实时作业的伪码；给出实时作业进入就绪队列的伪码，并逐行逐变量说明实现过程；

（2）（3分）在3.txt执行到第5条指令时按下实时按钮，设置断点跟踪生成与调度过程；录制视频文件讲解上述操作过程、核心变量变化过程以及屏幕显示结果。视频文件名：10-实时作业请求

**测试结果与分析：**

作业后备队列插入实时作业的伪码&实时作业进入就绪队列的伪码：

|  |
| --- |
| method CreateInstructions() |
| instructionCount = 20 |
| instructions = new List() |
| probabilities = [15, 18, 20] |
|  |
| for i from 1 to instructionCount |
| state = 0 |
| for j from 0 to length of probabilities |
| if i < probabilities[j] |
| state = j |
| break |
| instructions.add(new Instruction(i, state)) |
|  |
| return instructions |
|  |
| // 将作业加入作业后备队列 |
| method AddJobToJobQueue() |
| jobId = maxJobId + 1 |
| job = new Job(jobId, ClockInterruptHandlerThread.simulationTime, 20, CreateInstructions()) |
| OSKernel.jobQueue.add(job) |
|  |
| // 处理作业后备队列 |
| if OSKernel.jobQueue is not empty |
| countFree = 0 |
| sortedJobQueue = new PriorityQueue(sorted by Job.GetInTime) |
| sortedJobQueue.addAll(OSKernel.jobQueue) |
|  |
| tmpjob = sortedJobQueue.peek() |
| if tmpjob.GetInTime() <= ClockInterruptHandlerThread.simulationTime |
| print(ClockInterruptHandlerThread.simulationTime + " [作业后备队列：]") |
| invokeLater(ui.AddJobRequestMessage, ClockInterruptHandlerThread.simulationTime + " [作业后备队列：]") |
| OSKernel.loader.AddMessageToSaveList(ClockInterruptHandlerThread.simulationTime + " [作业后备队列：]") |
|  |
| for tempJob in sortedJobQueue |
| if tempJob.GetInTime() > ClockInterruptHandlerThread.simulationTime |
| break |
| message = " [作业" + tempJob.GetJobId() + " 请求时间" + tempJob.GetInTime() + |
| " 指令数量" + tempJob.GetInstructionCount() + |
| " 需要A" + tempJob.GetNeedA() + |
| " 需要B" + tempJob.GetNeedB() + "]" |
| print(message) |
| invokeLater(ui.AddJobRequestMessage, message) |
| OSKernel.loader.AddMessageToSaveList(message) |
|  |
| else if OSKernel.jobQueue is empty and countFree <= 3 |
| countFree = countFree + 1 |
| print(ClockInterruptHandlerThread.simulationTime + " [无作业]") |
| invokeLater(ui.AddJobRequestMessage, ClockInterruptHandlerThread.simulationTime + " [无作业]") |
| OSKernel.loader.AddMessageToSaveList(ClockInterruptHandlerThread.simulationTime + " [无作业]") |
|  |
| method Schedule() |
| CheckWaitQueue() |
| if (!pcbQueue.isEmpty()) |
| for i = 0 to size(pcbQueue) |
| if (AbleToInReadyQueue(peek(pcbQueue))) |
| pcb = poll(pcbQueue) |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddReady1ProcessMessage(info)) |
| AddMessageToSaveList(info) |
| else |
| pcb = poll(pcbQueue) |
| waitQueue.add(pcb) |
|  |
| MFQ3() |
|  |
| if (simulationTime != 0 and readyQueue1.isEmpty() and readyQueue2.isEmpty() and readyQueue3.isEmpty() |
| and inBlockQueue.isEmpty() and outBlockQueue.isEmpty()) |
| info = "info" |
| countFreeTime++ |
| if (countFreeTime <= 3) |
| print(info) |
| invokeLater(() -> ui.AddRunningProcessMessage(info)) |
| AddMessageToSaveList(info) |
| if (countFreeTime == 1 and simulationTime != 1) |
| SaveResults(GetCurrentTime()) |
| else |
| countFreeTime = 0 |

点击实时按钮后调用AddJobToJobQueue()方法以及CreateInstructions()方法，设置当前作业的ID为系统计数maxJobId+1，作业进入时间为系统时间，CreateInstructions()生成指令集，20条指令中包含15条计算类指令，3条键盘输入类指令，2条屏幕输出类指令，将之加入到后备作业队列中，后由作业调度函数每两秒扫描一次后备队列，先判断当前的作业队列是否为空，如果不空，对其中的内容按照进入时间进行升序排序，同时将其中所有进入时间小于当前时间的作业加入到预备pcb队列中，并从作业队列中删除，输出相关信息。进程调度函数Schedule()方法调用AbleToInReadyQueue()方法判断其是否符合调度条件，如果符合条件，则为其分配内存使其进入就绪队列同时在界面上输出相关信息。

