南京農業大學

计算机操作系统课程设计

实践报告

题　　目: 可视化仿真实现 Linux2.6 进程管理与内存管理

姓　　名: 陈扬　　梁嘉文

学　　院: 信息科技学院

专　　业: 计算机科学技术系

班　　级: 网工 161　　计科 161

学　　号: 19316117　　19216126

指导教师: 姜海燕　　职 称: 教授

2019 年 3 月 15 日

目录

南京農業大學

计算机操作系统课程设计

实践报告

题　　目: 可视化仿真实现 Linux2.6 进程管理与内存管理

姓　　名: 陈扬　　梁嘉文

学　　院: 信息科技学院

专　　业: 计算机科学技术系

班　　级: 网工 161　　计科 161

学　　号: 19316117　　19216126

指导教师: 姜海燕　　职 称: 教授

2019 年 3 月 15 日

目录

可视化仿真实现 Linux2.6 进程管理与内存管理

摘要

为了检验自己的操作系统课程的学习情况与掌握程度，以及将该课程知识用编程语言描述的技能，本次小组选择“可视化仿真实现 Linux2.6 进程管理与内存管理”的题目进行操作系统课程设计。该课程设计对于理解操作系统中进程管理与内存管理的知识有着重要作用，同时，管理手段采用 Linux2.6 内核的规则，可借此机会检验自己的编程水平与 Linux 核心代码的阅读水平。

系统模拟仿真了 Linux2.6 系统，并实现了作业及进程并发环境、MMU 地址变换、进程原语、页表生成与页面调度算法、三级作业调度过程及算法（需作业调度到指令集，至少实现三态转换）、页面分配与回收算法、进程同步互斥、进程死锁检测与撤销算法，并将实现原理过程通过可视化方式呈现。

按照计算机理论，在程序框架上，将系统分为硬件、驱动程序、系统管理模块、UI 界面四个模块。

在硬件层面，系统根据题目需要，设计了 CPU、内存、外存（硬盘）、地址线数据线四个硬件，CPU 中还包含计时器与 MMU。CPU 负责进程指令的执行与数据的传递，计时器负责发出中断与系统时间的计算，MMU 负责系统中地址的变换。内存与外存是系统中的存储设备，所有作业、进程以及页面的管理都以这两个硬件为基础进行设计。

系统管理模块分为作业管理、进程管理与页面管理。作业管理模块是基于 CPU 与硬盘硬件的系统模块之一。该模块的功能是为作业的创建、存入、删除以及作业的调入检测提供相关的支持。作业管理模块在 java 工程中写在 JobModule.java 文件中，同时该类被 Control.java 调用。

进程管理模块是基于 CPU 与内存的系统模块之一。该模块的功能是为作业调入后转换成的进程提供管理功能。进程管理是整个系统设计中最为复杂的部分，其包括低级调度管理、中级调度管理、高级调度管理、进程链表等功能。进程管理模块在 java 工程中写在 ProcessModule.java 文件中，同时该类被 Control.java 调用。

页面管理是负责系统中对于页面存入、读取、对换、换入换出功能的管理模块。在页面管理中，该模块与其他模块的信息交换全部都通过 Page 类来进行。当有页面换入、换出请求发出时，该模块先将页面信息写入 Page 类的对象，再将该对象传出，当其他模块收到该对象时，也可以对该对象进行操作，以减少操作的复杂度。页面管理模块在 java 工程中写在 PageModule.java 文件中，同时该类被 Control.java 调用。

系统的 UI 界面为 java 程序直接提供给用户进行操作的界面，通过该界面，用户可以方便快捷的使用所有系统功能并知晓系统所有功能模块以及硬件设备的实时信息。

综上所述，该系统很好地实现了课程设计的所有要求，同时，还提供生动形象的 UI 界面，方便用户进行操作管理。

关键字：Linux2.6、系统仿真、进程管理、内存管理、JAVA 编程

可视化仿真实现 Linux2.6 进程管理与内存管理

1 实践目的与意义

为了检验自己一学期的操作系统课程的学习情况与掌握程度，以及将该课程知识用编程语言描述的技能，所以本次选择“可视化仿真实现 Linux2.6 进程管理与内存管理”的题目进行操作系统课程设计。

该课程设计对于理解操作系统中进程管理与内存管理的知识有着重要作用，同时，管理手段采用 Linux2.6 内核的规则，可借此机会检验自己的编程水平与 Linux 核心代码的阅读水平。

2 实践任务与合作

根据 Linux2.6 进程管理与内存管理原理，仿真实现作业及进程并发环境、MMU 地址变换、进程原语、页表生成与页面调度算法、三级作业调度过程及算法（需作业调度到指令集，至少实现三态转换）、页面分配与回收算法、进程同步互斥、进程死锁检测与撤销算法，并将实现原理过程通过可视化方式呈现。为了实现以上任务目标，我们进行了如下分工：

组长（陈扬）：

系统整体框架的构思与搭建
程序设计规范的撰写
三级调度过程及算法
JCB、PCB 的设计
死锁检测与撤销算法
页表生成
可视化方式呈现过程
进程同步互斥的实现
进程与进程原语的设计实现

组员（梁嘉文）：

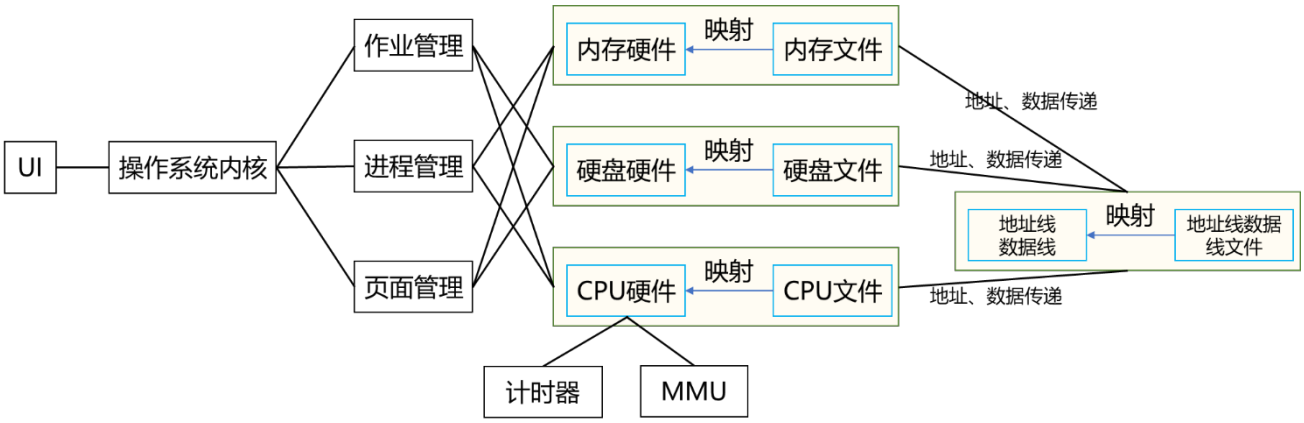
CPU 部件的仿真
内存空间的仿真实现
MMU 地址变换
页面设计实现
页面调度算法
页面分配与回收算法
可视化方式呈现过程

3 程序结构说明

本系统的结构设计参考了操作系统课本的设备管理章节，从最底层向上可分为：硬件、硬件驱动、系统管理模块、系统内核、UI 界面。同时，由于对于硬件部分的仿真本身就包含了存入、读取的功能，所以将硬件与硬件驱动进行合并。在系统管理模块部分，分为：作业管理、进程管理、页面管理三部分。在系统内核部分，系统提供了系统的全局变量与系统的操作类，用来将其下的三个模块进

行整合。再向上，即 JAVA 所提供的 UI 交互界面，可以方便用户使用该系统。

程序的总体结构设计如图所示。



4 裸机硬件仿真设计

4.1 CPU 设计

CPU 硬件中包含计时器与 MMU。在程序设计时，需要将两者分开，设计三个类，CPU 类“cpu”，计时器类“timer”，MMU 硬件类“mmu”。timer 与 MMU 只在 cpu 类中存有唯一一个实例，即可体现出计时器与 MMU 包含于 CPU 的概念。任何对计时器与 MMU 的调用，都需要通过 cpu 类中的实例来进行，而不能够直接访问。

CPU 类的定义为：

```
public class CPU
{
    public static CPU cpu=new CPU();
    public Timer ti;
    public MMU mm;
}
```

CPU 的结构及内容：

1、地址寄存器 PC

2、PSW 程序状态寄存器

3、IR（指令寄存器）

4、页基址寄存器 CR3

5、在时间片结束时执行完的指令数量 already_run

4.2 内存设计

内存大小为 32KB，故可根据该要求设计 memory 类，该类有一个总大小为 32KB 的对象数组，为 short 类型，因为系统要求地址线与数据线为 16 位，所以在该系统的设计时，统一使用 short 类型数据。需要注意：内存 memory 类为文件的映射，因此，在该类被实例化之前，其构造函数需要进行从文件到对象的映射，即读取文件内容，并根据文件内容初始化该实例。内存硬件类 memory 只是简单地对硬件进行模拟仿真，不需要过多复杂的操作，只需要提供两个基本操作即可，即数据的存入与取出。但是，数据的存入与取出都需要通过地址线与数据线。

内存类的定义为：

```
public class Memory
{
    public static Memory memory=new Memory();
    private byte []data=new byte[32*1024];          //32KB=32768B
}
```

内存总大小为 32KB，规定：前 16KB 为内核区，后 16KB 为用户区

内核区存储的内容：核心栈+系统内核，进程所有 PCB 信息

综上所述，内存的组成结构为：

核心栈+系统内核（1 页）、PCB 池（31 页）、用户区（32 页）

4.3 硬盘设计

与内存硬件的设计相似，硬盘类只提供简单的数据存入取出。但是，数据的存入与取出也都需要通过地址线与数据线。硬盘硬件只负责最基本的硬件仿真。硬盘 harddisk 类为文件的映射，因此，在该类被实例化之前，其构造函数需要进行从文件到对象的映射，即读取文件内容，并根据文件内容初始化该实例。

硬盘类定义为：

```
public class HardDisk
{
    public static HardDisk harddisk=new HardDisk();
    private byte [][][]data=new byte[32][64][512];
}
```

硬盘总大小为 1MB，规定：

前 48KB 为虚存区（与内存的 16KB 用户区组合，加起来共 64KB，也是 16 位地址线的最大寻址空间）

再 16KB 为系统文件区（用来模拟开机后需要被加载入内存的系统文件）

剩下的 960KB 都为文件区（用来存储作业）

综上所述，硬盘的组成结构为：

虚存区（96 页）+系统文件区（32 页）+文件区（1920 页）

4.4 地址线数据线

地址线数据线为计算机硬件中存在但是会被经常忽略的部件。在该系统的设计中，有必要对地址线与数据线进行仿真。

根据计算机组成原理与系统结构的知识，地址线与数据线为在物理上连接 CPU 与其他各个带有存储功能部件的物理结构。在设计时，若对每一个部件都进行地址线与数据线的存入取出处理，必然会增加系统设计的复杂程度。故地址线与数据线的设计可考虑进行适当地简化。

故将设计思想简化为：只有当硬件需要进行存取操作时，才开始调用地址线与数据线。

5 通用数据结构设计

5.1 页表设计

Page 类设计如下：

```
{
    private int page_num;      //页号
    private short[] data=new short[512/2]; //每页大小=512B=256 个 short 类型
```

}

由于地址线和数据线均为 16 位，故每页中单位大小设置为 16 位，采用 short 类型进行描述。采用统一的页大小设计后，在进行页面调度时可以只关注调度算法，避免数据类型的影响。

5.2 PCB 设计

PCB 类中定义了如下变量，用以描述进程控制块的信息：

```
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，page_table[i][0]为进程的页号，从 0 开始编号；page_table[i][1]为对应的物理页号
private ArrayList<Integer> instructions=new ArrayList<Integer>();   //该进程所有的指令

private short in_page_num=0;       //该 PCB 所在的页号
private short pool_location=-1;          //该 PCB 在 PCB 池的位置
private int total_runtime=0;          //总共运行的时间
public int ins_runtime=0;             //指令的执行总时间
public boolean if_in_p=false;         //是否处于 P 状态
public boolean if_p_success=true; //P 是否成功
```

分别设置了用来读取和修改数据成员的接口。

5.3 JCB 设计

```
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
private short in_page_num=0;       //该 JCB 所在的页号
```

分别设置了用来读取和修改数据成员的接口。

5.4 系统全局变量

kernel 类中定义了系统环境中的一些常量的值，以及一些变量不同值所对应的状态描述

```
/*系统基本信息*/
    public static int SINGLE_PAGE_SIZE=512;    //每一页/块的大小
    public static int MEMORY_SIZE=32*1024;    //内存大小，32KB
    public static int MEMORY_KERNEL_SPACE_SIZE=16*1024; //内存内核空间大小，16KB
    public static int MEMORY_KERNEL_CORESTACKANDOSKERNEL_SIZE=512; //核心栈+系统内核大小，1 页
    public static int MEMORY_KERNEL_PCB_POOL_SIZE=31*512;  //PCB 池大小，31 页
    public static int MEMORY_USER_SPACE_SIZE=16*1024;//内存用户空间大小（页表、页框使用），16KB
    public static int HARDDISK_SIZE=1*1024*1024; //硬盘空间大小，1MB
    public static int HARDDISK_VIRTUAL_MEMORY_SIZE=64*1024;    //虚存区大小，128 页，64KB
    public static int HARDDISK_SYSTEMFILE_SIZE=16*1024;  //系统文件大小，32 页，16KB
    public static int HARDDISK_FILE_SPACE_SIZE=944*1024;  //文件区大小，1888 页，944KB
    public static int HARDDISK_CYLINDER_NUM=32;  //磁盘磁道数
    public static int HARDDISK_SECTOR_NUM=64; //磁盘扇区数
    public static int HARDDISK_PAGE_SIZE=512;    //磁盘每页/块大小
    public static int SINGLE_INSTRUCTION_SIZE=8;    //单条指令的大小
    public static int INSTRUCTIONS_PER_PAGE=SINGLE_PAGE_SIZE/SINGLE_INSTRUCTION_SIZE;    //每一页的指令数目

    public static int INTERRUPTION_INTERVAL=10;    //系统发生中断的间隔
    public static int SYSTEM_TIME=0;        //系统内时间
    public static void SystemTimeAdd() {kernel.SYSTEM_TIME+=kernel.INTERRUPTION_INTERVAL;} //系统时间自增
    public static int TLB_LENGTH=kernel.MEMORY_USER_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE/2;  //TLB 快表的长度，16
    /*系统基本信息*/

    /*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*/

    /*Process PSW  程序状态字*/
    public final static byte PSW_KERNEL_STATE=0;  //管态
    public final static byte PSW_USER_STATE=1;            //目态
    /*Process PSW  程序状态字*/

    /*硬件初始化需要变量*/
    public static String MEMORYFILE_PATHNAME="./input/memory.dat";
    //内存文件地址
    public static String HARDDISKFILE_PATHNAME="./input/harddisk.dat";
    //硬盘文件地址
    public static String CPUFILE_PATHNAME="./input/cpu.dat";
    //CPU 文件地址
    /*硬件初始化需要变量*/

    /*60 条指令*/
    public static int GetInstructionType(int instruction){
        //获取指令类型
        if(instruction>=0&&instruction<=9)return 1;
        if(instruction>=10&&instruction<=19)return 2;
        if(instruction>=20&&instruction<=29)return 3;
        if(instruction>=30&&instruction<=39)return 4;
        if(instruction>=40&&instruction<=49)return 5;
        if(instruction>=50&&instruction<=59)return 6;
        return -1;}
    public static int GetInstructionTime(int instruction)
    {/*获取指令执行所需要时间*/return ((int)(instruction%10))*10+20;}
    public static int[] MUTEX= {-4,-3,-2,-1,0,1,2,3,4,5};          //临界区信号
量,10 个
    public static int[] SYSTEM_RESOURCE= {0,1,2,3,4,5,6,7,8,9};//系统资源
量，10 个
    public static int GetUseResourceNum(int instruction)
    {/*获取该指令所申请、释放的 PV、资源所在的数组序号*/return
instruction%10;}
    /*60 条指令*/
}
```

6 模块设计说明

6.1 作业管理

作业管理模块是基于 CPU 与硬盘硬件的系统模块之一。该模块的功能是为

作业的创建、存入、删除以及作业的调入检测提供相关的支持。

作业管理模块在 java 工程中写在 JobModule.java 文件中，同时该类被 Control.java 调用。

与作业管理有关的操作有：

```
public short GetJobNum()
    //获得当前磁盘中的作业总数量

public int GetCurrentCreateJobID()
    //获取当前要创建的作业的编号

public void SaveJobToHardDisk(JCB jcb)
    //将作业保存到外存，d1是JCB数据块，d2是包含所有指令的ArrayList，每条指令占8字节

public void GetJCBFromFile(File f)
    //从文件中读取所有的JCB

public ArrayList<JCB> GetAllJCB()
    //获取所有的JCB

public void NextJob()
    //已经读取完一个作业，进入到下一个作业

public int GetNextJobNum()
    //获取下一个将要被创建的JCB的序号

public boolean IsAllJobToProcess()
    //检测是否还有作业没有变成进程

public void RefreshJobList()
    //刷新作业后备队列

public boolean IsJobListEmpty()
    //查看作业后备队列是否为空
}
```

6.2 进程管理

进程管理模块是基于 CPU 与内存的系统模块之一。该模块的功能是为作业调入后转换成的进程提供管理功能。

进程管理是整个系统设计中最为复杂的部分，其包括低级调度管理、中级调度管理、高级调度管理、进程链表等功能。

作业管理模块在 java 工程中写在 ProcessModule.java 文件中，同时该类被 Control.java 调用。

与进程管理有关的操作有：

ProcessModule()//构造函数

public PCB TurnJCBToPCB(JCB jcb)
//将JCB变换为PCB

public void TransferJobCodeToSwapArea(JCB jcb,String apply)
//将指定作业的程序段存入虚存中
//jcb为作业控制块，apply为申请到的虚存空间分配字符串

public void WriteProcessPageTable(PCB pcb,String apply)
//将申请到的虚存页面写入到进程的页表中

public short GetFreePCBNumInPool()
//获取PCB池中可用的PCB数量

public void DeletePCBInPool(PCB pcb)
//将某一个PCB从PCB池中删除

public short ApplyOnePCBInPool()
//在PCB池中申请一个PCB

public void AddToPCBPool(PCB pcb)
//将PCB加入到PCB池中

public void TransferProcessToRunningQueue(PCB pcb)
//将进程移入运行队列
//遍历就绪队列、等待队列、挂起队列，将进程移出，只加入到运行队列
//移入运行队列

public void TransferProcessToReadyQueue(PCB pcb,boolean if_active)
//将进程移入就绪队列
//遍历运行队列、等待队列、挂起队列，将进程移出，只加入到就绪队列
//移入就绪队列

public void TransferProcessToWaitQueue(PCB pcb)
//将进程移入等待队列
//遍历运行队列、就绪队列、挂起队列，将进程移出，只加入到等待队列
//移入等待队列

public void TransferProcessToSuspendQueue(PCB pcb)
//将进程移入挂起队列
//遍历运行队列、就绪队列、等待队列，将进程移出，只加入到挂起队列
//移入挂起队列

```
public void TransferProcessToEndQueue(PCB pcb)
    //将进程移入完成队列
    //遍历运行队列、就绪队列、等待队列、挂起队列，将进程移出，加入到完成队列
    //移入完成队列

public void RefreshReadyQueueBitmap()
    //刷新就绪队列的bitmap

public void RefreshActiveExpired()
    //刷新active和expired指针

public boolean IfPageInMemory(short page_num)
    //检测需要的页是否在内存中
    //检测是否发生缺页中断

public void ChangePageTable(short ori_page_num,short changed_page_num)
    //将持有原来页的进程的页表更新

public void SolveMissingPage(PCB pcb,short need_page_num)
    //缺页中断的处理，need_page_num为需要的在外存中的页的页号

public boolean IfRunOver(PCB pcb)
    //检测某进程是否运行完毕

public boolean IfTimeSliceOver(PCB pcb)
    //检测时间片是否用完

public void AddToEndQueue(PCB pcb)
    //将作业加入到结束队列

public boolean IsRunningQueueEmpty()
    //检测运行队列是否为空

public boolean IsReadyQueueEmpty()
    //检测就绪队列是否为空

public boolean IsWaitQueueEmpty()
    //检测等待队列是否为空

public boolean IsSuspendQueueEmpty()
    //检测挂起队列是否为空

public int GetActivePoint()
```

```
        //获取active指针

    public int GetExpiredPoint()
        //获取expired指针

    public PCB GetPCBWithID(short id)
        //根据进程ID获取PCB

    public void RunType1(PCB pcb)
        //类型1指令的处理

    public void RunType2(PCB pcb)
        //类型2指令的处理

    public void RunType3(PCB pcb)
        //指令类型3的处理

    public void RunType4(PCB pcb)
        //指令类型4的处理

    public void RunType5(PCB pcb)
        //指令类型5的处理

    public void RunType6(PCB pcb)
        //类型6指令的处理
    }
```

6.3 页面管理

页面管理是负责系统中对于页面存入、读取、对换、换入换出功能的管理模块。在页面管理中，该模块与其他模块的信息交换全部都通过 Page 类来进行。当有页面换入、换出请求发出时，该模块先将页面信息写入 Page 类的对象，再将该对象传出，当其他模块收到该对象时，也可以对该对象进行操作，以减少操作的复杂度。

作业管理模块在 java 工程中写在 PageModule.java 文件中，同时该类被 Control.java 调用。

与页面管理相关的操作有：

```
PageModule()//构造函数
{
    InitSwapAreaUsage();   //初始化交换区页面被占用状态
    for(int i=0;i<6;i++) //实例化伙伴算法的空闲链表
        this.free_area[i]=new ArrayList<Short>();
    InitFreeAreaList();        //初始化空闲链表
    RefreshBitmap();           //刷新伙伴算法的Bitmap
}
```

```
private void InitSwapAreaUsage()
    //初始化交换区页的使用情况

private void InitFreeAreaList()
    //初始化空闲链表

public void RefreshBitmap()
    //刷新bitmap

private int SetOneBit(int num,int loca,int bit)
    //修改bitmap中的某一位信息

private int GetOneBit(int num,int loca)
    //获得bitmap中的某一位信息

public int GetFreePageNumInMemory()
    //返回当前物理内存中可用的页框数

public int GetFreePageNumInDisk()
    //返回当前虚存中可用的页框数
public Page GetPage(short num)
    //获得某一个页面

public boolean IfCouldApplyPageInDisk(short num)
    //检测在虚存中是否可以申请num个页面

public int CalculateCloest2Num(short num)
    //计算出与该数字最接近的2的次幂数的幂

private void SetBlockState(int list,int no,int set_num,int state)
    //在伙伴算法的链表中，在第list级别的第no块设置连续的set_num块为state状态

private void RefreshBlockList()
    //从底往上刷新块链表

public String ApplyPageInMemory(short num)
    //在内存中，向伙伴算法申请num个页面

public String ApplyPageInDisk(short page_num)
    //在虚存中申请page_num个页面，返回一个String类型的值
    //String格式的数据说明：从左到右编号为0-191，共192位，每一位的值为0/1，1代表该页面分配给该进程使用。在写入程序区时，必须按照分配的页面顺序从小到大写入
```

```
    //同时，在记录数组中记录这些申请的页框，将他们设置为已用状态

public void FreePageInMemory(short page_num)
    //在内存中，利用伙伴算法释放某一页

public void FreePageInDisk(short page_num)
    //在外存中，释放某一页

public void RecyclePage(short page_num)
    //回收页面，参数num为页面号（num从0开始编号）
    //将该页的内容全部清空，并在记录中使得该页表示为未被占用

public void MoveToMemory(short memory_page_num,short swap_page_num)
    //将指定的虚存页移动到指定的内存页中

public void MoveToDisk(short memory_page_num,short swap_page_num)
    //将指定的内存页移动到指定的虚存页中

public void ExchangePage(short memory_page_num,short swap_page_num)
    //交换两个页面的内容
public void CopyPage(short src_page_num,short des_page_num)
    //将序号src_page_num的页面复制到序号为des_page_num的页面中

public short GetOneFreePageInMemory()
    //在物理内存中找到一个空闲的页面，并返回该页面序号

public short GetOneFreePageInDisk()
    //在虚存中找到一个空闲的页面，并返回该页面序号

public void LRUVisitOnePage(int page_num)
    //LRU访问某一页

public short LRUGetLastPageNum()
    //获得应该调出的页面号

public boolean isBlockUsing(int i,int j)
    //获取内存中某一个页的使用情况

public boolean isPageUsing(int i)
    //获取虚存中某一页的使用情况
}
```

6.4 调度算法

调度算法是实现并发环境模拟的核心功能模块之一，分为高级调度、中级调

度和低级调度三个层次，从不同方面保证了系统的稳定高效运行，为多任务并发执行创造条件。

```
public void run()
    //线程执行函数，负责进行调度

public void UIRefresh()
    //不同UI的刷新

public void SuspendProcessWithPageNum(short num)
    //检测某一页所关联的进程，并将该进程加入到挂起态

public void HighLevelScheduling()
    //高级调度

public void MiddleLevelScheduling()
    //中级调度

public void LowLevelScheduling()
    //低级调度
}
```

6.5 界面模块

在程序执行之后，采用不同界面对系统的各个方面的信息进行展示，在监测系统状态的同时查看模拟系统的运行情况。

界面模块共分为 2 个部分，在 MainUI 中展示总体情况，并依托 MainUI 打开其他 UI 界面查看各 UI 所展示的信息。

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] Linux 的内存管理[EB/OL].
https://www.cnblogs.com/xelatex/p/3491301.html

[4] Linux2.6 内核进程调度分析[EB/OL].
https://blog.csdn.net/dlutbrucezhang/article/details/8694793

[5] Linux2.6 内核--进程调度理论[EB/OL].
https://www.cnblogs.com/joey-hua/p/5707780.html

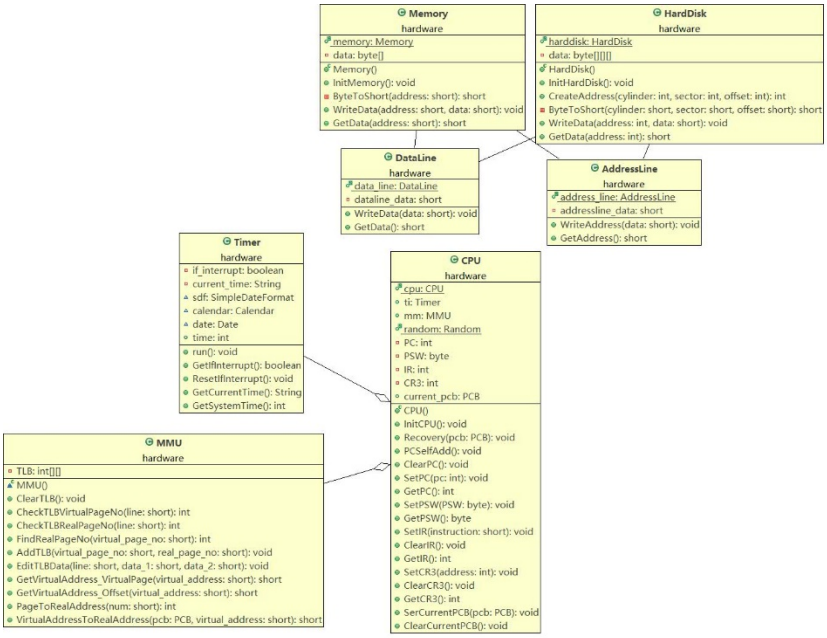
[6] Linux 2.6 调度系统分析[EB/OL].
https://blog.csdn.net/hzrandd/article/details/51034488

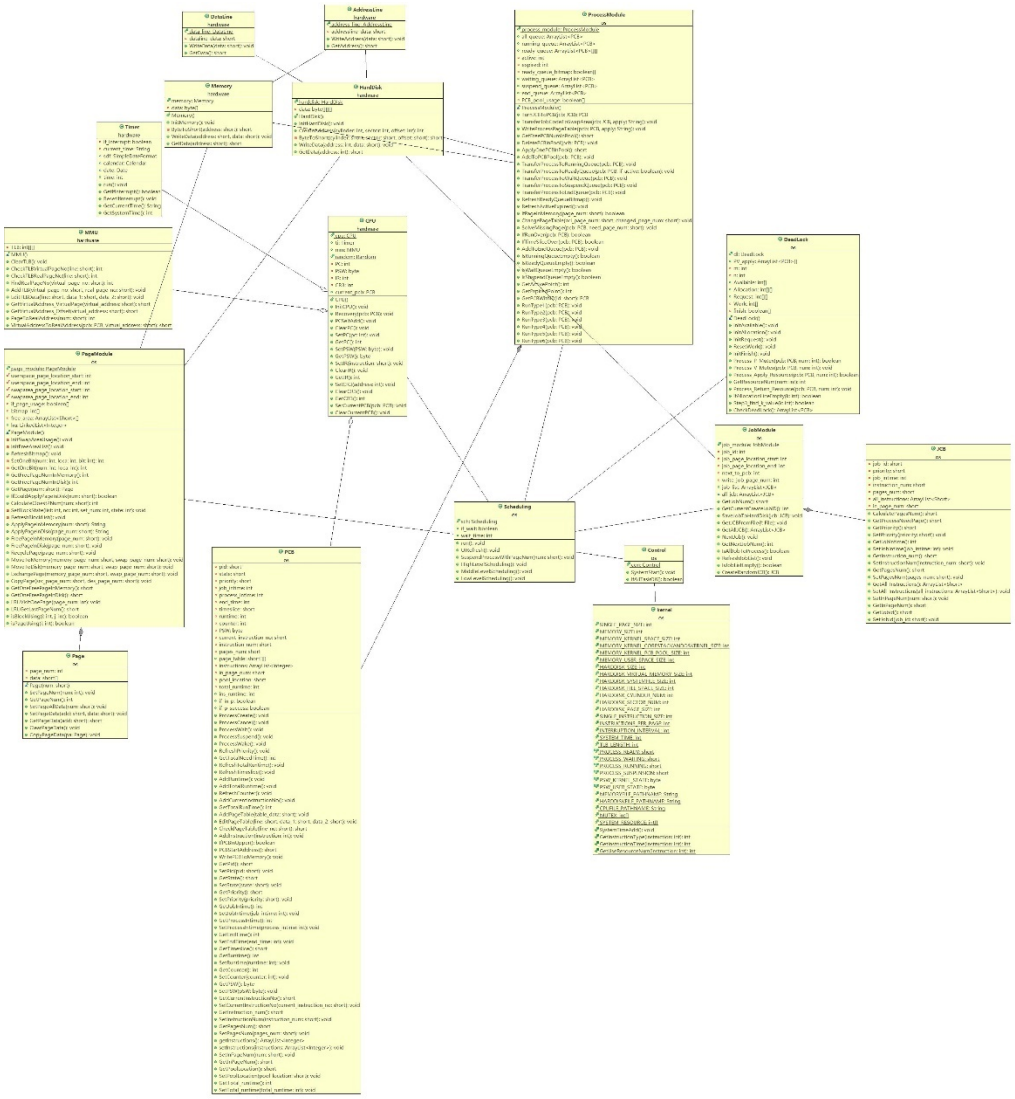
[7] Linux 内核中几个比较有意思的解释(进程调度算法，页面调度算法，非线性工作集)[EB/OL].
https://blog.51cto.com/dog250/1698404

附件 1：程序文件及结构说明

```
│    .classpath
│    .project
│
│
├──input
│        cpu.dat
│        harddisk.dat
│        memory.dat
│
└──src
    │    Main.java
    │
    ├──hardware
    │        AddressLine.java
    │        CPU.java
    │        DataLine.java
    │        HardDisk.java
    │        Memory.java
    │        MMU.java
    │        Timer.java
    │
    ├──os
    │        Control.java
    │        DeadLock.java
    │        JCB.java
    │        JobModule.java
    │        kernel.java
    │        Page.java
    │        PageModule.java
    │        PCB.java
    │        ProcessModule.java
    │        Scheduling.java
    │
    └──ui
            CPUInfoUI.java
            CreateJobUI.java
            HardDiskUI.java
            MainUI.java
            MemoryUI.java
            PageModuleUI.java
```

附件 2：类图说明


Memory
hardware
memory: Memory
data: byte[]
Memory()
InitMemory(): void
ByteToShort(address: short): short
WriteData(address: short, data: short): void
GetData(address: short): short
HardDisk
hardware
harddisk: HardDisk
data: byte[][][][]
HardDisk()
InitHardDisk(): void
CreateAddress(cylinder: int, sector: int, offset: int): int
ByteToShort(cylinder: short, sector: short, offset: short): short
WriteData(address: int, data: short): void
GetData(address: int): short
DataLine
hardware
data_line: DataLine
dataline_data: short
WriteData(data: short): void
GetData(): short
AddressLine
hardware
address_line: AddressLine
addressline_data: short
WriteAddress(data: short): void
GetAddress(): short
Timer
hardware
if_interrupt: boolean
current_time: String
sdf: SimpleDateFormat
calendar: Calendar
date: Date
time: int
run(): void
GetIfInterrupt(): boolean
ResetIfInterrupt(): void
GetCurrentTime(): String
GetSystemTime(): int
CPU
hardware
cpu: CPU
ti: Timer
mm: MMU
random: Random
PC: int
PSW: byte
IR: int
CR3: int
current_pcb: PCB
CPU()
InitCPU(): void
Recovery(pcb: PCB): void
PCSelfAdd(): void
ClearPC(): void
SetPC(pc: int): void
GetPC(): int
SetPSW(PSW: byte): void
GetPSW(): byte
SetIR(instruction: short): void
ClearIR(): void
GetIR(): int
SetCR3(address: int): void
ClearCR3(): void
GetCR3(): int
SerCurrentPCB(pcb: PCB): void
ClearCurrentPCB(): void
MMU
hardware
TLB: int[][]
MMU()
ClearTLB(): void
CheckTLBVirtualPageNo(line: short): int
CheckTLBRealPageNo(line: short): int
FindRealPageNo(virtual_page_no: short): int
AddTLB(virtual_page_no: short, real_page_no: short): void
EditTLBData(line: short, data_1: short, data_2: short): void
GetVirtualAddress_VirtualPage(virtual_address: short): short
GetVirtualAddress_Offset(virtual_address: short): short
PageToRealAddress(num: short): int
VirtualAddressToRealAddress(pcb: PCB, virtual_address: short): short



