|  |  |  |
| --- | --- | --- |
| 南农大  **计算机操作系统课程设计**  **实践报告**  XH2 | | |
|  | 题 目: | 可视化仿真实现Linux2.6进程管理与内存管理 |
|  | 姓 名: | 陈扬 |
|  | 学 院: | 信息科技学院 |
|  | 专 业: | 计算机科学技术系 |
|  | 班 级: | 网工161 |
|  | 学 号: | 19316117 |
|  | 指导教师: | 姜海燕 职称: 教授 |
| 2019年3月 15 日 | | |

**目录**

[**一、程序结构说明** 1](#_Toc3558399)

[**（一）基本硬件** 1](#_Toc3558400)

[**（二）管理模块** 1](#_Toc3558401)

[**（三）系统内核** 1](#_Toc3558402)

[**（四）UI界面** 2](#_Toc3558403)

[**二、功能设计与实现** 2](#_Toc3558404)

[**（一）高级调度** 2](#_Toc3558405)

[**（二）中级调度** 3](#_Toc3558406)

[**（三）低级调度** 3](#_Toc3558407)

[**（四）JCB、PCB设计实现** 4](#_Toc3558408)

[**（五）死锁检测与撤销** 5](#_Toc3558409)

[**（六）页表生成** 6](#_Toc3558410)

[**（七）四态转换** 7](#_Toc3558411)

[**（八）进程同步互斥** 7](#_Toc3558412)

[**（九）进程原语** 8](#_Toc3558413)

[**三、功能测试** 10](#_Toc3558414)

[**（一）系统常量** 10](#_Toc3558415)

[**（二）PCB测试** 10](#_Toc3558416)

[**（三）JCB测试** 11](#_Toc3558417)

[**（四）死锁检测** 11](#_Toc3558418)

[**（五）UI界面** 12](#_Toc3558419)

[**（六）三级调度** 16](#_Toc3558420)

[**（七）指令运行** 16](#_Toc3558421)

[**四、技术问题分析报告** 20](#_Toc3558422)

[**（一）精确时间计时** 20](#_Toc3558423)

[**（二）队列指定元素删除** 21](#_Toc3558424)

[**（三）cpu内部时间计数** 22](#_Toc3558425)

[**（四）三级调度的优先级控制** 23](#_Toc3558426)

[**五、实验心得** 23](#_Toc3558427)

[**六、参考文献** 24](#_Toc3558428)

**可视化仿真实现Linux2.6进程管理与内存管理**

**一、程序结构说明**

程序的设计遵循了操作系统的基本原理，从底层开始构建，逐级往上添加功能与模块。任何一个模块都有其对应的父模块与子模块。程序的结构层层调用且保持了层级结构，这样的设计使得程序日后的开发趋于模块化。

程序的结构（除了Main.java）文件之外，可分为四个模块：基本硬件、管理模块、系统内核、UI界面。在具体的代码结构上，可以分为三个package包，分别为：hardware、os、ui。

每一个模块的程序结构将在下文加以详细说明。

**（一）基本硬件**

基本硬件为package包下的内容。有：AddressLine.java、CPU.java、DataLine.java、HardDisk.java、Memory.java、MMU.java、Timer.java文件。

每一个文件中所定义的类以及在程序中的基本功能如下：

AddressLine.java：定义了AddressLine类，模拟抽象了地址线的功能，为系统底层硬件的地址传输提供支持。

CPU.java：定义了CPU类，模拟抽象了CPU的功能，为系统整体的程序运行提供支持，内部还有计时器、MMU硬件。

DataLine.java：定义了DataLine类，模拟抽象了数据线的功能，为系统底层硬件的数据传输提供支持。

HardDisk.java：定义了HardDisk类，模拟抽象了硬盘的功能，为系统提供外部存储的功能。在程序开始运行后，该模块作为系统底层将被频繁调用。

Memory.java：定义了Memory类，模拟抽象了内存的功能，为系统提供内存的功能。在程序开始运行后，该模块作为系统底层将被频繁调用。

MMU.java：定义了MMU类，模拟抽象了CPU中内存管理部件的功能，为系统从虚拟地址到物理地址的转换的功能。同时，在MMU还设计有TLB表，加快了地址的转换。

Timer.java：定义了Timer类，模拟抽象了CPU中计时器的功能。该模块可以按照一定的时间间隔产生时钟中断，以协调进行系统三级调度。

**（二）管理模块**

管理模块为os包下的部分内容。有：JobModule.java、ProcessModule.java、PageModule.java文件。

每一个文件中所定义的类以及在程序中的基本功能如下：

JobModule.java：定义了JobModule类，提供了对于系统作业的所有有关的管理功能。

ProcessModule.java：定义了ProcessModule类，提供了对于系统进程的所有有关的管理功能。

PageModule.java：定义了PageModule类，提供了对于内存页框与磁盘块的所有有关的管理功能。同时，伙伴算法、内存分配回收、虚存空间管理的功能都在该类中可以找到。

**（三）系统内核**

系统内核为os包下的部分内容。有：Control.java、DeadLock.java、JCB.java、kernel.java、Page.java、PCB.java、Scheduling.java文件。

每一个文件中所定义的类以及在程序中的基本功能如下：

Control.java：定义了Control类，该类是对于系统中所有功能模块的整合，并对外提供API接口方便UI调用系统功能。

DeadLock.java：定义了DeadLock类，该类是对于死锁功能的具体实现。在该类中，提供了PV操作指令和资源申请释放指令的具体操作，在任何情况下进行的死锁检测算法调度。

JCB.java：定义了JCB类，该类是对于作业控制块的具体描述。

kernel.java：定义了kernel类，该类为整个程序提供需要使用的常量定义。

Page.java：定义了Page类，该类是对于页面、页框、块的具体描述。

PCB.java：定义了PCB类，该类是对于进程控制块的具体描述。

Scheduling.java：定义了Scheduling类，该类中提供了低级调度、中级调度、高级调度三种不同调度的功能实现。

**（四）UI界面**

UI界面为ui包下的所有内容。有：CPUInfoUI.java、CreateJobUI.java、HardDisk.java、MainUI.java、Memory.java、PageModule.java文件。

每一个文件中所定义的类以及在程序中的基本功能如下：

CPUInfoUI.java：定义了CPUInfoUI类，该类是CPU信息展示的界面。

CreateJobUI.java：定义了CreateJobUI类，该类是提供创建作业功能的界面。

HardDisk.java：定义了HardDiskUI类，该类是提供查看硬盘内容功能的界面。

MainUI.java：定义了MainUI类，该类是整个程序开始执行后的最开始的界面，展示了各个队列的进程运行情况。同时，提供了调出其他窗口的功能。

MemoryUI.java：定义了MemoryUI类，该类是提供查看内存内容功能的界面。

PageModuleUI.java：定义了PageModuleUI类，该类是提供查看页面占用与分配情况的界面。

**二、功能设计与实现**

在本次操作系统课程设计实验中，我设计完成了多项功能，如：三级调度、JCB与PCB的设计实现、死锁检测与撤销、页表生成、页面调度算法、四态转换、进程同步互斥、进程原语功能。

这些功能是系统最核心最重要的功能。同时，某些功能还需要依赖于页面调度算法的实现。

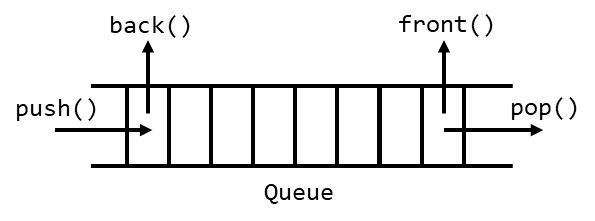
下文将对这些功能的实现原理进行深度阐述。

**（一）高级调度**

高级调度又称为作业调度或宏观调度。它用于确定把后备队列上的哪些作业调入内存,并为之建立进程,分配其所需的资源,然后将它挂在就绪队列上。

高级调度又称为作业调度或宏观调度。其主要功能是根据一定的算法，从输入的一批任务（作业）中选出若干个作业(从磁盘的作业后备队列中选择作业调入内存)，分配必要的资源并建立与作业相对应的进程，如内存、外设等，为它建立相应的用户作业进程和为其服务的系统进程（如输入/输出进程），最后把它们的程序和数据调入内存，等待进程调度程序对其执行调度，并在作业完成后作善后处理工作。

高级调度的基本原理即为队列的表现形式，如图：



高级调度将控制多道程序的道数，被选择进入内存的作业越多，每个作业所获得的CPU时间就越少，为了向用户提供满意的服务，有时需要限制内存中同时运行的进程数。每当有作业执行完毕并撤离时，作业调度会选择一个或多个作业补充进入内存。此外，如果CPU空闲时间超过一定的阈值，系统也会引出作业调度选择后备作业。

高级调度的实现在文件Scheduling.java中，具体实现的逻辑为：

1、刷新后备队列

2、判断后备队列中是否还有作业

3、若有，则按照FIFO算法，获取PCB队列中需要被调入的作业

4、检测PCB池空间是否足够

5、若足够，则检测虚存空间是否足够

6、若足够，则开始导入进程：

① 将JCB转换为PCB

② 在虚存中申请对应大小的空间

③ 将作业的所有指令区转移到虚存中

④ 写PCB中的快表

⑤ 将PCB写入到PCB池

⑥ “下一个作业”指针后移

⑦ 将该进程加入到就绪队列，等待被调度

**（二）中级调度**

中级调度又称为中程调度，引入中级调度的主要目的是为了提高内存的利用率和系统的吞吐量。内存中不能有太多的进程，把进程从内存移到外存，当内存有足够空间时，再将合适的进程换入内存，等待进程调度。中级调度实际上就是存储器管理中的对调功能。