## 4.6 连续动态内存分配与回收跟踪测试（申请ABC）

**评价标准：共11分。**

**程序断点设置正确，可以单步或者多步跟踪执行，显示中间过程、变量变化以后最后结果。结果正确，文字论述清晰，视频演示讲解清晰，得满分。**

**不能单步或者多步跟踪执行，不显示中间过程、变量变化，最后结果不正确。本题计0分**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据】**

**测试内容：**

（1）（3分）给出连续动态内存分配算法及可视化过程核心代码，并逐行逐变量说明实现过程；

（2）（3分）给出进程创建的核心代码，并逐行逐变量说明实现过程；

（3）（5分）按下“执行”按钮，运行3秒后，设置断点跟踪运行，讲解1.txt、2.txt、3.txt作业执行、内存分配的完整过程；录制视频文件讲解上述操作过程和结果。视频文件名：11-连续动态内存分配与回收

**测试结果与分析：**

连续动态内存分配算法及可视化过程核心代码：

|  |
| --- |
| method AllocateMemory(pcb) |
| calculateNum = pcb.GetCalculateNum() |
| neededBlock = calculateNum / MemoryBlock.blockSize |
| bestFitStartAddress = -1 |
| bestFitBlockSize = Infinity |
|  |
| if calculateNum % MemoryBlock.blockSize != 0 |
| neededBlock++ |
|  |
| // 在所有空闲块中查找最适合的块 |
| for each (startAddress, freeBlocks) in reflectMap |
| if freeBlocks >= neededBlock and freeBlocks < bestFitBlockSize |
| bestFitStartAddress = startAddress |
| bestFitBlockSize = freeBlocks |
|  |
| // 如果找到合适的内存块 |
| if bestFitStartAddress != -1 |
| OSKernel.totalProcess++ |
| pcb.SetPid(OSKernel.totalProcess) |
| blockId = bestFitStartAddress / MemoryBlock.blockSize |
| if bestFitStartAddress % MemoryBlock.blockSize > 0 |
| blockId++ |
|  |
| remainingSize = pcb.GetCalculateNum() |
|  |
| for i from blockId to blockId + neededBlock |
| memoryBlock[i].SetTakenUp(true) |
| memoryBlock[i].SetPid(pcb.GetPid()) |
| memoryBlock[i].SetJid(pcb.GetJobId()) |
|  |
| if remainingSize >= MemoryBlock.blockSize |
| memoryBlock[i].SetOccupiedSize(MemoryBlock.blockSize) |
| remainingSize -= MemoryBlock.blockSize |
| else |
| memoryBlock[i].SetOccupiedSize(remainingSize) |
| remainingSize = 0 |
|  |
| if remainingSize == 0 |
| pcb.SetStartAddress(blockId \* MemoryBlock.blockSize) |
| mmu.address.put(pcb.GetPid(), pcb.GetStartAddress()) |
|  |
| OSKernel.deviceA.Use(pcb.GetNeedA()) |
| OSKernel.deviceB.Use(pcb.GetNeedB()) |
| RefreshReflectMap() |
|  |
| invokeLater(ui.UpdateMemoryStatus, memoryBlock) |
| return true |
|  |
| return false |
|  |
| // 释放内存，同时释放设备 |
| method ReleaseMemory(pcb) |
| if pcb.GetState() == -1 |
| startPhysicalAddress = pcb.GetStartAddress() |
| blockNumber = pcb.GetCalculateNum() / MemoryBlock.blockSize |
|  |
| if pcb.GetCalculateNum() % MemoryBlock.blockSize != 0 |
| blockNumber++ |
|  |
| // 遍历进程所占有的所有块进行释放 |
| for i from 0 to blockNumber |
| blockId = (startPhysicalAddress / MemoryBlock.blockSize) + i |
| block = memoryBlock[blockId] |
| block.SetTakenUp(false) |
| block.SetOccupiedSize(0) |
| block.SetPid(0) |
| block.SetJid(0) |
|  |
| OSKernel.deviceA.Release(pcb.GetNeedA()) |
| OSKernel.deviceB.Release(pcb.GetNeedB()) |
| RefreshReflectMap() |
|  |
| invokeLater(ui.UpdateMemoryStatus, memoryBlock) |
|  |
| // 更新内存状态方法 |
| method UpdateMemoryStatus(memoryBlocks) |
| for i from 0 to 100 |
| block = memoryBlocks[i] |
| blockPanelInMemory = memoryGrid.getComponent(i) |
|  |
| blockPanelInMemory.removeAll() |
|  |
| if block.IsTakenUp() |
| label = new Label("<html>bid: " + i + "<br/>pid: " + block.GetPid() + "<br>jid: " + block.GetJid() + "</html>") |
| blockPanelInMemory.setBackground(RED) |
| else |
| label = new Label("<html>bid: " + i + "<br/></html>") |
| blockPanelInMemory.setBackground(GREEN) |
|  |
| label.setFont(Font(label.getFont().getName(), PLAIN, 9)) |
| blockPanelInMemory.add(label) |
|  |
| memoryPanel.revalidate() |
| memoryPanel.repaint() |