南京農業大學

计算机操作系统课程设计

实践报告

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

可视化仿真实现 Linux2.6 进程管理与内存管理

1 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。

1.1 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。

1.2 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。

1.3 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。

1.4 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。

2 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。

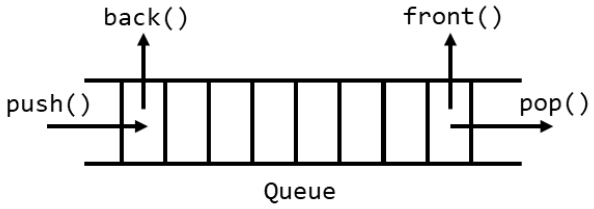
下文将对这些功能的实现原理进行深度阐述。

2.1 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

2.2 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

2.3 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
12、检测时间片是否用完，如果完毕，则进入就绪队列

2.4 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

2.5 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。

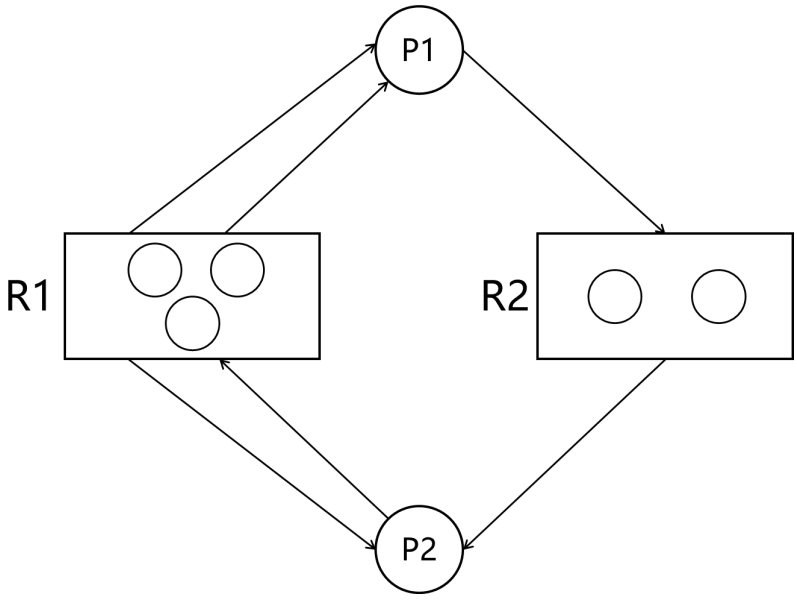
③资源不可剥夺

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。

2.6 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];
    }
```

2.7 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;
}
```

2.8 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
    }
```

2.9 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
    }
```

3 功能测试

3.1 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：

```
System.out.println(kernel.SINGLE_PAGE_SIZE);
System.out.println(kernel.MEMORY_SIZE);
System.out.println(kernel.MEMORY_KERNEL_SPACE_SIZE);
System.out.println(kernel.MEMORY_KERNEL_CORESTACKANDOSKERNEL_SIZE);
System.out.println(kernel.MEMORY_KERNEL_PCB_POOL_SIZE);
System.out.println(kernel.MEMORY_USER_SPACE_SIZE);
System.out.println(kernel.HARDDISK_SIZE);
System.out.println(kernel.HARDDISK_VIRTUAL_MEMORY_SIZE);
```

输出的结果如下：

```
<terminated> M
16384
512
15872
16384
1048576
65536
```

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。

3.2 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：

```
PCB t=new PCB();
t.SetPid((short) 100);
System.out.println(t.GetPid());
t.SetCounter(50);
System.out.println(t.GetCounter());
t.SetCurrentInstructionNo((short) 10);
System.out.println(t.GetCurrentInstructionNo());
```

输出的结果如下：

```
<terminated> M
100
50
10
```

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。

3.3 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：

```
JCB t=new JCB();
t.SetInPageNum((short) 10);
System.out.println(t.GetInPageNum());
t.SetInstructionNum((short) 100);
System.out.println(t.GetInstruction_num());
t.SetJobid((short) 2333);
System.out.println(t.GetJobid());
```

测试结果如下：

```
<terminated
10
100
2333
```

可见，测试结果正确。

3.4 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：

```
PCB t1=new PCB();
PCB t2=new PCB();
t1.SetPid((short) 1);
t2.SetPid((short) 2);
DeadLock.dl.Process_Apply_Resource(t1,1);
DeadLock.dl.Process_Apply_Resource(t2,2);
DeadLock.dl.Process_Apply_Resource(t2,2);
DeadLock.dl.Process_Apply_Resource(t2,1);
DeadLock.dl.Process_Apply_Resource(t1,2);
ArrayList<PCB> result=DeadLock.dl.CheckDeadLock();
if(result.size()!=0)
    System.out.println("检测到死锁");
else
    System.out.println("未发生死锁");
```

执行结果为：

```
<terminated> Mai
检测到死锁
```

可见，运行结果正确！

3.5 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

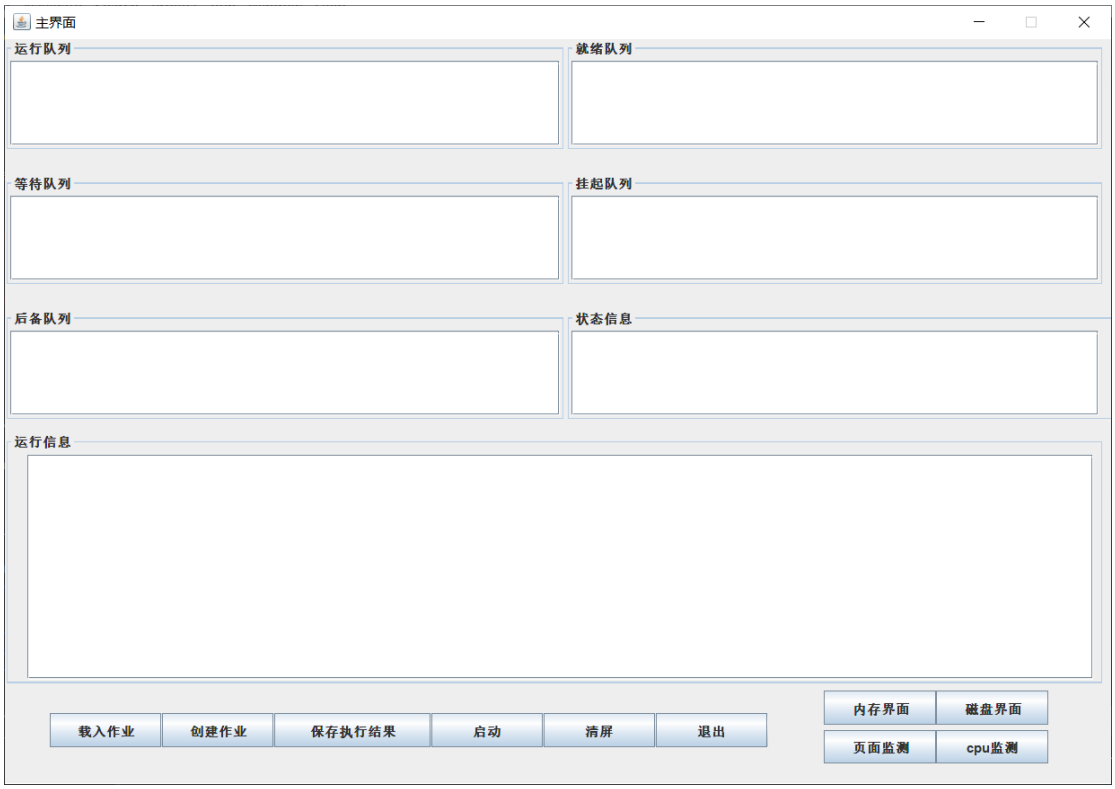
的含义，建议通过 Eclipse 自带的单步调试功能进行测试。

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：

```
public class Main {
    public static void main(String[] args)
    {
        MainUI.main_ui.SetVisible(true);
    }
}
```

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

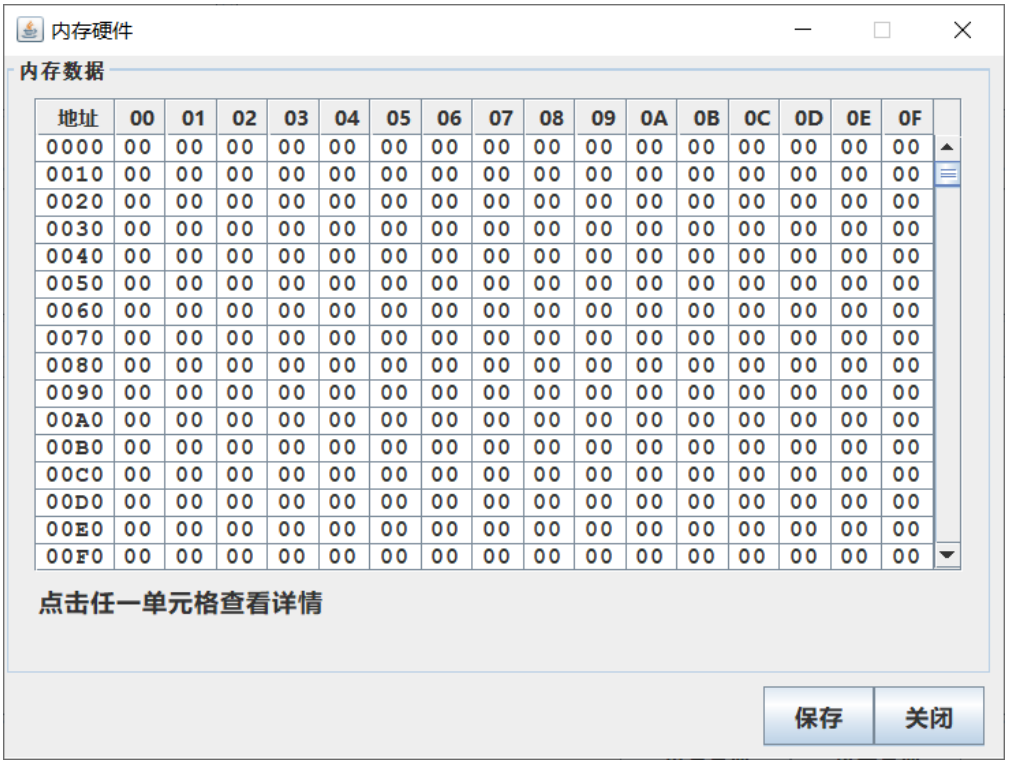
磁盘界面：打开磁盘数据查看界面，可以查看当前磁盘中所有的数据信息

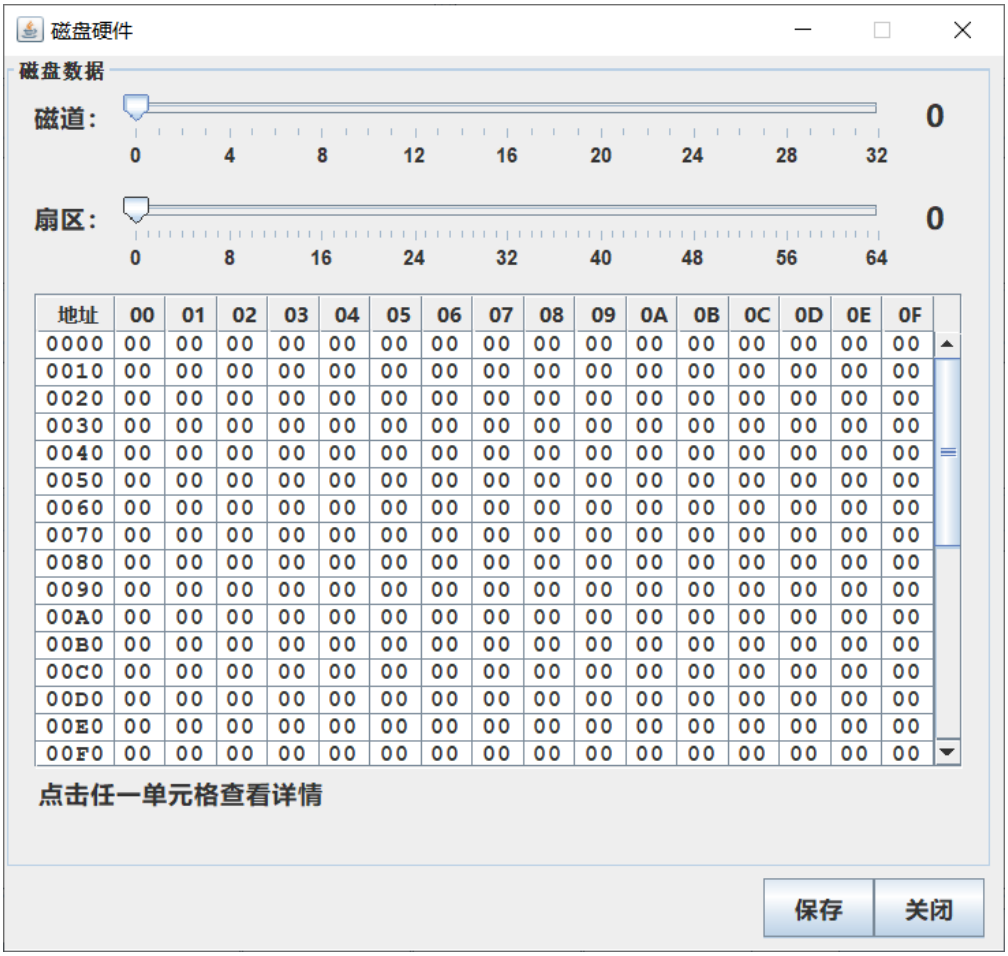
页面检测：打开页面管理的检测界面，可以查看内存、虚存、磁盘的空间占

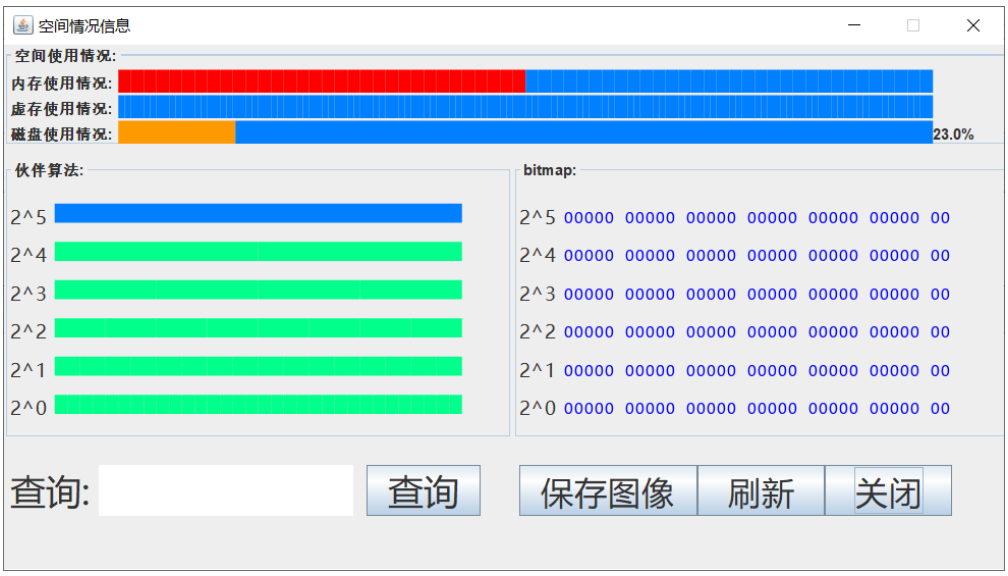
用情况和伙伴算法的运行原理和 bitmap

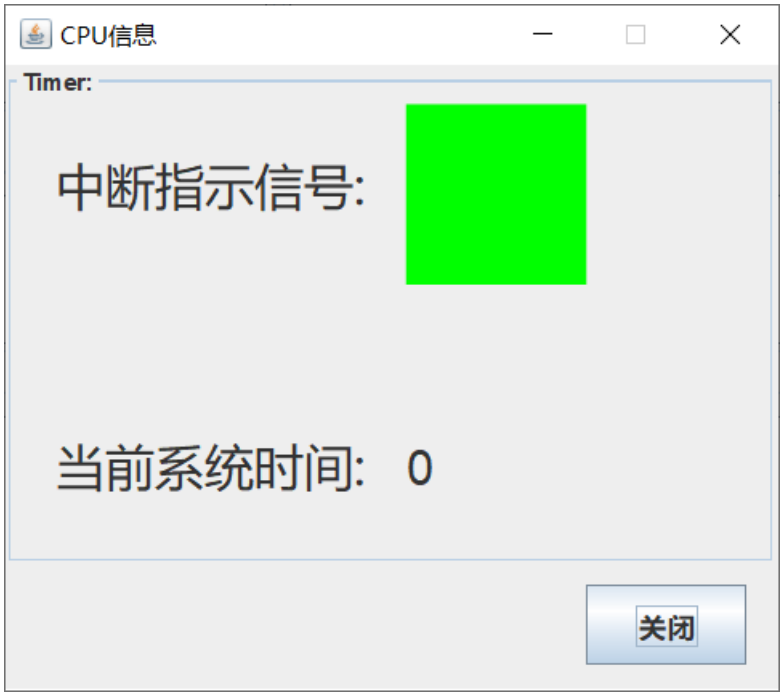
CPU 检测：打开 CPU 检测窗体，可以检测此时 CPU 的具体中断信息与当前系统时间。

这些窗体的截图如下：




磁盘硬件
磁盘数据
磁道:
0
0 4 8 12 16 20 24 28 32
扇区:
0
0 8 16 24 32 40 48 56 64
地址 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
点击任一单元格查看详情
保存
关闭


空间情况信息
空间使用情况:
内存使用情况:
虚存使用情况:
磁盘使用情况:
23.0%
伙伴算法:
2^5
2^4
2^3
2^2
2^1
2^0
bitmap:
2^5 00000 00000 00000 00000 00000 00000 00
2^4 00000 00000 00000 00000 00000 00000 00
2^3 00000 00000 00000 00000 00000 00000 00
2^2 00000 00000 00000 00000 00000 00000 00
2^1 00000 00000 00000 00000 00000 00000 00
2^0 00000 00000 00000 00000 00000 00000 00
查询:
查询
保存图像
刷新
关闭



3.6 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：

系统时间：0高级调度--检测后备队列是否有作业--后备队列有作业--作业序号：1等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB转换为PCB--在虚存中申请空间--作业载入--写入页表--写PCB池--进程加入到就绪队列，等待被调度

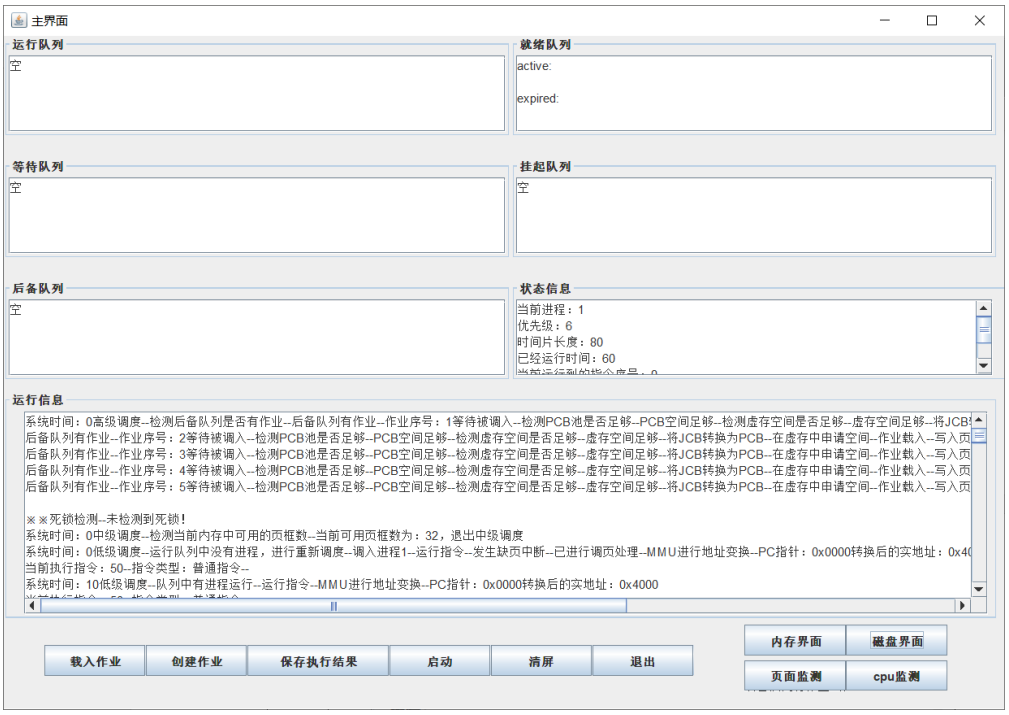
系统时间：2000中级调度--检测当前内存中可用的页框数--当前可用页框数为：27，退出中级调度

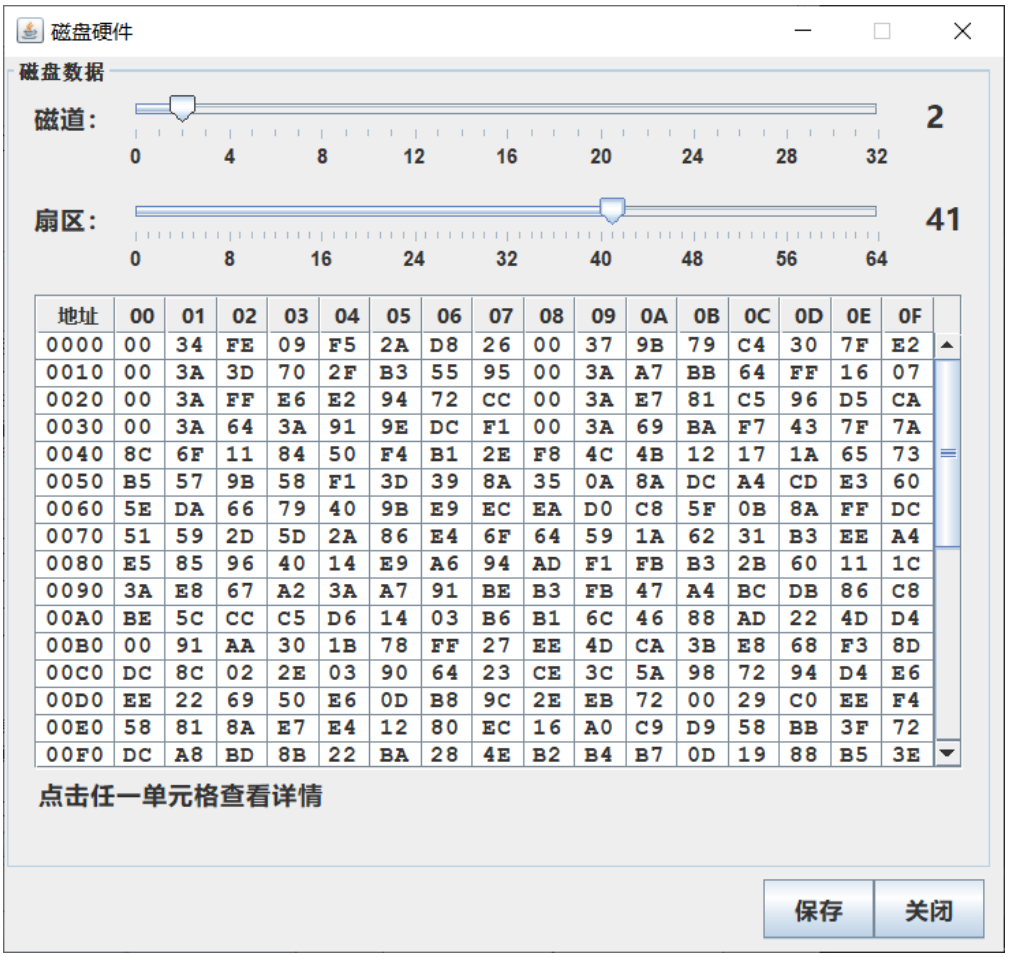
系统时间：2000低级调度--队列中有进程运行--运行指令--没有发生缺页中断--当前访问页表：0-34
MMU进行地址变换--PC指针：0x0018H，转换后的实地址：0x4418H
当前执行指令：58--指令类型：普通指令--PSW：用户态

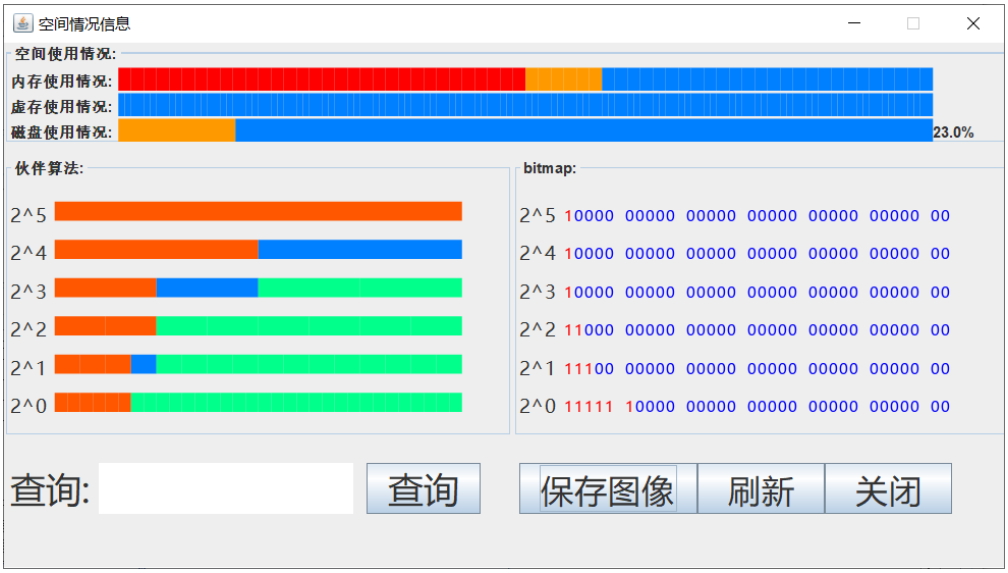
3.7 指令运行

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：


主界面
运行队列
空
就绪队列
active:
expired:
等待队列
空
挂起队列
空
后备队列
空
状态信息
当前进程：1
优先级：6
时间片长度：80
已经运行时间：60
运行信息
系统时间：0高级调度--检测后备队列是否有作业--后备队列有作业--作业序号：1等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB
后备队列有作业--作业序号：2等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB转换为PCB--在虚存中申请空间--作业载入--写入页
后备队列有作业--作业序号：3等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB转换为PCB--在虚存中申请空间--作业载入--写入页
后备队列有作业--作业序号：4等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB转换为PCB--在虚存中申请空间--作业载入--写入页
后备队列有作业--作业序号：5等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB转换为PCB--在虚存中申请空间--作业载入--写入页
※※死锁检测--未检测到死锁！
系统时间：0中级调度--检测当前内存中可用的页框数--当前可用页框数为：32，退出中级调度
系统时间：0低级调度--运行队列中没有进程，进行重新调度--调入进程1--运行指令--发生缺页中断--已进行调页处理--MMU进行地址变换--PC指针：0x0000转换后的实地址：0x4
当前执行指令：50--指令类型：普通指令--
系统时间：10低级调度--队列中有进程运行--运行指令--MMU进行地址变换--PC指针：0x0000转换后的实地址：0x4000
载入作业
创建作业
保存执行结果
启动
清屏
退出
内存界面
磁盘界面
页面监测
cpu监测


磁盘硬件
磁盘数据
磁道：
2
0 4 8 12 16 20 24 28 32
扇区：
41
0 8 16 24 32 40 48 56 64
地址 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000 00 34 FE 09 F5 2A D8 26 00 37 9B 79 C4 30 7F E2
0010 00 3A 3D 70 2F B3 55 95 00 3A A7 BB 64 FF 16 07
0020 00 3A FF E6 E2 94 72 CC 00 3A E7 81 C5 96 D5 CA
0030 00 3A 64 3A 91 9E DC F1 00 3A 69 BA F7 43 7F 7A
0040 8C 6F 11 84 50 F4 B1 2E F8 4C 4B 12 17 1A 65 73
0050 B5 57 9B 58 F1 3D 39 8A 35 0A 8A DC A4 CD E3 60
0060 5E DA 66 79 40 9B E9 EC EA D0 C8 5F 0B 8A FF DC
0070 51 59 2D 5D 2A 86 E4 6F 64 59 1A 62 31 B3 EE A4
0080 E5 85 96 40 14 E9 A6 94 AD F1 FB B3 2B 60 11 1C
0090 3A E8 67 A2 3A A7 91 BE B3 FB 47 A4 BC DB 86 C8
00A0 BE 5C CC C5 D6 14 03 B6 B1 6C 46 88 AD 22 4D D4
00B0 00 91 AA 30 1B 78 FF 27 EE 4D CA 3B E8 68 F3 8D
00C0 DC 8C 02 2E 03 90 64 23 CE 3C 5A 98 72 94 D4 E6
00D0 EE 22 69 50 E6 0D B8 9C 2E EB 72 00 29 C0 EE F4
00E0 58 81 8A E7 E4 12 80 EC 16 A0 C9 D9 58 BB 3F 72
00F0 DC A8 BD 8B 22 BA 28 4E B2 B4 B7 0D 19 88 B5 3E
点击任一单元格查看详情
保存
关闭

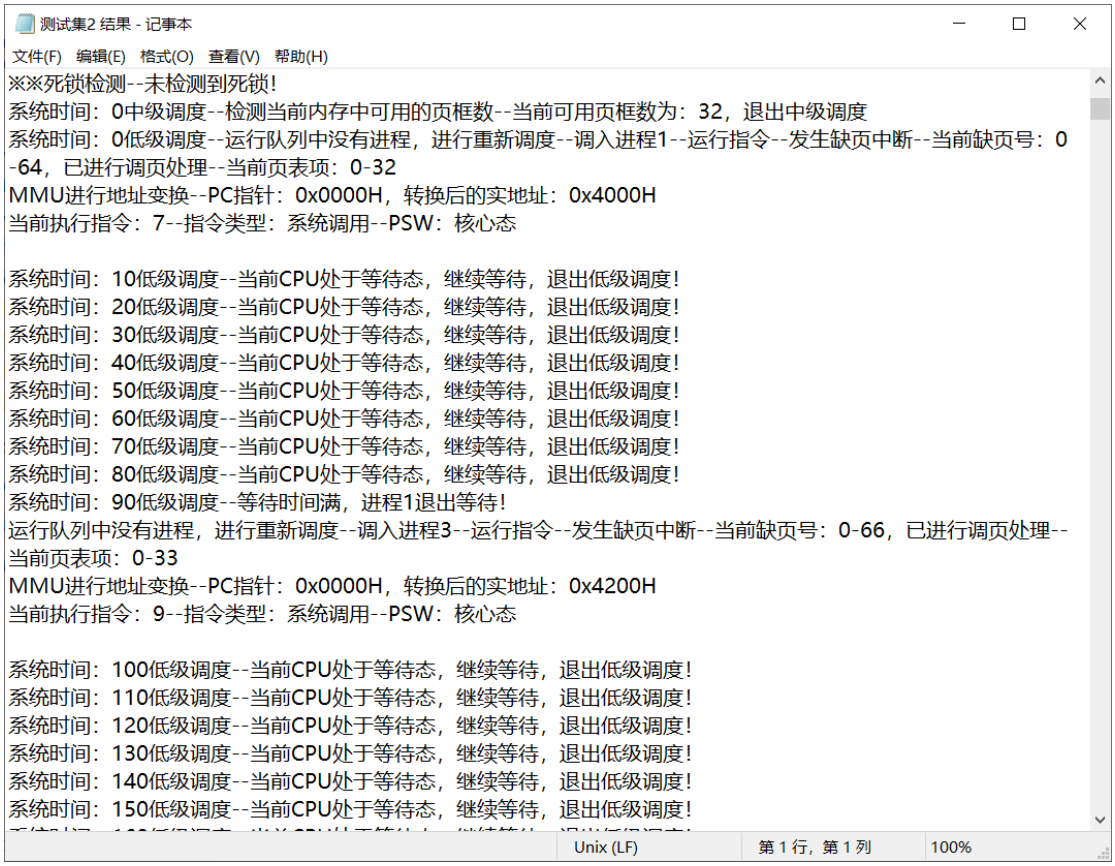


将程序的运行结果输出保存到文件，文件内容如下：

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。

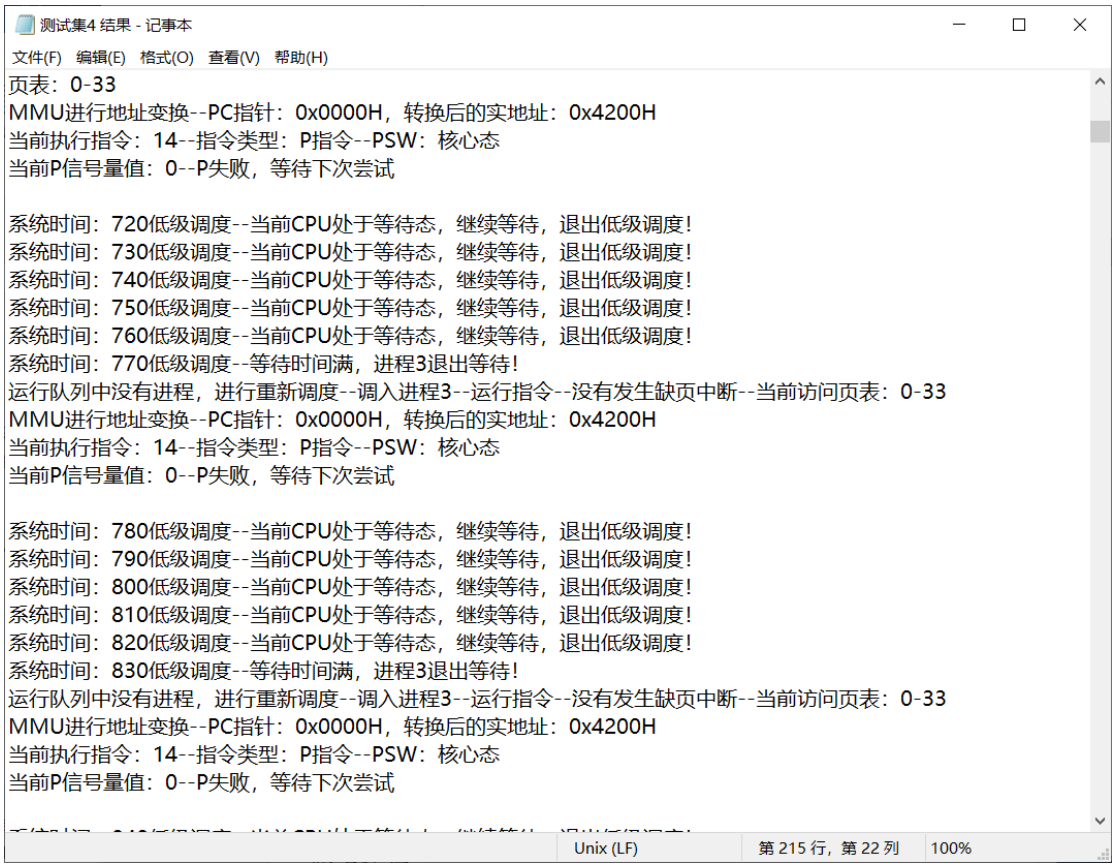
接着，开始测试系统调用的情况。

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：

```
系统时间：3480低级调度--当前CPU处于等待态，继续等待，退出低级调度!
系统时间：3490低级调度--当前CPU处于等待态，继续等待，退出低级调度!

※※死锁检测--检测到死锁！与死锁有关进程：
os.PCB@12cf638a，Process_ID：2

调用进程撤销原语，撤销进程!
系统时间：3500低级调度--运行队列中没有进程，进行重新调度--调入进程1--运行指令--没有发生缺页中断--当前访问页表：0-33
MMU进行地址变换--PC指针：0x0008H，转换后的实地址：0x4208H
当前执行指令：58--指令类型：普通指令--PSW：用户态
```