中级调度的实现在文件Scheduling.java中，具体实现的逻辑为：

1、将挂起队列中的进程全部取出

2、获取当前内存中可用的页框数

3、如果可用页框数小于10，则进行中级调度：

①根据LRU算法，取出最近最不经常使用的10个页面

②将这10个页面直接移出到虚存

③改写使用这些页面的进程的页表

④将这些进程变成挂起状态

**（三）低级调度**

低级调度又称为进程调度、短程调度，它决定就绪队列中的哪个进程将获得处理机，然后由分派程序执行把处理机分配给该进程的操作。在批处理，分时，实时三类系统中，进程调度必须被配置，因而是一种最基本的调度。与中级调度——交换，高级调度——作业调度相对应。

低级调度的实现在文件Scheduling.java中，具体实现的逻辑为：

1、检测当前是否正处于阻塞态，如果处于，则等待

2、检测是否已经退出阻塞态，如果是，则将阻塞态中的所有进程移入就绪态

3、刷新active和expired指针

4、查看当前运行队列是否有进程

5、如果没有，则从就绪队列中调入

6、检测当前正准备运行的指令所在的页是否在内存中

7、如果不在，发生缺页中断。如果在，将该页的访问次数+1

8、开始执行指令（涉及到MMU操作，需要取指令）

9、针对每条不同类型指令进行调度

10、刷新CPU与PCB状态

11、检测是否运行完毕，如果运行完毕，调用撤销原语

10、检测时间片是否用完，如果完毕，则进入就绪队列

**（四）JCB、PCB设计实现**

根据操作系统的设计，对于作业与进程的模拟是必须的。在操作系统中，作业分为：JCB、数据段、程序段；进程分为：PCB、核心栈、数据段、程序段。PCB为进程的静态刻画，JCB为作业的描述。通过JCB与PCB的构建，在程序运行时便可以自动的识别并处理作业、进程。

JCB与PCB的设计在JCB.java与PCB.java文件中实现。其具体的数据结构如下：

**JCB：**

private short job\_id; //作业ID

private short priority; //作业/进程的优先级

private int job\_intime; //作业进入时间

private short instruction\_num; //作业包含的指令数目

private short pages\_num; //作业所占用的页面数目

private ArrayList<Short> all\_instructions=new ArrayList<Short>(); //所有指令的链表

**PCB：**

private short pid; //进程标识符

private short state; //进程状态。就绪态、等待态、运行态、挂起态

private short priority; //进程优先级

private int job\_intime; //作业创建时间

private int process\_intime; //进程创建时间

private int end\_time; //作业/进程结束时间

private short timeslice; //时间片长度

private int runtime; //每次运行时，进程已经运行时间

private int counter; //该进程处于运行状态下的时间片余额

private byte PSW; //程序状态字。管态、目态

private short current\_instruction\_no; //当前运行到的指令编号

private short instruction\_num; //该进程总共包含的指令数目

private short pages\_num; //该作业/进程所占用的页面数目

public short [][]page\_table=new short[(kernel.MEMORY\_USER\_SPACE\_SIZE)/

kernel.SINGLE\_PAGE\_SIZE][2];

//页表

private ArrayList<Integer> instructions=new ArrayList<Integer>();

//该进程所有的指令

**（五）死锁检测与撤销**

多个进行相互等待对方资源，在得到所有资源继续运行之前，都不会释放自己已有的资源，这样造成了循环等待的现象，称为死锁。

死锁的发生有以下几个条件：

①资源互斥/资源不共享

每个资源要么已经分配给了一个进程，要么是可用的，只有这两种状态，资源不可以被共享使用，所以所谓的互斥是指：资源不共享，如果被使用，只能被一个进程使用。

②占有和等待/请求并保持

已经得到资源的进程还能继续请求新的资源，所以个人觉得叫占有并请求也许更好理解。

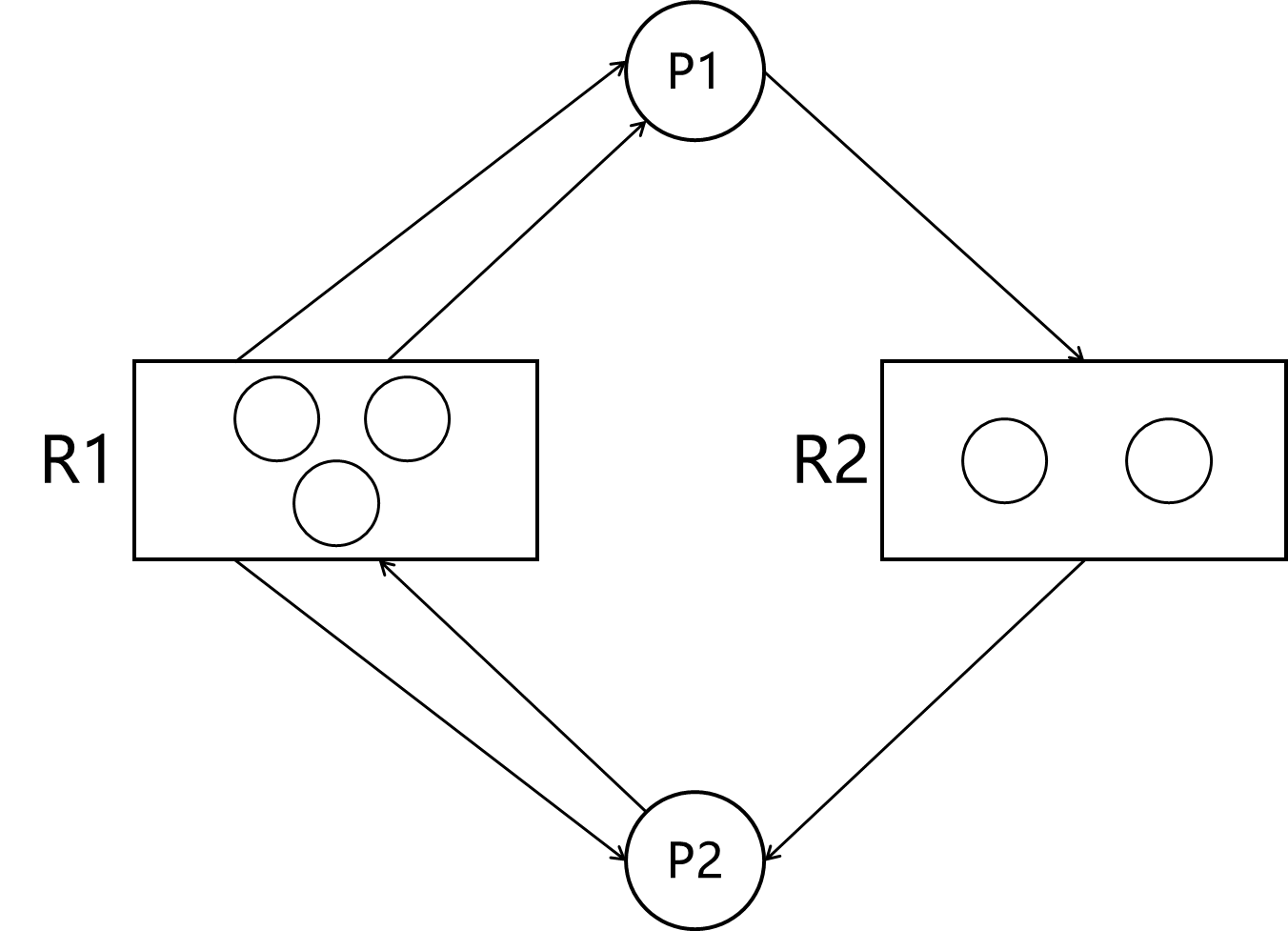
③资源不可剥夺

当一个资源分配给了一个进程后，其它需要该资源的进程不能强制性获得该资源，除非该资源的当前占有者显示地释放该资源。

④环路等待

死锁发生时，系统中一定有由两个或两个以上的进程组成的一条环路，环路上的每个进程都在等待下一个进程所占有的资源。

死锁的示意图如下：



进程P1申请2个资源R1，申请1个R2但未成功；进程P2申请资源R1与R2，试图再申请R1但并未成功。此时，可以发现，系统发生死锁。

死锁的检测算法为：

① ***Available***[m]是长度为m的向量，说明每类资源中可供分配的资源数目。

② ***Allocation*** [n,m]是n×m矩阵，说明已分配给每个进程的每类资源数目。

③ ***Request*** [n,m]是n×m矩阵，说明当前每个进程对每类资源的申请数目。

④ ***Work*** [m]是长度为m的工作向量。

⑤ ***finish*** [n]是长度为n的布尔型工作向量。

令 *k* =1,2,…,*n*,死锁检测算法的步骤如下：

① ***Work*** [\*] = ***Available*** [\*]

② 如果***Allocation*** [k,\*] ≠ 0，令***finish*** [k]=***false***；否则***finish*** [k] = ***true***

③ 寻找一个k，应满足条件

（***finish*** [k] == ***false***）&&（***Request*** [k,\*]≤***Work*** [\*]）

若找不到这样的k，则转向步骤⑤

④ 修改***Work*** [\*] = ***Work*** [\*] + ***Allocation*** [k,\*]，***finish*** [k] = true，然后转向步骤③

⑤ 如果存在***k***（1≤k≤n），***finish*** [k] = false，则系统处于死锁状态，并且***finish*** [k] = false的Pk是处于死锁的进程。

死锁的检测和恢复往往配套使用，当死锁被检测到后，采用各种方法解除系统死锁以恢复到可运行状态的常用方法有资源剥夺法、进程回退法、进程撤销法和系统重启法。

在本程序设计中，采用了进程撤销的办法进行死锁恢复，即当程序检测到死锁之后，会自动将与死锁有关的进程全部撤销。

**（六）页表生成**

进程中的页表为存储虚拟页面与实际物理页框的映射关系的数据结构，页表的定义在PCB数据结构下，具体代码为：

public short [][]page\_table=new short[(kernel.MEMORY\_USER\_SPACE\_SIZE)/

kernel.SINGLE\_PAGE\_SIZE][2];