在AllocateMemory()方法中，先获取当前pcb所需要的内存空间数量，之后转换为其所需要的内存块的数目，遍历二维HashMap reflectMap查找是否有一个空白的连续内存块可以满足当前进程的需求，如果找到满足需求的内存块，将其分配给当前pcb，同时对设备A、B进行占用操作，之后将空白内存块减少相应块数。释放内存时，遍历pcb所占有的内存块，将其状态设置为未被占用的状态。

在对内存区进行可视化显示时，遍历所有的内存块，查询其被占用的信息，如果被占用，渲染为红色，同时加上内存块id以及占用改内存块的作业以及进程id，如果未被占用，则将其渲染为绿色，仅显示内存块id。

进程创建的核心代码：

|  |
| --- |
| method Schedule() |
| CheckWaitQueue() |
| if (!pcbQueue.isEmpty()) |
| for i = 0 to size(pcbQueue) |
| if (AbleToInReadyQueue(peek(pcbQueue))) |
| pcb = poll(pcbQueue) |
| info = "info" |
| print(info) |
| invokeLater(() -> ui.AddReady1ProcessMessage(info)) |
| AddMessageToSaveList(info) |
| else |
| pcb = poll(pcbQueue) |
| waitQueue.add(pcb) |
|  |
| MFQ3() |
|  |
| if (simulationTime != 0 and readyQueue1.isEmpty() and readyQueue2.isEmpty() and readyQueue3.isEmpty() |
| and inBlockQueue.isEmpty() and outBlockQueue.isEmpty()) |
| info = "info" |
| countFreeTime++ |
| if (countFreeTime <= 3) |
| print(info) |
| invokeLater(() -> ui.AddRunningProcessMessage(info)) |
| AddMessageToSaveList(info) |
| if (countFreeTime == 1 and simulationTime != 1) |
| SaveResults(GetCurrentTime()) |
| else |
| countFreeTime = 0 |

创建进程时，进程调度函数Schedule()方法先调用AbleToInReadyQueue()方法判断其是否符合调度条件，如果符合条件，则为其分配内存使其进入就绪队列同时在界面上输出相关信息。

## 4.7 进程阻塞唤醒线程跟踪测试与结果分析（10）

**评价标准：共12分。**

**程序断点设置正确，可以单步或者多步跟踪执行，显示中间过程、变量变化以后最后结果。结果正确，文字论述清晰，视频演示讲解清晰，得满分。**

**不能单步或者多步跟踪执行，不显示中间过程、变量变化，最后结果不正确。本题计0分**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据**

**测试内容：**

（1）（4分）分别给出进程阻塞、进程再调度过程伪码，并逐行逐变量说明；

（2）（3分）分别给出输入、显示等线程操作过程伪码，并逐行逐变量说明；

（3）（5分）按下“执行”按钮，运行2秒后，设置断点跟踪运行1.txt、2.txt，讲解阻塞线程唤醒过程，与进程调度线程并发过程，讲解1.txt、2.txt完整的运行过程。录制视频文件讲解上述操作过程和结果。视频文件名：12-阻塞唤醒过程