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。

4 技术问题分析报告

4.1 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。

4.2 队列指定元素删除

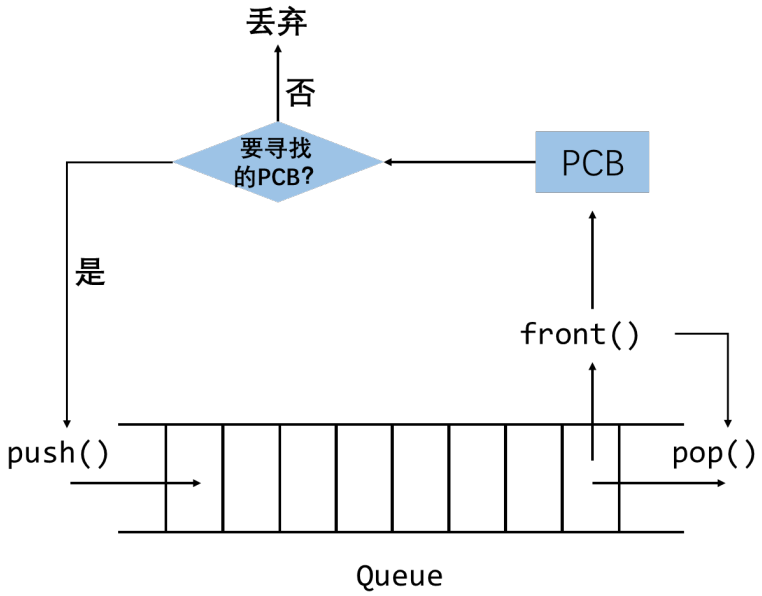
1. 问题描述

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。



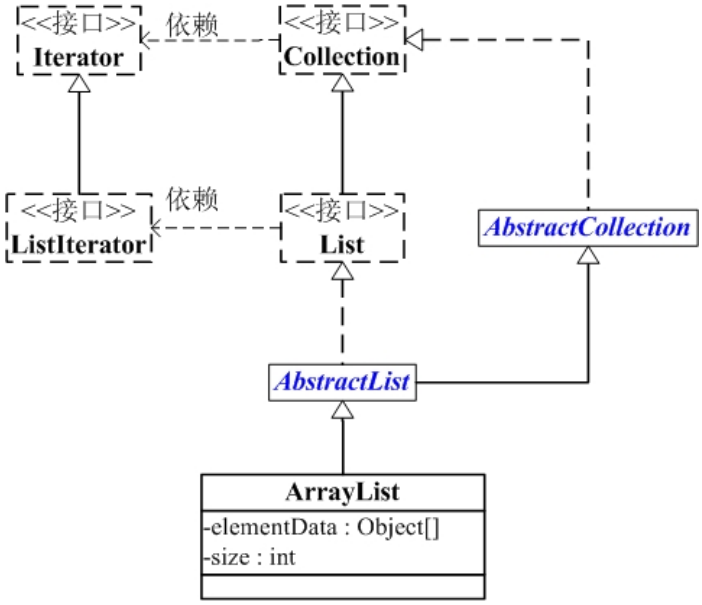
在后来的程序撰写中，又发现了一种 JAVA 常用的数据结构 ArrayList，ArrayList 是利用数组的形式模拟仿真链表。不仅可以体现出链表、队列的特性，还能够像数组一样在指定位置插入元素。

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。

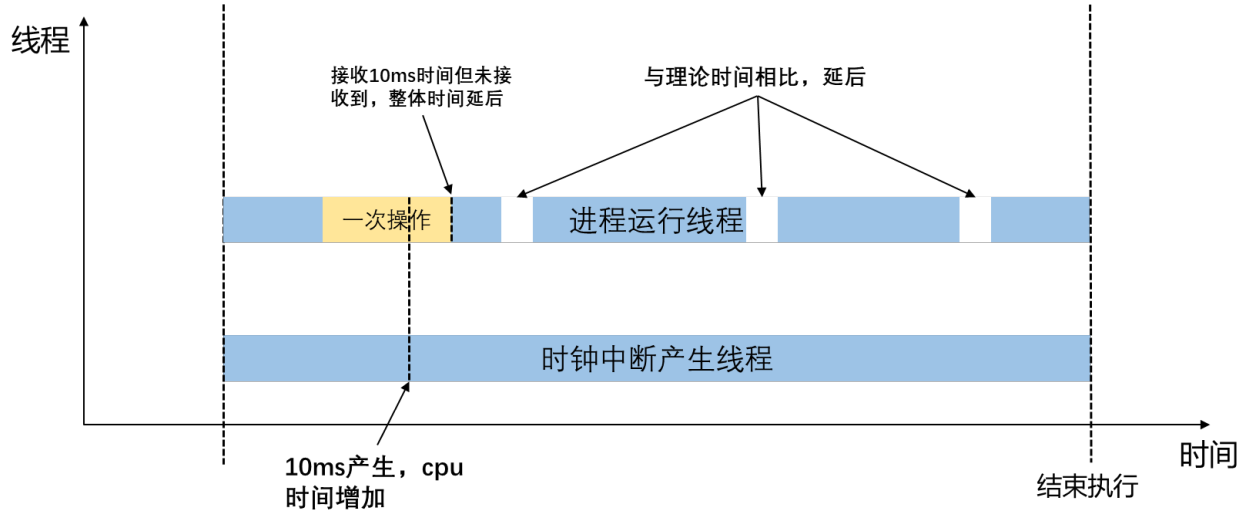
4.3 cpu 内部时间计数

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。



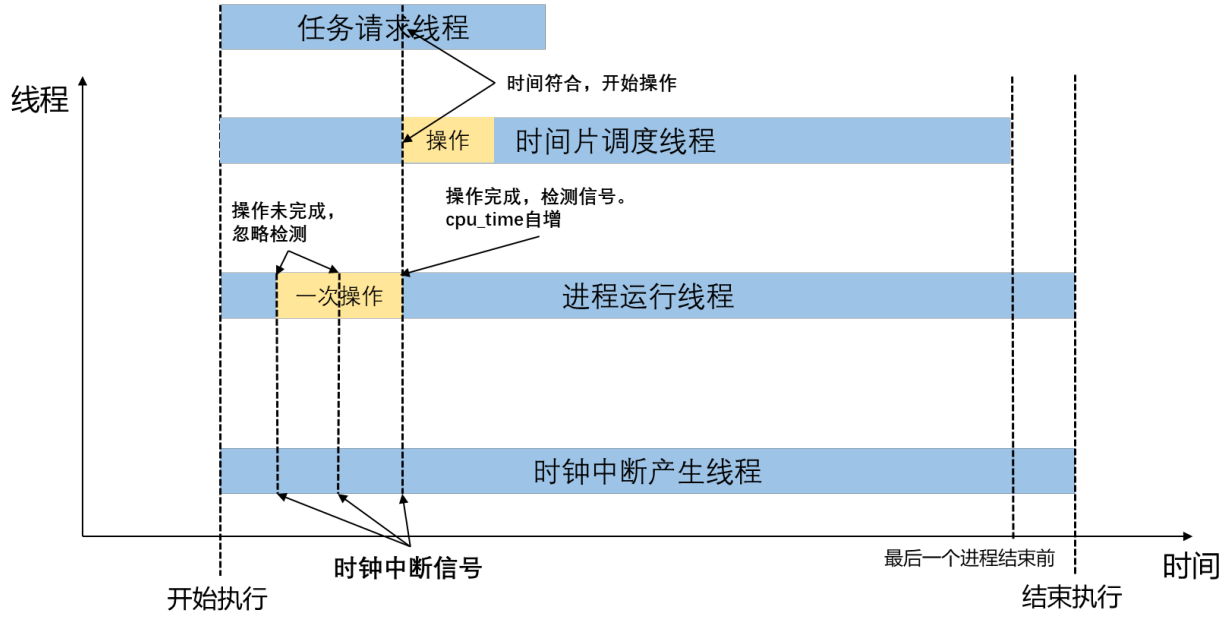
cpu 时间延后问题

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。



4.4 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。

5 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。

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

南京農業大學

计算机操作系统课程设计

实践报告

题　　目: Linux2.6 进程管理系统的仿真实现

姓　　名: 梁嘉文

学　　院: 信息科技学院

专　　业: 计算机科学技术系

班　　级: 计科 161

学　　号: 19216126

指导教师: 姜海燕　职 称: 教授

2019 年 3 月 15 日

目录

1 程序结构说明

1.1 程序结构图

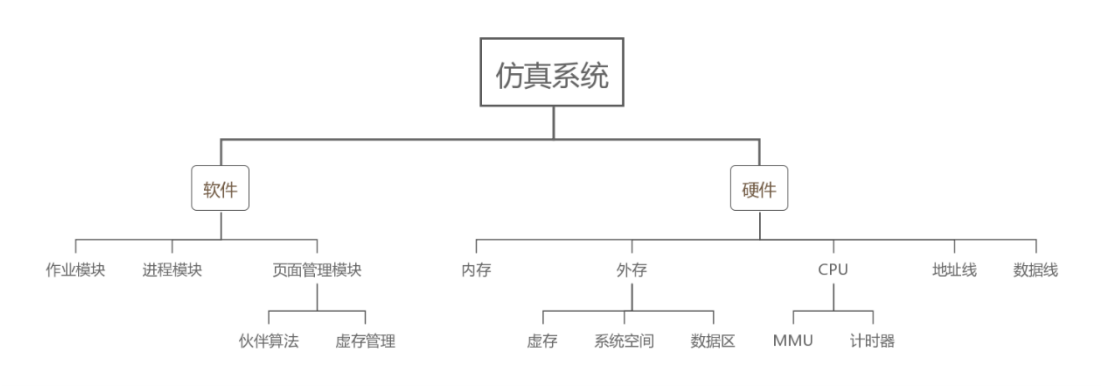


图 1 程序结构图

1.2 程序结构描述

在仿真系统的软件部分中，页面管理部分涉及到了对于内存的管理和对于虚存空间的管理。在讲所有空间进行分页之后，进行内存管理时，采用 linux2.6 所使用的伙伴算法；在管理虚存空间时，使用简单页式存储管理进行统一管理。

在硬件部分中，对内存、外存、cpu 三个部件分别建立文件作为仿真设备，系统中的类对象体现为文件的映射；对地址线和数据线两个部件只进行类对象的实例化，模拟系统中实际存在的硬件设备，但并没有对应的文件仿真。

在 CPU 类中设计 MMU 单元和计时器单元，负责模拟 CPU 中的地址变换机构和 CPU 内的计时器部件。

2 模块论述设计与实现

2.1 内存模块

内存作为操作系统中最核心的部件之一，需要频繁与 CPU 进行数据交互，频繁进行读写操作。而内存本身作为存储器，并没有任何可编程的能力和执行操作的能力。故在裸机设计中，对于内存模块的功能定位为空间有限的高速存储设备，在设计时着重体现其读写功能。

由于内存类的实例对象是内存文件的映射，故设置内存类对象在加载时将文件中所有数据导入。

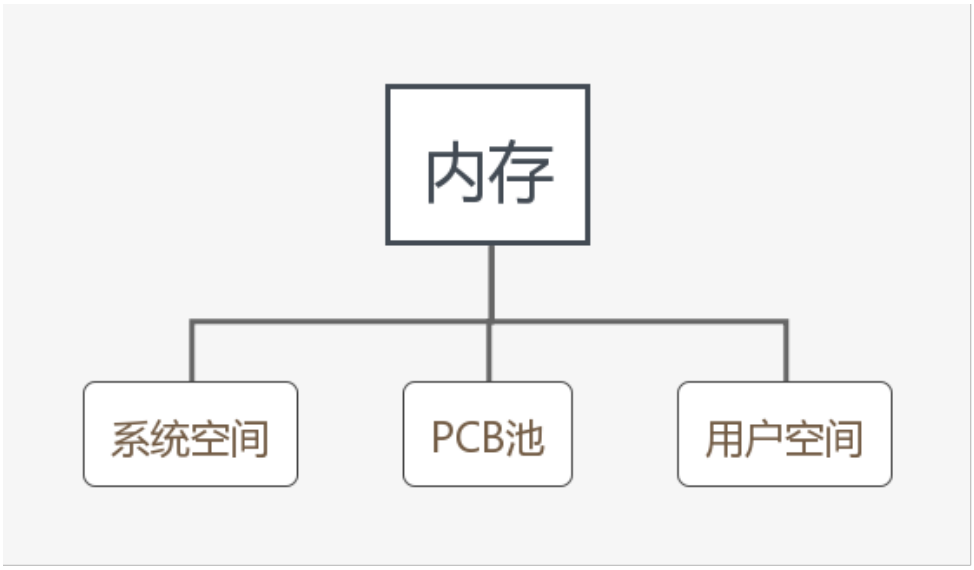


图 2 内存设计图

内存空间共 32KB，用户空间共 16KB，按照每 512B 一页的设计，占据页号 32-63，分配时使用伙伴算法，回收时以单个页面为单位进行回收。

2.1.1 读取内存数据

GetData(地址 address)
{
将 address 信息写入地址线 address_line
读取地址线信息 address_line 作为偏移地址在内存类的数据中进行寻址
将寻址到的内存单元中的数据组合成 short 型数据并写入数据线 data_line 中
读取 data_line 中的数据并作为返回值返回
}

2.1.2 写入内存数据

WriteData(地址 address，数据 data)
{
将 address 信息写入地址线 address_line
将 data 信息写入数据线 data_line 中
读取地址线信息 address_line 作为偏移地址在内存类的数据中进行寻址
向寻址到的内存单元中写入 data_line 中的数据，并将该 short 型数据拆分成两个字节分别存储
}

2.2 硬盘设计

硬盘作为计算机系统中存储数据的中药设备，需要存储大量数据信息，以及存储程序文件。但硬盘本身作为存储器，同样并没有可编程的能力和执行操作的能力。故在裸机设计中，对于硬盘的功能定位为空间足够的长期存储设备，在设计时需要体现其读写功能。

由于硬盘类的实例对象是硬盘文件的映射，故设置硬盘类对象在加载时将文件中所有数据导入。

由于硬盘空间大小超出 16 位地址线寻址空间，故对地址线采取分时复用的方式，分三次传输物理地址的偏移量、扇区号、磁道号，并按照特定方式进行组合，使三维地址与一维地址之间建立映射关系，既满足系统中对于地址统一管理的需要，又满足硬件条件的物理限制。

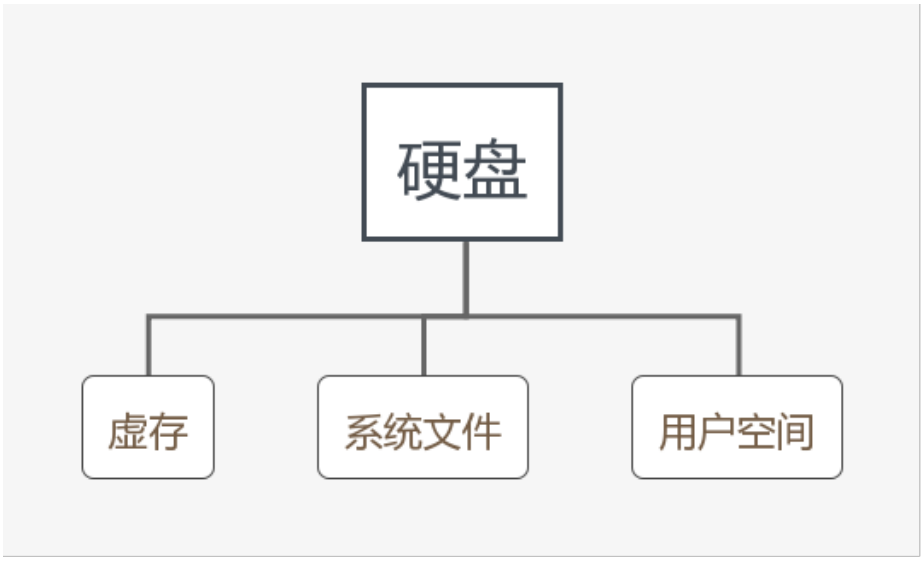


图 3 硬盘设计图

硬盘空间共 1MB，前 64KB 为硬盘与内存的交换区，即虚存空间，中间 16KB 存储系统文件，在开机时调入内存的系统区，其余空间均为用户空间，可存储作业信息和数据。在管理虚存空间时，对于虚存空间的申请以页为单位，在可用空间中分配足够数量的页后停止。释放时可对单独的页进行释放。

2.2.1 读取硬盘数据

GetData(地址 address)
{
将 address 信息写入地址线 address_line
读取地址线信息 address_line 作为偏移地址在硬盘类的数据中进行寻址
将寻址到的硬盘单元中的数据组合成 short 型数据并写入数据线 data_line 中
读取 data_line 中的数据并作为返回值返回
}

2.2.2 写入硬盘数据

WriteData(地址 address，数据 data)
{
将 address 信息写入地址线 address_line
将 data 信息写入数据线 data_line 中
读取地址线信息 address_line 作为偏移地址在硬盘类的数据中进行寻址
向寻址到的硬盘单元中写入 data_line 中的数据，并将该 short 型数据拆分成两个字节分别存储
}

2.3 cpu 设计

Cpu 作为系统的核心部件，包括寄存器堆、mmu、计时器等几个部件，用于支持系统的各个功能模块的工作，在各个功能模块中协调工作。

2.3.1 mmu 设计

Mmu 负责将虚拟地址转换为实际地址，由于所有空间均采用页式空间管理，故在 mmu 中的转换逻辑应为虚拟地址转换为页号，页号转换为页框号，再根据页框号找到对应的实际地址。同时在 mmu 中，设置 TLB 用于快速查询页号所对应页框号，提高 CPU 利用率。在查询页号对应地址时，若 TLB 命中则直接返回结果；若 TLB 未命中且内存命中，则将该块信息添加至 TLB 中，替换从 TLB 中删除 LRU 算法选择的块；若 TLB 与内存均为命中，则产生缺页中断，同时更新 TLB 与内存，均使用 LRU 算法选择被替换的单元。

在保证该页位于内存中时，可使用以下函数进行地址转换

```
VirtualAddressToRealAddress (pcb 块，虚拟地址 virtual_address)
{
提取虚拟地址的偏移地址
在 TLB 中查找 virtual_address 对应的页号
If(存在)  使用页号查询物理地址
Else
{
根据逻辑地址计算页号
将虚拟页号与物理页号信息添加至 TLB 被替换单元中
根据物理页号查询物理地址
}
获取物理地址作为基地址，加上偏移地址后作为实际的物理地址返回
}

PageToRealAddress(物理页号 page_num)
{
If(page_num 范围在内存中)根据页号计算物理地址
Else
{
根据页号计算物理地址
将物理地址转换成硬盘的三维地址
将三维地址按照硬盘的地址转换方式重新组合成一维地址进行返回
}
}
```

2.3.2 计时器设计

计时器类中运行一个线程负责产生时间戳，对时间戳进行自增处理模拟系统内核时间的自增。同时每隔一段时间使中断信号发生改变，产生一次中断信号，

2.3.3 寄存器设计

在 cpu 中设置 PC、IR、CR3、PSW 等寄存器，存储程序的执行情况。同时

设置 pcb 指针 current_pcb 用于定位当前执行的进程的 pcb 块，在切换进程之后负责恢复现场。

2.4 地址线和数据线

由于地址线与数据线并不是独立存在的硬件单元，但在硬件仿真中不可缺少，故对这两个对象分别抽象为类，设置对应的读写函数用于操作，但不设置文件作为硬件的虚拟实现方式。

2.5 页面管理

将所有存储空间（32KB 内存+1MB 硬盘）统一按照 512B 每页的形式进行划分，共计 2112 页，第 0-63 页为内存空间，64-192 页为虚存空间，第 224-2111 页为硬盘的用户空间。对于不同区间的页面，采取不同的管理方法。

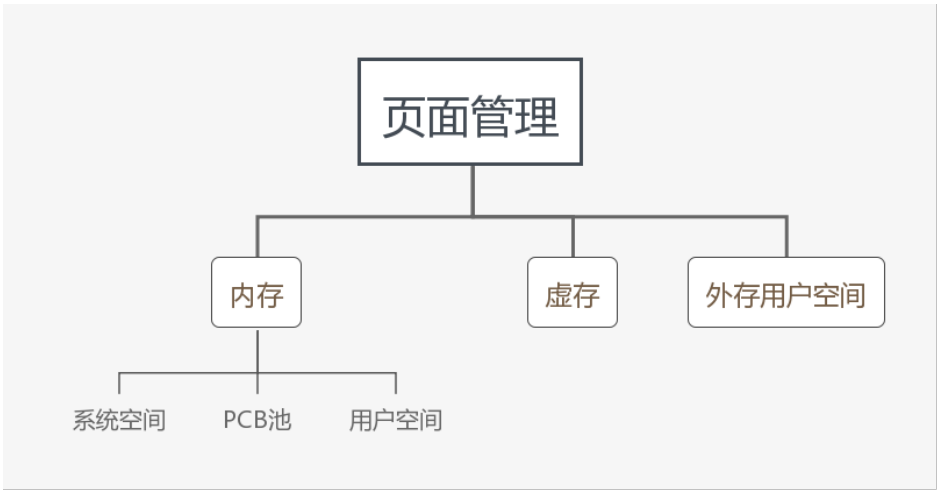


图 4 页面管理模块设计图

由于虚存空间相对而言足够大，故选择在高级调度时将作业的所有信息全部调入至虚存中，再由缺页中断调入内存；相应的，在内存空间不足时，也会使用 LRU 算法将某些进程中最久未使用的页中的数据调回至虚存中，增加内存可用空间。

2.5.1 伙伴算法分配内存

```
Applymemory(需求的页号数量 page_need)
{
向上取整计算 page_need 最接近的 2 的整数次幂 k
在空闲链表中找到大小为 k 的链中的第一个空闲块号
if(还有符合条件的空闲块)
{将空间分配给申请的进程，并自上而下修改 bitmap 中记录数据|
else
{返回空，分配内存失败}
}
```

2.5.2 回收内存空间

```
Recyclepage(需要回收的页号 page_num)
{
```

将 page_num 页所对应内存空间填充为全 0
递归修改page_num所在bitmap图，若有伙伴块被一起释放则修改上层bitmap图状态
}

2.5.3 虚存空间分配内存

ApplyDisk(需求的页号数量 Page_need)
{
While(Page_need>=0)
{
在虚存中顺序查找未使用的块号
向申请的进程分配一个空闲块
继续查找下一个空闲块，并将 Page_need-1
}
返回分配情况，使用字符串存储分配信息
}

2.5.4 回收虚存空间

Recyclepage(需要回收的页号 page_num)
{
将 page_num 页所对应内存空间填充为全 0
将 page_num 页的使用情况修改为未使用
在申请该页的进程中删除对该页的占用信息
}

2.5.5 页面生成

在新建页对象时，使用构造函数生成页，数据产生于内存中页面对应地址。
Page(页号 num)
{
调用 mmu 中的功能，将页号转换为物理地址 address
For(i=0;i<512;i+=2)
{
将地址 address+i 处 short 类型数据复制到 page 类数据对象中
}
}

2.5.6 页面修改

在修改页中数据时，不仅修改页面中数据成员，同时重新寻址并修改对应内存中数据。

在交换操作时，由于页对象已生成，对其数据无法进行修改，故需要重新生成页查看修改后的信息。

3 技术问题分析报告

3.1 问题一

在页中设置数组存放页内数据，在对页面数据进行修改时无法同步对内存中数据进行修改。

解决方法：取消页中存放数据的数组，在修改页数据时新建页对象，直接使用内存空间作为页面数据存放的空间，在修改页面时直接修改内存数据。符合页面作为虚拟逻辑结构的实际情况。

3.2 问题二

问题：伙伴算法的设计

解决方法：通过查询资料，使用 bitmap 算法，建立不同页面数大小的空闲队列， 使用位运算作为提取 bit 情况的工具。设计时考虑到设计的统一性，所有队列均采用最大空间即 3 2 位设计，不应被使用的位补 0，避免误操作。同时，在分配和回收页面时采用递归设计，分配时自上而下分配指定大小的页面空间给进程，且额度恒定位 2 的 k 次方，在可允许的页面浪费情况下减少了资源回收和分配的算法复杂度。k 较大的页所包含的页为 k-1 层两个相邻的页，即处于伙伴关系的页。

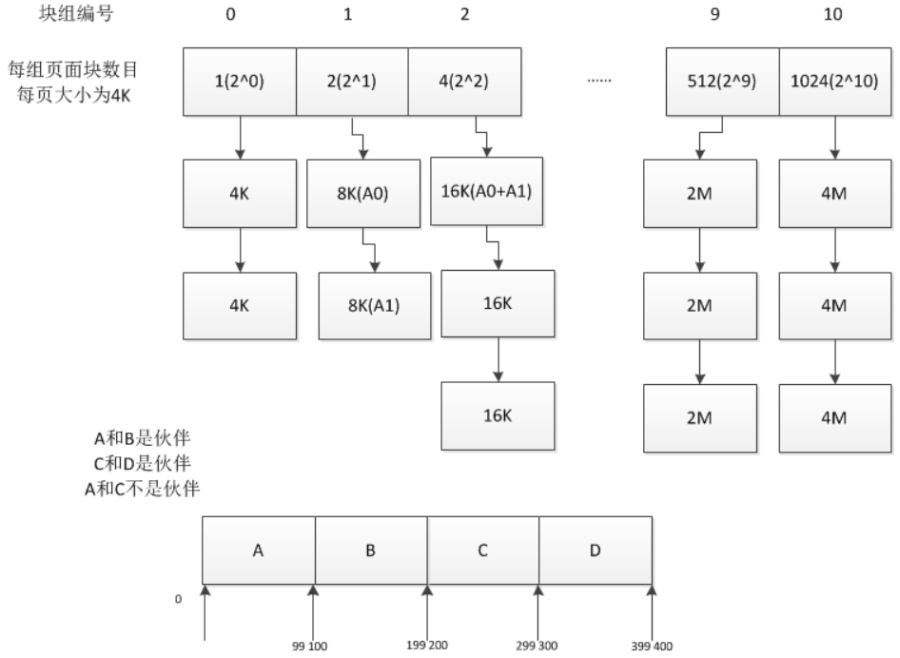


图 5 伙伴算法及 bitmap 的设计

3.3 问题三

分配页号时需考虑不同设备、不同逻辑区间的区别

解决方法：在系统区间中，允许根据页号进行内存单元的读取操作，但禁止访写入操作；在用户空间中，资源的申请和释放需要通过伙伴算法相关函数进行，通过页号可查询到是否处在使用中；在续存空间中，可直接申请需要的页面数量，并进行分配，释放时可直接根据页面序号进行释放。统一编排页号后在具体函数中添加条件判断，用于检测页号是否在合法空间中，实现对非法操作的拒绝处理。

3.4 问题四

在设计页面监测的 UI 界面时需要对界面进行动态刷新

解决方法：经过查阅资料，对于不同功能的模块分别建立类并进行实例化，对每个类分别继承 JPanel 类，重载其 paint 函数，实现对于界面的重绘，完成功能需求。

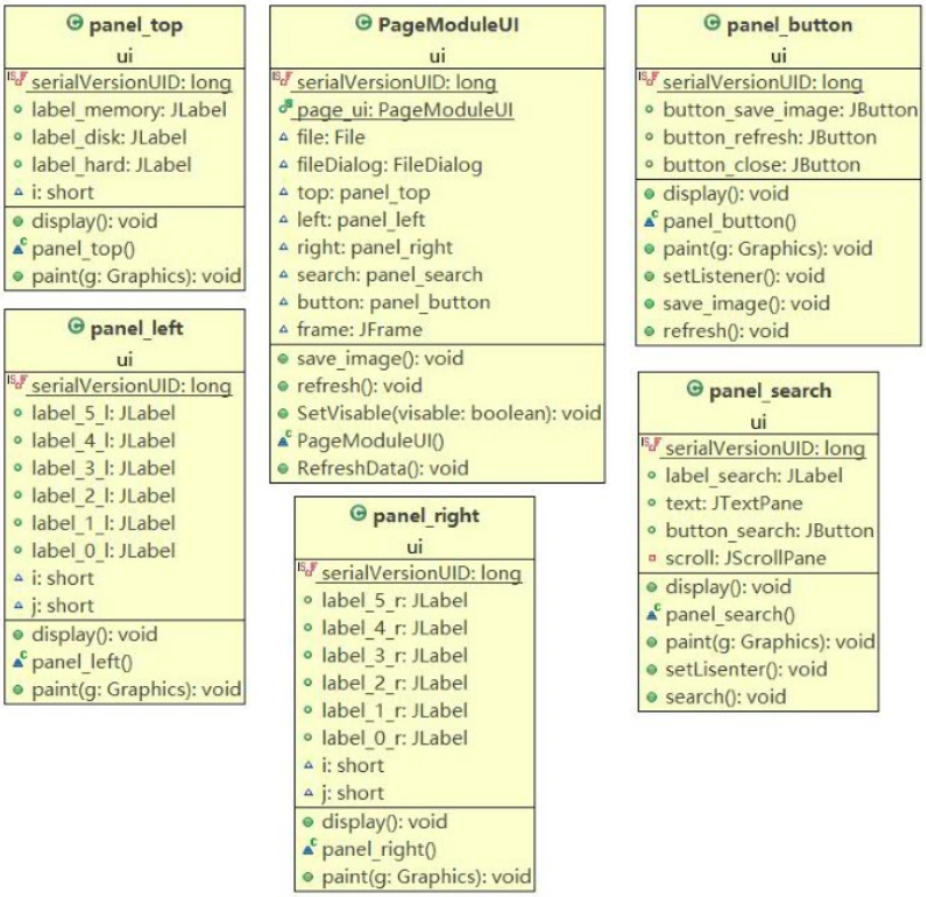


图 6 在主类中实例化子类对象

3.5 问题五

调用其他类的实例化对象时保证结果唯一

解决方法：对于每一个可能在外部进行调用的类，分别创建该类的静态对象，保证对象的唯一性，同时避免了在生成类对象时可能的系统情况差异，保证了程序逻辑的合理。

4 个人总结

经过本次操作系统课设，我对于操作系统的理解程度从纸上谈兵的阶段加深到了略懂一二的程度。可以说，操作系统的每一个部分都要经过精心的设计，才能在构筑成整体的操作系统时相互协作，更好的完成操作系统的任务。

在本次课程设计实验中，我们的选题是模拟仿真 Linux2.6 的系统中的部分模块，由于 linux 系统的设计十分巧妙，花费了许多时间阅读相关参考文献才在心中有了各个模块大致的实现方法。

在此次的课程设计中，我主要负责：CPU 部件的仿真、内存空间的仿真实现、MMU 地址变换、页表设计实现、页面调度算法、页面分配与回收算法、等几个方面内容的撰写，同时设计了内存检测窗口和 CPU 信息查看窗口。

由于功能较为底层，在设计时尽可能精简，只保留了该部分最核心的功能，以便搭建起逻辑较为简单的仿真系统运行的平台，在不影响平台运行的情况下降低逻辑复杂度。

在此基础上，对整个存储空间使用页式存储管理，既方便了伙伴算法的建立，又降低了逻辑复杂度。代价是对程序员较不友好，但在本次仿真中并不考虑程序员方的需求，故此代价可忽略。

由于能力不足，系统中尚有不足之处，需在今后的时间里逐步完善，最终完成一个较为贴近真实的 linux2.6 仿真系统的编写。

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.