该数据结构使用了二维数组的形式。在存储页表时，从上往下进行扫描，若碰到某一行未被写入，则将其进行替换。当需要更新时，选择指定的行进行替换即可。PCB类提供了对页表的添加、修改、查询、删除操作，用户可以通过PCB对象进行调用，有关的函数如下：

public void AddPageTable(short table\_data)

{

//增加页表项

int location=0;

while(this.page\_table[location][0]!=-1)

{

location++;

}

this.page\_table[location][0]=(short) location;

this.page\_table[location][1]=table\_data;

}

public void EditPageTable(short line,short data\_1,short data\_2)

{

//修改页表

this.page\_table[line][0]=data\_1;

this.page\_table[line][1]=data\_2;

}

public short CheckPageTable(short line\_no)

{

//查询第line\_no项表的值

return this.page\_table[line\_no][1];

}

**（七）四态转换**

在程序中，设计了四种不同的状态，分别为：运行态、就绪态、等待态、挂起态。当进程执行不同的指令时，其所处于的状态也会随之改变。

进程状态的描述在kernel.java中进行描述，使用全局静态常量进行定义：

/\*Process State 进程状态参数\*/

public final static short PROCESS\_READY = 0; //就绪态

public final static short PROCESS\_WAITING = 1; //等待态

public final static short PROCESS\_RUNNING = 2; //运行态

public final static short PROCESS\_SUSPENSION = 3; //挂起态

/\*Process State 进程状态参数\*/

在PCB中，有专门的数据项state进行描述。当进程触发其进程原语时，便会自动检测当前所处的状态与时机，自动的修改状态。

当程序需要检测进程的状态时，可以使用PCB所提供的API接口进行查询，但是，对于设置state的API有严格要求，不允许外界直接调用，只允许进程管理模块调用。

与state变量有关的函数操作为：

public short GetState() {

return state;

}

public void SetState(short state) {

this.state = state;

}

**（八）进程同步互斥**

为了模拟仿真进程的同步互斥，需要使用PV的信号量。根据操作系统原理，同步互斥是两个并发进程的执行顺序关系的描述。

多个进程由于争夺同一资源而导致同步互斥，根据这一原理，可以在系统初始化时模拟PV信号量的值，设计不同的P指令与V指令，各自对应不同的信号量。

信号量的值的定义在kernel.java文件中，定义为：

public static int[] MUTEX= {-4,-3,-2,-1,0,1,2,3,4,5};

当进程执行到P指令或V指令时，首先检测信号量的值是否满足条件（≥1），若满足，则可以申请并申请成功；若不满足，则进入等待队列，下一次继续执行该指令尝试。

与PV信号量检测有关的函数为：

public boolean Process\_P\_Mutex(PCB pcb,int num)

{

//某个进程P第num个mutex信号量，只有成功了才将其放入apply队列

//返回值为能够P成功

if(kernel.MUTEX[num]<=0) //当前没有资源，不能P

return false;

else //有资源，可以P

{

kernel.MUTEX[num]--;

this.PV\_apply[num].add(pcb);

return true;

}

}

public void Process\_V\_Mutex(PCB pcb,int num)

{

//某个进程V第num个mutex信号量

kernel.MUTEX[num]++; //释放资源

this.PV\_apply[num].remove(pcb); //将pcb移出

}

**（九）进程原语**

为了实现进程的不同状态的转换以及进程的创建、撤销功能，在PCB中定义声明了进程原语。

进程控制原语包括：进程的建立、进程的撤销、进程的等待和进程的唤醒。

操作系统执行和监督进程控制操作，往往通过执行各种原语操作实现。计算机控制器的机器指令是微操作构成的，原语是机器指令的延伸，是由若干条机器指令构成用以完成特定功能的一段程序。

原语的定义在PCB.java文件中可以找到，分为：进程创建、进程撤销、进程阻塞、进程挂起、进程唤醒。其定义为：

public void ProcessCreate()

{

//进程原语：进程创建

//将该进程设置为就绪态，加入到就绪队列

ProcessModule.process\_module.all\_queue.add(this);

this.state=kernel.PROCESS\_READY;

ProcessModule.process\_module.TransferProcessToReadyQueue(this,true);

}

public void ProcessCancel()

{

//进程原语：进程撤销

//设置进程结束时间、移入完成队列

this.end\_time=kernel.SYSTEM\_TIME;

ProcessModule.process\_module.TransferProcessToEndQueue(this);

//清空页表

for(int i=0;i<(kernel.MEMORY\_USER\_SPACE\_SIZE)/kernel.SINGLE\_PAGE\_SIZE;i++)

{

page\_table[i][0]=-1;

page\_table[i][1]=-1;

}

//释放该进程占有的所有页

for(int i=0;i<(kernel.MEMORY\_USER\_SPACE\_SIZE)/kernel.SINGLE\_PAGE\_SIZE;i++)

{

if(this.page\_table[i][1]!=-1)

PageModule.page\_module.RecyclePage(this.page\_table[i][1]);

}

//释放进程占用的所有PV信号量和资源

//释放PV信号量

for(int i=0;i<10;i++)

{

if(DeadLock.dl.PV\_apply[i].contains(this)==true) //如果该PCB申请了资源

{

DeadLock.dl.PV\_apply[i].remove(this); //将该进程移出

kernel.MUTEX[i]++;

}

}

//释放申请的资源

for(int i=0;i<10;i++)

{

if(DeadLock.dl.Allocation[this.GetPid()][i]>=0)

{

DeadLock.dl.Available[i]+=DeadLock.dl.Allocation[this.GetPid()][i];

DeadLock.dl.Allocation[this.GetPid()][i]=0;

DeadLock.dl.Request[this.GetPid()][i]=0;

}

}

//将该PCB从PCB池中删除

ProcessModule.process\_module.DeletePCBInPool(this);

}

public void ProcessWait()

{

//进程原语：进程阻塞

//将进程加入到阻塞队列

ProcessModule.process\_module.TransferProcessToWaitQueue(this);

this.state=kernel.PROCESS\_WAITING;

}

public void ProcessSuspend()

{

//进程原语：进程挂起

//将进程加入到挂起队列

ProcessModule.process\_module.TransferProcessToSuspendQueue(this);

this.state=kernel.PROCESS\_SUSPENSION;

}

public void ProcessWake()

{

//进程原语：进程唤醒

//将该进程从等待态移出，加入到就绪队列

ProcessModule.process\_module.TransferProcessToReadyQueue(this,true);

this.state=kernel.PROCESS\_READY; //修改进程状态为就绪态

}

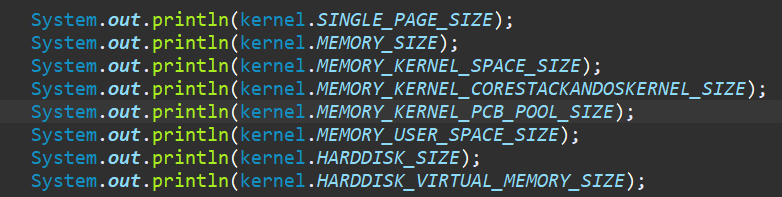
**三、功能测试**

**（一）系统常量**

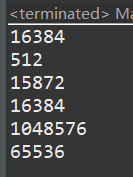
程序设计了多种系统常量，读者可以在kernel.java中进行查看。同时，这些系统常量为静态类型，即说明在任何地方都可以进行使用。

测试方法如下：

为了调用这些变量，可以在Main函数中直接调用system.out.println函数进行测试，测试代码如下：



输出的结果如下：



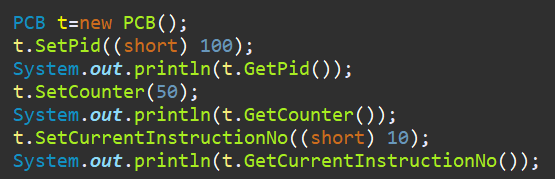
可见，输出的结果正确。

根据测试，可以得到正确的输出值。此处只测试了一小部分变量，在测试其他变量时，读者可以使用kernel进行引导测试。此处的测试只是为了说明在JAVA中，静态变量是一个全局通用的变量类型。

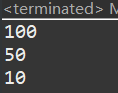
**（二）PCB测试**

在测试PCB时，可以在Main函数中实例化一个PCB类的对象，并对其进行操作。程序在设计PCB时，提供了多种不同的Get与Set函数，可以保证在测试时对PCB结构的完全控制。

测试方法如下：



输出的结果如下：



可见，测试结果正确。

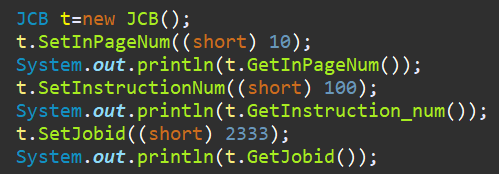
对于PCB的使用，对于PCB类的初始化与回收，是在程序运行过程中常用的一个功能。具体的使用方法，可以参考三级调度函数的写法与功能。

**（三）JCB测试**

程序设计时，创建了JCB的数据结构（其结构可以见上文的描述）。JCB结构是对于作业的描述，其中也包含有一定数量的变量可供使用。程序为此提供了Get方法与Set方法，读者可以在Main函数中实例化一个JCB类的实例化对象，然后对其进行调控测试。

测试的方法与上文中描述的PCB的测试方法一样，在此不做赘述。

测试方法：



测试结果如下：



可见，测试结果正确。

**（四）死锁检测**

程序的功能之一便是实现了死锁的模拟仿真与检测算法。在控制台界面下，读者可以不通过UI界面直接测试死锁检测算法的正确性。

在程序设计中，DeadLock类对外提供了多种函数API（详情见上文），读者可以调用他们来模拟每一个进程的申请操作。

主要用到的API有：

public boolean Process\_Apply\_Resource(PCB pcb,int num)