**测试结果与分析：**

进程阻塞、进程再调度过程伪码

|  |
| --- |
| // 运行进程 |
| method RunProcess() |
| if GetCurrentProcess() == null |
| print("当前没有进程在运行") |
| return |
|  |
| currentPC = GetCurrentProcess().GetPc() |
| irState = GetCurrentProcess().GetInstructionState() |
|  |
| // 对当前 PC 进行赋值 |
| pc = currentPC |
|  |
| // 计算类指令，要求不能中断，直接运行 |
| if irState == 0 |
| SetPsw(0) |
| GetCurrentProcess().INCCount() |
| info = ClockInterruptHandlerThread.GetCurrentTime() + " [运行进程：进程" + GetCurrentProcess().GetPid() + " 的第" + (currentPC + 1) + "条指令，计算类指令，编号为0，物理地址为：" + OSKernel.memory.mmu.AddressTransformer(GetCurrentProcess()) + "数据大小100B]" |
| print(info) |
| invokeLater(ui.AddRunningProcessMessage, info) |
| OSKernel.loader.AddMessageToSaveList(info) |
| currentPC++ |
| GetCurrentProcess().SetPc(currentPC) |
| psw = 0 |
|  |
| // 键盘输入类指令，进入输入阻塞队列 |
| else if irState == 1 |
| SetPsw(1) |
| pcb = GetCurrentProcess() |
|  |
| switch pcb.GetTimeSlice() |
| case 1 |
| OSKernel.readyQueue1.remove(pcb) |
| case 2 |
| OSKernel.readyQueue2.remove(pcb) |
| case 4 |
| OSKernel.readyQueue3.remove(pcb) |
|  |
| OSKernel.inBlockQueue.add(pcb) |
| OSKernel.loader.CheckInputBlock(pcb.GetPid(), "请求时间：" + ClockInterruptHandlerThread.GetCurrentTime()) |
|  |
| info = ClockInterruptHandlerThread.GetCurrentTime() + " [阻塞进程：进程" + pcb.GetPid() + " 进入输入阻塞队列，开始调度下一个进程]" |
| print(info) |
| invokeLater(ui.AddRunningProcessMessage, info) |
| OSKernel.loader.AddMessageToSaveList(info) |
| OSKernel.loader.AddInfoToBlockQueue(ClockInterruptHandlerThread.GetCurrentTime()) |
| psw = 1 |
|  |
| // 屏幕输出类指令，进入输出阻塞队列 |
| else |
| SetPsw(1) |
| pcb = GetCurrentProcess() |
|  |
| switch pcb.GetTimeSlice() |
| case 1 |
| OSKernel.readyQueue1.remove(pcb) |
| case 2 |
| OSKernel.readyQueue2.remove(pcb) |
| case 4 |
| OSKernel.readyQueue3.remove(pcb) |
|  |
| OSKernel.outBlockQueue.add(pcb) |
| OSKernel.loader.CheckOutputBlock(pcb.GetPid(), "请求时间：" + ClockInterruptHandlerThread.GetCurrentTime()) |
|  |
| info = ClockInterruptHandlerThread.GetCurrentTime() + " [阻塞进程：进程" + pcb.GetPid() + " 进入输出阻塞队列，开始调度下一个进程]" |
| print(info) |
| invokeLater(ui.AddRunningProcessMessage, info) |
| OSKernel.loader.AddMessageToSaveList(info) |
| OSKernel.loader.AddInfoToBlockQueue(ClockInterruptHandlerThread.GetCurrentTime()) |
| psw = 1 |
|  |
| // 进程回到就绪队列 |
| method BackToReadyQueue(pcb) |
| switch pcb.GetTimeSlice() |
| case 1: |
| OSKernel.readyQueue1.add(pcb) |
| pcb.readQueue.add(ClockInterruptHandlerThread.GetCurrentTime()) |
| info = ClockInterruptHandlerThread.GetCurrentTime() + " [唤醒进程：进程" + pcb.GetPid() + " 再次进入就绪队列1,待执行指令条数：" + (pcb.GetInstructionCount() - pcb.GetPc()) + "]" |
| case 2: |
| OSKernel.readyQueue2.add(pcb) |
| pcb.readQueue.add(ClockInterruptHandlerThread.GetCurrentTime()) |
| info = ClockInterruptHandlerThread.GetCurrentTime() + " [唤醒进程：进程" + pcb.GetPid() + " 再次进入就绪队列2,待执行指令条数：" + (pcb.GetInstructionCount() - pcb.GetPc()) + "]" |
| case 4: |
| OSKernel.readyQueue3.add(pcb) |
| pcb.readQueue.add(ClockInterruptHandlerThread.GetCurrentTime()) |
| info = ClockInterruptHandlerThread.GetCurrentTime() + " [唤醒进程：进程" + pcb.GetPid() + " 再次进入就绪队列3,待执行指令条数：" + (pcb.GetInstructionCount() - pcb.GetPc()) + "]" |
|  |
| print(info) |
| invokeLater(OSKernel.ui.AddOutBlockMessage, info) |
| OSKernel.loader.AddMessageToSaveList(info) |
| OSKernel.loader.PrintBackToReadyQueue(ClockInterruptHandlerThread.GetCurrentTime()) |