[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] 27Up. 伙伴算法 buddy
[EB/OL].https://blog.csdn.net/lcl497049972/article/details/82954124

[8] Unique-You.伙伴(buddy)算法及位图
[EB/OL].https://blog.csdn.net/qq_22238021/article/details/80208630

[9] yilongdashi.操作系统-虚拟存储管理技术之虚拟页式存储管理
[EB/OL].https://blog.csdn.net/yilongdashi/article/details/82728780

[10] Givefine.Java 中的多线程你只要看这一篇就够了
[EB/OL].https://www.cnblogs.com/wxd0108/p/5479442.html

[11]大脑补丁. Java 用 JFrame、JPanel、Graphics 绘图案例讲解
[EB/OL].https://blog.csdn.net/x541211190/article/details/77414861

[12] 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

附件 1：组长测试报告

一、完成任务与功能说明

1. 作业调度

1.1 作业调度模块数据成员

```
private int job_id=0;//作业号
private int job_page_location_start=(kernel.MEMORY_SIZE+kernel.HARDDISK_VIRTUAL_MEMORY_SIZE+kernel.HARDDISK_SYSTEMFILE_SIZE)/kernel.SINGLE_PAGE_SIZE;  //作业存储区的地址开始
private int job_page_location_end=(kernel.MEMORY_SIZE+kernel.HARDDISK_SIZE)/kernel.SINGLE_PAGE_SIZE;  //作业存储区的地址结束
private int next_to_pcb=0;  //下一个将要读入的作业序号
private int write_job_page_num=job_page_location_start;  //写入作业的编号
public ArrayList<JCB> job_list=new ArrayList<JCB>();  //作业后备队列
```

设置起始地址用来限制访问空间，避免越界访问引起系统错误。

1.2 JCB 设计

1.2.1 JCB 数据成员

```
private short job_id; //作业 ID
private short priority;  //作业/进程的优先级
rivate 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
private short in_page_num=0;  //该 JCB 所在的页号
```

以上数据总大小小于 512B，故可使用一页进行存储。

1.2.2 计算作业所占页面数

```
public short CalculatePagesNum()
 {
     //每条指令占 8 个字节，1 页有 64 条指令，加上 JCB 占 1 页空间
     if(this.instruction_num==0)
         return 1+1;
     else if((this.instruction_num%64==0)&&(this.instruction_num!=0))
         return (short) (this.instruction_num/64+1);
     else
         return (short) ((this.instruction_num/64+1)+1);
 }
```

由于 JCB 的设计为第一页存放作业信息，故每个作业所占空间为指令数/8 向上取整再加一。

1.3 将作业转存到外存

```
public void SaveJobToHardDisk(JCB jcb)
 {
      //将作业保存到外存，d1 是 JCB 数据块，d2 是包含所有指令的 ArrayList，每条指令占 8 字节
      Page jcb_page=new Page((short) this.write_job_page_num++);
//获取应写入 JCB 的那一页
      jcb_page.SetPageData((short)0, (short) j);          //写入 JCB 的标识，01 位
      jcb_page.SetPageData((short)2, (short) job_id++);//写入作业的序号 ID，23 位
      jcb_page.SetPageData((short)4, jcb.GetPriority());//写入作业优先级，45 位
      jcb_page.SetPageData((short)6, (short) jcb.GetJobIntime());
//写入作业进入时间，67 位
      jcb_page.SetPageData((short)8, jcb.GetInstruction_num());       //写入指令数目，89 位
      jcb.SetPagesNum(jcb.CalculatePagesNum());//计算该进程所需要占的页面数
      jcb_page.SetPageData((short)10, jcb.GetPagesNum());
//写入占用页面数目，1011 位
      //开始随机填充接下来的部分，模拟数据区
      for(short i=12;i<kernel.SINGLE_PAGE_SIZE;i+=2)
      {
           jcb_page.SetPageData(i, (short) CPU.random.nextInt());;
      }
      //将所有指令写入接下来的页
      int already_write_instruction_num=0;
      for(short i=0;i<jcb.GetPagesNum()-1;i++)
      {
           Page t=new Page((short) (this.write_job_page_num+i));     //获取该页面
           //写入页的内容
           for(short j=0;j<512;j+=8)
           {
                if(already_write_instruction_num<=jcb.GetInstruction_num())
//指令还未写完
                {
                     //写入指令
 t.SetPageData(j, jcb.GetAll_Instructions().get(already_write_instruction_num++));
                     //随机填充数据
      for(short k=(short) (j+2);k<j+kernel.SINGLE_INSTRUCTION_SIZE;k+=2)
                          t.SetPageData(k, (short) CPU.random.nextInt());
                }
                else //指令已经写完
                {
                     //随机填充数据
           for(short k=(short) j;k<j+kernel.SINGLE_INSTRUCTION_SIZE;k+=2)
                          t.SetPageData(k, (short) CPU.random.nextInt());
                }
           }
```

```
        }
        this.write_job_page_num+=jcb.GetPagesNum()-1;            //指针后移
    }
```

按照计算好的页号需求量，在外存的连续空间上写入作业信息。

1.4 获取所有 JCB 信息

```
public JCB[] GetAllJCB()
    {
        int job_num=GetJobNum();                //获取总作业数量
        JCB []all_jcb=new JCB[job_num];             //创建存储所有 jcb 的对象数组
        int read_job_num=0;             //已经读取的作业数量
        int read_page_num=this.job_page_location_start;
//当前读取的页面页号，初始化为存储区开始点
        while(read_job_num!=job_num)
        {
            //获取当前页面
            Page pa=new Page((short) read_page_num);
            if(pa.GetPageData((short) 0)!=j)    //当前页不是 JCB 所在页
                continue;       //直接跳过，读取下一页面
            //实例化 JCB 并存入数据
            all_jcb[read_job_num]=new JCB();
            all_jcb[read_job_num].SetInPageNum((short) read_page_num);              // 设 置
JCB 所在页面序号
            all_jcb[read_job_num].SetJobid(pa.GetPageData((short) 2));              // 设 置
作业的 ID
            all_jcb[read_job_num].SetPriority(pa.GetPageData((short) 4));
//设置作业优先级
            all_jcb[read_job_num].SetJobIntime(pa.GetPageData((short) 6));          // 设 置
作业进入时间
            all_jcb[read_job_num].SetInstructionNum(pa.GetPageData((short) 8));     // 设 置
作业指令数量
            all_jcb[read_job_num].SetPagesNum(pa.GetPageData((short) 10));          // 设 置
作业占用页面数
            //获取所有指令
            read_page_num++;        //读取下一页面
            Page instru_page=new Page((short) read_page_num); //获取指令页面
            for(int i=0;i<all_jcb[read_job_num].GetInstruction_num();i++)
            {
 all_jcb[read_job_num].GetAll_Instructions().add(instru_page.GetPageData((short)
((i%kernel.INSTRUCTIONS_PER_PAGE)*kernel.SINGLE_INSTRUCTION_SIZE)));
                if((i+1)%kernel.INSTRUCTIONS_PER_PAGE==0)
                {
                    read_page_num++;
                    instru_page=new Page((short) read_page_num);//获取下一页面
                }
```

```
            }

if(all_jcb[read_job_num].GetInstruction_num()%kernel.INSTRUCTIONS_PER_PAGE==0)
                read_page_num--;
            else
                read_page_num++;
            read_job_num++;         //已经读取完一个作业
        }
        return all_jcb;
}
```

将所有作业信息读取，存放到 JCB 数组中，为作业转换为进程做准备。

2. 进程的三级调度

2.1 高级调度

```
public void HighLevelScheduling()
{
    //高级调度
    //检测后备队列是否有新的未被导入的作业
MainUI.main_ui.run_info+="系统时间："+kernel.SYSTEM_TIME+"高级调度--";
    JobModule.job_module.RefreshJobList();                  //刷新后备队列
    MainUI.main_ui.run_info+="检测后备队列是否有作业--";
    if(!JobModule.job_module.IsAllJobToProcess()) //后备队列中还有作业
    {
        while(!JobModule.job_module.job_list.isEmpty())
        {
            MainUI.main_ui.run_info+="后备队列有作业--";
        JCB jcb=JobModule.job_module.job_list.get(0);  //获取最近的一个作业
    MainUI.main_ui.run_info+="作业序号："+jcb.GetJobid()+"等待被调入--";
            //检查 PCB 池是否空间足够
            MainUI.main_ui.run_info+="检测 PCB 池是否足够--";
            if(ProcessModule.process_module.GetFreePCBNumInPool()>=1)
            {
                MainUI.main_ui.run_info+="PCB 空间足够--";
                //PCB 池空间足够
                //检测虚存空间是否足够
                MainUI.main_ui.run_info+="检测虚存空间是否足够--";
if(jcb.GetPagesNum()<=PageModule.page_module.GetFreePageNumInDisk()-1)
                {
                    //虚存空间足够
                    MainUI.main_ui.run_info+="虚存空间足够--";
                    //第一步，JCB 转换为 PCB
                    MainUI.main_ui.run_info+="将 JCB 转换为 PCB--";
                PCB pro=ProcessModule.process_module.TurnJCBToPCB(jcb);
                    //第二步，在虚存中申请对应大小的空间
```

```
                        MainUI.main_ui.run_info+="在虚存中申请空间--";
                        String
swap_area_apply=PageModule.page_module.ApplyPageInDisk(pro.GetPagesNum());
                        //第三步，将作业的所有指令区调入到虚存中
                        MainUI.main_ui.run_info+="作业载入--";

 ProcessModule.process_module.TransferJobCodeToSwapArea(jcb,swap_area_apply);
                        //第四步，写 PCB 中的页表
                        MainUI.main_ui.run_info+="写入页表--";
                        ProcessModule.process_module.WriteProcessPageTable(pro,
swap_area_apply);
                        //第五步，将 PCB 写入到 PCB 池
                        MainUI.main_ui.run_info+="写 PCB 池--";
                        ProcessModule.process_module.AddToPCBPool(pro);
                        //第六步，指针后移
                        JobModule.job_module.NextJob();
            JobModule.job_module.RefreshJobList();              //刷新后备队列
                        //第七步，将该进程加入到就绪队列，等待被调度
                MainUI.main_ui.run_info+="进程加入到就绪队列，等待被调度\n";
                        ProcessModule.process_module.TransferProcessToReadyQueue(pro,
true);
                        //pro.ProcessCreate();    //调用进程创建原语
                    }
                    else //虚存空间不足没有空间
                    MainUI.main_ui.run_info+="虚存空间不足，高级调度退出\n";
                }
                else //PCB 池空间不足
                MainUI.main_ui.run_info+="PCB 池空间不足，高级调度退出\n";
            }
        }
        else   //后备队列中没有作业
 MainUI.main_ui.run_info+="在后备队列中没有检测到作业，高级调度退出\n";
 }
```

高级调度通过判断后备队列是否有作业判断是否启动，判断虚存中是否有空间和 PCB 池中是否有空间决定能否执行高级调度。高级调度成功之后即将一个作业转换为一个进程，全部信息调入虚存中，并将进程加入就绪队列等待低级调度。

2.2 中级调度

```
public void MiddleLevelScheduling()
 {
     //中级调度
 MainUI.main_ui.run_info+="系统时间："+kernel.SYSTEM_TIME+"中级调度--";
     //将挂起队列中的进程全部取出
     for(int i=0;i<ProcessModule.process_module.suspend_queue.size();i++)
     {
```

```
            PCB t=ProcessModule.process_module.suspend_queue.get(i);
            ProcessModule.process_module.TransferProcessToReadyQueue(t,true);  // 将该进程加入到就绪队列
        }
        //第一步，获取当前内存中可用的页框数
        MainUI.main_ui.run_info+="检测当前内存中可用的页框数--";
        int free_page_num=PageModule.page_module.GetFreePageNumInMemory();
        //第二步，如果可用页框数小于 10，则进行中级调度
        if(free_page_num<10)
        {
            MainUI.main_ui.run_info+="当前可用页框数为："+free_page_num+"，立即进行中级调度--";
            //页框数小于 10
            for(int j=22;j<PageModule.page_module.lru.size();j++)
            {
                int page=PageModule.page_module.lru.get(j);
                SuspendProcessWithPageNum((short) page);
            }
        }
        else //空闲页框数足够
            MainUI.main_ui.run_info+="当前可用页框数为："+free_page_num+"，退出中级调度\n";
    }
```

当内存中可用页框数量较低时，进行中级调度，将内存中最近最久未访问的页框释放，并将该页框所属于的进程设为挂起态，直到内存中可用页框数不低于 10 个时截止。

2.3 低级调度

```
public void LowLevelScheduling()
 {
    //低级调度
    MainUI.main_ui.run_info+="系统时间："+kernel.SYSTEM_TIME+"低级调度--";
    kernel.SystemTimeAdd();                                    //系统时间自增
    this.wait_time-=kernel.INTERRUPTION_INTERVAL;     //等待时间自减
    if(wait_time<=0)
    {
        if_wait=false;
        if(!ProcessModule.process_module.waiting_queue.isEmpty())
        {
            PCB t=ProcessModule.process_module.waiting_queue.get(0);
            MainUI.main_ui.run_info+="等待时间满，进程"+t.GetPid()+"退出等待态！\n";
            if(t.if_in_p==true)
            {
                t.if_in_p=false;
                if(t.if_p_success)
```

```
                    t.AddCurrentInstructionNo();
                }
                else
                    t.AddCurrentInstructionNo();
                t.ins_runtime=0;
                if_wait=false;
                t.RefreshCounter();       //刷新时间片余额
                ProcessModule.process_module.TransferProcessToReadyQueue(t,false);
            }
        }
        //第一步，检测当前是否正处于等待态，如果处于，则 CPU 等待
        if(if_wait==true)
        {
            MainUI.main_ui.run_info+="当前 CPU 处于等待态，继续等待，退出低级调度！
\n";
            PCB t=ProcessModule.process_module.waiting_queue.get(0);
            t.AddRuntime();           //模拟已经运行过一次
            t.AddTotalRuntime();    //总运行时间增加
            t.ins_runtime+=kernel.INTERRUPTION_INTERVAL;
            //处于等待态，则检查等待时间
            return;
        }
        //第二步，刷新 active 和 expired 指针
        ProcessModule.process_module.RefreshActiveExpired();
        //第三步，查看当前运行队列是否还有进程
        if(ProcessModule.process_module.IsRunningQueueEmpty()==true)
        {
            MainUI.main_ui.run_info+="运行队列中没有进程，进行重新调度--";
            for(int i=0;i<140;i++)
            {
                if(ProcessModule.process_module.IsRunningQueueEmpty()==false)
                    break;

 if(!ProcessModule.process_module.ready_queue[ProcessModule.process_module.GetActive
Point()][i].isEmpty())
                {
                    PCB
t=ProcessModule.process_module.ready_queue[ProcessModule.process_module.GetActivePo
int()][i].get(0);
                    ProcessModule.process_module.TransferProcessToRunningQueue(t);
                    MainUI.main_ui.run_info+="调入进程"+t.GetPid()+"--";
                }
            }
        }
```

```
        else
            MainUI.main_ui.run_info+="队列中有进程运行--";

        //第四步，检测当前正准备执行的指令所在的页是否在内存中
        if(ProcessModule.process_module.running_queue.isEmpty())
        {
            MainUI.main_ui.run_info+="当前没有可运行进程！\n";
            return;
        }
        PCB run=ProcessModule.process_module.running_queue.get(0);
        //CPU 恢复现场
        CPU.cpu.Recovery(run);
        //检测是否发生缺页中断
        MainUI.main_ui.run_info+="运行指令--";
 if(ProcessModule.process_module.IfPageInMemory(run.page_table[run.GetCurrentInstructio
nNo()/(kernel.SINGLE_PAGE_SIZE/kernel.SINGLE_INSTRUCTION_SIZE)][1])==false)
        {
            //如果所需要的页面不在内存中，则发生缺页中断
            //缺页中断的处理
            MainUI.main_ui.run_info+="发生缺页中断--";
            ProcessModule.process_module.SolveMissingPage(run,
run.page_table[run.GetCurrentInstructionNo()/(kernel.SINGLE_PAGE_SIZE/kernel.SINGLE
_INSTRUCTION_SIZE)][1]);
            MainUI.main_ui.run_info+="已进行调页处理--";

 PageModule.page_module.LRUVisitOnePage(run.page_table[run.GetCurrentInstructionNo()
/(kernel.SINGLE_PAGE_SIZE/kernel.SINGLE_INSTRUCTION_SIZE)][1]);
        }
        else

 PageModule.page_module.LRUVisitOnePage(run.page_table[run.GetCurrentInstructionNo()
/(kernel.SINGLE_PAGE_SIZE/kernel.SINGLE_INSTRUCTION_SIZE)][1]);
        //MMU 的地址变换，取指令
        MainUI.main_ui.run_info+="MMU 进行地址变换 --PC 指针：
0x"+String.format("%04x", CPU.cpu.GetPC()).toUpperCase()+
                "转换后的实地址：0x"+String.format("%04x",
CPU.cpu.mm.VirtualAddressToRealAddress(run, (short) CPU.cpu.GetPC())).toUpperCase();
        //执行指令
        if(run.getInstructions().size()<=run.GetCurrentInstructionNo())
            return;
        int
current_ins_type=kernel.GetInstructionType(run.getInstructions().get(run.GetCurrentInstructi
onNo()));
        MainUI.main_ui.run_info+="当前执行指令：
```

```
"+run.getInstructions().get(run.GetCurrentInstructionNo())+"--";
        //每一种指令的不同运行过程
        run.RefreshTimeslice();        //刷新时间片
        run.SetCounter(0);                    //设置已用时间片
        CPU.cpu.mm.ClearTLB();          //清空快表
        switch(current_ins_type)
        {
        case 1:
            MainUI.main_ui.run_info+="指令类型：系统调用--";
            ProcessModule.process_module.RunType1(run);
            break;
        case 2:
            MainUI.main_ui.run_info+="指令类型：P 指令--";
            ProcessModule.process_module.RunType2(run);
            break;
        case 3:
            MainUI.main_ui.run_info+="指令类型：V 指令--";
            ProcessModule.process_module.RunType3(run);
            break;
        case 4:
            MainUI.main_ui.run_info+="指令类型：申请资源--";
            //ProcessModule.process_module.RunType4(run);
            break;
        case 5:
            MainUI.main_ui.run_info+="指令类型：释放资源--";
            //ProcessModule.process_module.RunType5(run);
            break;
        case 6:
            MainUI.main_ui.run_info+="指令类型：普通指令--";
            ProcessModule.process_module.RunType6(run);
            break;
        }
        //检测是否运行完毕，如果运行完毕，调用撤销原语
        if(ProcessModule.process_module.IfRunOver(run)==true)
        {
            MainUI.main_ui.run_info+="\n 进程"+run.GetPid()+"运行完毕！\n";
            run.ProcessCancel();          //调用进程撤销原语
        }
        //检测时间片是否用完，如果完毕，则进入就绪队列
 if(ProcessModule.process_module.IfTimeSliceOver(run)==true&&this.if_wait!=true)
        {
            MainUI.main_ui.run_info+="\n 进程"+run.GetPid()+"时间片到期！\n";
            //设置 counter 为 0
            run.SetCounter(0);
```

```
            run.SetRuntime(0);
            //动态调整优先级
            run.RefreshPriority();
            //将该进程放入就绪队列
        ProcessModule.process_module.TransferProcessToReadyQueue(run, false);
        }
        MainUI.main_ui.run_info+="\n";
    }
```

低级调度即进程的三态转换，同时在低级调度时，对处于运行态的进程所申请的页面产生缺页中断的响应，将申请的页面调入至内存申请的页框之中。低级调度所使用的算法为 O(1)算法，该算法使用分散计算时间片的方式，每次都是选取优先级最高而且还有剩余时间片的进程来运行，重新计算时间片的时间复杂度也仅有 O(1)。通过设置活动进程和过期进程提高了调度效率。

3. 死锁检测

3.1 系统资源量设计

```
public static int[] MUTEX= {-4,-3,-2,-1,0,1,2,3,4,5};          //临界区信号量,10 个
public static int[] SYSTEM_RESOURCE= {0,1,2,3,4,5,6,7,8,9}; //系统资源量，10 个
```

3.2 死锁检测与释放算法

```
public ArrayList<PCB> CheckDeadLock()
 {
        //死锁检测
        //第一步，令 Work[*]=Available[*]
        ResetWork();
        InitFinish();
        //第二步，如果 Allocation[k,*]不等于 0，令 finish[k]=false;否则 finish[k]=true
        for(int k=0;k<n;k++)
        {
            if(!IfAllocationLineEmpty(k))
                finish[k]=false;
            else
                finish[k]=true;
        }
        //第三步，寻找一个 k 值
        for(int k=0;k<n;k++)
        {
            if(Step3_find_k_value(k)==true)
                //第四步，修改 Work[*]=Work[*]+Allocation[k,*],finish[k]=true
                for(int j=0;j<m;j++)
                    Work[j]=Work[j]+Allocation[k][j];
            finish[k]=true;
        }
        //第五步，查找处于死锁的进程
        ArrayList<PCB> dl_pcb=new ArrayList<PCB>();
```

```
        for(int k=0;k<n;k++)
        {
            if(finish[k]==false)
            dl_pcb.add(ProcessModule.process_module.GetPCBWithID((short) k));
        }
        return dl_pcb;
    }
```

仿照银行家算法设计死锁检测算法，在发生死锁时撤销所有发生死锁的进程。

4. 进程管理

4.1 进程 PCB 设计

4.1.1 PCB 数据成员

```
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
private                    short                    [][]page_table=new
short[(kernel.MEMORY_USER_SPACE_SIZE)/kernel.SINGLE_PAGE_SIZE][2];
 //页表，page_table[i][0]为进程的页号，从 0 开始编号；page_table[i][1]为对应的物理页
号
private ArrayList<Integer> instructions=new ArrayList<Integer>();
//该进程所有的指令
private short in_page_num=0;        //该 PCB 所在的页号
private short pool_location=-1;              //该 PCB 在 PCB 池的位置
```

PCB 类中包含了进程和作业的全部信息，在创建时生成。

4.1.2 进程原语

分别设置进程的创建、撤销、阻塞、唤醒、挂起原语。

4.1.3 更新优先级

```
public void RefreshPriority()
 {
        //随机生成-5~5 之间的数字  更新进程的优先级
        short changed=(short) (new Random().nextInt(5-(-5))+(-5));
        this.priority+=changed;
        if(this.priority>=19)
            this.priority=19;
```

```
        if(this.priority<=-20)
            this.priority=-20;
    }
    public void RefreshTimeslice()
    {
        //根据更新后的优先级更新该进程能够获得的时间片
        if(this.priority<0)
            this.timeslice=(short) (((short)(-35*this.priority+100))/10*10);
        else
            this.timeslice=(short) (((short)(-2.6316*this.priority+100))/10*10);
    }
```

使用动态优先级配合时间片以实现 linux2.6 的调度算法。

4.1.4 页表操作

支持对本进程分配页表的查询、修改，以及向分配页表记录中添加新的页表等操作。

4.1.5 PCB 操作

支持对 PCB 各数据项的读取与写入操作。

4.2 进程管理模块数据成员

```
private ArrayList<PCB> all_queue=new ArrayList<PCB>();      //所有进程链表
private ArrayList<PCB> running_queue=new ArrayList<PCB>();      //运行队列
@SuppressWarnings("unchecked")
private ArrayList<PCB> [][]ready_queue=new ArrayList[2][140];      //就绪队列
private int active=0; //active 指针
private int expired=1;      //expired 指针
private boolean []ready_queue_bitmap=new boolean[140];
//就绪队列位图，范围 0 - 139，共 140 位
private ArrayList<PCB> waiting_queue=new ArrayList<PCB>();      //等待队列
private ArrayList<PCB> suspend_queue=new ArrayList<PCB>();      //挂起队列
private ArrayList<PCB> end_queue=new ArrayList<PCB>();
//已经运行结束的进程
private boolean[] PCB_pool_usage=new boolean[kernel.MEMORY_KERNEL_PCB_POOL_SIZE/(kernel.SINGLE_PAGE_SIZE/2)];
      //PCB 池使用情况
```

分别设置运行、就绪、等待、挂起、结束队列管理处于不同状态的进程信息，为进程调度提供支持。同时设置 PCB 池，在限制系统并发度的同时提高对进程管理的操作空间。

4.3 作业转换为进程

```
public PCB TurnJCBToPCB(JCB jcb)
{
    //将 JCB 变换为 PCB
    PCB pcb=new PCB();
    //进程标识符
    pcb.SetPid(jcb.GetJobid());
    //进程状态
    pcb.SetState(kernel.PROCESS_READY);
    //进程优先级
```

```
        pcb.SetPriority(jcb.GetPriority());
        //作业创建时间
        pcb.SetJobIntime(pcb.GetJobIntime());
        //进程创建时间
        pcb.SetProcessIntime(kernel.SYSTEM_TIME);
        //作业/进程结束时间
        pcb.SetEndTime(-1);            //在创建时不设置，等到进程撤销时设置
        //时间片长度
        pcb.RefreshTimeslice();
        //进程已经运行时间
        pcb.SetRuntime(0);
        //该进程处于运行状态下的时间片余额
        pcb.SetCounter(-1);       //在创建时不设置
        //程序状态字
        pcb.SetPSW(kernel.PSW_USER_STATE);
        //当前运行到的指令编号
        pcb.SetCurrentInstructionNo((short) 0);
        //该进程总共包含的指令数目
        pcb.SetInstructionNum(jcb.GetInstruction_num());
        //该进程所占用的页面数目
        pcb.SetPagesNum((short) (jcb.GetPagesNum()-1));
        //初始化页表
        for(short
i=0;i<(kernel.MEMORY_USER_SPACE_SIZE+kernel.HARDDISK_VIRTUAL_MEMORY
_SIZE)/kernel.SINGLE_PAGE_SIZE;i++)
        {
            pcb.EditPageTable(i,(short)-1,(short)-1);
        }
        //添加所有指令
        for(int i=0;i<pcb.GetInstruction_num();i++)
        {
            pcb.AddInstruction(jcb.GetAll_Instructions().get(i));
        }
        //所在页号
        pcb.SetInPageNum((short) -1);//由 void AddToPCBPool(PCB pcb)函数修改
        return pcb;
    }
```

在作业转换到进程时根据作业信息和系统信息设置 PCB 的数据成员。

```
public short ApplyOnePCBInPool()
    {
        //在 PCB 池中申请一个 PCB
        for(short
i=0;i<kernel.MEMORY_KERNEL_PCB_POOL_SIZE/(kernel.SINGLE_PAGE_SIZE/2);i++)
            if(this.PCB_pool_usage[i]==false)
```

```
            {
                this.PCB_pool_usage[i]=true;
                return i;
            }
        return -1;
}
```

之后将生成的 PCB 加入 PCB 池中，若 PCB 池已满则无法创建进程。

4.4 将进程调入虚存

```
public void TransferJobCodeToSwapArea(JCB jcb,String apply)
{
    //将指定作业的程序段存入虚存中
    //jcb 为作业控制块，apply 为申请到的虚存空间分配字符串
    int already_transfer=0;          //已经转移的页面数量
    short i=0;
    while(already_transfer!=jcb.GetProcessNeedPage())
    {
        if(apply.charAt(i)==1)
        {
PageModule.page_module.CopyPage((short)(jcb.GetInPageNum()+1+already_transfer), i);
            i++;
            already_transfer++;       //已经分配完一页
        }
        else
        {
            i++;
            continue;
        }
    }
}
public void WriteProcessPageTable(PCB pcb,String apply)
{
    //将申请到的虚存页面写入到进程的页表中
    for(int i=0;i<apply.length();i++)
    {
        if(apply.charAt(i)==1)
            pcb.AddPageTable((short)i);
    }
}
```

根据 PCB 申请的页面数量和虚存管理所分配的页面信息，将进程数据调入制定虚存页面中，并将申请到的虚存页面添加至 PCB 的相关记录中。

4.5 进程状态切换

使用进程原语完成进程的状态切换，并将进程移动到制定队列中。

4.6 缺页中断

```
public void SolveMissingPage(PCB pcb,short need_page_num)
```

```
{
    //缺页中断的处理，need_page_num 为需要的在外存中的页的页号
    if(PageModule.page_module.GetFreePageNumInMemory()<1)  //内存满了
    {
        int out_page_num=PageModule.page_module.LRUGetLastPageNum();    //LRU
进行页面置换
    PageModule.page_module.ExchangePage((short) out_page_num, need_page_num); //
页面置换
        ChangePageTable((short) out_page_num,need_page_num); //更新对应的进程的
页表
    }
    else       //内存没有满
    {
        short in_page_num=PageModule.page_module.GetOneFreePageInMemory();  //
在内存中申请一页
        PageModule.page_module.MoveToMemory(in_page_num, need_page_num);  //
将交换区的页移入
        PageModule.page_module.RecyclePage(need_page_num);
//回收交换区
        ChangePageTable(in_page_num,need_page_num);
//更新页表
    }
}
```

5. 指令集设计

5.1 指令类型

由于页大小为 512B，设指令字长为 8B，共 6 种类型指令 60 条。

类型 1 为系统调用；

类型 2 为 P 操作；

类型 3 为 V 操作；

类型 4 为资源申请操作；

类型 5 为资源释放操作；

类型 6 为普通指令。

```
public void Type_1_Instruction_Slove(PCB pcb)
{
    //指令类型 1 的处理，普通系统调用
    //设置 PCB 的 PSW 状态
    //设置 CPU 的 PSW 状态
    pcb.SetPSW(kernel.PSW_USER_STATE);
    CPU.cpu.SetPSW(kernel.PSW_USER_STATE);
}

public void Type_2_Instruction_Slove(PCB pcb)
{
```

```
        //指令类型 2 的处理，P 操作
        //设置 PCB 的 PSW 状态
        //设置 CPU 的 PSW 状态
        pcb.SetPSW(kernel.PSW_KERNEL_STATE);
        CPU.cpu.SetPSW(kernel.PSW_KERNEL_STATE);
    }

    public void Type_3_Instruction_Slove(PCB pcb)
    {
        //指令类型 3 的处理，V 操作
        //设置 PCB 的 PSW 状态
        //设置 CPU 的 PSW 状态
        pcb.SetPSW(kernel.PSW_KERNEL_STATE);
        CPU.cpu.SetPSW(kernel.PSW_KERNEL_STATE);
    }

    public void Type_4_Instruction_Slove(PCB pcb)
    {
        //指令类型 4 的处理，资源申请操作
        //设置 PCB 的 PSW 状态
        //设置 CPU 的 PSW 状态
        pcb.SetPSW(kernel.PSW_KERNEL_STATE);
        CPU.cpu.SetPSW(kernel.PSW_KERNEL_STATE);
    }

    public void Type_5_Instruction_Slove(PCB pcb)
    {
        //指令类型 5 的处理，资源释放操作
        //设置 PCB 的 PSW 状态
        //设置 CPU 的 PSW 状态
        pcb.SetPSW(kernel.PSW_KERNEL_STATE);
        CPU.cpu.SetPSW(kernel.PSW_KERNEL_STATE);
    }

    public void Type_6_Instruction_Slove(PCB pcb)
    {
        //指令类型 6 处理，普通指令
        //设置 PCB 的 PSW 状态
        //设置 CPU 的 PSW 状态
        pcb.SetPSW(kernel.PSW_USER_STATE);
        CPU.cpu.SetPSW(kernel.PSW_USER_STATE);
    }
```

5.2 指令设计

直接依据指令的数值进行区分，不设计各字段的具体含义。

6. 内存、硬盘、调度等界面

该部分为 UI 设计部分，代码较多，在此不方便进行展示。若读者有查看的需要，请浏览/ui/文件夹下的文件。

7. PCB、JCB 数据结构设计

该部分将在下文详细描述。

二、测试安装用例详细说明

一、测试环境

系统：Windows10 1809

内存：16GB

硬盘：240GB SSD + 500GB HDD

JDK：java 11.0.2 2019-01-15 LTS

二、测试方法

1. 调入作业记录

将整个工程导入到程序中（建议使用 Eclipse），然后点击运行，开始执行该程序。

当程序开始执行后，将会出现以下界面。点击“载入作业”，在程序目录中选择 job.txt 加载。

本次测试我们提供了 7 组不同的数据进行测试，读者可以根据自己的需求进行选择并导入。

注意，导入的测试样例请勿随意修改，若修改，将会发生不可预知的情况，这种情况并不属于测试范围之内，而是关乎到 JAVA 本身的异常处理机制，可以直接省略。

当然，我们也鼓励读者使用自己编写的测试样例进行测试。编写的方法可以根据测试样例进行模仿。具体的指令代号含义如下：

0-9：系统调用

10-19：P 指令

20-29：V 指令

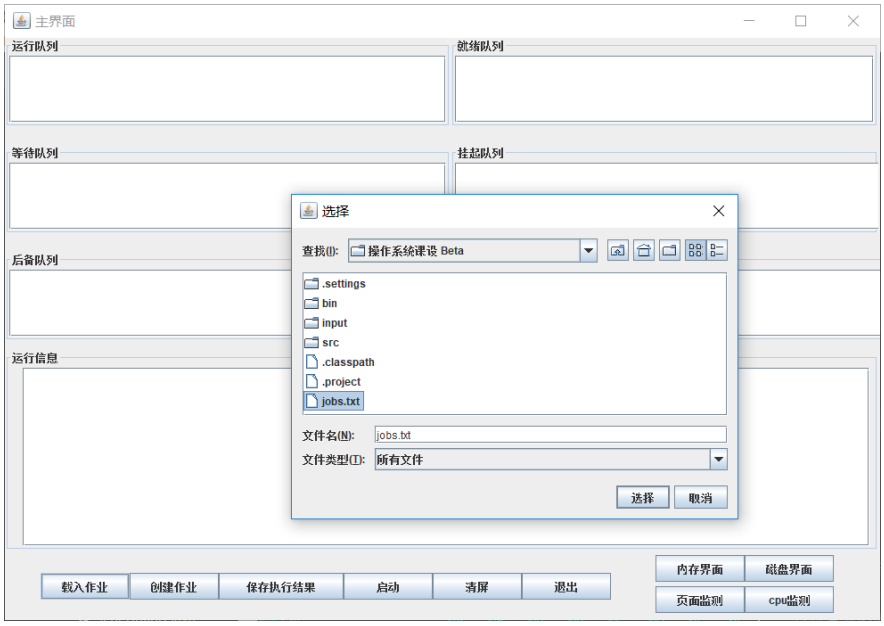
30-39：资源申请

40-49：资源释放

50-59：普通指令

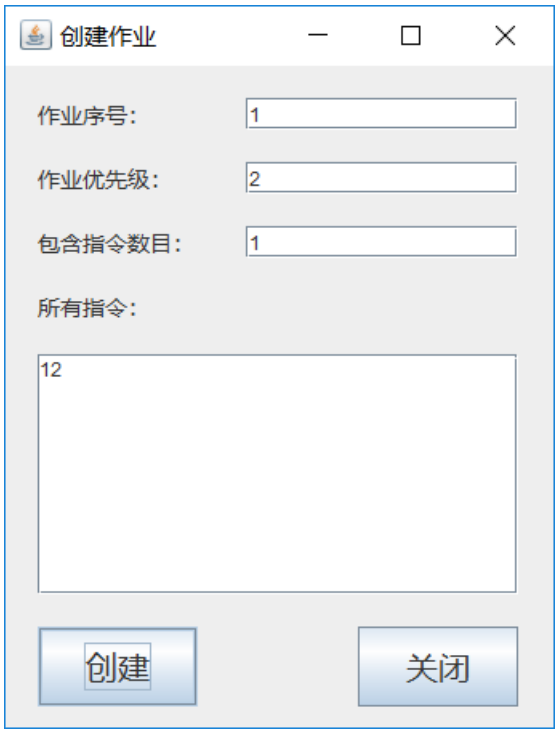
每一条指令都有各自不同的执行时间，具体的执行时间计算方法为：

time = instruction%10*10+20



2. 创建作业

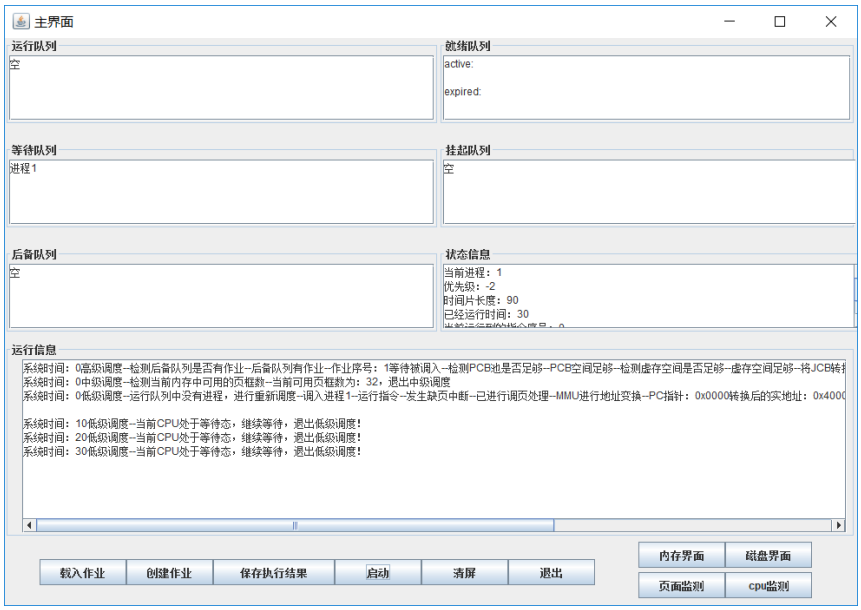
选择创建作业，填写信息后即可创建作业。



创建后直接启动，即可看到系统中运行一个进程。

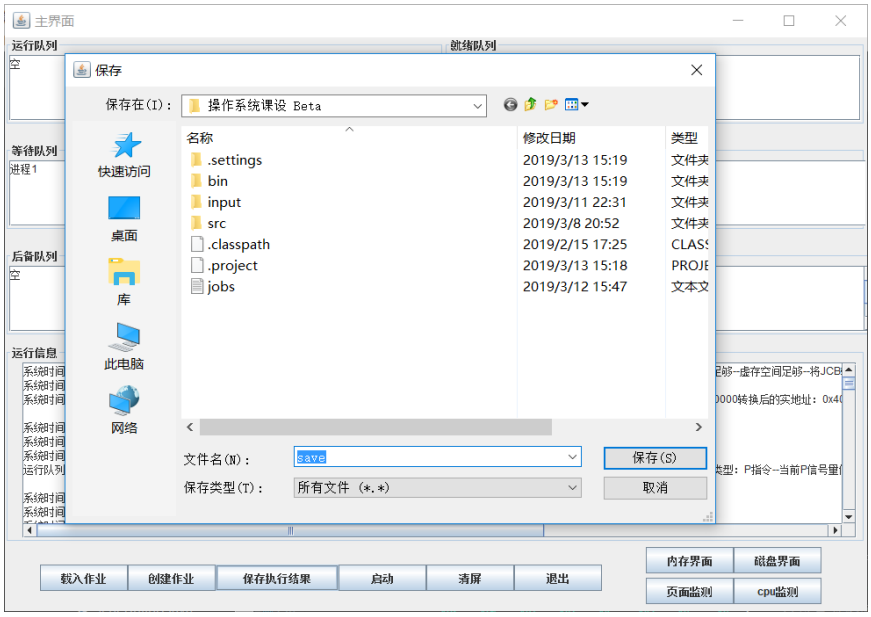
注意，当创建作业后，程序并不会直接将该作业加入到后备队列，需要读者手动点击“启动”按钮方可开始执行。