//某个进程申请某个资源

public int GetResourceNum(int num)

//获取某类资源的数目

public void Process\_Return\_Resource(PCB pcb,int num)

//某个进程归还资源

public ArrayList<PCB> CheckDeadLock()

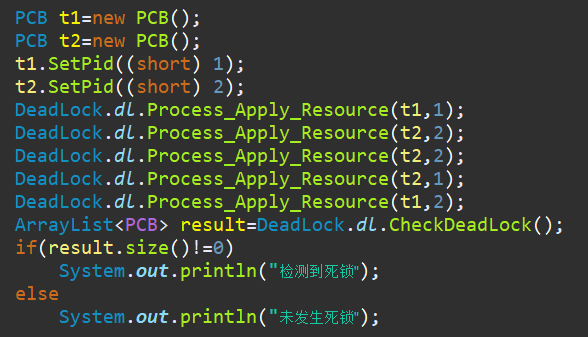
//死锁检测

通过这些函数，读者便可以模拟仿真每一个进程的资源申请请求与死锁的检测。

同时，由于这些函数本身并不面向控制台提供功能，所以比较难以调用。读者可能会因为不理解本程序的设计而不能够操作。

在此，提供测试的方法。

现在，假设死锁产生的条件：有A资源1个，B资源2个。有进程1与进程2，进程1与进程2申请资源的顺序为：进程1申请1个A资源并成功，进程2连续申请2个B资源并成功，进程1申请B资源失败并等待，进程2申请A资源失败并等待。以上模拟的是一种比较简单的死锁，将其中代码的形式表示可以为：



执行结果为：



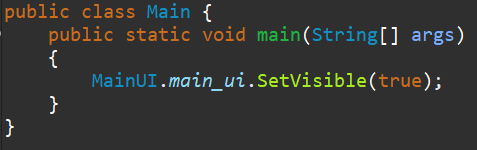
可见，运行结果正确！

**（五）UI界面**

本程序的三级调度功能无法通过控制台界面与Main函数接口直接调用进行测试。因此，下文将展示如何通过UI界面进行操作，来检验不同的函数的正确性。

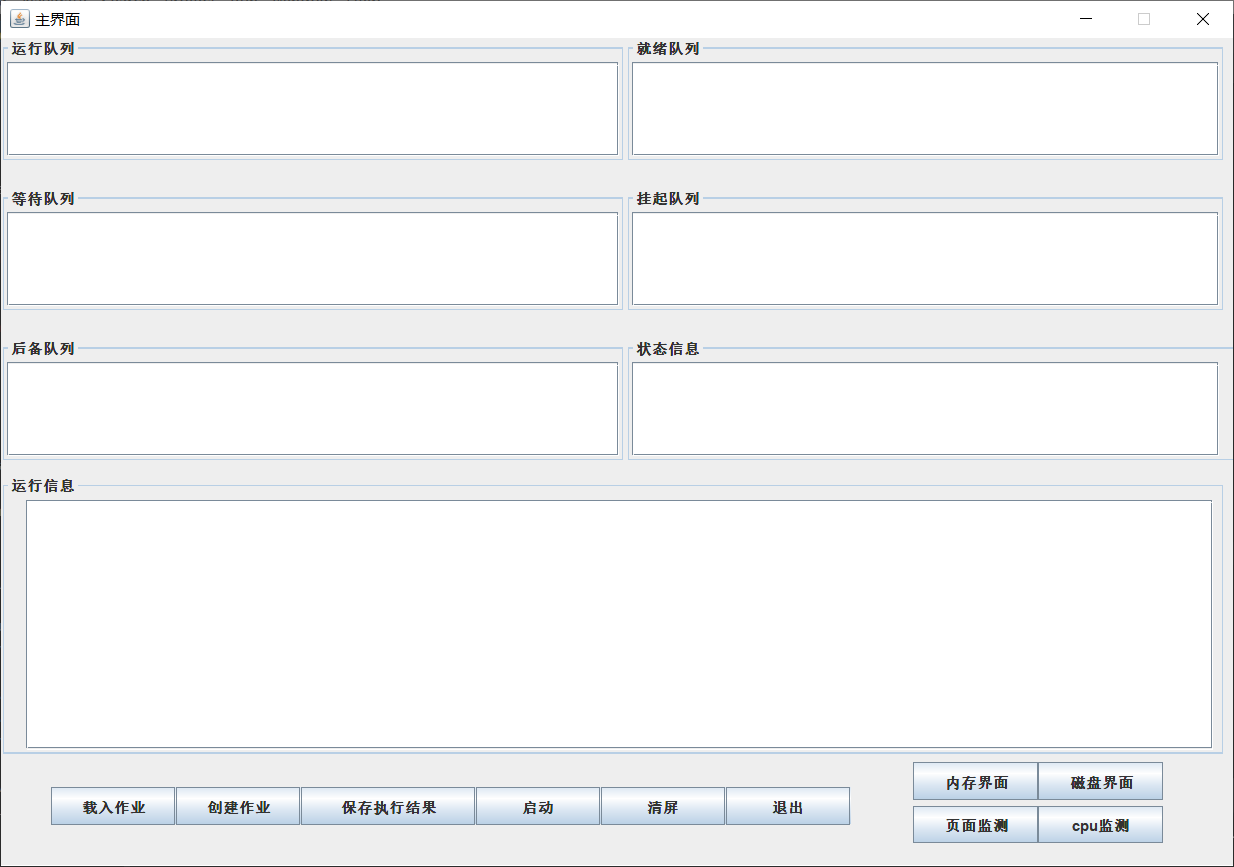
同时，由于程序的函数实现较多，逻辑上较为复杂。读者若不能够理解程序的含义，建议通过Eclipse自带的单步调试功能进行测试。