在CPU的RunProcess()方法中，先判断当前的进程是否为空，如果是空，则直接返回，不是空则获取当前进程的已运行指令数、指令状态，同时根据指令状态判断时计算类指令还是输入输出类指令。如果是计算类指令，将当前进程的pc加一，表示该指令已经执行完毕，同时修改psw。如果指令为输入输出类指令，则将psw设置为1，根据pcb的时间片判断其在哪一级就绪队列中，将其从所在的就绪队列中移除同时加入输入输出阻塞队列中等待输入输出的完成。

在返回就绪队列时，根据pcb的时间片决定应该返回哪一级就绪队列的末尾同时将信息输出。

输入、显示等线程操作伪代码

// 处理所有 I/O 操作（输入阻塞）

|  |
| --- |
| method ALL\_IO() |
| // 检查输入阻塞队列是否为空 |
| if !OSKernel.inBlockQueue.isEmpty() |
| pcb = OSKernel.inBlockQueue.peek() |
| outputCount++ |
|  |
| // 如果满足唤醒时间条件 |
| if outputCount == inTime |
| OSKernel.inBlockQueue.remove(pcb) |
| OSKernel.loader.CheckInputBlock(pcb.GetPid(), "唤醒时间：" + ClockInterruptHandlerThread.GetCurrentTime()) |
|  |
| pc = pcb.GetPc() |
| info = ClockInterruptHandlerThread.GetCurrentTime() + " [IO操作：进程" + pcb.GetPid() + "的第" + (pc + 1) + "条指令" + pcb.StateToString() + "执行完成]" |
| print(info) |
| invokeLater(OSKernel.ui.AddInBlockMessage, info) |
| OSKernel.loader.AddMessageToSaveList(info) |
|  |
| pc++ |
|  |
| // 进程执行结束，释放资源 |
| if pc >= pcb.GetInstructionCount() |
| pcb.SetFinishTime(ClockInterruptHandlerThread.GetCurrentTime()) |
| pcb.SetState(-1) |
| OSKernel.memory.ReleaseMemory(pcb) |
|  |
| info2 = ClockInterruptHandlerThread.GetCurrentTime() + " [终止进程" + pcb.GetPid() + "，结束时间：" + ClockInterruptHandlerThread.GetCurrentTime() + " 请求时间," + pcb.GetInTime() + "，总耗时：" + (pcb.GetFinishTime() - pcb.GetCreateTime()) + "]" |
|  |
| if OSKernel.readyQueue3.contains(pcb) |
| OSKernel.readyQueue3.remove(pcb) |
|  |
| OSKernel.loader.PrintBackToReadyQueue(ClockInterruptHandlerThread.GetCurrentTime()) |
| print(info2) |
| invokeLater(OSKernel.ui.AddRunningProcessMessage, info2) |
| OSKernel.loader.AddMessageToSaveList(info2) |
| OSKernel.loader.PrintBlockInfo() |
|  |
| // 进程未结束，返回就绪队列 |
| else if pc < pcb.GetInstructionCount() |
| pcb.SetPc(pc) |
| BackToReadyQueue(pcb) |
|  |
| outputCount = 0 |
|  |
| // 处理所有 I/O 操作（输出阻塞） |
| method ALL\_IO() |
| if !OSKernel.outBlockQueue.isEmpty() |
| pcb = OSKernel.outBlockQueue.peek() |
| outCount++ |
|  |
| // 如果满足唤醒时间条件 |
| if outCount == outTime |
| OSKernel.outBlockQueue.remove(pcb) |
| OSKernel.loader.CheckOutputBlock(pcb.GetPid(), "唤醒时间：" + ClockInterruptHandlerThread.GetCurrentTime()) |
|  |
| pc = pcb.GetPc() |
| info = ClockInterruptHandlerThread.GetCurrentTime() + " [IO操作：进程" + pcb.GetPid() + "的第" + (pc + 1) + "条指令" + pcb.StateToString() + "执行完成]" |
| print(info) |
| invokeLater(OSKernel.ui.AddOutBlockMessage, info) |
| OSKernel.loader.AddMessageToSaveList(info) |
|  |
| pc++ |
|  |
| // 进程执行结束，释放资源 |
| if pc >= pcb.GetInstructionCount() |
| pcb.SetFinishTime(ClockInterruptHandlerThread.GetCurrentTime()) |
| pcb.SetState(-1) |
| OSKernel.memory.ReleaseMemory(pcb) |
|  |
| info2 = ClockInterruptHandlerThread.GetCurrentTime() + " [终止进程" + pcb.GetPid() + "，结束时间：" + ClockInterruptHandlerThread.GetCurrentTime() + " 请求时间," + pcb.GetInTime() + "，总耗时：" + (pcb.GetFinishTime() - pcb.GetCreateTime()) + "]" |
|  |
| if OSKernel.readyQueue3.contains(pcb) |
| OSKernel.readyQueue3.remove(pcb) |
|  |
| OSKernel.loader.PrintBackToReadyQueue(ClockInterruptHandlerThread.GetCurrentTime()) |
| print(info2) |
| invokeLater(OSKernel.ui.AddRunningProcessMessage, info2) |
| OSKernel.loader.AddMessageToSaveList(info2) |
| OSKernel.loader.PrintBlockInfo() |
|  |
| // 进程未结束，返回就绪队列 |
| else if pc < pcb.GetInstructionCount() |
| pcb.SetPc(pc) |
| BackToReadyQueue(pcb) |
| outCount = 0 |