在添加指令时，请严格遵守上文所述的作业编写规范，不然所产生的错误我们不负责任。



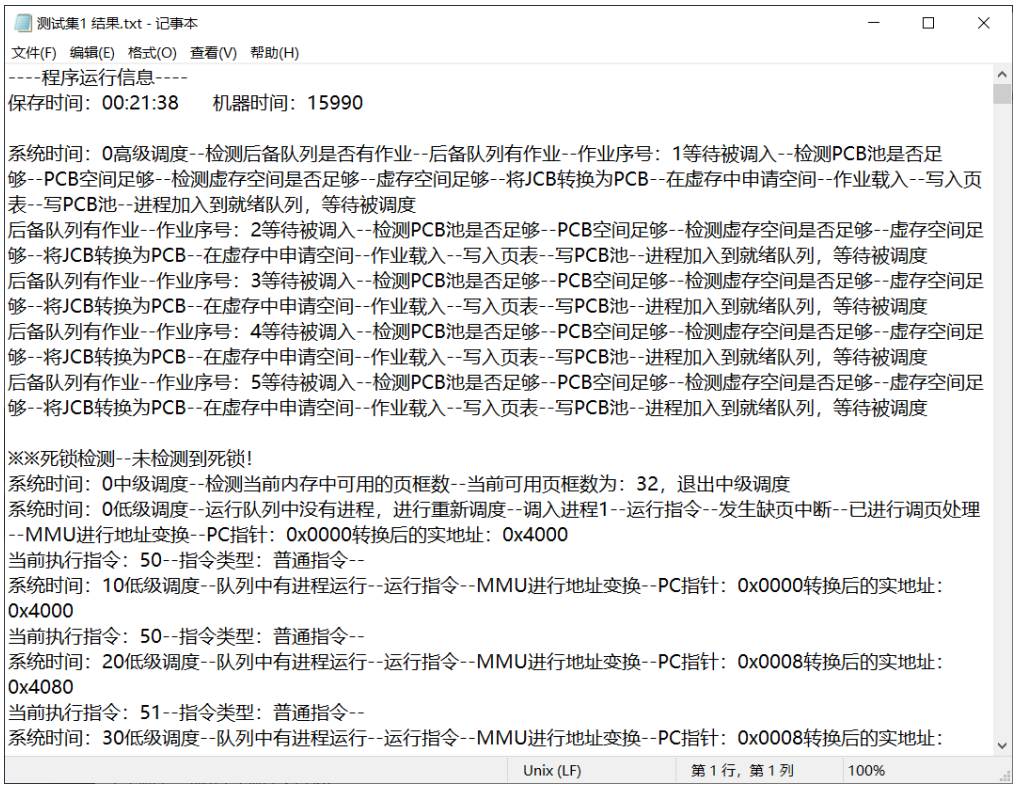
3. 保存执行结果

选择保存执行结果，保存到磁盘中。



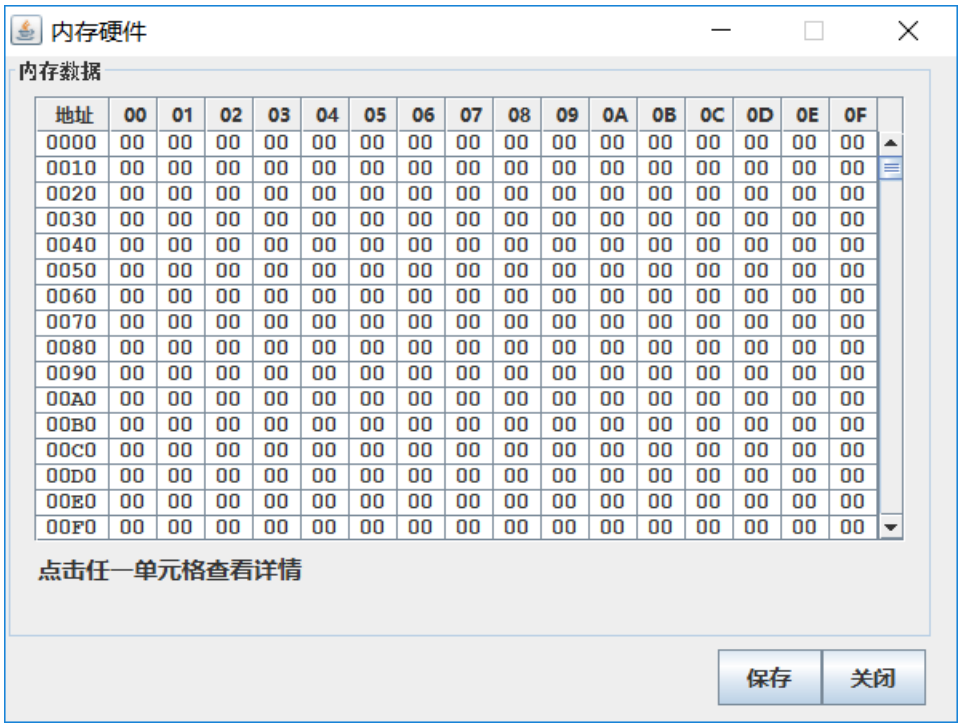
将运行信息保存到磁盘后，便可以打开该文件查看详情。

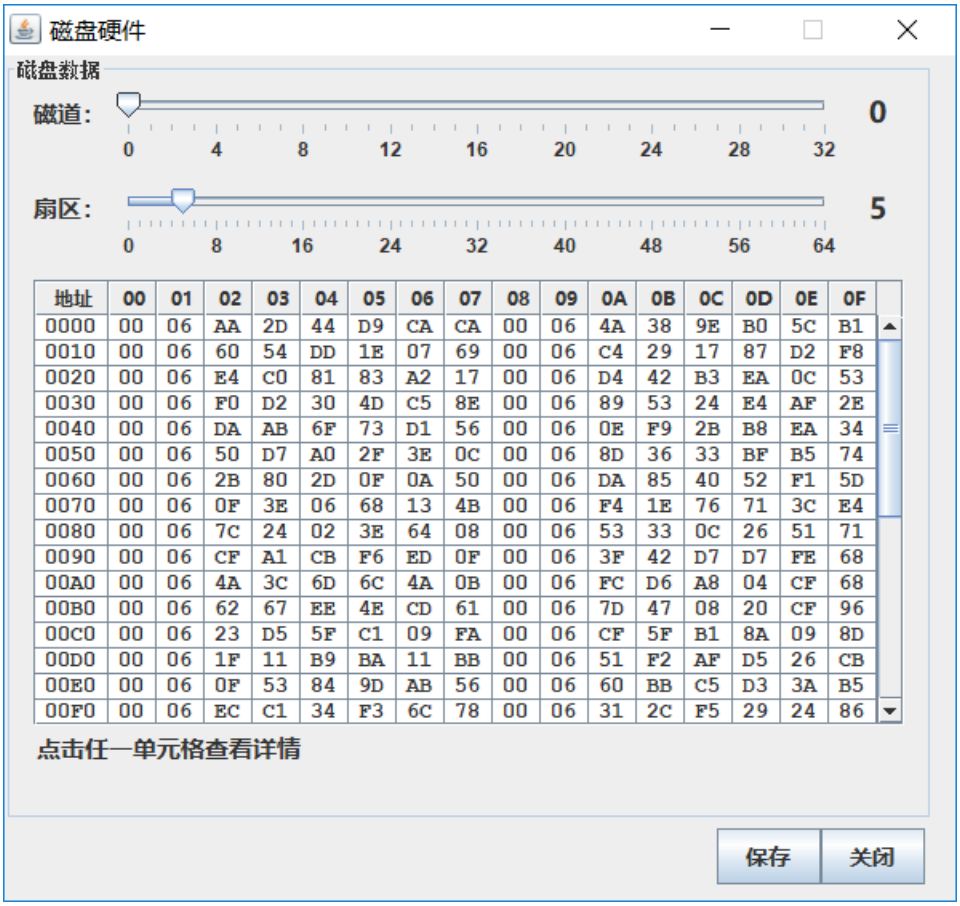
截图情况如下：

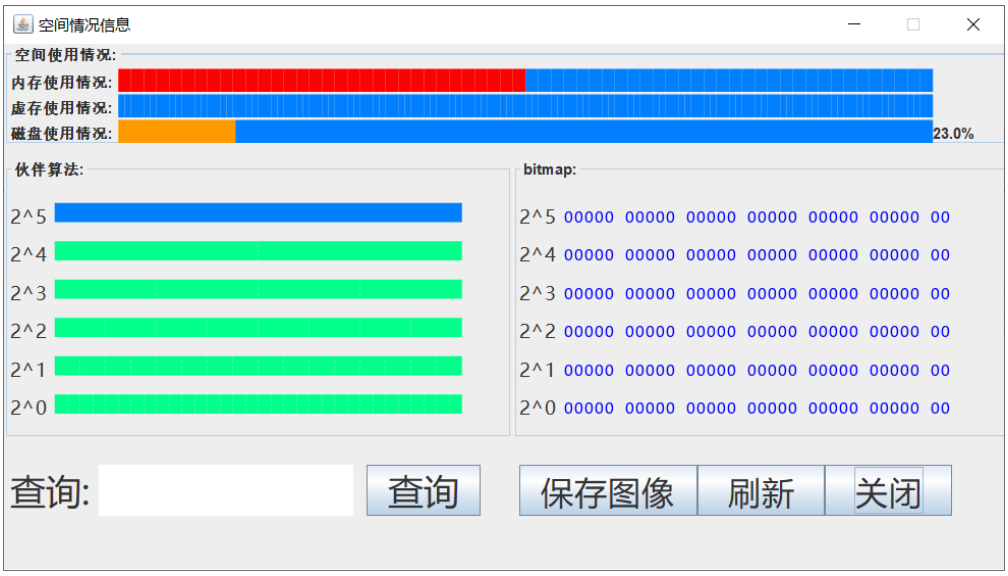


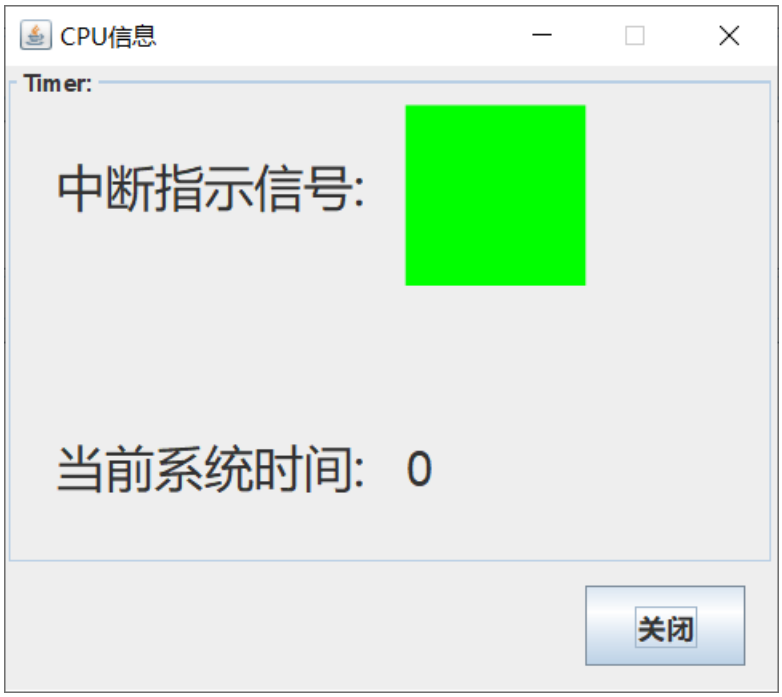
4. 查看各个界面信息

右下角可分别查看磁盘信息、内存信息、页面信息和 CPU 信息。




磁盘硬件
磁盘数据
磁道:
0
0 4 8 12 16 20 24 28 32
扇区:
5
0 8 16 24 32 40 48 56 64
地址 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000 00 06 AA 2D 44 D9 CA CA 00 06 4A 38 9E B0 5C B1
0010 00 06 60 54 DD 1E 07 69 00 06 C4 29 17 87 D2 F8
0020 00 06 E4 C0 81 83 A2 17 00 06 D4 42 B3 EA 0C 53
0030 00 06 F0 D2 30 4D C5 8E 00 06 89 53 24 E4 AF 2E
0040 00 06 DA AB 6F 73 D1 56 00 06 0E F9 2B B8 EA 34
0050 00 06 50 D7 A0 2F 3E 0C 00 06 8D 36 33 BF B5 74
0060 00 06 2B 80 2D 0F 0A 50 00 06 DA 85 40 52 F1 5D
0070 00 06 0F 3E 06 68 13 4B 00 06 F4 1E 76 71 3C E4
0080 00 06 7C 24 02 3E 64 08 00 06 53 33 0C 26 51 71
0090 00 06 CF A1 CB F6 ED 0F 00 06 3F 42 D7 D7 FE 68
00A0 00 06 4A 3C 6D 6C 4A 0B 00 06 FC D6 A8 04 CF 68
00B0 00 06 62 67 EE 4E CD 61 00 06 7D 47 08 20 CF 96
00C0 00 06 23 D5 5F C1 09 FA 00 06 CF 5F B1 8A 09 8D
00D0 00 06 1F 11 B9 BA 11 BB 00 06 51 F2 AF D5 26 CB
00E0 00 06 0F 53 84 9D AB 56 00 06 60 BB C5 D3 3A B5
00F0 00 06 EC C1 34 F3 6C 78 00 06 31 2C F5 29 24 86
点击任一单元格查看详情
保存
关闭


空间情况信息
空间使用情况:
内存使用情况:
虚存使用情况:
磁盘使用情况:
23.0%
伙伴算法:
2^5
2^4
2^3
2^2
2^1
2^0
bitmap:
2^5 00000 00000 00000 00000 00000 00000 00
2^4 00000 00000 00000 00000 00000 00000 00
2^3 00000 00000 00000 00000 00000 00000 00
2^2 00000 00000 00000 00000 00000 00000 00
2^1 00000 00000 00000 00000 00000 00000 00
2^0 00000 00000 00000 00000 00000 00000 00
查询:
查询
保存图像
刷新
关闭



三、裸机硬件部件仿真设计

1. 内存仿真设计

内存总大小为32KB，规定：
前16KB为内核区
后16KB为用户区
内核区存储的内容：核心栈+系统内核+进程所有PCB信息
综上所述，内存的组成结构为：
核心栈+系统内核（1页）、PCB池（31页）、用户区（32页）

2. 硬盘仿真设计

硬盘总大小为1MB（1024KB），规定：
前64KB为虚存区
再16KB为系统文件区
剩下的944KB都为文件区
综上所述，硬盘的组成结构为：
虚存区（128页）+系统文件区（32页）+文件区（1888页）

3. CPU仿真设计

CPU中包含以下部件：
地址寄存器PC
指令寄存器IR
页基址寄存器CR3
随机数生成器random
程序状态寄存器PSW
当前执行进程 PCB 的指针 currend_pcb

四、通用数据结构设计

1. **Page类设计**

private int page_num; //页号

private short[] data=new short[kernel.SINGLE_PAGE_SIZE/2];

//每页大小=512B=256个short类型

Page类中定义了对于页的基本操作，如读取写入清空等，支持对页内具体数据的读写操作和对页面整体的读写操作。

2. **PCB类设计**

PCB类中存储了该进程的全部信息，其中包含由作业转换为进程时存储的作业信息，以及运行过程中所需要的控制用信息。PCB中同时设置有下述数据成员的set和get函数进行读写操作。

PCB为动态生成，因进程创建而产生，因进程结束而消亡，没有静态的PCB对象可以使用。

PCB类中同时设计了进程的创建、撤销、阻塞、唤醒、挂起原语，用于在三级调度时进行状态切换。

3. **JCB类设计**

在JCB中存储了作业的相关信息，并设置有对应数据的set和get函数进行读写操作，在作业被创建为进程时将上述信息添加至PCB中使用。

4. **全局变量设计**

kernel类中定义了需要用到的静态变量

/*系统基本信息*/

public static int SINGLE_PAGE_SIZE=512;//每一页/块的大小

public static int MEMORY_SIZE=32*1024;//内存大小，32KB

public static int MEMORY_KERNEL_SPACE_SIZE=16*1024;//内存内核空间大小，16KB

public static int MEMORY_KERNEL_CORESTACKANDOSKERNEL_SIZE=512; //核心栈+系统内核大小，1页

public static int MEMORY_KERNEL_PCB_POOL_SIZE=31*512; //PCB池大小，31页

public static int MEMORY_USER_SPACE_SIZE=16*1024; //内存用户空间大小（页表、页框使用），16KB

public static int HARDDISK_SIZE=1*1024*1024; //硬盘空间大小，1MB

public static int HARDDISK_VIRTUAL_MEMORY_SIZE=64*1024; //虚存区大小，128页，64KB

public static int HARDDISK_SYSTEMFILE_SIZE=16*1024; //系统文件大小，32页，16KB

public static int HARDDISK_FILE_SPACE_SIZE=944*1024; //文件区大小，1888页，944KB

public static int HARDDISK_CYLINDER_NUM=32;//磁盘磁道数

public static int HARDDISK_SECTOR_NUM=64; //磁盘扇区数

public static int HARDDISK_PAGE_SIZE=512;//磁盘每页/块大小

public static int SINGLE_INSTRUCTION_SIZE=8; //单条指令的大小

public static int INSTRUCTIONS_PER_PAGE=SINGLE_PAGE_SIZE/SINGLE_INSTRUCTION_SIZE;//每一页的指令数目

public static int INTERRUPTION_INTERVAL=10; //系统发生中断的间隔

```
public static int SYSTEM_TIME=0;          //系统内时间
public static void SystemTimeAdd()
{kernel.SYSTEM_TIME+=kernel.INTERRUPTION_INTERVAL;}  //系统时间自增
public static int
TLB_LENGTH=kernel.MEMORY_USER_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE/2;
//TLB快表的长度，16
/*系统基本信息*/

/*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*/

/*Process PSW 程序状态字*/
public final static byte PSW_KERNEL_STATE=0;     //管态
public final static byte PSW_USER_STATE=1;          //目态
/*Process PSW 程序状态字*/

/*硬件初始化需要变量*/
public static String MEMORYFILE_PATHNAME="./input/memory.dat";       //内存文件
地址
public static String HARDDISKFILE_PATHNAME="./input/harddisk.dat";    //硬盘文件
地址
public static String CPUFILE_PATHNAME="./input/cpu.dat";              //CPU文件地址
/*硬件初始化需要变量*/

/*60条指令*/
public static int GetInstructionType(int instruction){
      //获取指令类型
      if(instruction>=0&&instruction<=9)return 1;
      if(instruction>=10&&instruction<=19)return 2;
      if(instruction>=20&&instruction<=29)return 3;
      if(instruction>=30&&instruction<=39)return 4;
      if(instruction>=40&&instruction<=49)return 5;
      if(instruction>=50&&instruction<=59)return 6;
      return -1;}
public static int GetInstructionTime(int instruction)
{/*获取指令执行所需要时间*/return (instruction%10)*10+20;}
public static int[] MUTEX= {-4,-3,-2,-1,0,1,2,3,4,5};        //临界区信号量,10个
public static int[] SYSTEM_RESOURCE= {0,1,2,3,4,5,6,7,8,9};     //系统资源量，10个
public static int GetUseResourceNum(int instruction)
{/*获取该指令所申请、释放的PV、资源所在的数组序号*/return instruction%10;}
```

```
/*60 条指令*/
```

五、代码结构与函数说明

1. PCB类

```
public class 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，page_table[i][0]为进程的页号，从0开始编号；page_table[i][1]为对应的物理页号
 private ArrayList<Integer> instructions=new ArrayList<Integer>();    //该进程所有的指令

 private short in_page_num=0;             //该PCB所在的页号
 private short pool_location=-1;          //该PCB在PCB池的位置
 private int total_runtime=0;          //总共运行的时间
 public int ins_runtime=0;                //指令的执行总时间
 public boolean if_in_p=false;        //是否处于P状态
 public boolean if_p_success=true; //P是否成功

 public void ProcessCreate()
      //进程原语：进程创建

 public void ProcessCancel()
      //进程原语：进程撤销

 public void ProcessWait()
      //进程原语：进程阻塞

 public void ProcessSuspend()
      //进程原语：进程挂起
```

```
public void ProcessWake()
    //进程原语：进程唤醒
public void RefreshPriority()
    //随机生成-5~5之间的数字 更新进程的优先级

public int GetTotalNeedTime()
    //获取此PCB运行完成总共需要的时间

public void RefreshTotalRuntime()
    //刷新已经运行的时间

public void RefreshTimeslice()
    //根据更新后的优先级更新该进程能够获得的时间片

public void AddRuntime()
    //增加进程已经运行时间runtime

public void AddTotalRuntime()
    //增加进程已经运行的总时间

public void RefreshCounter()
    //更新进程的时间片余额

public void AddCurrentInstructionNo()
    //更新current_instruction_no，当前正要运行的指令编号

public int GetTotalRunTime()
    //获取该进程总共的运行时间

public void AddPageTable(short table_data)
    //增加页表项

public void EditPageTable(short line,short data_1,short data_2)
    //修改页表

public short CheckPageTable(short line_no)
    //查询第line_no项表的值

public void AddInstruction(int instruction)
    //添加指令

public boolean IfPCBInUpper()
    //判断该PCB是否位于页的上半段
```

```
public short PCBStartAddress()
    //获取该PCB在页中的起始地址
public void WritePCBToMemory()
    //将此PCB写入到内存的PCB池中

public short GetPid()
    //获取此进程PID

public void SetPid(short pid)
    //设置此进程PID

public short GetState()
    //获取此进程状态

public void SetState(short state)
    //设置此进程状态

public short GetPriority()
    //获取此进程优先级

public void SetPriority(short priority)
    //设置此进程优先级

public int GetJobIntime()
    //获取此作业创建时间

public void SetJobIntime(int job_intime)
    //设置此作业创建时间

public int GetProcessIntime()
    //获取此进程创建时间

public void SetProcessIntime(int process_intime)
    //修改此进程创建时间

public int GetEndTime() {
    //获取此进程/作业结束时间

public void SetEndTime(int end_time)
    //设置此进程/作业结束时间

public short GetTimeslice()
    //获取此进程时间片长度
```

```
public int GetRuntime()
    //获取此进程运行时间长度
public void SetRuntime(int runtime)
    //获取此进程时间片长度

public int GetCounter()
    //获取此进程时间片余额

public void SetCounter(int counter)
    //设置此进程时间片余额

public byte GetPSW()
    //获取此进程状态字

public void SetPSW(byte pSW)
    //设置此进程状态字

public short GetCurrentInstructionNo()
    //获取此进程当前指令序号

public void SetCurrentInstructionNo(short current_instruction_no)
    //设置此进程当前指令序号
}
public short GetInstruction_num()
    //获取此进程包含指令数

public void SetInstructionNum(short instruction_num)
    //设置此进程包含指令数

public short GetPagesNum()
    //获取此进程包含页面数

public void SetPagesNum(short pages_num)
    //设置此进程包含指令数

public ArrayList<Integer> getInstructions()
    //获取此进程指令序列

public void setInstructions(ArrayList<Integer> instructions)
    //设置此进程指令序列

public void SetInPageNum(short num)
    //设置此PCB所在页号
```

```
public short GetInPageNum()
    //获得此PCB所在页号
public short GetPoolLocation()
    //获取此进程在PCB池中位置

public void SetPoolLocation(short pool_location)
    //设置此进程在PCB池中位置

public int GetTotal_runtime()
    //获取此进程总运行时间

public void SetTotal_runtime(int total_runtime)
    //设置此进程总运行时间
}
```

2. JCB类

```
public class 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
 private short in_page_num=0;              //该JCB所在的页号

 public short CalculatePagesNum()
     //计算作业所占用的页面数目

 public short GetProcessNeedPage()
     //获取作业所需页面数

 public short GetPriority()
     //获取作业优先级

 public void SetPriority(short priority)
     //设置作业优先级

 public int GetJobIntime()
     //获取作业进入时间

 public void SetJobIntime(int job_intime)
     //设置作业进入时间
```

```
public short GetInstruction_num()
    //获取作业进入时间

public void SetInstructionNum(short instruction_num)
    //设置作业指令序列

public short GetPagesNum()
    //获取作业所需页面数

public void SetPagesNum(short pages_num)
    //设置作业所需页面数

public ArrayList<Short> GetAll_Instructions()
    //获取作业指令序列

public void SetAll_Instructions(ArrayList<Short> all_instructions)
    //设置页面指令序列

public void SetInPageNum(short num)
    //设置作业所在页面号

public short GetInPageNum()
    //获取作业所在页面号

public short GetJobid()
    //获取作业ID

public void SetJobid(short job_id)
    //设置作业ID
}
```

3. 进程管理

```
public class ProcessModule
{
public static ProcessModule process_module=new ProcessModule();

public ArrayList<PCB> all_queue=new ArrayList<PCB>();      //所有进程链表
public ArrayList<PCB> running_queue=new ArrayList<PCB>();      //运行队列
@SuppressWarnings("unchecked")
public ArrayList<PCB> [][]ready_queue=new ArrayList[2][140];      //就绪队列
private int active=0;      //active指针
private int expired=1;     //expired指针
private boolean []ready_queue_bitmap=new boolean[140];              //就绪队列位图，范围
0 - 139，共140位
```

```
public ArrayList<PCB> waiting_queue=new ArrayList<PCB>();          //等待队列
public ArrayList<PCB> suspend_queue=new ArrayList<PCB>();          //挂起队列
public ArrayList<PCB> end_queue=new ArrayList<PCB>();              //已经运行结束的进程

private boolean[] PCB_pool_usage=new boolean[kernel.MEMORY_KERNEL_PCB_POOL_SIZE/(kernel.SINGLE_PAGE_SIZE/2)];
    //PCB池使用情况

ProcessModule()//构造函数

public PCB TurnJCBToPCB(JCB jcb)
    //将JCB变换为PCB

public void TransferJobCodeToSwapArea(JCB jcb,String apply)
    //将指定作业的程序段存入虚存中
    //jcb为作业控制块，apply为申请到的虚存空间分配字符串

public void WriteProcessPageTable(PCB pcb,String apply)
    //将申请到的虚存页面写入到进程的页表中

public short GetFreePCBNumInPool()
    //获取PCB池中可用的PCB数量

public void DeletePCBInPool(PCB pcb)
    //将某一个PCB从PCB池中删除

public short ApplyOnePCBInPool()
    //在PCB池中申请一个PCB

public void AddToPCBPool(PCB pcb)
    //将PCB加入到PCB池中

public void TransferProcessToRunningQueue(PCB pcb)
    //将进程移入运行队列
    //遍历就绪队列、等待队列、挂起队列，将进程移出，只加入到运行队列
    //移入运行队列

public void TransferProcessToReadyQueue(PCB pcb,boolean if_active)
    //将进程移入就绪队列
    //遍历运行队列、等待队列、挂起队列，将进程移出，只加入到就绪队列
    //移入就绪队列

public void TransferProcessToWaitQueue(PCB pcb)
```

```
    //将进程移入等待队列
    //遍历运行队列、就绪队列、挂起队列，将进程移出，只加入到等待队列
    //移入等待队列

public void TransferProcessToSuspendQueue(PCB pcb)
    //将进程移入挂起队列
    //遍历运行队列、就绪队列、等待队列，将进程移出，只加入到挂起队列
    //移入挂起队列

public void TransferProcessToEndQueue(PCB pcb)
    //将进程移入完成队列
    //遍历运行队列、就绪队列、等待队列、挂起队列，将进程移出，加入到完成队列
    //移入完成队列

public void RefreshReadyQueueBitmap()
    //刷新就绪队列的bitmap

public void RefreshActiveExpired()
    //刷新active和expired指针

public boolean IfPageInMemory(short page_num)
    //检测需要的页是否在内存中
    //检测是否发生缺页中断

public void ChangePageTable(short ori_page_num,short changed_page_num)
    //将持有原来页的进程的页表更新

public void SolveMissingPage(PCB pcb,short need_page_num)
    //缺页中断的处理，need_page_num为需要的在外存中的页的页号

public boolean IfRunOver(PCB pcb)
    //检测某进程是否运行完毕

public boolean IfTimeSliceOver(PCB pcb)
    //检测时间片是否用完

public void AddToEndQueue(PCB pcb)
    //将作业加入到结束队列

public boolean IsRunningQueueEmpty()
    //检测运行队列是否为空

public boolean IsReadyQueueEmpty()
    //检测就绪队列是否为空
```

```
public boolean IsWaitQueueEmpty()
    //检测等待队列是否为空

public boolean IsSuspendQueueEmpty()
    //检测挂起队列是否为空

public int GetActivePoint()
    //获取active指针

public int GetExpiredPoint()
    //获取expired指针

public PCB GetPCBWithID(short id)
    //根据进程ID获取PCB

public void RunType1(PCB pcb)
    //类型1指令的处理

public void RunType2(PCB pcb)
    //类型2指令的处理

public void RunType3(PCB pcb)
    //指令类型3的处理

public void RunType4(PCB pcb)
    //指令类型4的处理

public void RunType5(PCB pcb)
    //指令类型5的处理

public void RunType6(PCB pcb)
    //类型6指令的处理
}
```

4. 调度算法

```
public class Scheduling extends Thread
{
 public static Scheduling sch=new Scheduling();

 public boolean if_wait=false;       //当前是否正处于等待态
 public int wait_time=-1;            //等待态还剩余时间

 public void run()//线程执行函数，负责进行调度
```

```
public void UIRefresh()
    //不同UI的刷新

public void SuspendProcessWithPageNum(short num)
    //检测某一页所关联的进程，并将该进程加入到挂起态

public void HighLevelScheduling()
    //高级调度

public void MiddleLevelScheduling()
    //中级调度

public void LowLevelScheduling()
    //低级调度
}
```