整个程序由Main函数进入，由UI界面进行整个程序的调控。所以，Main函数的设计较为简单，为：



通过此函数的调用，可以直接打开主界面。

当程序运行后，出现主界面，截图如下：



在开始时，主界面没有任何内容，需要读者手动导入作业并运行。

作业导入后，点击“启动”按钮便可以开始运行，点击“清屏”按钮，便可以清空当前“运行信息”文本框中的信息。

其他按钮的功能为：

创建作业：打开一个创建作业的窗体，进行创建作业。

退出：退出程序

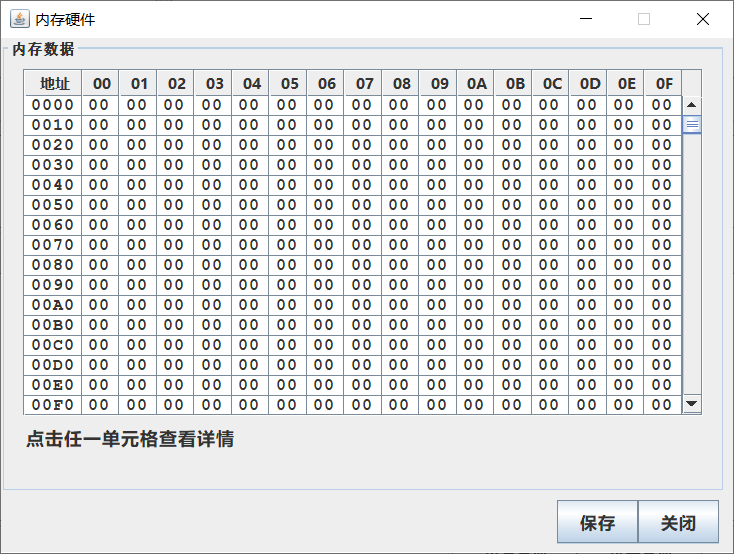
内存界面：打开内存数据查看界面，可以查看当前内存中所有的数据信息

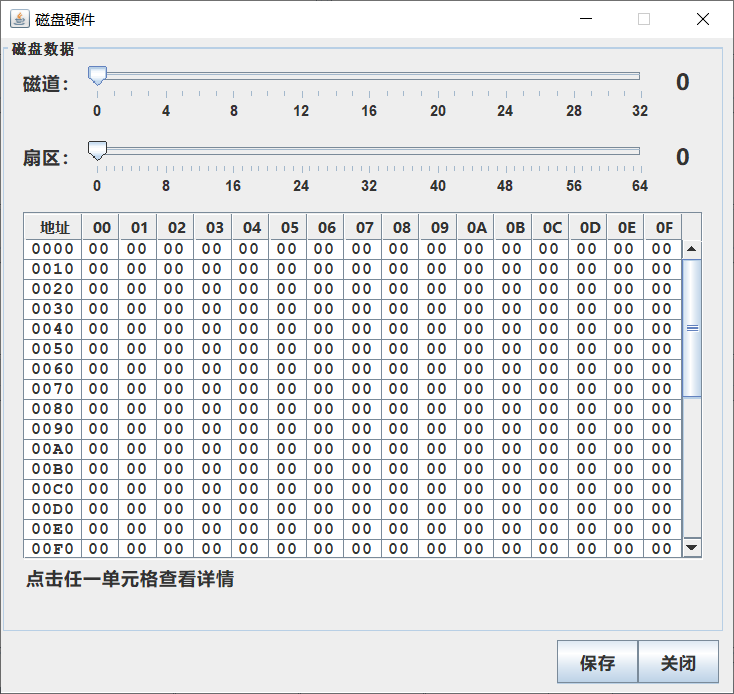
磁盘界面：打开磁盘数据查看界面，可以查看当前磁盘中所有的数据信息

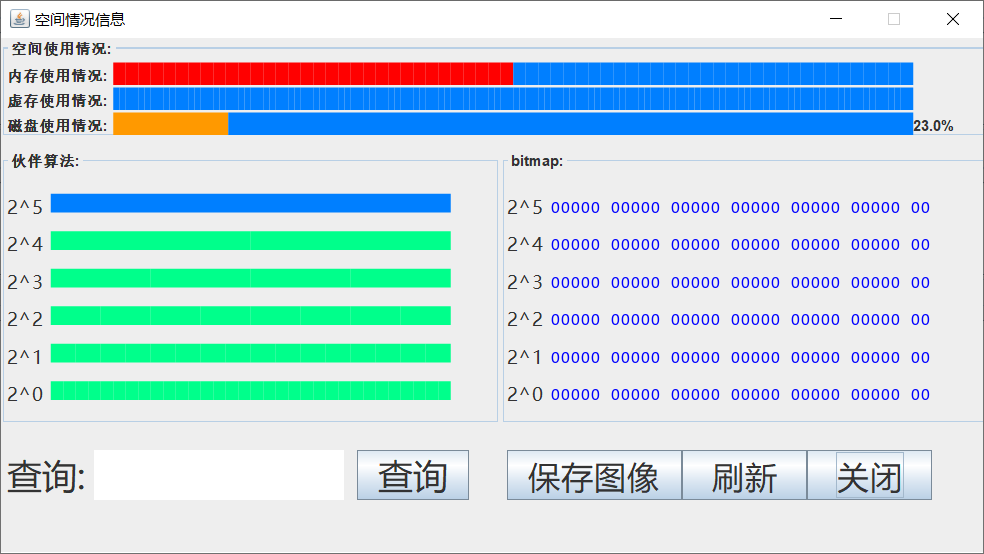
页面检测：打开页面管理的检测界面，可以查看内存、虚存、磁盘的空间占用情况和伙伴算法的运行原理和bitmap

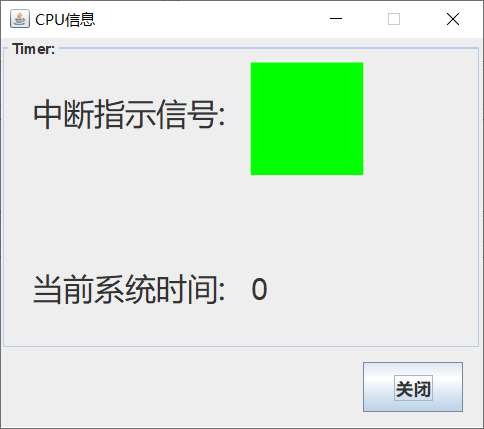
CPU检测：打开CPU检测窗体，可以检测此时CPU的具体中断信息与当前系统时间。