输入输出阻塞线程中，先判断阻塞队列头部的进程在阻塞队列中的时间是否已经满足出阻塞队列的时间，如果满足，将其从ui因的阻塞队列中移除，同时判断该进程是否已经结束，如果结束，将进程的状态设置为结束，释放相应的内存以及外部设备，输出相关的信息。如果进程未结束运行，则对其已运行的指令数加一，再调用BackToReadyQueue()方法将其放回就绪队列中。

## 4.8 内存缓冲区的设计访问与测试

**评价标准：共14分。**

**程序断点设置正确，可以单步或者多步跟踪执行，显示中间过程、变量变化以后最后结果。结果正确，文字论述清晰，视频演示讲解清晰，得满分。**

**不能单步或者多步跟踪执行，不显示中间过程、变量变化，最后结果不正确。本题计0分**

**录制视频讲解与文字论述不一致，本题计0分。**

**视频讲解不清晰、不完整，视频演示讲解成绩得分不超过该项总成绩的一半。**

**其他情况，评阅人举证给出得分依据**

**测试内容：**

（1）（3分）给出输入缓冲区、输出缓冲区仿真设计、可视化显示操作的函数过程、相关数据结构等核心伪码，并逐行逐变量说明；

（2）（4分）给出输入线程利用输入缓冲区实现进程变量赋值、PV操作缓冲区位置变量、唤醒输入变量阻塞队列等核心伪码，并逐行逐变量说明；

（3）（3分）给出输出线程利用输入缓冲区实现进程变量赋值、PV操作缓冲区位置变量、唤醒输入变量阻塞队列等核心伪码，并逐行逐变量说明；

（4）（5分）按下“执行”按钮，运行2秒后，设置断点跟踪运行1.txt、2.txt，讲解阻塞线程唤醒过程，与进程调度线程并发过程，讲解1.txt、2.txt完整的运行过程。讲解视频文件名：13-内存缓冲区的设计

**测试结果与分析：**

5参考文献

【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]董博南,杨秋松,李明树.M-RSF：面向Unikernel的一种多级反馈队列任务调度机制[J].通信学报,2024,45(05):54-69.

[2]刘政文,赵曙光,刘彷平.基于“调度+反馈”策略的多队列数据流发送控制系统设计[J].东华大学学报(自然科学版),2023,49(03):130-135.DOI:10.19886/j.cnki.dhdz.2022.0091.

[3]鲍敬源,韩锡方,钱超.基于多级反馈队列调度算法的舰船补给流程优化[J].中国水运(下半月),2021,21(10):40-42.

[4]赵丽敏,郑文艳,王文博.轮转调度算法中动态时间片的CPN实现[J].软件,2020,41(08):129-131.

[5]米热古丽·克比尔,顾英杰.嵌入式操作系统负载均衡时间片轮转算法仿真[J].计算机仿真,2019,36(11):247-250.

[1]刘亚杰.基于多线程技术的C语言程序并行化改造策略研究[J].电子元器件与信息技术,2024,8(08):39-41.DOI:10.19772/j.cnki.2096-4455.2024.8.012.