5. 作业调度

```
public class JobModule
{
 public static JobModule job_module=new JobModule();

 private int job_id=0;      //作业号
 private int job_page_location_start=(kernel.MEMORY_SIZE+kernel.HARDDISK_VIRTUAL_MEMORY_SIZE+kernel.HARDDISK_SYSTEMFILE_SIZE)/kernel.SINGLE_PAGE_SIZE;
 //作业存储区的地址开始
 private int job_page_location_end=(kernel.MEMORY_SIZE+kernel.HARDDISK_SIZE)/kernel.SINGLE_PAGE_SIZE;
 //作业存储区的地址结束
 private int next_to_pcb=0;    //下一个将要读入的作业序号
 private int write_job_page_num=job_page_location_start;        //写入作业的编号
 public ArrayList<JCB> job_list=new ArrayList<JCB>();          //作业后备队列
 ArrayList<JCB> all_jcb=new ArrayList<JCB>();              //存储所有JCB的数组

 public short GetJobNum()
     //获得当前磁盘中的作业总数量
     short job_num=0;

 public int GetCurrentCreateJobID()
     //获取当前要创建的作业的编号

 public void SaveJobToHardDisk(JCB jcb)
```

```
    //将作业保存到外存，d1是JCB数据块，d2是包含所有指令的ArrayList，每条指令占
8字节

    public void GetJCBFromFile(File f)
        //从文件中读取所有的JCB

    public ArrayList<JCB> GetAllJCB()
        //获取所有的JCB

    public void NextJob()
        //已经读取完一个作业，进入到下一个作业

    public int GetNextJobNum()
        //获取下一个将要被创建的JCB的序号

    public boolean IsAllJobToProcess()
        //检测是否还有作业没有变成进程

    public void RefreshJobList()
        //刷新作业后备队列

    public boolean IsJobListEmpty()
        //查看作业后备队列是否为空
    }
```

6. 死锁检测

```
public class DeadLock
{
    public static DeadLock dl=new DeadLock();//该类的静态对象
    @SuppressWarnings("unchecked")
    public ArrayList<PCB> []PV_apply=new ArrayList[10];    //PV信号量当前的占用情况（P
过该资源但是没有释放的进程）
    private int m=10;   //向量长度m，每类资源中可供分配的资源数目
    private int n=500;  //向量长度n，进程个数

    public int []Available=new int[m];              //每类资源中可供分配的资源数目
    public int [][]Allocation=new int[n][m];        //已分配给每个进程的每类资源数目
    public int [][]Request=new int[n][m];           //每个进程对每类资源的申请数目
    private int []Work=new int[m];                  //长度为m的工作向量
    private boolean []finish=new boolean[n];        //长度为n的布尔型向量

    DeadLock()
    {
```

```
        for(int i=0;i<10;i++)
            PV_apply[i]=new ArrayList<PCB>();
        InitAvailable();
        InitAllocation();
        InitRequest();
        ResetWork();
        InitFinish();
    }

    public void InitAvailable()
        //初始化Available向量

    public void InitAllocation()
        //初始化Allocation向量

    public void InitRequest()
        //初始化Request向量

    public void ResetWork()
        //设置Work向量

    public void InitFinish()
        //初始化finish向量

    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

    public void Process_V_Mutex(PCB pcb,int num)
        //某个进程V第num个mutex信号量

    public boolean Process_Apply_Resource(PCB pcb,int num)
        //某个进程申请某个资源
        //返回值为是否能够申请成功

    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 boolean IfAllocationLineEmpty(int k)
        //判断Allocation向量的第k行是否为空
```

```
public boolean Step3_find_k_value(int k)
    //第3步，找到符合条件的k值
    //满足条件：finish[k]==false && Request[k , *] <= Work[*]
    //判断当前输入的k值是否符合条件

public ArrayList<PCB> CheckDeadLock()
    //死锁检测
```

六、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：

```
System.out.println(kernel.SINGLE_PAGE_SIZE);
System.out.println(kernel.MEMORY_SIZE);
System.out.println(kernel.MEMORY_KERNEL_SPACE_SIZE);
System.out.println(kernel.MEMORY_KERNEL_CORESTACKANDOSKERNEL_SIZE);
System.out.println(kernel.MEMORY_KERNEL_PCB_POOL_SIZE);
System.out.println(kernel.MEMORY_USER_SPACE_SIZE);
System.out.println(kernel.HARDDISK_SIZE);
System.out.println(kernel.HARDDISK_VIRTUAL_MEMORY_SIZE);
```

输出的结果如下：

```
<terminated> M
16384
512
15872
16384
1048576
65536
```

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：

```
PCB t=new PCB();
t.SetPid((short) 100);
System.out.println(t.GetPid());
t.SetCounter(50);
System.out.println(t.GetCounter());
t.SetCurrentInstructionNo((short) 10);
System.out.println(t.GetCurrentInstructionNo());
```

输出的结果如下：

```
<terminated> M
100
50
10
```

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：

```
JCB t=new JCB();
t.SetInPageNum((short) 10);
System.out.println(t.GetInPageNum());
t.SetInstructionNum((short) 100);
System.out.println(t.GetInstruction_num());
t.SetJobid((short) 2333);
System.out.println(t.GetJobid());
```

测试结果如下：

```
<terminated
10
100
2333
```

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：

```
PCB t1=new PCB();
PCB t2=new PCB();
t1.SetPid((short) 1);
t2.SetPid((short) 2);
DeadLock.dl.Process_Apply_Resource(t1,1);
DeadLock.dl.Process_Apply_Resource(t2,2);
DeadLock.dl.Process_Apply_Resource(t2,2);
DeadLock.dl.Process_Apply_Resource(t2,1);
DeadLock.dl.Process_Apply_Resource(t1,2);
ArrayList<PCB> result=DeadLock.dl.CheckDeadLock();
if(result.size()!=0)
    System.out.println("检测到死锁");
else
    System.out.println("未发生死锁");
```

执行结果为：

```
<terminated> Mai
检测到死锁
```

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。

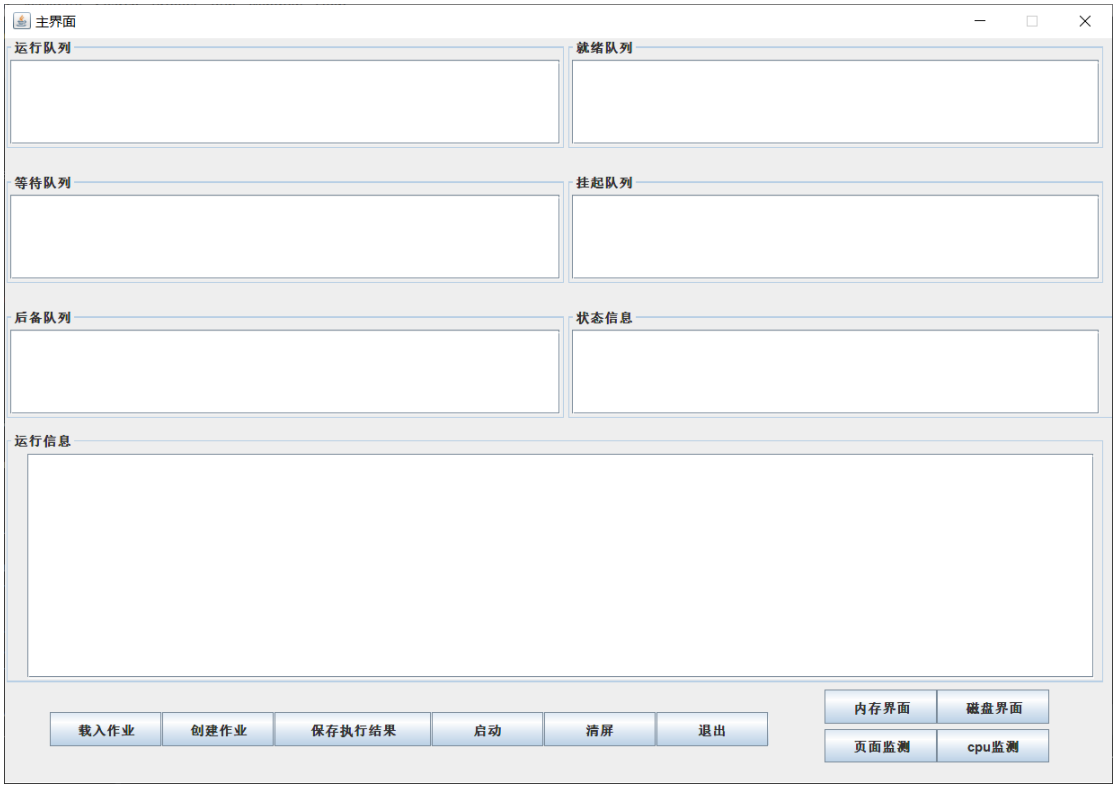
同时，由于程序的函数实现较多，逻辑上较为复杂。读者若不能够理解程序的含义，建议通过 Eclipse 自带的单步调试功能进行测试。

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：

```
public class Main {
    public static void main(String[] args)
    {
        MainUI.main_ui.SetVisible(true);
    }
}
```

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

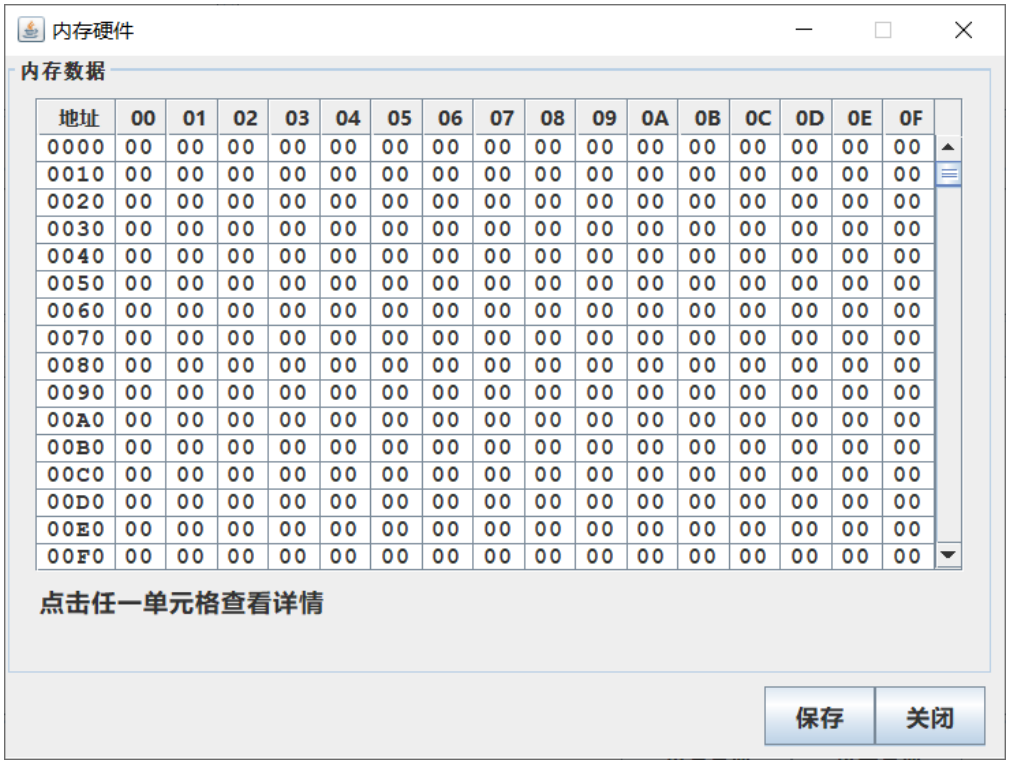
内存界面：打开内存数据查看界面，可以查看当前内存中所有的数据信息

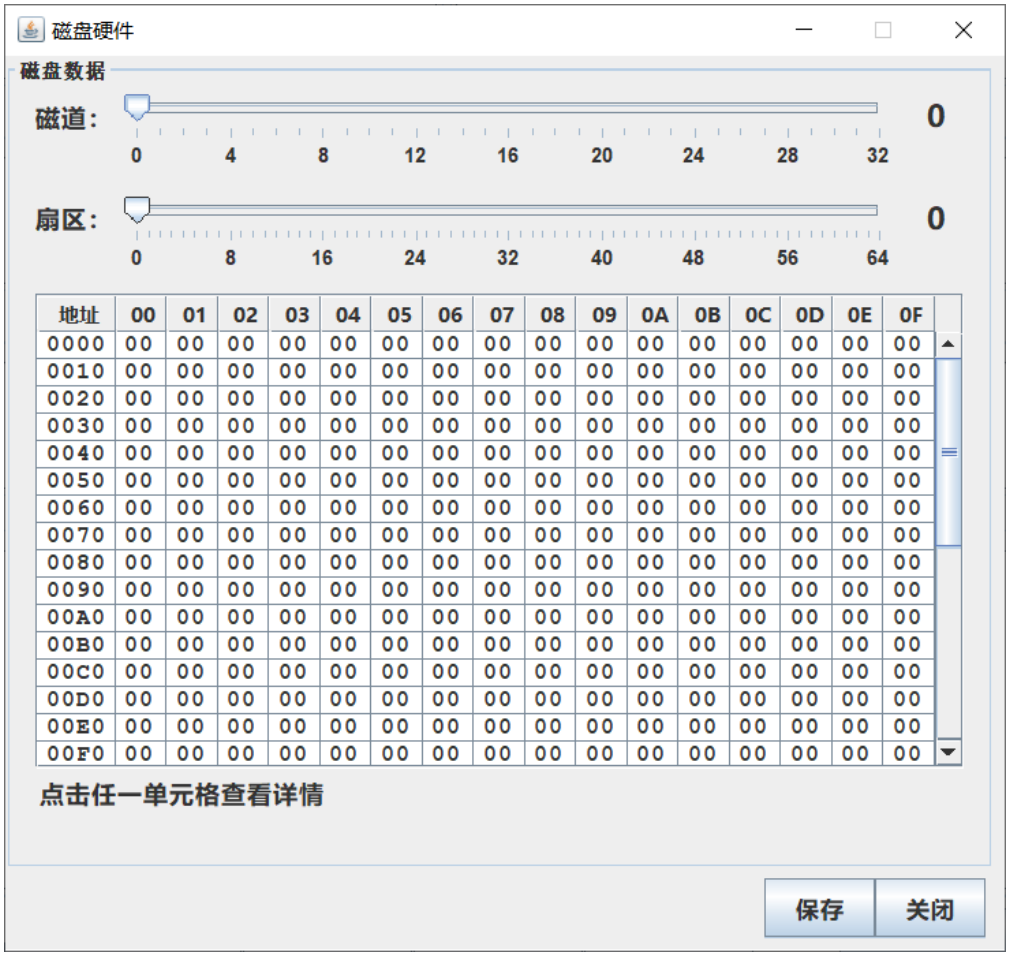
磁盘界面：打开磁盘数据查看界面，可以查看当前磁盘中所有的数据信息

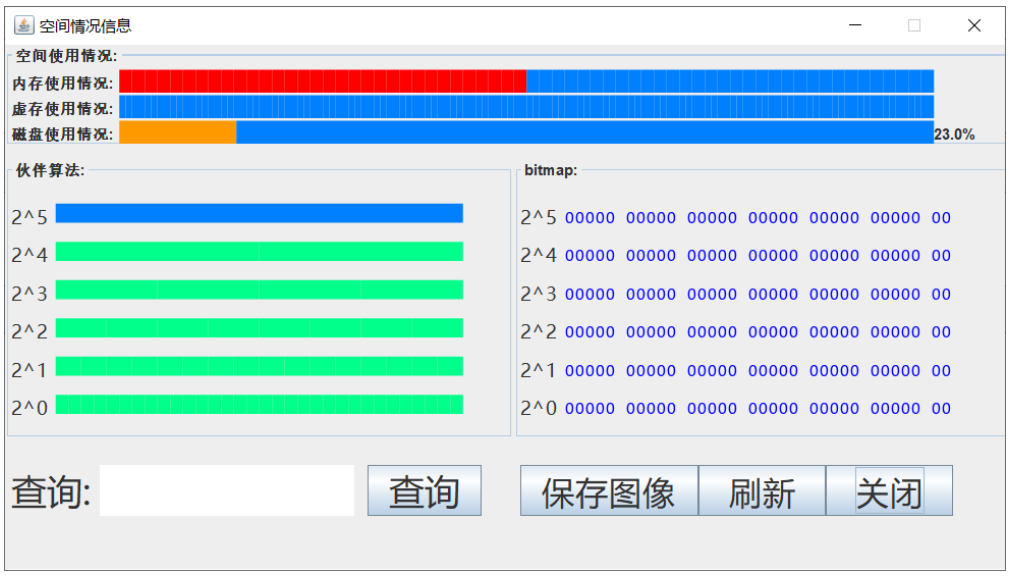
页面检测：打开页面管理的检测界面，可以查看内存、虚存、磁盘的空间占用情况和伙伴算法的运行原理和 bitmap

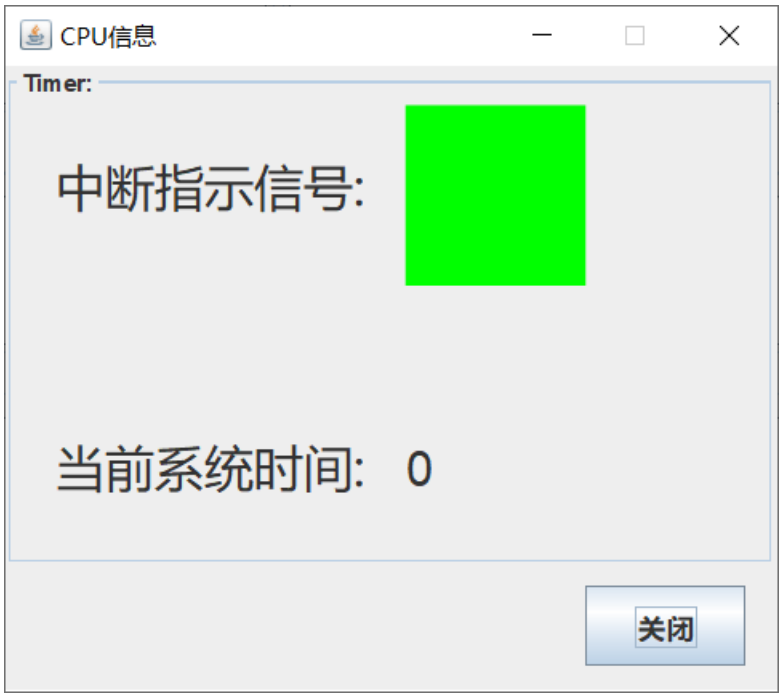
CPU 检测：打开 CPU 检测窗体，可以检测此时 CPU 的具体中断信息与当前系统时间。

这些窗体的截图如下：




磁盘硬件
磁盘数据
磁道:
0
0 4 8 12 16 20 24 28 32
扇区:
0
0 8 16 24 32 40 48 56 64
地址 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
点击任一单元格查看详情
保存
关闭


空间情况信息
空间使用情况:
内存使用情况:
虚存使用情况:
磁盘使用情况:
23.0%
伙伴算法:
2^5
2^4
2^3
2^2
2^1
2^0
bitmap:
2^5 00000 00000 00000 00000 00000 00000 00
2^4 00000 00000 00000 00000 00000 00000 00
2^3 00000 00000 00000 00000 00000 00000 00
2^2 00000 00000 00000 00000 00000 00000 00
2^1 00000 00000 00000 00000 00000 00000 00
2^0 00000 00000 00000 00000 00000 00000 00
查询:
查询
保存图像
刷新
关闭



（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：

系统时间：0高级调度--检测后备队列是否有作业--后备队列有作业--作业序号：1等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB转换为PCB--在虚存中申请空间--作业载入--写入页表--写PCB池--进程加入到就绪队列，等待被调度

系统时间：2000中级调度--检测当前内存中可用的页框数--当前可用页框数为：27，退出中级调度

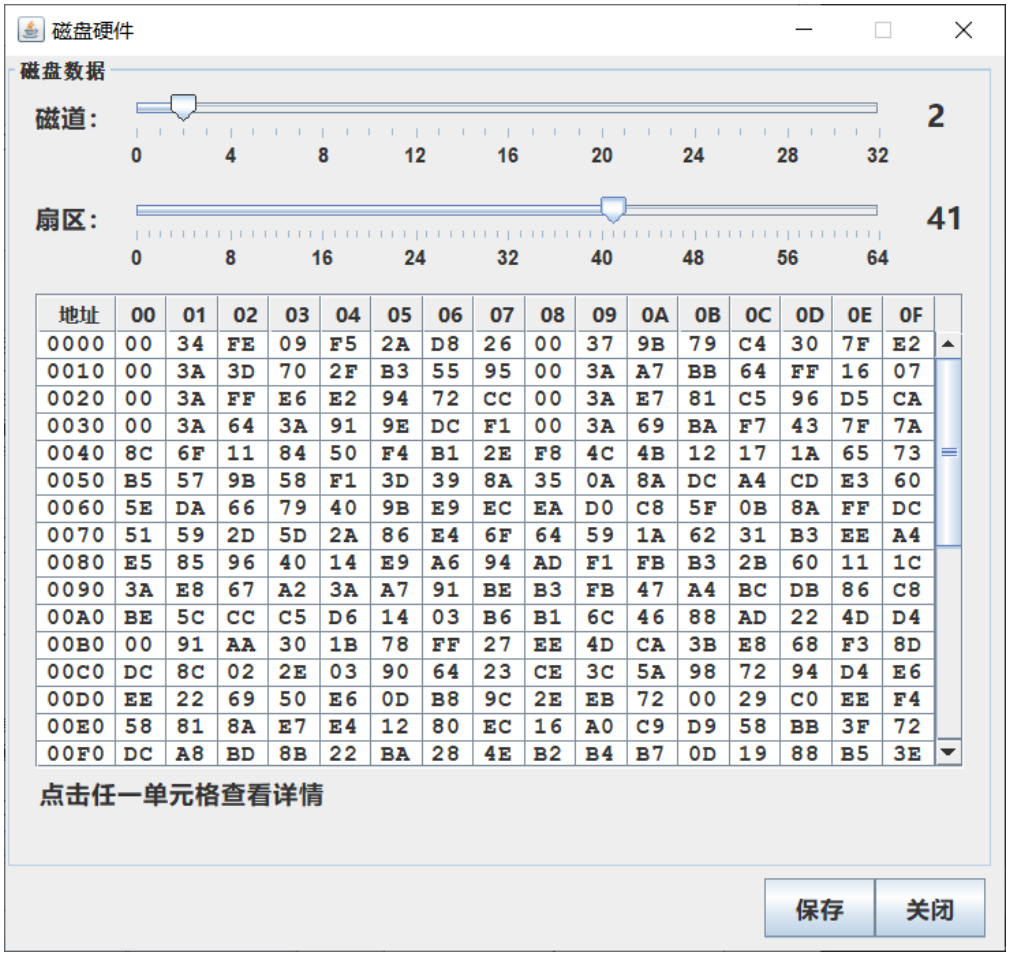
系统时间：2000低级调度--队列中有进程运行--运行指令--没有发生缺页中断--当前访问页表：0-34
MMU进行地址变换--PC指针：0x0018H，转换后的实地址：0x4418H
当前执行指令：58--指令类型：普通指令--PSW：用户态

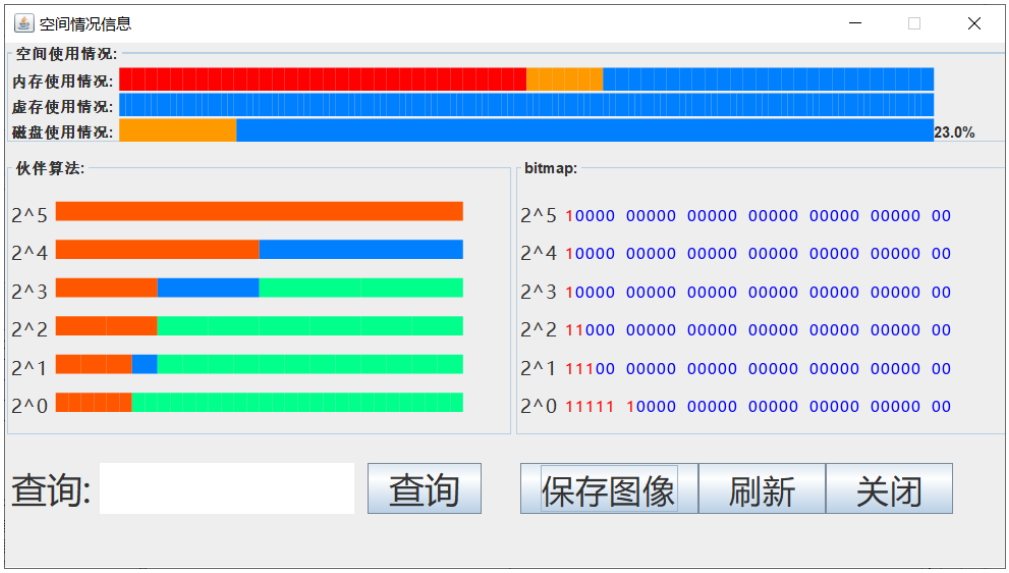
（七）指令运行

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：


主界面
运行队列
空
就绪队列
active:
expired:
等待队列
空
挂起队列
空
后备队列
空
状态信息
当前进程：1
优先级：6
时间片长度：80
已经运行时间：60
运行信息
系统时间：0高级调度--检测后备队列是否有作业--后备队列有作业--作业序号：1等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB
后备队列有作业--作业序号：2等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB转换为PCB--在虚存中申请空间--作业载入--写入页
后备队列有作业--作业序号：3等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB转换为PCB--在虚存中申请空间--作业载入--写入页
后备队列有作业--作业序号：4等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB转换为PCB--在虚存中申请空间--作业载入--写入页
后备队列有作业--作业序号：5等待被调入--检测PCB池是否足够--PCB空间足够--检测虚存空间是否足够--虚存空间足够--将JCB转换为PCB--在虚存中申请空间--作业载入--写入页
※※死锁检测--未检测到死锁！
系统时间：0中级调度--检测当前内存中可用的页框数--当前可用页框数为：32，退出中级调度
系统时间：0低级调度--运行队列中没有进程，进行重新调度--调入进程1--运行指令--发生缺页中断--已进行调页处理--MMU进行地址变换--PC指针：0x0000转换后的实地址：0x40
当前执行指令：50--指令类型：普通指令--
系统时间：10低级调度--队列中有进程运行--运行指令--MMU进行地址变换--PC指针：0x0000转换后的实地址：0x4000
载入作业
创建作业
保存执行结果
启动
清屏
退出
内存界面
磁盘界面
页面监测
cpu监测


磁盘硬件
磁盘数据
磁道：
2
0 4 8 12 16 20 24 28 32
扇区：
41
0 8 16 24 32 40 48 56 64
地址 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000 00 34 FE 09 F5 2A D8 26 00 37 9B 79 C4 30 7F E2
0010 00 3A 3D 70 2F B3 55 95 00 3A A7 BB 64 FF 16 07
0020 00 3A FF E6 E2 94 72 CC 00 3A E7 81 C5 96 D5 CA
0030 00 3A 64 3A 91 9E DC F1 00 3A 69 BA F7 43 7F 7A
0040 8C 6F 11 84 50 F4 B1 2E F8 4C 4B 12 17 1A 65 73
0050 B5 57 9B 58 F1 3D 39 8A 35 0A 8A DC A4 CD E3 60
0060 5E DA 66 79 40 9B E9 EC EA D0 C8 5F 0B 8A FF DC
0070 51 59 2D 5D 2A 86 E4 6F 64 59 1A 62 31 B3 EE A4
0080 E5 85 96 40 14 E9 A6 94 AD F1 FB B3 2B 60 11 1C
0090 3A E8 67 A2 3A A7 91 BE B3 FB 47 A4 BC DB 86 C8
00A0 BE 5C CC C5 D6 14 03 B6 B1 6C 46 88 AD 22 4D D4
00B0 00 91 AA 30 1B 78 FF 27 EE 4D CA 3B E8 68 F3 8D
00C0 DC 8C 02 2E 03 90 64 23 CE 3C 5A 98 72 94 D4 E6
00D0 EE 22 69 50 E6 0D B8 9C 2E EB 72 00 29 C0 EE F4
00E0 58 81 8A E7 E4 12 80 EC 16 A0 C9 D9 58 BB 3F 72
00F0 DC A8 BD 8B 22 BA 28 4E B2 B4 B7 0D 19 88 B5 3E
点击任一单元格查看详情
保存
关闭

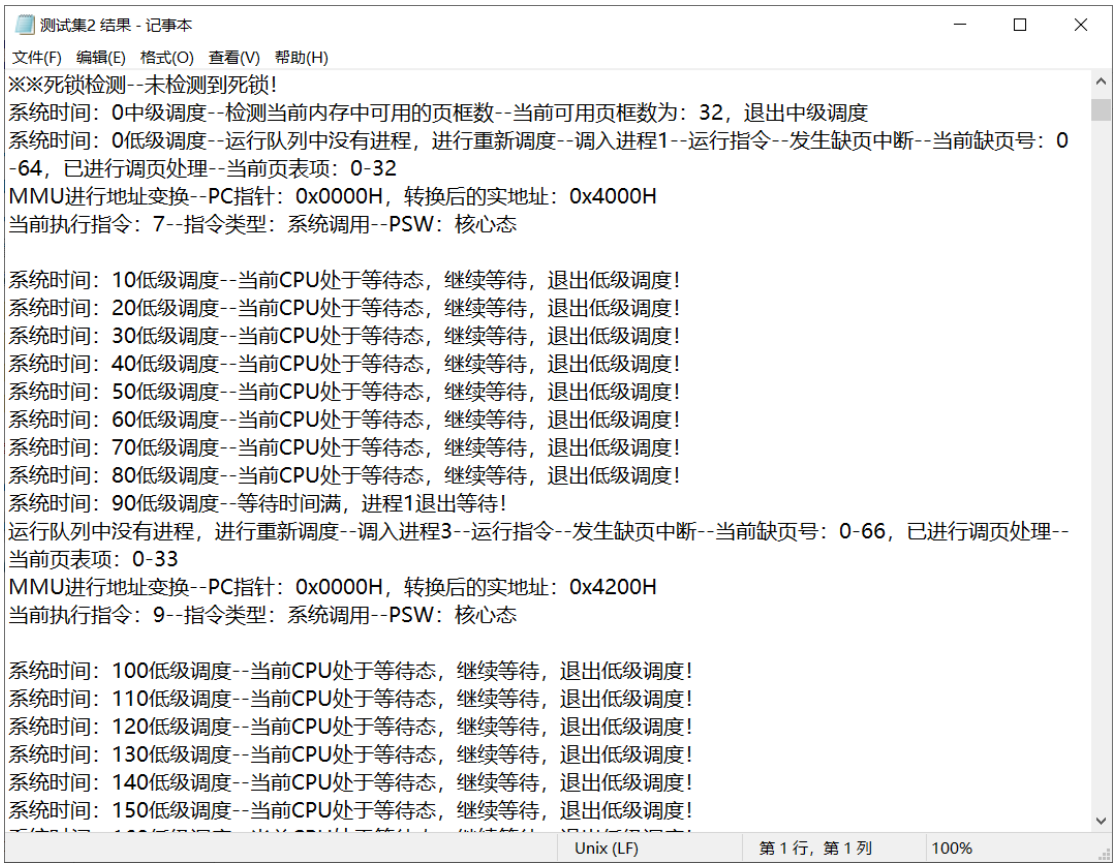


将程序的运行结果输出保存到文件，文件内容如下：

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。

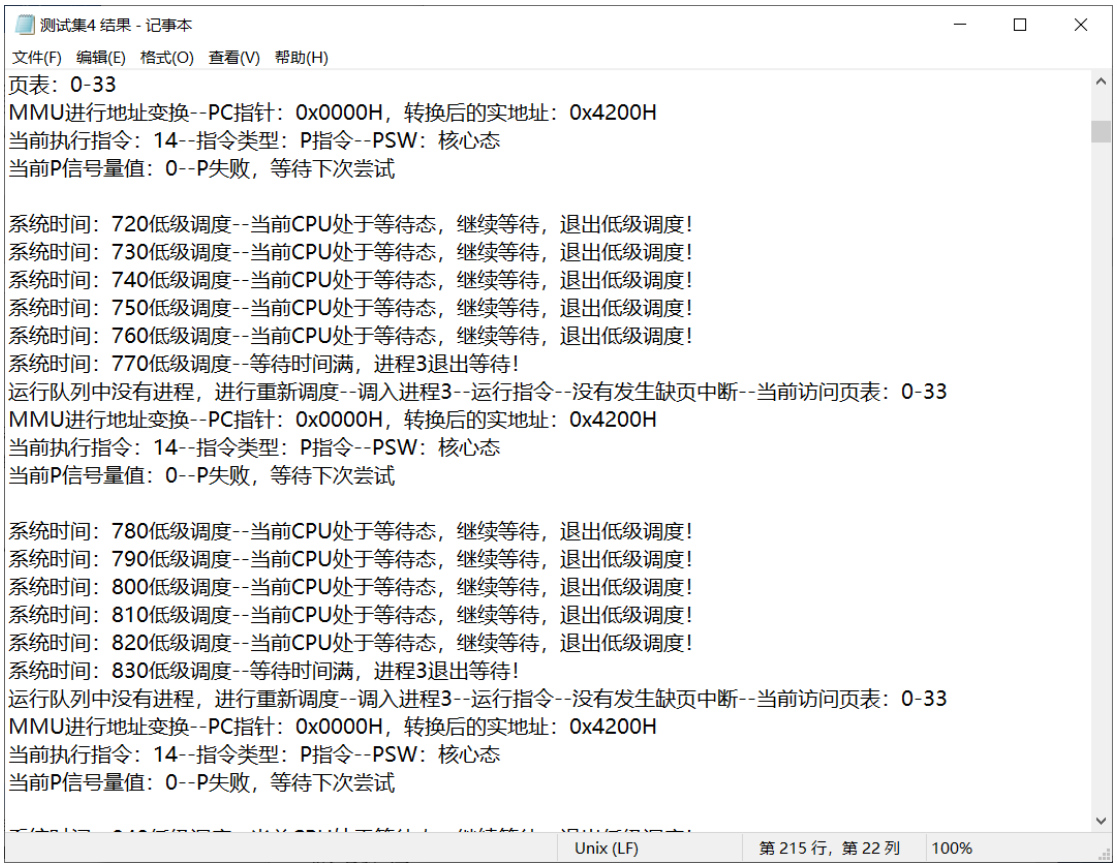
接着，开始测试系统调用的情况。

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：

```
系统时间：3480低级调度--当前CPU处于等待态，继续等待，退出低级调度!
系统时间：3490低级调度--当前CPU处于等待态，继续等待，退出低级调度!

※※死锁检测--检测到死锁！与死锁有关进程：
os.PCB@12cf638a，Process_ID：2

调用进程撤销原语，撤销进程!
系统时间：3500低级调度--运行队列中没有进程，进行重新调度--调入进程1--运行指令--没有发生缺页中断--当前访问页表：0-33
MMU进行地址变换--PC指针：0x0008H，转换后的实地址：0x4208H
当前执行指令：58--指令类型：普通指令--PSW：用户态
```