这些窗体的截图如下：







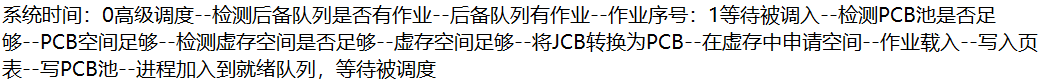


**（六）三级调度**

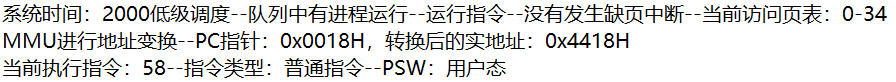
为了模拟实现作业、进程的三级调度，程序在界面中将以直观的方式予以体现。

以上述步骤进行作业导入后，点击“启动”开始程序。在程序的运行结果输出中，程序将按照时间间隔的不同输出三级调度的不同信息。

如图：



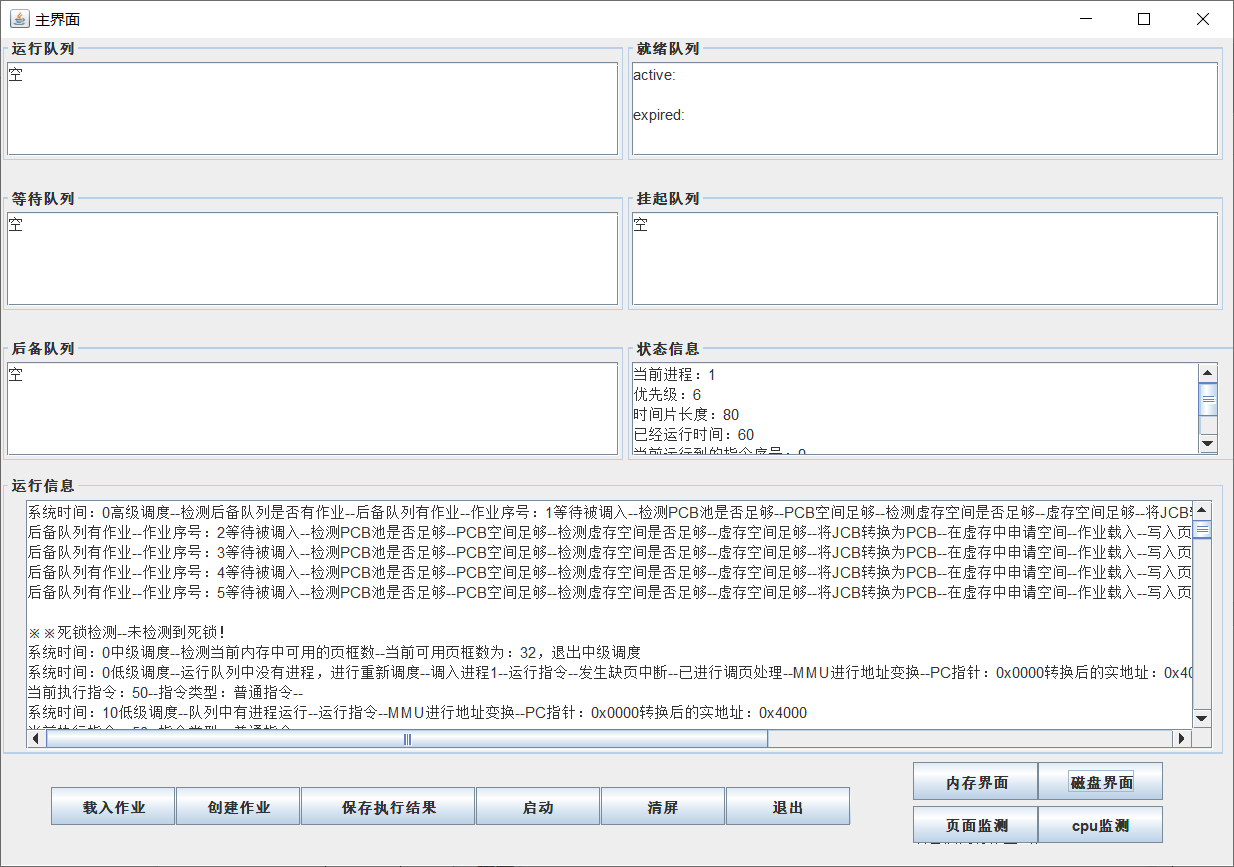


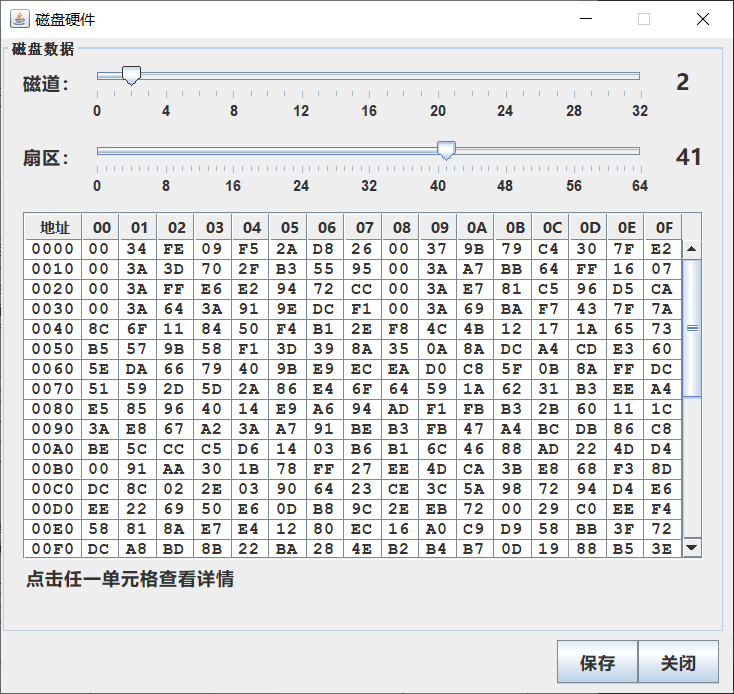


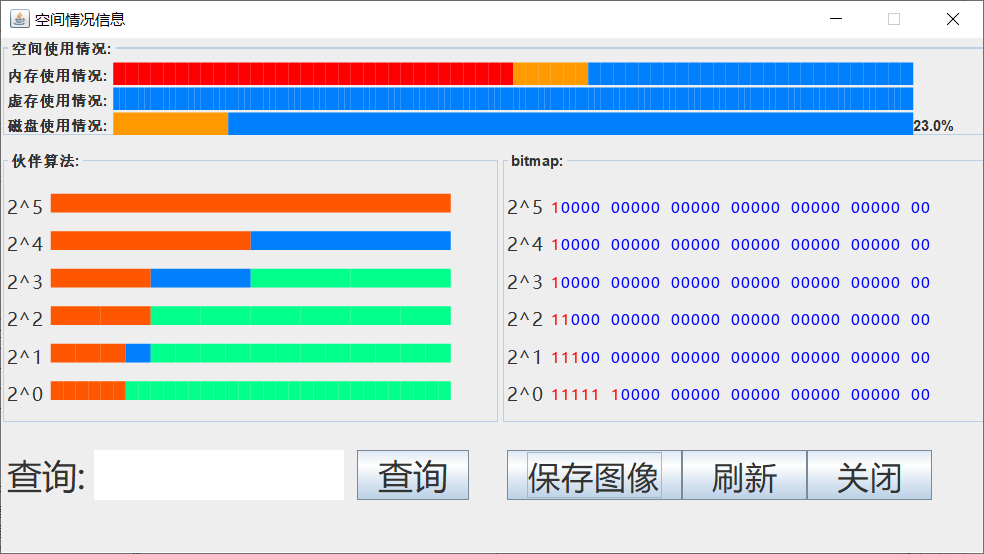
**（七）指令运行**

程序最基本也是最重要的功能便是指令的运行。在该测试中，将测试程序对于指令的运行模拟，并分析其正确性。

点击“载入作业”，从文件中读取作业，此处使用了“/测试输入”文件夹下的样例1进行测试，程序的运行结果如下：







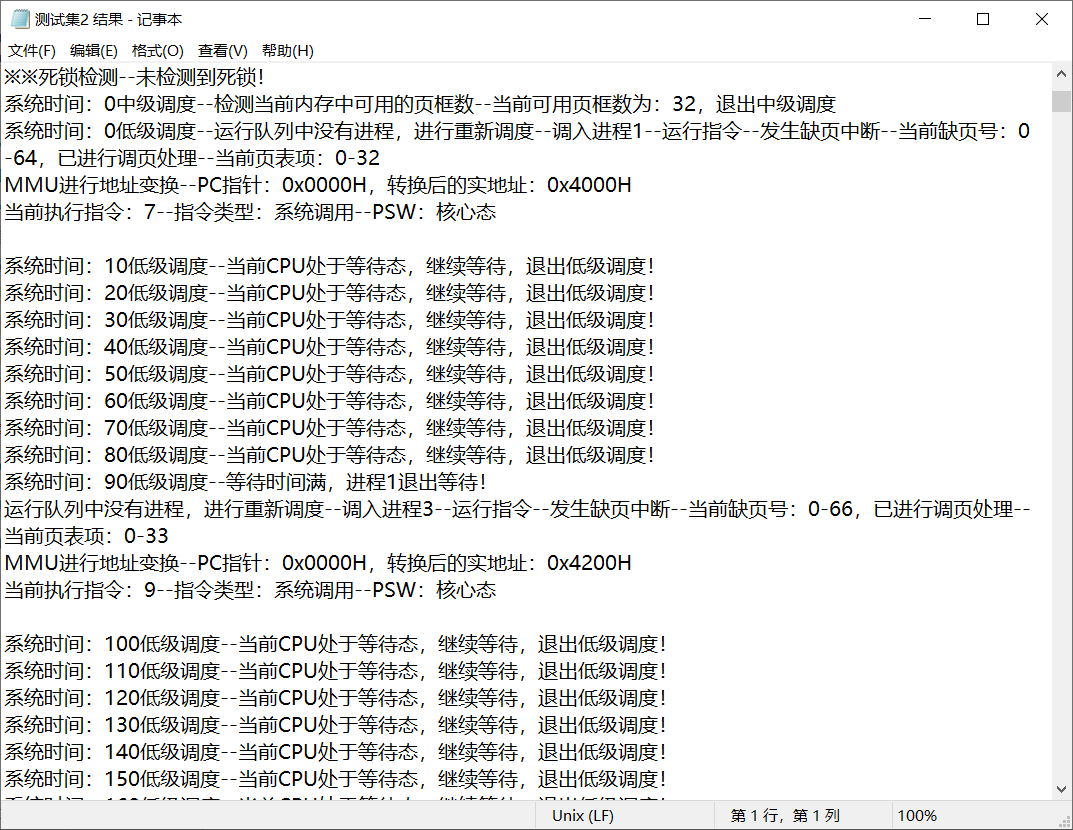
将程序的运行结果输出保存到文件，文件内容如下：



该作业中全部为普通类型指令，即不存在系统调用，因此不会进入等待队列。当进程运行到时间片到时，则进行进入就绪态，进行重新调度。根据进程的优先级不同，调入优先级最高的进程进行运行。

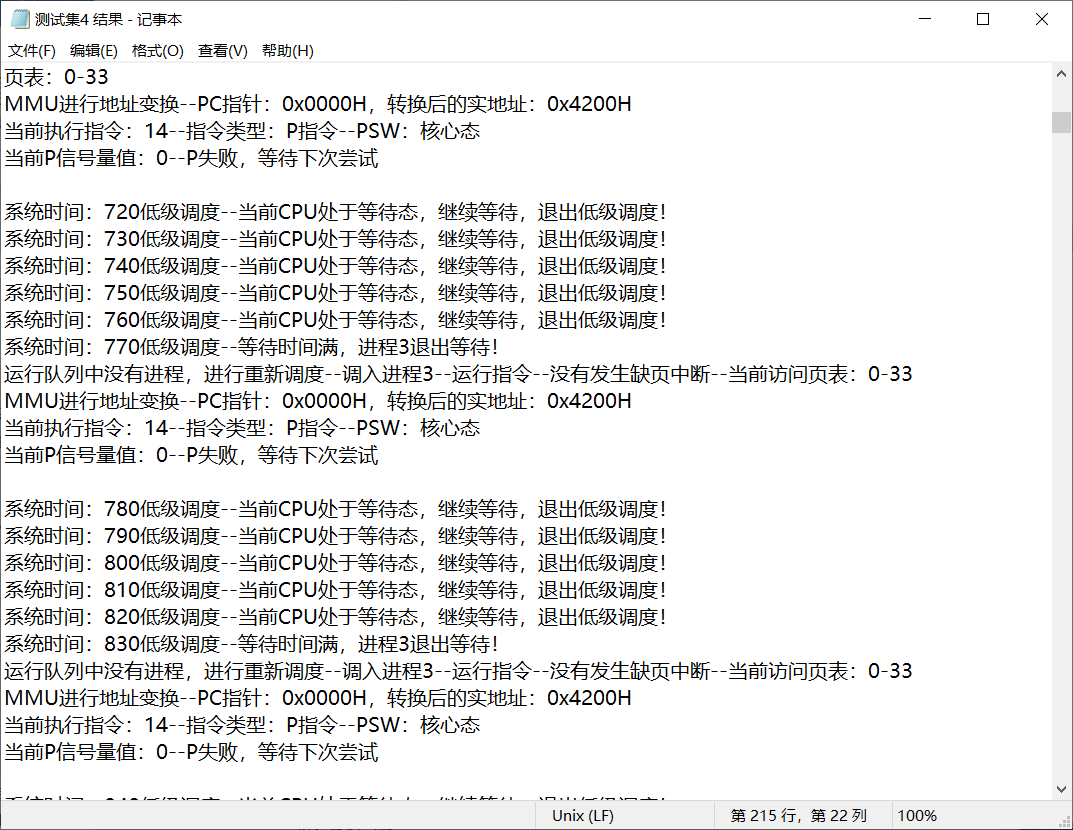
接着，开始测试系统调用的情况。

选择“/测试输入”文件夹下的测试样例2文件进行导入，查看运行结果。结果如图所示：

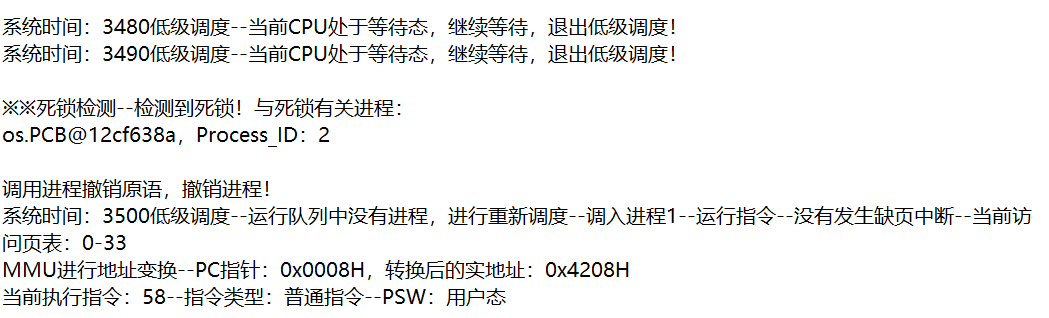


可以看到，当执行到系统调用指令时，CPU直接进入等待状态。在下一次进行低级调度时，CPU直接会检测到当前的等待队列中有进程且还没有能够达到出等待态的条件，这时，CPU直接放弃低级调度，继续等待等待队列中的指令执行完成。

测试PV操作的情况，同样可以使用“/测试输入”文件夹中的对应文件进行输入，输出的结果，可以反映出程序试图P和试图V信号量时的具体信息。通过这些测试输入，便可以体现出PV操作的同步互斥的状态，如图：



测试死锁状态检测功能，同样可以导入对应的测试样例，检查输出文件：



可以看到，程序自动检测到了死锁，并调用了该进程的撤销原语进行撤销。当该死锁进程被撤销后，其他的进程便可以继续开始运行，直至所有进程运行结束并退出。

**四、技术问题分析报告**

**（一）精确时间计时**

**1. 问题描述**

为了能够准确的发生10ms的中断时间，故需要设计中断的产生函数。

开始，拟采取每一次检测当前系统时间，调用系统API函数GetLocalTime()进行检测，然后每一次与上一次获得的时间相减，判断是否为10ms。

经过分析，该种方法所需要消耗的CPU资源过于庞大，不适合该程序的运行与设计，且JAVA为解释性语言，对于系统调用的实现过于缓慢，不适合在系统上运行。

**2. 解决方法**

利用JAVA Thread线程中的Sleep()函数，可以将睡眠时间精确到ms级别。同时，又因为该函数为线程，独立的执行，因为不会对其他程序产生影响。

**（二）队列指定元素删除**

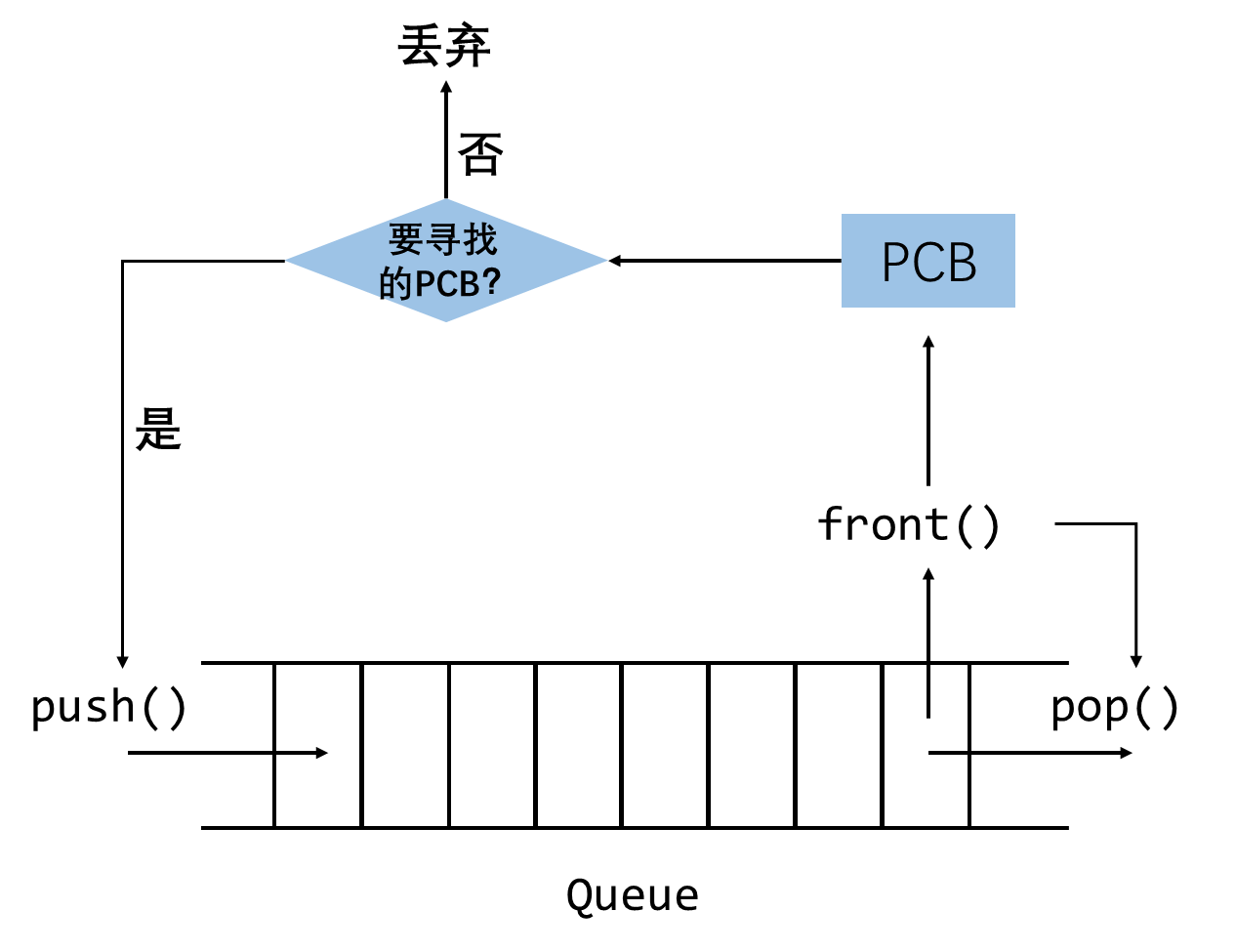
**1. 问题描述**

当PCB的状态转换时，需要从当前队列中移出。但是，因为队列Queue的特性，只能够先进先出，无法直接移出指定位置的元素。所以，需要编写代码实现从队列中删除指定序号的元素。

**2. 解决方法**

设计了算法进行该操作，将指定元素进行删除。该算法利用了队列先进先出的特性，每一次弹出一个元素进行比较，若不是，则将其插入到队列的最后，若是，则不将其插入到最后。

实现过程的原理如图所示。



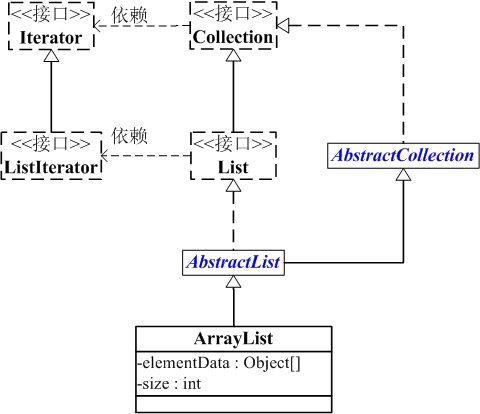
在后来的程序撰写中，又发现了一种JAVA常用的数据结构ArrayList，ArrayList是利用数组的形式模拟仿真链表。不仅可以体现出链表、队列的特性，还能够像数组一样在指定位置插入元素。

ArrayList 是一个数组队列，相当于 动态数组。与Java中的数组相比，它的容量能动态增长。它继承于AbstractList，实现了List, RandomAccess, Cloneable, java.io.Serializable这些接口。

ArrayList 继承了AbstractList，实现了List。它是一个数组队列，提供了相关的添加、删除、修改、遍历等功能。

ArrayList 实现了RandmoAccess接口，即提供了随机访问功能。RandmoAccess是java中用来被List实现，为List提供快速访问功能的。在ArrayList中，我们即可以通过元素的序号快速获取元素对象；这就是快速随机访问。稍后，我们会比较List的“快速随机访问”和“通过Iterator迭代器访问”的效率。

ArrayList与Collection关系如下图：



可见，ArrayList中可以兼容各种不同类型的数据结构，方便了PCB的存储。

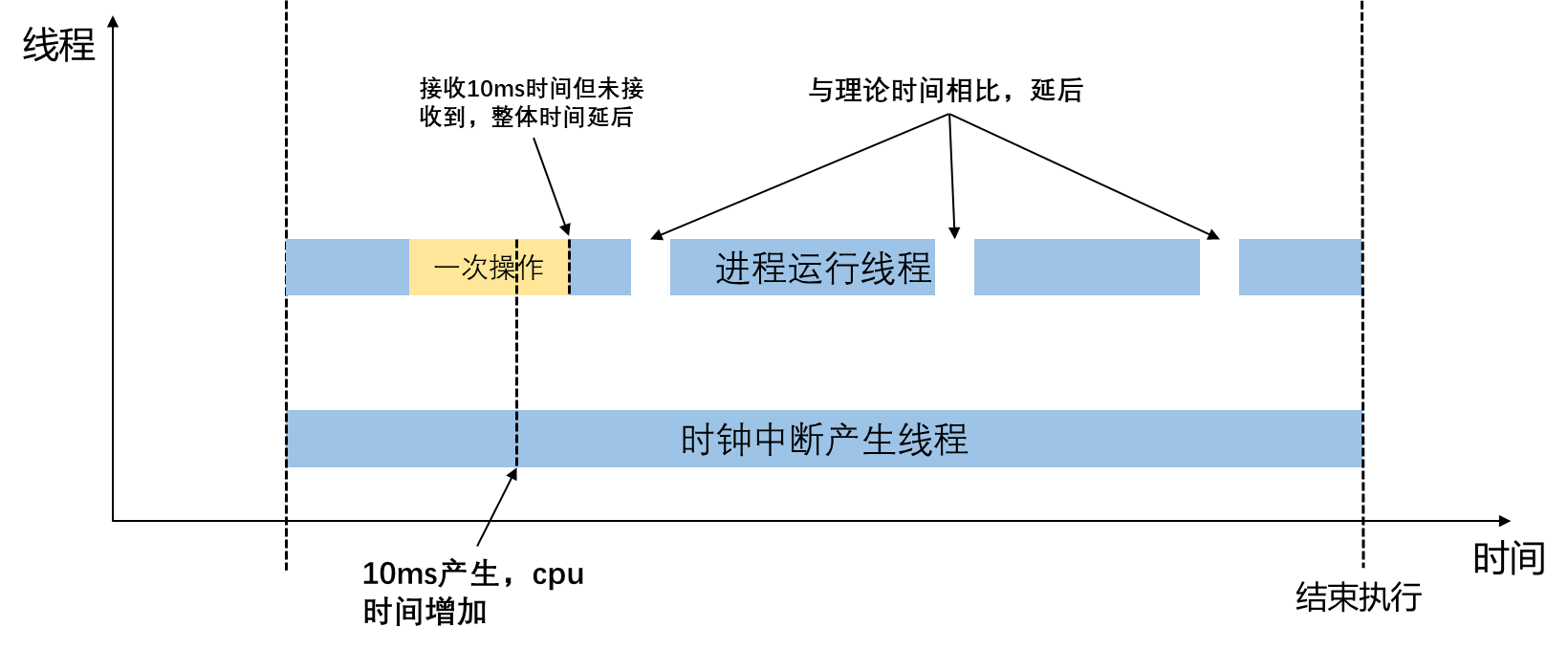
**（三）cpu内部时间计数**

**1. 问题描述**

为了能够精确的描述系统内部的时间，故需要使用一个全局唯一的cpu\_time变量来给出时间，并且随着每一次时钟中断的产生，其时间也应相应的做出变化。

cpu\_time变量应只让一个函数拥有修改权限，对其他所有的函数只有读取权限。但是，这个权限如何分配是一个问题。

若让时钟中断产生线程接管该权限，则会有时候导致时间不准的问题。具体表现为：当一个线程的一次处理操作未进行完成时，时钟中断已经到来，这时，该线程无法准确接收中断信号，导致后面的操作被整体的延后，与前面的时间产生脱节，如图 时间延后问题所示。