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。

七、总结与改进建议

总结：

该程序完整的实现了本次操作系统课程设计所需要实现的全部功能，如：作业及进程并发环境、MMU 地址变换、进程原语、页表生成与页面调度算法、三级作业调度过程及算法、页面分配与回收算法、进程同步互斥、进程死锁检测与撤

销算法，并将实现原理过程通过可视化方式呈现。同时，在三态转换的基础上，本程序又引入了“挂起态”的概念，并通过中级调度保证内存保留有一定阈值的剩余空间。
在系统底层设计上，程序模拟仿真了内存、硬盘、CPU、计时器、MMU、地址线数据线的硬件，并能够在并发环境共用这些硬件的情况下不出错的运行。
在功能模块上，程序设计了页面管理、进程管理、作业管理三个模块，每一个模块都有自己的独特功能，同时也与其他模块进行联动。通过这三个大模块的协调工作，系统才得以运行。
在顶层设计上，程序设计了 Control 控制类对程序进行控制。系统的常用变量存放在 kernel.java 文件中，都设计为了静态常量，方便被全局调用。
在界面设计上，程序设计了 CPU 界面、内存界面、磁盘界面、页面管理界面、进程管理界面进行展示，各个界面每隔一定的时间能够自动刷新，更新该模块当前的内容，方便使用者掌握当前指令的运行情况。

建议：

虽然系统能够运行，但是，在设计过程中，仍有许多的不足留待改进：
1、程序的模块分解太过细化，阻碍了最后整体功能的实现。建议将程序的大功能与大模块直接进行封装，单独测试，有助于提升测试的效率与准确性
2、程序的三级调度模块逻辑过于复杂，不利于日后的修改。建议将此模块进行功能上的整合，将细小的部分合并到打模块中
3、程序的死锁检测与三级调度，在微观上为串行关系，在此方面没有能够很好地体现出 JAVA 作为设计并发程序语言的优越性。如果有可能，建议将三级调度与死锁检测单个装入一个线程，并通过 JAVA 本身提供的 synchronized 修饰符进行控制。
4、程序的可视化界面在运行过程中会有闪动情况，需要将窗口晃动才能够继续准确输出。经过网上查阅，该问题可以通过“双缓冲”的页面绘制方式来控制，但是由于本身能力的不足，未能够掌握此技能，希望留待以后继续改进。
5、程序的中断发生间隔为 10ms，该时间对于肉眼观测来说速度过快，建议在程序中加入“单步调试”的功能，以方便对系统模拟的精确控制与度量。
6、输出文件的文本组织形式过于复杂，有很多的信息常常会集中于一行上进行显示。建议优化文件输出的形式，将文件输出以清晰的逻辑进行展示。

附件 2：组员测试报告

一、完成任务与功能说明

1.内存的虚拟仿真与功能实现

1.1 内存类数据成员

```
private byte []data=new byte[32*1024];        //32KB=32768B
```

对应二进制文件 memory.dat 共32KB

按照字节存储，共32K，使用一维数组模拟内存线性空间。

1.2 初始化内存数据

```
public synchronized void InitMemory()
 {
      //从文件中读取内存文件的内容，放入data数组，以字节为单位
      try
      {
          FileReader reader=new FileReader(kernel.MEMORYFILE_PATHNAME);
           BufferedReader br=new BufferedReader(reader);
           int ch=0,address=0;
           for(int i=0;i<kernel.MEMORY_SIZE;i++)
           {
                ch=br.read();
                this.data[address++]=(byte) ch;
           }
           reader.close();
           br.close();
      } catch (Exception e)
      {
           e.printStackTrace();
      }
 }
```

由于文件大小设置为32K，故内存空间与内存文件为一一对应关系，不需要进行进一步的转换。

1.3 内存数据的读取

```
public synchronized short GetData(short address)
 {
      AddressLine.address_line.WriteAddress(address);
//      向      地      址      线      中      写      入      地      址      信      息
 DataLine.data_line.WriteData(ByteToShort(AddressLine.address_line.GetAddress()));
      return DataLine.data_line.GetData();
      //根据地址线中的地址信息取出内存中相应的数据
 }
```

首先提取地址线中设置的地址数据，作为参数传入ByteToShort函数中，经过加工将当前地址与高一位地址组合成short型数据，写入数据线中等待读取。

1.4 内存数据的写入

```
public synchronized void WriteData(short address,short data)
 {
      AddressLine.address_line.WriteAddress(address);                  //向地址线中写入地址信息
      DataLine.data_line.WriteData(data);                              //向数据线中写入数据信息
      short address_temp=AddressLine.address_line.GetAddress();
      short data_temp=DataLine.data_line.GetData();

      if(address_temp%2==0)
      {
           this.data[address_temp]=(byte) (data_temp>>8);          //取高8位
           this.data[address_temp+1]=(byte) (data_temp);           //取低8位
      }
      else
      {
           this.data[address_temp-1]=(byte) (data_temp>>8);            //取高8位
           this.data[address_temp]=(byte) (data_temp);                 //取低8位
      }
      //将数据线中的信息写入到地址线指示的内存地址
 }
```

先将地址信息写入地址线中进行寻址，之后向数据线写入数据信息，依据寻址结果向目标地址低地址写入数据低8位信息，向目标地址高地址写入目标数据高8位信息，完成数据写入。

2. 硬盘的虚拟仿真与功能实现

2.1 硬盘类数据成员

```
private byte [][][]data=new byte[32][64][512];
```

对应二进制文件 harddisk.dat 共1MB

依据CHS寻址方式原理，设计将1MB磁盘空间分成32个磁道，每个磁道64个扇区，每个扇区512个字节，依据三个维度的地址可以精确寻址到每一个字节。

2.2 初始化硬盘数据

```
public synchronized void InitHardDisk()
 {
      //从文件中读取硬盘文件的内容，放入data数组，以字节为单位
      try
      {
      FileReader reader=new FileReader(kernel.HARDDISKFILE_PATHNAME);
           BufferedReader br=new BufferedReader(reader);
           int ch=0;
           for(int i=0;i<kernel.HARDDISK_CYLINDER_NUM;i++)
           {
                for(int j=0;j<kernel.HARDDISK_SECTOR_NUM;j++)
                {
                     for(int k=0;k<kernel.HARDDISK_PAGE_SIZE;k++)
```

```
                    {
                        ch=br.read();
                        this.data[i][j][k]=(byte) ch;
                    }
                }
            }
            reader.close();
            br.close();
        } catch (Exception e)
        {
            e.printStackTrace();
        }
}
```

与初始化内存时相同，1MB的文件大小对应了1MB的实际仿真磁盘大小，因此只需将磁盘文件中的数据全部导入至仿真系统的磁盘数组即可。

2.3 磁盘数据的读取

```
    public synchronized short GetData(int address)
{
    short address_0_15=0;
    short address_16_31=0;
    AddressLine.address_line.WriteAddress((short) (address&0x0000FFFF));
    address_0_15=AddressLine.address_line.GetAddress();
AddressLine.address_line.WriteAddress((short) ((address>>16)&0x0000FFFF));
    address_16_31=AddressLine.address_line.GetAddress();
    short offset=(short) (address_0_15&0x01FF);
    short sector=(short) ((address_0_15>>9)&0x03F);
    short cylinder=(short) ((short) ((address_0_15&0x08000)>>11)|(short)
(address_16_31&0x0F));
    DataLine.data_line.WriteData(ByteToShort(cylinder,sector,offset));
    return DataLine.data_line.GetData();
}
```

先将地址写入地址线，然后通过转换函数转换为对应地址的三维坐标，进而定位到磁盘数组，将低8位与高8位组合，返回一个short型数据至数据线中等待读取。

2.4 一维地址转换为三维地址

```
public int CreateAddress(int cylinder,int sector,int offset)
{
    //cylinder磁道、sector扇区、offset偏移
    //地址结构：偏移——扇区——磁道
    //占用位数：偏移0-8    扇区9-14   磁道15-19
    int address=0;
    address=(address|offset)&0x001FF;
    address=(address|(sector<<9))&0x07FFF;
    address=(address|(cylinder<<15))&0xFFFFF;
    return address;
```

```
}
```

由于地址线为16位，可以表示的范围不足1MB空间，所以将地址设为三个维度，使用一个32位数据保存，将三个维度的数据保存在32位数据的不同位中，直接提取即可得到三个维度的地址数据进行寻址。

2.5 硬盘数据的写入

```
public synchronized void WriteData(int address,short data)
 {
      short address_0_15=0;
      short address_16_31=0;
      AddressLine.address_line.WriteAddress((short) (address&0x0000FFFF));
      address_0_15=AddressLine.address_line.GetAddress();
      AddressLine.address_line.WriteAddress((short) ((address>>16)&0x0000FFFF));
      address_16_31=AddressLine.address_line.GetAddress();
      short offset=(short) (address_0_15&0x01FF);
      short sector=(short) ((address_0_15>>9)&0x03F);
      short cylinder=(short) ((short) ((address_0_15&0x08000)>>11)|(short)
(address_16_31&0x0F));
      if(offset%2==0)
      {
           DataLine.data_line.WriteData(data);
           short data_temp=DataLine.data_line.GetData();
           this.data[cylinder][sector][offset]=(byte) ((data_temp>>8)&0x00FF);
           this.data[cylinder][sector][offset+1]=(byte) (data_temp&0x00FF);
      }
      else
      {
           DataLine.data_line.WriteData(data);
           short data_temp=DataLine.data_line.GetData();
           this.data[cylinder][sector][offset-1]=(byte) ((data_temp>>8)&0x00FF);
           this.data[cylinder][sector][offset]=(byte) (data_temp&0x00FF);
      }
 }
```

先将地址写入地址线，待存数据写到数据线，然后提取地址线数据进行寻址，并将数据分别写入到对应的高低地址中。

3. 地址线与数据线的仿真设计与功能实现（以地址线为例）

3.1 地址线、数据线数据成员

```
private short addressline_data=0;
```

由于地址线为16位，故使用一个short数据即可表示地址线。数据线同理。

3.2 地址线、数据线功能模块

```
public synchronized short GetAddress()
 {
      short temp=this.addressline_data;
      this.addressline_data=0;
      return temp;
```

```
}
public synchronized void WriteAddress(short data)
{
    this.addressline_data=data;
}
```

由于地址线与数据线并不是独立存在的硬件单元，但在硬件仿真中不可缺少，故对这两个对象分别抽象为类，设置对应的读写函数用于操作，但不设置文件作为硬件的虚拟实现方式。

4. CPU的仿真设计与功能实现

4.1 CPU数据成员

```
public Timer ti;
public MMU mm;
 public static Random random=new Random();          //CPU中的随机数生成器
 private int PC=0;          //地址寄存器PC
 private byte PSW=kernel.PSW_KERNEL_STATE;     //程序状态寄存器
 private int IR=0;     //指令寄存器
 private int CR3=0;  //页基址寄存器
 public PCB current_pcb=null;            //当前执行的pcb
```

对应二进制文件 cpu.dat

由于仿真的CPU并不需要进行实际的运算，故在设计中默认CPU存在运算器部件，不对其进行功能实现。CPU中实际由计时器、MMU以及寄存器三个部分组成，通过一个文件进行硬件的虚拟。

4.2 计时器类的设计与实现

4.2.1 计时器类的数据成员

```
private boolean if_interrupt=false;        //是否发生中断的标志位
private String current_time="";                  //存储当前时间的字符串
SimpleDateFormat sdf=new SimpleDateFormat("HH:mm:ss");
Calendar calendar;
Date date;
private int time=-10;//计时器的时间
```

计时器类继承Thread类，单独开启一个线程进行计时。

4.2.2 计时器类方法

针对每个私有数据成员分别设置set和get函数进行读写操作。

4.2.3 中断模拟

```
public void run()
{
    while(true)
    {
        try
        {
            sleep(kernel.INTERRUPTION_INTERVAL); //睡眠一定时间
            if_interrupt=true;
            time+=kernel.INTERRUPTION_INTERVAL; //CPU内时间自增
            SetUIRefresh();          //刷新UI界面
```

```
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
```

在run方法中模拟中断信号，每隔固定时间将中断标志设置为true，模拟发出一次中断信号。每隔一定时间发出中断信号后，各个UI界面所表示的各个方面的处理系统会针对中断信号进行处理，待处理完成后更新各个UI窗口的信息。

4.3 MMU类的设计与实现

4.3.1 MMU类的数据成员

```
private int[][] TLB=new int[kernel.TLB_LENGTH][2];      //TLB快表
```

由于TLB中只存储虚拟页号和实际页框号，为了简化操作，同时节约空间，将其虚拟为一个二维数组。

4.3.2 页号转换

```
public synchronized short VirtualAddressToRealAddress(PCB pcb,short virtual_address)
    {
        //将虚拟地址转换为实地址，此时必须要保证所需要的页在内存中
        short virtual_page_no=GetVirtualAddress_VirtualPage(virtual_address);
        short virtual_offset=GetVirtualAddress_Offset(virtual_address);
        int real_page_no=FindRealPageNo(virtual_page_no); //在TLB中查找对应的页框号
        if(real_page_no==-1)            //快表中不存在该项目
        {
            //去内存中查询该页号对应的页框号，放入到TLB中
            AddTLB(virtual_page_no,pcb.CheckPageTable(virtual_page_no));
            //重新查询TLB
            real_page_no=FindRealPageNo(virtual_page_no);
        }
        return (short) (PageToRealAddress((short) real_page_no)+virtual_offset);
    }
```

将指令的虚拟地址转换到内存中的实际地址，由于所有存储空间均采取等大的页面进行分割，故只需将偏移地址提取出来，将虚拟页号映射至实际页号即可。

若实地址所在页面不在快表中，即将本页信息按照TLB序号添加至TLB中。

4.3.3 页号查询地址

```
    public synchronized int PageToRealAddress(short num)
    {
        if(num<kernel.MEMORY_SIZE/kernel.SINGLE_PAGE_SIZE)
        {
            return num*kernel.SINGLE_PAGE_SIZE;
        }
        else
        {
        num=(short) (num-kernel.MEMORY_SIZE/kernel.SINGLE_PAGE_SIZE);
            //cylinder磁道、sector扇区、offset偏移
```

```
            int cylinder=num/kernel.HARDDISK_SECTOR_NUM;     //计算磁道
            int sector=num%kernel.HARDDISK_SECTOR_NUM;       //计算扇区
            int offset=0;                                    //计算偏移
            return HardDisk.harddisk.CreateAddress(cylinder, sector, offset);
        }
    }
```

若内存中没有对应页的信息，即发生缺页中断时，通过本方法将传入的页/块号num转换成为该页/块的基地址，之后将页调入内存和TLB。该方法对内存与磁盘的地址均有效。

4.3.4 查询TLB信息

TLB数组的两列分别存储了页面的虚拟页号和实际页号，通过对应的set和get函数可以完成读写操作。

4.4 CPU的初始化与转存

初始化CPU时从文件中读入CPU各数据项的数据，关闭程序时将CPU当前状态保存至文件中。

4.5 CPU功能模块

4.5.1 PC指针自增

```
public void PCSelfAdd()
 {
      this.PC+=kernel.SINGLE_INSTRUCTION_SIZE;
 }
```

自增步长为一个指令长度。

4.5.2 CPU各数据项的set和get函数，可进行读写操作

5. PageModule的仿真设计与实现

5.1 PageModule类数据成员

```
//用户区起始页号
 private final int userspace_page_location_start=kernel.MEMORY_KERNEL_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE;
 //用户区结束页号
 private final int userspace_page_location_end=kernel.MEMORY_SIZE/kernel.SINGLE_PAGE_SIZE;
 //交换区起始页号
 private final int swaparea_page_location_start=kernel.MEMORY_SIZE/kernel.SINGLE_PAGE_SIZE;
 //交换区结束页号
 private final int swaparea_page_location_end=(kernel.MEMORY_SIZE+kernel.HARDDISK_VIRTUAL_MEMORY_SIZE)/kernel.SINGLE_PAGE_SIZE;
 //全部页面的使用情况
private boolean []if_page_usage=new boolean[(kernel.MEMORY_SIZE+kernel.HARDDISK_VIRTUAL_MEMORY_SIZE)/kernel.SINGLE_PAGE_SIZE];
 //伙伴算法
private int []bitmap=new int[6];     //每位对应1、2、4、8、16、32个页框，正好对应32位
```

```
@SuppressWarnings("unchecked")
private ArrayList<Short>[] free_area=new ArrayList[6];     //空闲链表
private LinkedList<Integer> lru=new LinkedList<Integer>(); //LRU算法实现
```

5.2 Page类设计

5.2.1 Page类数据成员

```
private int page_num;     //页号
private short[] data=new short[kernel.SINGLE_PAGE_SIZE/2];
//每页大小=512B=256个short类型
```

5.2.2 Page类功能模块

基于Page类的数据成员分别设置set及get函数进行读写操作。

设定了具有参数的构造函数，可在生成Page对象时选定具体的页，直接通过页操作调整对应页的内存空间的数据。

5.3 伙伴算法

5.3.1 bitmap设计

由于用户内存空间共16KB，按照每512B一页可分为32页，正好对应2^5，故使用bitmap算法，使用6个int型数据，分别表示2^0 – 2^5个页的使用情况。

5.3.2 分配空间

```
public String ApplyPageInMemory(short num)
{
    //在内存中，向伙伴算法申请num个页面
    int pow=CalculateCloest2Num(num);  //求出该页面所需要申请的块所在的链表级别
    String apply_str="null"; //默认为该页面无法找到
    for(int i=0;i<(kernel.MEMORY_USER_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE)/(int)Math.pow(2, pow);i++)
    {
        if(free_area[pow].get(i)==0)  //找到未分配的块
        {
            apply_str=""+pow+":"+i;//写入字符串，表示分配方式:"链表:块号"
            SetBlockState(pow,i,1,1);      //设置块链表的该位为1
            RefreshBlockList();            //刷新块链表
            break;
        }
    }
    return apply_str;
}
```

通过计算申请的页数量，直接进入对应bitmap中查找可用块，若有可用块则递归修改该块所对应bitmap图状态；若无可用块则递归检查更大的块有无可用空间，若所有块均分配完毕则停止分配空间。

5.3.3 回收空间

```
public void RecyclePage(short page_num)
{
    //回收页面，参数num为页面号（num从0开始编号）
    //将该页的内容全部清空，并在记录中使得该页表示为未被占用
```

```
 if(page_num>=this.userspace_page_location_start&&page_num<this.userspace_page_locati
on_end)
            FreePageInMemory(page_num);  //释放用户空间中的页
 if(page_num>=this.swaparea_page_location_start&&page_num<this.swaparea_page_locatio
n_end)
            FreePageInDisk(page_num); //释放交换区中的页
 }
public void FreePageInMemory(short page_num)
 {
      //在内存中，利用伙伴算法释放某一页
      //将这一页内容清空
      Page pa=new Page(page_num);
      for(short i=0;i<kernel.SINGLE_PAGE_SIZE;i+=2)
           pa.SetPageData(i, (short) 0);
      //在伙伴算法块链表中，将这一页所在的块清空
      SetBlockState(0,page_num-32,1,0);     //设置块链表的该位为0
      RefreshBlockList();                   //刷新块链表
 }
```

单次执行方法释放一个页，之后通过函数中的递归执行完成伙伴块的合并及bitmap状态的更新，完成伙伴算法空间的回收。

5.4 缺页中断

5.4.1 选定被替换页面

```
public void LRUVisitOnePage(int page_num)
 {
      //LRU访问某一页
      lru.remove(Integer.valueOf(page_num));          //将该页号从原来的里面删除
      lru.addFirst(Integer.valueOf(page_num));    //将访问的该页置顶
 }
```

将最近访问页面放置到LRU队列头。

```
 public short LRUGetLastPageNum()
 {
      //获得应该调出的页面号
      return lru.getLast().shortValue();         //获得链表的最后一项
 }
```

队列尾的页号就是最近最久未访问的页号，即为即将被替换的页号。

5.4.2 执行缺页中断

```
public void MoveToMemory(short memory_page_num,short swap_page_num)
 {
      //将指定的虚存页移动到指定的内存页中
      Page mem_page=new Page(memory_page_num);
      Page swap_page=new Page(swap_page_num);
      for(short i=0;i<kernel.SINGLE_PAGE_SIZE;i+=2)
      {
           mem_page.CopyPageData(swap_page);      //内容转移
```

```
            swap_page.ClearPageData();              //内容清空
        }
    }
```

将需要执行PageIn操作的页调入至内存LRU算法选中的页。

```
    public void MoveToDisk(short memory_page_num,short swap_page_num)
    {
        //将指定的内存页移动到指定的虚存页中
        Page mem_page=new Page(memory_page_num);
        Page swap_page=new Page(swap_page_num);
        for(short i=0;i<kernel.SINGLE_PAGE_SIZE;i+=2)
        {
            swap_page.CopyPageData(mem_page);       //内容转移
            mem_page.ClearPageData();               //内容清空
        }
    }
```

再讲LRU算法选中的页调出至执行PageIn操作的页，完成交换。

5.5 虚存管理

5.5.1 申请虚存空间

```
public String ApplyPageInDisk(short page_num)
 {
        if(page_num>GetFreePageNumInDisk())
            return "null";
        int count=0;
        String str="";
        for(int i=0;i<this.swaparea_page_location_end;i++)
        {
            if(i>=this.swaparea_page_location_start)
            {
                if(this.if_page_usage[i]==false)    //找到空闲的页面
                {
                    str+="1";
                    count++;
                    this.if_page_usage[i]=true;
                }
                else
                    str+="0";
            }
            else
                str+="0";
            if(count==page_num)
                break;
        }
        return str;
    }
```

在虚存中申请page_num个页面，返回一个String类型的值，该值从左到右编号为0-191，共192位，每一位的值为0/1，1代表该页面分配给该进程使用。在写入程序区时，必须按照分配的页面顺序从小到大写入。同时，在记录数组中记录这些申请的页框，将他们设置为已用状态。

5.5.2 释放虚存空间

```
public void FreePageInDisk(short page_num)
 {
      //在外存中，释放某一页
      //将这一页内容清空
      Page pa=new Page(page_num);
      for(short i=0;i<kernel.SINGLE_PAGE_SIZE;i+=2)
           pa.SetPageData(i, (short) 0);
      //将这一页设置为未占用状态
      this.if_page_usage[page_num]=false;
 }
```

通过多次调用此方法，完成对占用的虚存空间的释放

5.6 可用空间查询

使用if_page_usage数组记录内存和虚存中每个页的使用情况，并通过以下方法查询空间使用情况。

```
public int GetFreePageNumInMemory()
 {
      RefreshBitmap();
      int count=0;
      for(int
i=0;i<(kernel.MEMORY_USER_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE);i++)
      {
           if(GetOneBit(this.bitmap[0],i)==0)
                count++;
      }
      return count;
 }
```

该方法返回当前物理内存中可用的页框数。

```
 public int GetFreePageNumInDisk()
 {
      int count=0;
      for(int i=this.swaparea_page_location_start;i<this.swaparea_page_location_end;i++)
      {
           if(this.if_page_usage[i]==false)
                count++;
      }
      return count;
 }
```

该方法返回当前虚存中可用的页框数。

6. 可视化显示 CPU 情况与存储空间信息情况

二、测试安装用例详细说明

1. 直接执行 jar 程序查看默认数据测试结果
2. 将工程文件导入至 eclipse
3. 在 Main 函数中使用页面调度部分的函数，执行程序即可测试页面调度。
4. 在 Main 函数中使用页面部分的函数，执行程序即可测试页面相关功能。
5. 测试内存分配时，在 Main 函数中调用 PageModule.page_module.ApplyPageInMemory()进行测试，参数为申请的页号数量，范围 0-32。该函数返回值为 String 类型，记录了分配的情况，可直接输出查看结果。
6. 测试虚存分配时，在 Main 函数中调用 PageModule.page_module.ApplyPageInDisk()进行测试，参数为申请的页号数量，范围 0-128。该函数返回值为 String 类型，记录了分配的情况，可直接输出查看结果。
7. 测试内存空间回收时，需在分配内存之后进行，使用函数 PageModule.page_module.RecyclePage()进行回收，参数为要回收的页面号，范围是 32-63。例如申请 7 个页面，分配结果为 3;0，说明该进程分配了 8 个页面，程序实际使用页面号为 32-38,39 号页面被浪费。若要完整回收页面需手动释放 32-39 号页面，才可完成对全部空间的释放。
8. 测试虚存空间回收时，需在分配虚存之后进行，根据分配空间，使用函数 PageModule.page_module.RecyclePage()根据分配空间进行回收，参数为要回收的页面号，范围是 64-191。例如申请 5 个页面，分配结果为 00000000000000000000000000000000000000000000000000000000000011111，在分配 5 个页面之后停止函数，该字符串从左至右依次表示内存+虚存中的各块是否被该进程占有。释放时根据字符串信息进行释放即可。

三、裸机硬件部件仿真设计

4. 内存仿真设计
   内存总大小为32KB，规定：
   前16KB为内核区
   后16KB为用户区
   内核区存储的内容：核心栈+系统内核+进程所有PCB信息
   综上所述，内存的组成结构为：
   核心栈+系统内核（1页）、PCB池（31页）、用户区（32页）
5. 硬盘仿真设计
   硬盘总大小为1MB（1024KB），规定：
   前64KB为虚存区
   再16KB为系统文件区
   剩下的944KB都为文件区
   综上所述，硬盘的组成结构为：
   虚存区（128页）+系统文件区（32页）+文件区（1888页）
6. CPU仿真设计
   CPU中包含以下部件：
   地址寄存器PC
   指令寄存器IR
   页基址寄存器CR3

随机数生成器random
程序状态寄存器PSW
当前执行进程 PCB 的指针 currend_pcb

四、通用数据结构设计

5. Page类设计

```
private int page_num;    //页号
private short[] data=new short[kernel.SINGLE_PAGE_SIZE/2];
//每页大小=512B=256个short类型
```

Page类中定义了对于页的基本操作，如读取写入清空等，支持对页内具体数据的读写操作和对页面整体的读写操作。

6. PCB类设计

PCB类中存储了该进程的全部信息，其中包含由作业转换为进程时存储的作业信息，以及运行过程中所需要的控制用信息。PCB中同时设置有下述数据成员的set和get函数进行读写操作。

PCB为动态生成，因进程创建而产生，因进程结束而消亡，没有静态的PCB对象可以使用。

PCB类中同时设计了进程的创建、撤销、阻塞、唤醒、挂起原语，用于在三级调度时进行状态切换。

7. JCB类设计

在JCB中存储了作业的相关信息，并设置有对应数据的set和get函数进行读写操作，在作业被创建为进程时将上述信息添加至PCB中使用。

8. 全局变量设计

kernel类中定义了需要用到的静态变量

```
/*系统基本信息*/
public static int SINGLE_PAGE_SIZE=512;//每一页/块的大小
public static int MEMORY_SIZE=32*1024; //内存大小，32KB
public static int MEMORY_KERNEL_SPACE_SIZE=16*1024; //内存内核空间大小，16KB
public static int MEMORY_KERNEL_CORESTACKANDOSKERNEL_SIZE=512;      //核心栈+系统内核大小，1页
public static int MEMORY_KERNEL_PCB_POOL_SIZE=31*512;  //PCB池大小，31页
public static int MEMORY_USER_SPACE_SIZE=16*1024;   //内存用户空间大小（页表、页框使用），16KB
public static int HARDDISK_SIZE=1*1024*1024;    //硬盘空间大小，1MB
public static int HARDDISK_VIRTUAL_MEMORY_SIZE=64*1024;    //虚存区大小，128页，64KB
public static int HARDDISK_SYSTEMFILE_SIZE=16*1024;  //系统文件大小，32页，16KB
public static int HARDDISK_FILE_SPACE_SIZE=944*1024;  //文件区大小，1888页，944KB
public static int HARDDISK_CYLINDER_NUM=32; //磁盘磁道数
public static int HARDDISK_SECTOR_NUM=64;    //磁盘扇区数
public static int HARDDISK_PAGE_SIZE=512; //磁盘每页/块大小
public static int SINGLE_INSTRUCTION_SIZE=8;   //单条指令的大小
```

```
public static int INSTRUCTIONS_PER_PAGE=SINGLE_PAGE_SIZE/SINGLE_INSTRUCTION_SIZE;//每一页的指令数目

public static int INTERRUPTION_INTERVAL=10;          //系统发生中断的间隔
public static int SYSTEM_TIME=0;          //系统内时间
public static void SystemTimeAdd() {kernel.SYSTEM_TIME+=kernel.INTERRUPTION_INTERVAL;}   //系统时间自增
public static int TLB_LENGTH=kernel.MEMORY_USER_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE/2;
//TLB快表的长度，16
/*系统基本信息*/

/*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*/

/*Process PSW  程序状态字*/
public final static byte PSW_KERNEL_STATE=0;     //管态
public final static byte PSW_USER_STATE=1;          //目态
/*Process PSW  程序状态字*/

/*硬件初始化需要变量*/
public static String MEMORYFILE_PATHNAME="./input/memory.dat";        //内存文件地址
public static String HARDDISKFILE_PATHNAME="./input/harddisk.dat";     //硬盘文件地址
public static String CPUFILE_PATHNAME="./input/cpu.dat";               //CPU文件地址
/*硬件初始化需要变量*/

/*60条指令*/
public static int GetInstructionType(int instruction){
      //获取指令类型
      if(instruction>=0&&instruction<=9)return 1;
      if(instruction>=10&&instruction<=19)return 2;
      if(instruction>=20&&instruction<=29)return 3;
      if(instruction>=30&&instruction<=39)return 4;
      if(instruction>=40&&instruction<=49)return 5;
      if(instruction>=50&&instruction<=59)return 6;
      return -1;}
public static int GetInstructionTime(int instruction)
```

```
{/*获取指令执行所需要时间*/return (instruction%10)*10+20;}
public static int[] MUTEX= {-4,-3,-2,-1,0,1,2,3,4,5};          //临界区信号量,10个
public static int[] SYSTEM_RESOURCE= {0,1,2,3,4,5,6,7,8,9};      //系统资源量，10个
public static int GetUseResourceNum(int instruction)
{/*获取该指令所申请、释放的PV、资源所在的数组序号*/return instruction%10;}
/*60 条指令*/
```

五、代码结构与函数说明

1. CPU类

```
public class CPU
{
 public static CPU cpu=new CPU();
 public Timer ti;
 public MMU mm;
 public static Random random=new Random();            //CPU中的随机数生成器
 private int PC=0;           //地址寄存器PC
 private byte PSW=kernel.PSW_KERNEL_STATE;     //程序状态寄存器
 private int IR=0;     //指令寄存器
 private int CR3=0;  //页基址寄存器
 public PCB current_pcb=null;              //当前执行的pcb

 public CPU()//构造函数

 public synchronized void InitCPU()
      //从文件读取CPU的信息，完成从文件到类的映射

 public synchronized void CloseCPU()
      //将CPU的信息存入文件，完成从文件到类的映射

 public void PCSelfAdd()
      //PC指针的自增

 public void ClearPC()
      //PC指针内容的清空

 public int GetPC()
      //获得PC指针

 public void SetPSW(byte PSW)
      //设置PSW

 public byte GetPSW()
      //获取PSW
```

```
public void SetIR(short instruction)
        //设置指令

public void ClearIR()
        //清空指令

public int GetIR()
        //获取IR指令

public void SetCR3(int address)
        //设置CR3基址寄存器

public void ClearCR3()
        //清空CR3基址寄存器

public int GetCR3()
        //获取CR3

public void SerCurrentPCB(PCB pcb)
        //设置当前执行的PCB

public void ClearCurrentPCB()
        //清空当前的PCB指针
}
```

2. MMU类

```
public class MMU
{
 private int[][] TLB=new int[kernel.TLB_LENGTH][2];      //TLB快表

 MMU()//构造函数

 public synchronized void ClearTLB()
        //清空快表

 public synchronized int CheckTLBVirtualPageNo(short line)
        //在TLB中检测第line行的虚拟页号

 public synchronized int CheckTLBRealPageNo(short line)
        //在TLB中检测第line行中的实际页框号

 public synchronized int FindRealPageNo(short virtual_page_no)
        //在TLB中查询对应的虚拟页号对应的实际页框号
```

```
public synchronized void AddTLB(short virtual_page_no,short real_page_no)  {
    //添加TLB数据

public synchronized void EditTLBData(short line,short data_1,short data_2)
    //修改TLB快表的值

public synchronized short GetVirtualAddress_VirtualPage(short virtual_address)
    //获取虚拟地址的页号
public synchronized short GetVirtualAddress_Offset(short virtual_address)
    //获取虚拟地址的偏移，双字节数据的位移

public synchronized int PageToRealAddress(short num)
    //将传入的页/块号num转换成为该页/块的基地址
    //该功能对内存与磁盘地址同样有效

public synchronized short VirtualAddressToRealAddress(PCB pcb,short virtual_address)
    //将虚拟地址转换为实地址
    //注意，此时必须要保证所需要的页在内存中!!!!
}
```

3. Timer类

```
public class Timer extends Thread
{
private boolean if_interrupt=false;        //是否发生中断的标志位
private String current_time="";                  //存储当前时间的字符串
/*日期信息*/
SimpleDateFormat sdf=new SimpleDateFormat("HH:mm:ss");
Calendar calendar;
Date date;
/*日期信息*/
private int time=-10;//计时器的时间

public void run()//线程执行函数

public boolean GetIfInterrupt()
    //获取中断标志

public void ResetIfInterrupt()
    //重置中断标志为空

public String GetCurrentTime()
    //获取当前时间

public int GetSystemTime()
```

```
        //获取系统时间

 private void SetUIRefresh()
        //根据计时器，设置UI界面的刷新
}
```

4. Memory类

```
public class Memory
{
 public static Memory memory=new Memory();
 private byte []data=new byte[32*1024];       //32KB=32768B空间

 public Memory()//构造函数

 public synchronized void InitMemory()
        //从文件中读取内存文件的内容，放入data数组，以字节为单位

 private synchronized short ByteToShort(short address)
        //将byte组装成short类型数据

 public synchronized void WriteData(short address,short data)
        //将数据线中的信息写入到地址线指示的内存地址

 public synchronized short GetData(short address)
        //根据地址线中的地址信息取出内存中相应的数据
}
```

5. HardDisk类

```
public class HardDisk
{
 public static HardDisk harddisk=new HardDisk();
 private byte [][][]data=new byte[32][64][512];//32*64*512=1024KB空间

 public HardDisk()//构造函数

 public synchronized void InitHardDisk()
        //从文件中读取硬盘文件的内容，放入data数组，以字节为单位

 public int CreateAddress(int cylinder,int sector,int offset)
 {
        //根据磁道、扇区、偏移计算地址

 private synchronized short ByteToShort(short cylinder,short sector,short offset)
        //将byte组装成short类型数据
```

```
        //cylinder磁道、sector扇区、offset偏移
        //地址结构：偏移——扇区——磁道
        //占用位数：偏移0-8     扇区9-14    磁道15-19

 public synchronized void WriteData(int address,short data)
        //向硬盘给定地址处写入给定数据

 public synchronized short GetData(int address)
        //从硬盘给定地址处读取数据
}
```

6. DataLine类和AddressLine类（以AddressLine类为例）

```
public class AddressLine
{
 public static AddressLine address_line=new AddressLine();//地址线的静态变量
 private short addressline_data=0;  //地址线存储单元

 public synchronized void WriteAddress(short data)
        //存入地址信息

 public synchronized short GetAddress()
        //取出地址信息
}
```

7. PageModule类

```
public class PageModule
{
 public static PageModule page_module=new PageModule();

 /*一些常量*/
 //用户区起始页号
 private final int userspace_page_location_start=kernel.MEMORY_KERNEL_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE;
 //用户区结束页号
 private final int userspace_page_location_end=kernel.MEMORY_SIZE/kernel.SINGLE_PAGE_SIZE;
 //交换区起始页号
 private final int swaparea_page_location_start=kernel.MEMORY_SIZE/kernel.SINGLE_PAGE_SIZE;
 //交换区结束页号
 private final int swaparea_page_location_end=(kernel.MEMORY_SIZE+kernel.HARDDISK_VIRTUAL_MEMORY_SIZE)/kernel.SINGLE_PAGE_SIZE;
```

```
/*一些常量*/

//全部页面的使用情况
private boolean []if_page_usage=new boolean[(kernel.MEMORY_SIZE+kernel.HARDDISK_VIRTUAL_MEMORY_SIZE)/kernel.SINGLE_PAGE_SIZE];
//伙伴算法
private int []bitmap=new int[6];    //每位对应1、2、4、8、16、32个页框，正好对应32位
@SuppressWarnings("unchecked")
private ArrayList<Short>[] free_area=new ArrayList[6];     //空闲链表

private LinkedList<Integer> lru=new LinkedList<Integer>();

PageModule()//构造函数
{
    InitSwapAreaUsage();   //初始化交换区页面被占用状态
    for(int i=0;i<6;i++) //实例化伙伴算法的空闲链表
        this.free_area[i]=new ArrayList<Short>();
    InitFreeAreaList();        //初始化空闲链表
    RefreshBitmap();           //刷新伙伴算法的Bitmap
}

private void InitSwapAreaUsage()
    //初始化交换区页的使用情况

private void InitFreeAreaList()
    //初始化空闲链表

public void RefreshBitmap()
    //刷新bitmap

private int SetOneBit(int num,int loca,int bit)
    //修改bitmap中的某一位信息

private int GetOneBit(int num,int loca)
    //获得bitmap中的某一位信息

public int GetFreePageNumInMemory()
    //返回当前物理内存中可用的页框数

public int GetFreePageNumInDisk()
    //返回当前虚存中可用的页框数

public Page GetPage(short num)
```

```
    //获得某一个页面

public boolean IfCouldApplyPageInDisk(short num)
    //检测在虚存中是否可以申请num个页面
public int CalculateCloest2Num(short num)
    //计算出与该数字最接近的2的次幂数的幂

private void SetBlockState(int list,int no,int set_num,int state)
    //在伙伴算法的链表中，在第list级别的第no块设置连续的set_num块为state状态

private void RefreshBlockList()
    //从底往上刷新块链表

public String ApplyPageInMemory(short num)
    //在内存中，向伙伴算法申请num个页面

public String ApplyPageInDisk(short page_num)
    //在虚存中申请page_num个页面，返回一个String类型的值
    //String格式的数据说明：从左到右编号为0-191，共192位，每一位的值为0/1，1代表该页面分配给该进程使用。在写入程序区时，必须按照分配的页面顺序从小到大写入
    //同时，在记录数组中记录这些申请的页框，将他们设置为已用状态

public void FreePageInMemory(short page_num)
    //在内存中，利用伙伴算法释放某一页

public void FreePageInDisk(short page_num)
    //在外存中，释放某一页

public void RecyclePage(short page_num)
    //回收页面，参数num为页面号（num从0开始编号）
    //将该页的内容全部清空，并在记录中使得该页表示为未被占用

public void MoveToMemory(short memory_page_num,short swap_page_num)
    //将指定的虚存页移动到指定的内存页中

public void MoveToDisk(short memory_page_num,short swap_page_num)
    //将指定的内存页移动到指定的虚存页中

public void ExchangePage(short memory_page_num,short swap_page_num)
    //交换两个页面的内容

public void CopyPage(short src_page_num,short des_page_num)
    //将序号src_page_num的页面复制到序号为des_page_num的页面中
```

```
    public short GetOneFreePageInMemory()
        //在物理内存中找到一个空闲的页面，并返回该页面序号
    public short GetOneFreePageInDisk()
        //在虚存中找到一个空闲的页面，并返回该页面序号

    public void LRUVisitOnePage(int page_num)
        //LRU访问某一页

    public short LRUGetLastPageNum()
        //获得应该调出的页面号

    public boolean isBlockUsing(int i,int j)
        //获取内存中某一个页的使用情况

    public boolean isPageUsing(int i)
        //获取虚存中某一页的使用情况
}
```

六、功能测试

（按照拟完成功能和函数逐项测试，写清楚测试数据和输出结果，并对每套数据输出结果分析正确性并说明。测试数据需要多组并有一定代表性）

1. 伙伴算法分配内存空间

```
String info = PageModule.page_module.ApplyPageInMemory((short) 7);
```

在Main函数中调用函数，申请7个页面，当前系统中内存为空，伙伴算法中没有已分配空间，故函数返回结果为3:0，即2^3链表第0页，共8页，同时查看其它链表分配情况，发现结果正确。

```
aft = PageModule.page_module.GetFreePageNumInDisk();
text.append("还有几个可用块"+aft+"\n");
```

```
还有几个可用块24
分配到的链表号和块号3:0
```

```
String info2 = PageModule.page_module.ApplyPageInMemory((short) 4);
```

再次调用函数申请4个页面返回2:2，即2^2链表第2页，因0-1共8页已被分配，故分配第2块空闲页，结果正确。

```
还有几个可用块20
分配到的链表号和块号2:2
```

```
String info3 = PageModule.page_module.ApplyPageInMemory((short) 20);
```

```
还有几个可用块20
分配到的链表号和块号null
```

再次调用函数申请20个页面返回null。根据计算，内存中共有32页，已分配11块，剩余21页并没有连续32页的空间满足20页的申请，故无法进行分配，返回null结果正确。

2. 释放内存空间

```
PageModule.page_module.RecyclePage((short) 32);
PageModule.page_module.RecyclePage((short) 33);
PageModule.page_module.RecyclePage((short) 34);
PageModule.page_module.RecyclePage((short) 35);
PageModule.page_module.RecyclePage((short) 36);
PageModule.page_module.RecyclePage((short) 37);
PageModule.page_module.RecyclePage((short) 38);
PageModule.page_module.RecyclePage((short) 39);
```

分别调用函数释放32-38页，释放后可用空间由20变为28，结果正确。

```
还有几个可用块28
```

3. 分配虚存空间

Main 函数中调用函数申请 5 个虚存页面

```
info = PageModule.page_module.ApplyPageInDisk((short) 5);
```

函数返回结果从左至右依次为内存+虚存中每一块是否被此进程占用的情况分析，由于当前系统虚存为空，虚存起始页号为 64，而函数分配页号为 64-68，执行后虚存中可用页数从 128 变为 123，所以结果正确。

```
aft = PageModule.page_module.GetFreePageNumInDisk();
text.append("还有几个可用块"+aft+"\n");
```

```
还有几个可用块128
还有几个可用块123
```

```
分配到的块号00000000000000000000000000000000000000000000000000000000000011111
```

```
info2 = PageModule.page_module.ApplyPageInDisk((short) 8);
```

再次在 Main 函数中调用函数申请 8 个虚存页面，函数返回结果从左至右依次为内存+虚存中每一页是否被此进程占用的情况分析，由于当前系统虚存为空，虚存起始页号为 64，而函数分配页号为 69-76，执行后虚存中可用块数从 123 变为 115，所以结果正确。

```
还有几个可用块123
还有几个可用块115
```

```
分配到的块号00000000000000000000000000000000000000000000000000000000011111111
```

4. 释放虚存空间

```
PageModule.page_module.RecyclePage((short) 64);
PageModule.page_module.RecyclePage((short) 65);
PageModule.page_module.RecyclePage((short) 66);
```

对 64、65、66 页执行函数，执行后可用页数从 115 变为 118，结果正确。

```
还有几个可用块115
还有几个可用块118
```

5. 通过页面修改内存数据

```
Page page = new Page((short) 40);
Page test = new Page((short) 39);
page.SetPageData((short) 0,(short)  233);
test.SetPageData((short) 0,(short)  2333);
```

```
text.append("页40中数据"+page.GetPageData((short) 0)+"\n");
text.append("页39中数据"+test.GetPageData((short) 0)+"\n");
```

```
页40中数据233
页39中数据2333
```

```
text.append("第40页内存地址0处的数据"+Memory.memory.GetData((short) (CPU.cpu.mm.PageToRealAddress((short) 40)))+"\n");
text.append("第39页内存地址0处的数据"+Memory.memory.GetData((short) (CPU.cpu.mm.PageToRealAddress((short) 39)))+"\n");
```

```
第40页内存地址0处的数据233
第39页内存地址0处的数据2333
```

获取第 39、40 页，向第 40 页地址 0 处写入数据 233，向第 39 页地址 0 处写入数据 2333，调用交换函数交换两个页面，查看页面数据和内存数据。

```
PageModule.page_module.ExchangePage((short) 40, (short) 39);
```

```
text.append("页40中数据"+page.GetPageData((short) 0)+"\n");
text.append("页39中数据"+test.GetPageData((short) 0)+"\n");
```

```
交换后页40中数据233
交换后页39中数据2333
```

```
text.append("第40页内存地址0处的数据"+Memory.memory.GetData((short) (CPU.cpu.mm.PageToRealAddress((short) 40)))+"\n");
text.append("第39页内存地址0处的数据"+Memory.memory.GetData((short) (CPU.cpu.mm.PageToRealAddress((short) 39)))+"\n");
```

```
第40页内存地址0处的数据2333
第39页内存地址0处的数据233
```

发现页中数据已被更改，但内存中数据并未被更改。重新获取第 39、40 页查看数据。

```
Page page_new = new Page((short) 40);
Page test_new = new Page((short) 39);
```

```
交换后重新生成页40中数据2333
交换后重新生成页39中数据233
```

交换成功，结果正确。

向伙伴算法申请 4 个页面，并设置页 32 中地址处数据为 123，查看页面数据和内存第 32 页处数据。

```
String info = PageModule.page_module.ApplyPageInMemory((short) 4);
```

```
还有几个可用块28
分配到的链表号和块号2:0
页32中数据123
第32页内存地址0处的数据123
```

调用函数释放页 32。

```
PageModule.page_module.RecyclePage((short) 32);
```

查看页面数据和内存第 32 页处数据。

```
释放后页32中数据123
第32页内存地址0处的数据0
```

同样重新获取第 32 页后查看数据。

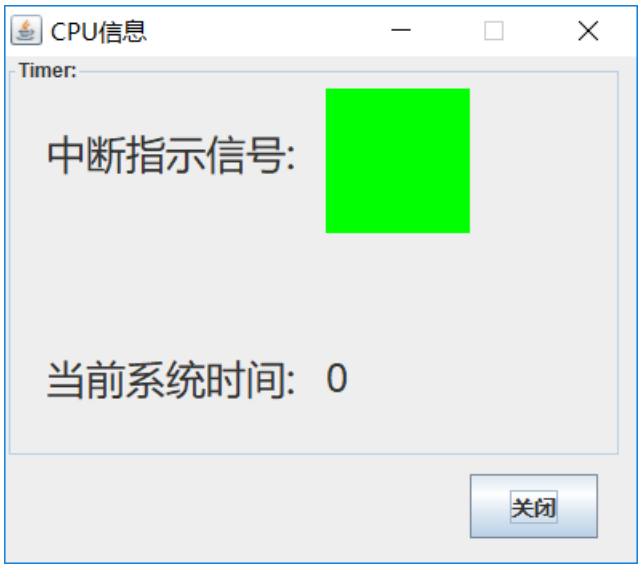
```
Page page_new2 = new Page((short) 32);
```

```
释放后重新生成页32中数据0
内存中的数据0
```

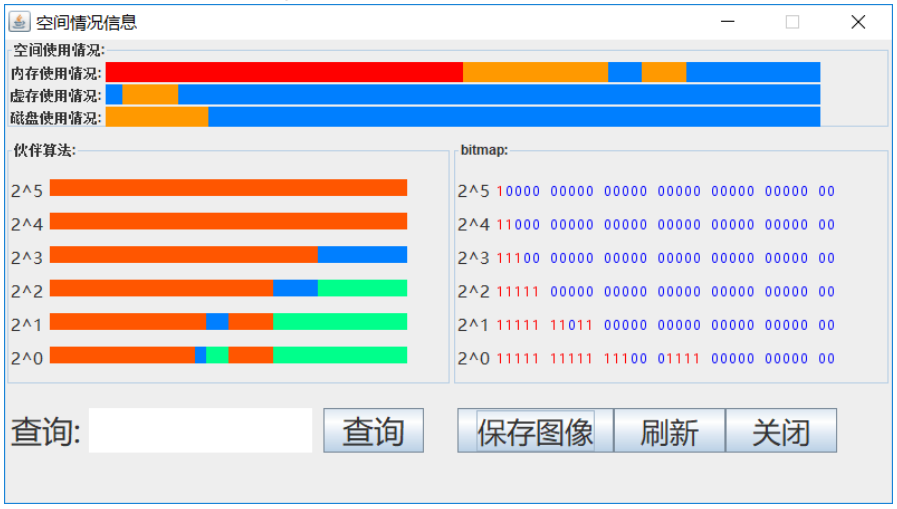
结果正确。

6. 界面信息查看

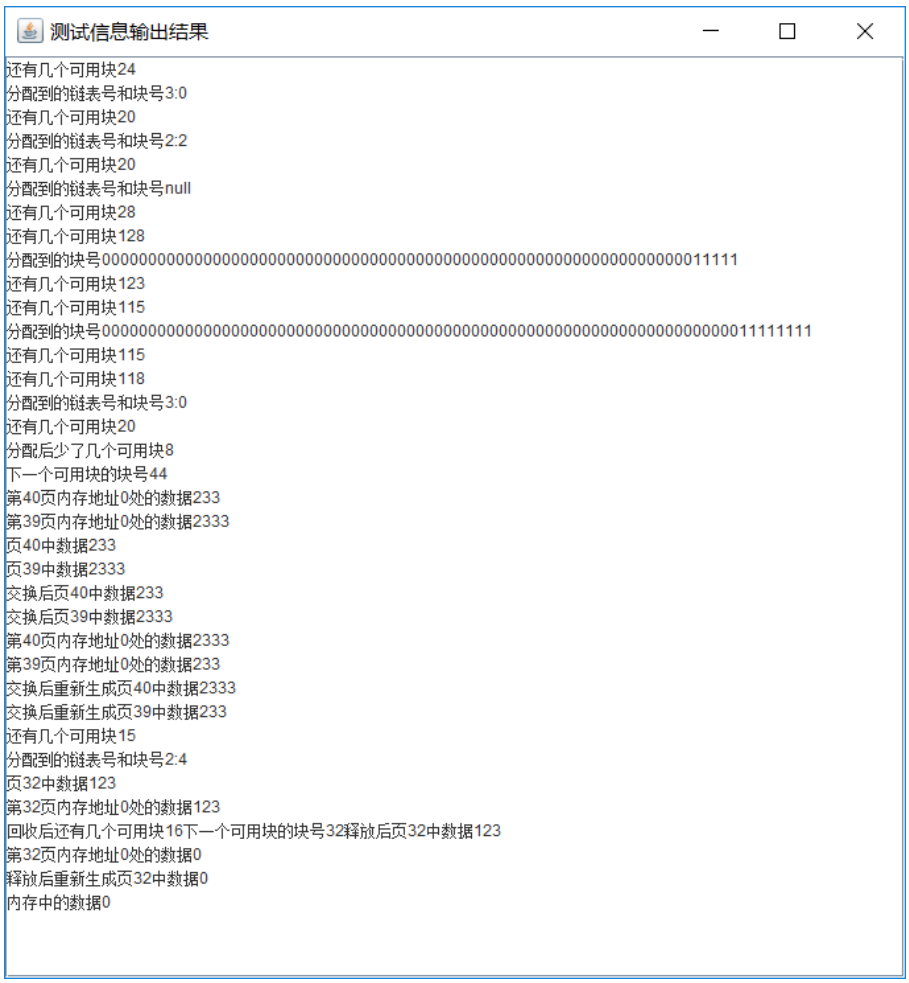
在 Main 函数中设置 CpuInfo 窗口显示，查看情况。



在 Main 函数中设置 PageModule 窗口显示，查看情况。



完整的测试程序输出。



七、总结与改进建议

1. 通过页面修改内存中数据时，内存中数据可以完成操作，但已获取的页面没有办法重新获得对应内存中的数据，需要重新获取该页面才可更新页面中的相关数据信息。可在内存中添加标志位判断是否进行修改，若有修改则自动更新对应地址的页面数据。

2. 页面编排时将所有页面按照统一编排进行序号设置，方便统一操作但降低了对于编程人员的友好性，可在内部结构中添加转换机制，在申请页面时将页面号由统一编排转换成独立编排，方便程序员进行申请，同时保证内部核心的稳定性。

3. 在根据伙伴算法释放页面时，应该添加判断信息，只有满足该页面已被伙伴算法分配且未被回收的条件，才可被回收。

4. 对于伙伴算法分配页面时产生的页面浪费现象，可以添加标志位，若被浪费的页的伙伴页被释放，则被浪费的页面也应该被同时释放，并同时更新 bitmap 信息。

5. bitmap 算法对于大量数据的 1/0 判断具有显著的性能提升，可应用在许多需要标志位数组的场景中提升性能。

6. UI 设计时没有考虑到双缓冲问题，导致画面会有闪烁或消失现象，可通过增加双缓冲功能解决。

7. 没有判断页是否被修改，在 LRU 算法时造成一定程度的效率下降。应在页表中设置修改位，同时将 LRU 算法判断标准加以改进，提高算法效率。

附件 3：核心代码

```
public static DeadLock dl=new DeadLock();//该类的静态对象
@SuppressWarnings("unchecked")
public ArrayList<PCB> []PV_apply=new ArrayList[10];    //PV 信号量当前的占用情况（P 过该资源但是没有释放的进程）
private int m=10;   //向量长度 m，每类资源中可供分配的资源数目
private int n=500;  //向量长度 n，进程个数

public int []Available=new int[m];              //每类资源中可供分配的资源数目
public int [][]Allocation=new int[n][m];        //已分配给每个进程的每类资源数目
public int [][]Request=new int[n][m];           //每个进程对每类资源的申请数目
private int []Work=new int[m];                      //长度为 m 的工作向量
private boolean []finish=new boolean[n];        //长度为 n 的布尔型向量

public boolean Step3_find_k_value(int k)
{
    //第 3 步，找到符合条件的 k 值
    //满足条件：finish[k]==false && Request[k , *] <= Work[*]
    //判断当前输入的 k 值是否符合条件
    boolean if_ok=true;
    for(int i=0;i<m;i++)
    {
        if(!(Request[k][i]<=Work[i]))
            if_ok=false;
    }
    return (finish[k]==false)&&if_ok;
}

public ArrayList<PCB> CheckDeadLock()
{
    //死锁检测

    //第一步，令 Work[*]=Available[*]
    ResetWork();
    InitFinish();
    //第二步，如果 Allocation[k,*]不等于 0，令 finish[k]=false;否则 finish[k]=true
    for(int k=0;k<n;k++)
    {
        if(!IfAllocationLineEmpty(k))
            finish[k]=false;
        else
            finish[k]=true;
    }
```

```
        //第三步，寻找一个 k 值
        for(int k=0;k<n;k++)
        {
            if(Step3_find_k_value(k)==true)
                //第四步，修改 Work[*]=Work[*]+Allocation[k,*],finish[k]=true
            {
                for(int j=0;j<m;j++)
                    Work[j]=Work[j]+Allocation[k][j];
                finish[k]=true;
                k=0;
            }
        }
        //第五步，查找处于死锁的进程
        ArrayList<PCB> dl_pcb=new ArrayList<PCB>();
        for(int k=0;k<n;k++)
        {
            if(finish[k]==false)
                dl_pcb.add(ProcessModule.process_module.GetPCBWithID((short) k));
        }
        return dl_pcb;
    }

    public void TransferJobCodeToSwapArea(JCB jcb,String apply)
    {
        //将指定作业的程序段存入虚存中
        //jcb 为作业控制块，apply 为申请到的虚存空间分配字符串
        int already_transfer=0;          //已经转移的页面数量
        short i=0;
        while(already_transfer!=jcb.GetProcessNeedPage())
        {
            if(apply.charAt(i)==1)
            {

PageModule.page_module.CopyPage((short)(jcb.GetInPageNum()+1+already_transfer), i);
                i++;
                already_transfer++;       //已经分配完一页
            }
            else
            {
                i++;
                continue;
            }
        }
    }
```

```
    public void WriteProcessPageTable(PCB pcb,String apply)
    {
        //将申请到的虚存页面写入到进程的页表中
        short count=0;
        for(int i=0;i<apply.length();i++)
        {
            if(apply.charAt(i)==1)
            {
                pcb.EditPageTable(count, count, (short) i);
                count++;
            }
        }
    }

    public void AddToPCBPool(PCB pcb)
    {
        //将 PCB 加入到 PCB 池中
        this.all_queue.add(pcb);        //加入到所有进程队列
        short pool_num=ApplyOnePCBInPool();            //向 PCB 池中申请一个 PCB，获取
PCB 序号
        pcb.SetPoolLocation((short) (pool_num));          //设置该 PCB 所在的具体页号
        pcb.SetInPageNum((short) (1+pool_num/2));              //设置 PCB 所在的页号
        pcb.WritePCBToMemory();          //PCB 内容写入内核区
    }

    public void RefreshActiveExpired()
    {
        //刷新 active 和 expired 指针
        boolean flag=true;
        for(int i=0;i<140;i++)
        {
            if(ready_queue[0][i].isEmpty()==false)        //该队列中还有进程
            {
                flag=false;
                break;
            }
        }
        if(flag==true)  //队列 0 为空
        {
            active=1;
            expired=0;
        }
        else
```

```
            {
                active=0;
                expired=1;
            }
        }

        public void ChangePageTable(short ori_page_num,short changed_page_num)
        {
            //将持有原来页的进程的页表更新
            for(int i=0;i<all_queue.size();i++)
            {
                PCB t=all_queue.get(i);
                for(short
j=0;j<(kernel.MEMORY_USER_SPACE_SIZE)/kernel.SINGLE_PAGE_SIZE;j++)
                {
                    if(t.CheckPageTable(j)==ori_page_num)
                        t.EditPageTable(j, j, changed_page_num);
                    if(t.CheckPageTable(j)==changed_page_num)
                        t.EditPageTable(j, j, ori_page_num);
                }
            }
        }

        public void SolveMissingPage(PCB pcb,short need_page_num)
        {
            //缺页中断的处理，need_page_num 为需要的在外存中的页的页号
            if(PageModule.page_module.GetFreePageNumInMemory()<1)        //内存满了
            {
                int out_page_num=PageModule.page_module.LRUGetLastPageNum();    //LRU
进行页面置换
                PageModule.page_module.ExchangePage((short) out_page_num, need_page_num);
        //页面置换
                ChangePageTable((short) out_page_num,need_page_num);//更新对应的进程的
页表
            }
            else        //内存没有满
            {
                short in_page_num=PageModule.page_module.GetOneFreePageInMemory();  //
在内存中申请一页
                PageModule.page_module.MoveToMemory(in_page_num, need_page_num);  //
将交换区的页移入
                PageModule.page_module.RecyclePage(need_page_num);
        //回收交换区
                ChangePageTable(in_page_num,need_page_num);
```

```
    //更新页表
        }
    }

    public static int SINGLE_PAGE_SIZE=512;//每一页/块的大小
    public static int MEMORY_SIZE=32*1024;//内存大小，32KB
    public static int MEMORY_KERNEL_SPACE_SIZE=16*1024;//内存内核空间大小，16KB
    public static int MEMORY_KERNEL_CORESTACKANDOSKERNEL_SIZE=512;      //核心栈+系统内核大小，1 页
    public static int MEMORY_KERNEL_PCB_POOL_SIZE=31*512;  //PCB 池大小，31 页
    public static int MEMORY_USER_SPACE_SIZE=16*1024;   //内存用户空间大小（页表、页框使用），16KB
    public static int HARDDISK_SIZE=1*1024*1024;    //硬盘空间大小，1MB
    public static int HARDDISK_VIRTUAL_MEMORY_SIZE=64*1024;     //虚存区大小，128 页，64KB
    public static int HARDDISK_SYSTEMFILE_SIZE=16*1024;  //系统文件大小，32 页，16KB
    public static int HARDDISK_FILE_SPACE_SIZE=944*1024;  //文件区大小，1888 页，944KB
    public static int HARDDISK_CYLINDER_NUM=32;//磁盘磁道数
    public static int HARDDISK_SECTOR_NUM=64;    //磁盘扇区数
    public static int HARDDISK_PAGE_SIZE=512; //磁盘每页/块大小
    public static int SINGLE_INSTRUCTION_SIZE=8;   //单条指令的大小
    public static int INSTRUCTIONS_PER_PAGE=SINGLE_PAGE_SIZE/SINGLE_INSTRUCTION_SIZE;      //每一页的指令数目

    public static int INTERRUPTION_INTERVAL=10;          //系统发生中断的间隔
    public static int SYSTEM_TIME=0;          //系统内时间
    public static void SystemTimeAdd() {kernel.SYSTEM_TIME+=kernel.INTERRUPTION_INTERVAL;}  //系统时间自增
    public static int TLB_LENGTH=kernel.MEMORY_USER_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE/2;
    //TLB 快表的长度，16
    /*系统基本信息*/

    /*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
```

```
    private void SetBlockState(int list,int no,int set_num,int state)
    {
        //在伙伴算法的链表中，在第 list 级别的第 no 块设置连续的 set_num 块为 state 状
态
        if(list>=0)
        {
            for(int i=no;i<no+set_num;i++)
            {
                free_area[list].set(i, (short) state);
            }
            SetBlockState(list-1,no*2,set_num*2,state);       //递归调用自身，修改该节点往
下的所有节点
        }
        else
            return;
    }

    public void RefreshBitmap()
    {
        //刷新 bitmap
        for(int i=0;i<6;i++)
        {
            for(int
j=0;j<(kernel.MEMORY_USER_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE)/((int)Math.pow(2,
i));j++)
            {
                this.bitmap[i]=SetOneBit(this.bitmap[i],j,this.free_area[i].get(j)==1?1:0);
            }
        }
    }

    private int SetOneBit(int num,int loca,int bit)
    {
        if(bit==1)
        {
            num|=(1<<loca);
        }
        else if(bit==0)
        {
            num&=~(1<<loca);
        }
        return num;
    }
```

```
    private int GetOneBit(int num,int loca)
    {
        return (num>>loca)&0x00000001;
    }

    private void RefreshBlockList()
    {
        //从底往上刷新块链表
        for(int i=0;i<=4;i++)
        {
            for(int
j=0;j<(kernel.MEMORY_USER_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE)/((int)Math.pow(2,
i));j+=2)
            {
                //对于占用的处理
                if(free_area[i].get(j)==1||free_area[i].get(j+1)==1)
                    free_area[i+1].set(j/2, (short) 1);
                //对于空闲的处理
                if(free_area[i].get(j)==0&&free_area[i].get(j+1)==0)
                    free_area[i+1].set(j/2, (short) 0);
            }
        }
        RefreshBitmap();   //刷新 bitmap
    }

    public String ApplyPageInMemory(short num)
    {
        //在内存中，向伙伴算法申请 num 个页面
        int pow=CalculateCloest2Num(num);  //求出该页面所需要申请的块所在的链表级
别
        String apply_str="null"; //默认为该页面无法找到
        for(int
i=0;i<(kernel.MEMORY_USER_SPACE_SIZE/kernel.SINGLE_PAGE_SIZE)/(int)Math.pow(2,
pow);i++)
        {
            if(free_area[pow].get(i)==0)  //找到未分配的块
            {
                apply_str=""+pow+":"+i;          //写入字符串，表示分配方式："链表:块号
"
                SetBlockState(pow,i,1,1);     //设置块链表的该位为 1
                RefreshBlockList();           //刷新块链表
                break;
            }
        }
```

```
        return apply_str;
    }

    public String ApplyPageInDisk(short page_num)
    {
        //在虚存中申请 page_num 个页面，返回一个 String 类型的值
        //String 格式的数据说明：从左到右编号为 0-191，共 192 位，每一位的值为 0/1，1 代表该页面分配给该进程使用。在写入程序区时，必须按照分配的页面顺序从小到大写入
        //同时，在记录数组中记录这些申请的页框，将他们设置为已用状态
        if(page_num>GetFreePageNumInDisk())
            return "null";
        int count=0;
        String str="";
        for(int i=0;i<this.swaparea_page_location_end;i++)
        {
            if(i>=this.swaparea_page_location_start)
            {
                if(this.if_page_usage[i]==false)    //找到空闲的页面
                {
                    str+="1";
                    count++;
                    this.if_page_usage[i]=true;
                }
                else
                    str+="0";
            }
            else
                str+="0";
            if(count==page_num)
                break;
        }
        return str;
    }

    public void FreePageInMemory(short page_num)
    {
        //在内存中，利用伙伴算法释放某一页
        //将这一页内容清空
        Page pa=new Page(page_num);
        for(short i=0;i<kernel.SINGLE_PAGE_SIZE;i+=2)
            pa.SetPageData(i, (short) 0);
        //在伙伴算法块链表中，将这一页所在的块清空
        SetBlockState(0,page_num-32,1,0);     //设置块链表的该位为 0
        RefreshBlockList();                   //刷新块链表
```

```
    }

    public void FreePageInDisk(short page_num)
    {
        //在外存中，释放某一页
        //将这一页内容清空
        Page pa=new Page(page_num);
        for(short i=0;i<kernel.SINGLE_PAGE_SIZE;i+=2)
            pa.SetPageData(i, (short) 0);
        //将这一页设置为未占用状态
        this.if_page_usage[page_num]=false;
    }

    public void RecyclePage(short page_num)
    {
        //回收页面，参数 num 为页面号（num 从 0 开始编号）
        //将该页的内容全部清空，并在记录中使得该页表示为未被占用

    if(page_num>=this.userspace_page_location_start&&page_num<this.userspace_page_locati
on_end)
            FreePageInMemory(page_num);  //释放用户空间中的页

    if(page_num>=this.swaparea_page_location_start&&page_num<this.swaparea_page_locatio
n_end)
            FreePageInDisk(page_num); //释放交换区中的页
    }

    public void LRUVisitOnePage(int page_num)
    {
        //LRU 访问某一页
        lru.remove(Integer.valueOf(page_num));          //将该页号从原来的里面删除
        lru.addFirst(Integer.valueOf(page_num));    //将访问的该页置顶
    }

    public short LRUGetLastPageNum()
    {
        //获得应该调出的页面号
        return lru.getLast().shortValue();          //获得链表的最后一项
    }

    public void SetPageAllData(short num)
    {
        SetPageNum(num);       //设置页号
        //初始化页面的数据
```

```
        if(page_num<kernel.MEMORY_SIZE/kernel.SINGLE_PAGE_SIZE)
        {
            //该页在内存中
            for(int i=0;i<kernel.SINGLE_PAGE_SIZE/2;i++)
            {
                data[i]=Memory.memory.GetData((short)
(CPU.cpu.mm.PageToRealAddress((short) this.page_num)+i*2));
            }
        }
        else
        {
            //该页在外存中
            for(int i=0;i<kernel.SINGLE_PAGE_SIZE/2;i++)
            {
                data[i]=HardDisk.harddisk.GetData(CPU.cpu.mm.PageToRealAddress((short)
this.page_num)+i*2);
            }
        }
    }

    public int PageToRealAddress(short num)
    {
        //将传入的页/块号 num 转换成为该页/块的基地址
        //该功能对内存与磁盘地址同样有效
        if(num<kernel.MEMORY_SIZE/kernel.SINGLE_PAGE_SIZE)
            return num*kernel.SINGLE_PAGE_SIZE;
        else
        {
            num=(short) (num-(kernel.MEMORY_SIZE/kernel.SINGLE_PAGE_SIZE));
            //cylinder 磁道、sector 扇区、offset 偏移
            int cylinder=num/kernel.HARDDISK_SECTOR_NUM;    //计算磁道
            int sector=num%kernel.HARDDISK_SECTOR_NUM;          //计算扇区
            int offset=0;                                                   //计算偏移
            return HardDisk.harddisk.CreateAddress(cylinder, sector, offset);
        }
    }

    public short VirtualAddressToRealAddress(PCB pcb,short virtual_address)
    {
        //将虚拟地址转换为实地址
        //注意，此时必须要保证所需要的页在内存中!!!!
        short virtual_page_no=GetVirtualAddress_VirtualPage(virtual_address);
        short virtual_offset=GetVirtualAddress_Offset(virtual_address);
        int real_page_no=FindRealPageNo(virtual_page_no); //在 TLB 中查找对应的页框号
```

```
        if(real_page_no==-1)          //快表中不存在该项目
        {
            //去内存中查询该页号对应的页框号，放入到 TLB 中
            AddTLB(virtual_page_no,pcb.CheckPageTable(virtual_page_no));
            //重新查询 TLB
            real_page_no=FindRealPageNo(virtual_page_no);
        }
        //else  快表中存在该项目
        if(PageToRealAddress((short) real_page_no)+virtual_offset<0x4000)
            return (short) ((short) (PageToRealAddress((short) real_page_no)+virtual_offset)+0x4000);
        else
            return (short) (PageToRealAddress((short) real_page_no)+virtual_offset);
```