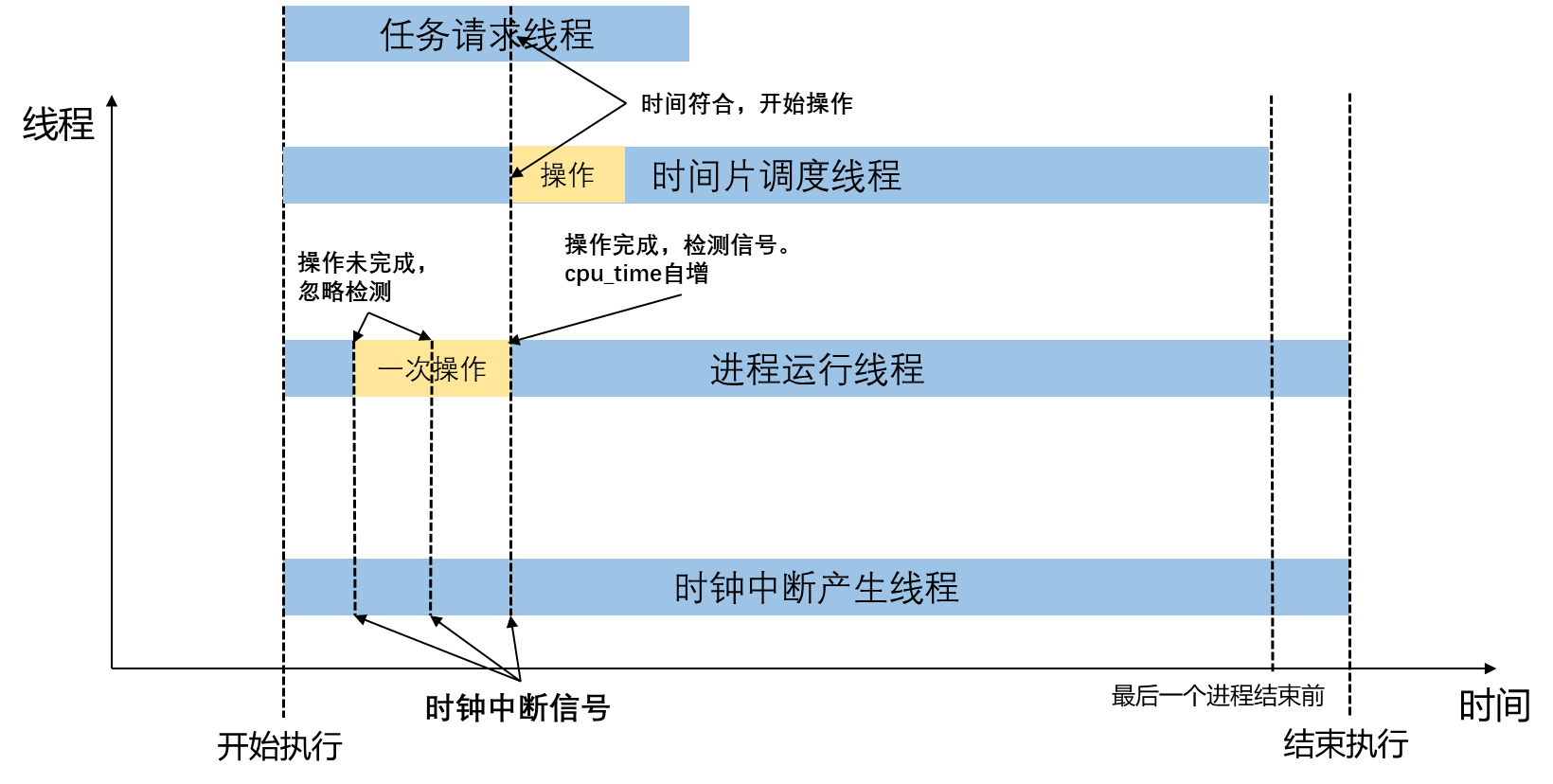


cpu时间延后问题

**2. 解决方法**

为了解决时间不同步的问题，可以将时间的修改权限转移至“进程运行进程”。因为进程运行进程与时钟中断产生进程是同步开始与结束的，所以权限的转移并不会影响。对于真实时间与cpu时间不同步的问题，可以有选择的忽略一些10ms的时钟中断，每一次的cpu\_time的计时以线程实际接收的为准。

每一次当线程接收到中断信号后，cpu\_time才能够自增。利用这种方式，虽然于现实的时间略有误差，但能够保证cpu时间的绝对准确。同时，给除了时钟中断产生线程和进程运行线程之外的其他线程加上信号检测，可以实现多个线程之间的信号传递，进一步保证时间的准确性。该实现方法的原理图如图6-3 时间延后解决原理所示。



**（四）三级调度的优先级控制**

**1.问题描述**

为了能够使得程序在确定的时间开始进行三级调度，需要设计三级调度的具体实现。最理想的方案为，将三级不同的调度放到不同的线程中去。但是，由于JAVA本身的机制问题，三个不同的线程同时启动，会导致他们同时调用同一个API函数来检测时间，这样的机制有可能会导致线程之间调度顺序的混乱，无法做到：高级调度🡪中级调度🡪低级调度 的有效顺序。

**2.解决方法**

在经过了多次的尝试过后，由于自身的能力以及JAVA平台的限制，最终决定放弃使用多个调度多线程运行的机制。现在，将多级调度写入到同一个线程中，由这一个线程启动所有的调度，顺序为：高级调度🡪中级调度🡪低级调度。通过这样的方式，可以保证三级调度的有效进行，但是，这也大大降低了系统设计“并发性”的要求。

**五、实验心得**

经过一个寒假的努力，与队友付出了艰苦卓绝的努力，现在的系统终于全部完成。虽然现在的系统在运行时仍有一些瑕疵与小问题，但这些多是由于JAVA的多线程本身的问题所导致。

经过操作系统课程设计的完整的程序设计与开发，我对于操作系统整体概念与进程管理、内存管理的理解愈加深刻。

在实验过程中，令我印象最为深刻的是对于操作系统理论的理解与对于程序框架的搭建。

个人认为，在本次课程设计实验中，为了设计出一个符合要求，能够模拟仿真Linux2.6的系统，最为重要的是对于操作系统的整体概念的理解掌握。有了这些基础知识，才能够理清楚系统中各个部分、各个功能的含义以及他们与其他模块的联合调用关系，不然，在设计系统时将会举步维艰，寸步难行。

有了基础知识，其次便是程序结构的设计。一个好的程序结构，应该充分考虑系统的各方面特性，如系统与Linux2.6进程管理与内存管理的仿真如何实现，系统的模块划分是否合理，系统的内聚与耦合程度如何……在经过了艰苦卓绝的程序框架构思与设计后，小组终于设计出了一个很满意的程序框架。在此框架上编写的程序代码，相信能够有很好的表现。

其次，良好的团队合作也是本次课程设计能够顺利进行的关键。为了避免两个人合作开发的效率比一个人单独开发还要慢，小组编写了多份程序开发规范，以方便两个人之间编写开发的代码能够互相交流与调用。最后的系统功能集成是由我来负责，有了之前编写好的开发规范，最后对于系统的集成便可以事半功倍，大大的提高效率。通过这次合作，也让我明白了在团队合作中，有一个好的规范的重要性。

在此次的课程设计中，我主要负责：系统整体框架的构思与搭建、程序设计规范的撰写、三级作业调度过程及算法、进程死锁检测与撤销算法、可视化方式呈现过程、进程同步互斥的实现、系统功能集成测试。

通过测试，系统完整的实现了实验要求任务书上的全部功能，测试通过。同时，在多种复杂的情况下皆可以测试通过。

综上所述，本次操作系统课设不仅锻炼了我对于JAVA程序开发语言的编写能力，又加深了我对于Linux系统内核的理解。

**六、参考文献**

[1]费翔林,骆斌.操作系统教程(第五版)[M].北京:高等教育出版社,2014

[2] Daniel P. Bovet,Marco Cesati.深入理解linux内核(第三版)[M].北京:中国电力出版社,2008

[3] 沈勇,王志平,庞丽萍.对伙伴算法内存管理的讨论[J].计算机与数字工程,2004(03):40-43.

[4] 岳笑含,刘艳秋.基于Linux 2.6进程调度系统的实时性研究[J].沈阳化工学院学报,2010,24(01):79-83.

[5] 薛峰.Linux内核伙伴系统分析[J].计算机系统应用,2018,27(01):174-179.

[6] 刘磊锋,刘皓章.伙伴算法在Linux内核中的应用及其改进[J].软件导刊,2010,9(10):59-61.

[7] 仇阳.Linux内核进程调度算法发展[J].电子世界,2017(07):85+87.

[8] 戴红.嵌入式Linux的预防死锁算法[J].吉林工程技术师范学院学报,2004(06):4-7.

[9] 马瑾利.PV操作实现进程同步互斥的研究[J].科学技术创新,2018(29):90-91.

[10] 鲁力,韩洁,徐琴.PV操作解决进程同步问题的难点研究与实现[J].电脑知识与技术,2017,13(13):38-39.

[11] Linux---死锁及避免死锁的方法

<https://blog.csdn.net/qq_37934101/article/details/81869245>

[12] Linux 2.6 schedule() 切换进程时没有释放rq->lock却又为何不会导致死锁？

<https://blog.csdn.net/jektonluo/article/details/50085051>

[13] Linux 伙伴算法简介

<https://www.cnblogs.com/cherishui/p/4246133.html>

[14] Linux内存管理伙伴算法

<https://www.cnblogs.com/alantu2018/p/8527821.html>

[15] 高级调度-百度百科

<https://baike.baidu.com/item/%E9%AB%98%E7%BA%A7%E8%B0%83%E5%BA%A6/9256523?fr=aladdin>

[16] 中级调度

<https://baike.baidu.com/item/%E4%B8%AD%E7%BA%A7%E8%B0%83%E5%BA%A6/9256594>

[17] 低级调度

<https://baike.baidu.com/item/%E4%BD%8E%E7%BA%A7%E8%B0%83%E5%BA%A6/9256559>