**《计算机操作系统课程设计》**

**测试报告（自测）**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **题目**  **名称** | **可视化仿真实现Linux2.6进程管理与内存管理** | | | | | |
| **院系** | **信息学院** | **班级** | **网工161** | **测试时间** | **2019/03/13** | |
| **指导老师** | **姜海燕教授** | **助教**  **姓名** | **钱峥远** | | **电话**  **QQ** | **18013886150** |
| **姓名**  **学号** | **陈扬**  **19316117** | **手机**  **QQ** | **17761703297**  **824678119** | | **申请成绩** | **A+** |
| **是否组长** | **是** | **组长**  **手机** |  | | **组长**  **申请成绩** |  |
| **完**  **成**  **任**  **务**  **与**  **功**  **能**  **说**  **明** | 1. **作业调度**    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. **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。   1. **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.GetActivePoint()][i].isEmpty())  {  PCB t=ProcessModule.process\_module.ready\_queue[ProcessModule.process\_module.GetActivePoint()][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.GetCurrentInstructionNo()/(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.GetCurrentInstructionNo()));  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。   1. **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。   1. **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  }  }   1. **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。   1. **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。 | | | | | |
| **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。 | | | | | |
| 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）   1. **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）   1. **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 | | | | | |
| 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。   1. **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。   1. **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。   1. **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\*/ | | | | | |
| **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 | | | | | |
| **功**  **能**  **测**  **试**  **功**  **能**  **测**  **试** | **（一）系统常量**  程序设计了多种系统常量，读者可以在kernel.java中进行查看。同时，这些系统常量为静态类型，即说明在任何地方都可以进行使用。  测试方法如下：  为了调用这些变量，可以在Main函数中直接调用system.out.println函数进行测试，测试代码如下：    输出的结果如下：    可见，输出的结果正确。  根据测试，可以得到正确的输出值。此处只测试了一小部分变量，在测试其他变量时，读者可以使用kernel进行引导测试。此处的测试只是为了说明在JAVA中，静态变量是一个全局通用的变量类型。  **（二）PCB测试**  在测试PCB时，可以在Main函数中实例化一个PCB类的对象，并对其进行操作。程序在设计PCB时，提供了多种不同的Get与Set函数，可以保证在测试时对PCB结构的完全控制。  测试方法如下：    输出的结果如下：    可见，测试结果正确。  对于PCB的使用，对于PCB类的初始化与回收，是在程序运行过程中常用的一个功能。具体的使用方法，可以参考三级调度函数的写法与功能。  **（三）JCB测试**  程序设计时，创建了JCB的数据结构（其结构可以见上文的描述）。JCB结构是对于作业的描述，其中也包含有一定数量的变量可供使用。程序为此提供了Get方法与Set方法，读者可以在Main函数中实例化一个JCB类的实例化对象，然后对其进行调控测试。  测试的方法与上文中描述的PCB的测试方法一样，在此不做赘述。  测试方法：    测试结果如下：    可见，测试结果正确。  **（四）死锁检测**  程序的功能之一便是实现了死锁的模拟仿真与检测算法。在控制台界面下，读者可以不通过UI界面直接测试死锁检测算法的正确性。  在程序设计中，DeadLock类对外提供了多种函数API（详情见上文），读者可以调用他们来模拟每一个进程的申请操作。  主要用到的API有：  public boolean Process\_Apply\_Resource(PCB pcb,int num)  //某个进程申请某个资源  public int GetResourceNum(int num)  //获取某类资源的数目  public void Process\_Return\_Resource(PCB pcb,int num)  //某个进程归还资源  public ArrayList<PCB> CheckDeadLock()  //死锁检测  通过这些函数，读者便可以模拟仿真每一个进程的资源申请请求与死锁的检测。  同时，由于这些函数本身并不面向控制台提供功能，所以比较难以调用。读者可能会因为不理解本程序的设计而不能够操作。  在此，提供测试的方法。  现在，假设死锁产生的条件：有A资源1个，B资源2个。有进程1与进程2，进程1与进程2申请资源的顺序为：进程1申请1个A资源并成功，进程2连续申请2个B资源并成功，进程1申请B资源失败并等待，进程2申请A资源失败并等待。以上模拟的是一种比较简单的死锁，将其中代码的形式表示可以为：    执行结果为：    可见，运行结果正确！  **（五）UI界面**  本程序的三级调度功能无法通过控制台界面与Main函数接口直接调用进行测试。因此，下文将展示如何通过UI界面进行操作，来检验不同的函数的正确性。  同时，由于程序的函数实现较多，逻辑上较为复杂。读者若不能够理解程序的含义，建议通过Eclipse自带的单步调试功能进行测试。  整个程序由Main函数进入，由UI界面进行整个程序的调控。所以，Main函数的设计较为简单，为：    通过此函数的调用，可以直接打开主界面。  当程序运行后，出现主界面，截图如下：    在开始时，主界面没有任何内容，需要读者手动导入作业并运行。  作业导入后，点击“启动”按钮便可以开始运行，点击“清屏”按钮，便可以清空当前“运行信息”文本框中的信息。  其他按钮的功能为：  创建作业：打开一个创建作业的窗体，进行创建作业。  退出：退出程序  内存界面：打开内存数据查看界面，可以查看当前内存中所有的数据信息  磁盘界面：打开磁盘数据查看界面，可以查看当前磁盘中所有的数据信息  页面检测：打开页面管理的检测界面，可以查看内存、虚存、磁盘的空间占用情况和伙伴算法的运行原理和bitmap  CPU检测：打开CPU检测窗体，可以检测此时CPU的具体中断信息与当前系统时间。  这些窗体的截图如下：          **（六）三级调度**  为了模拟实现作业、进程的三级调度，程序在界面中将以直观的方式予以体现。  以上述步骤进行作业导入后，点击“启动”开始程序。在程序的运行结果输出中，程序将按照时间间隔的不同输出三级调度的不同信息。  如图：        **（七）指令运行**  程序最基本也是最重要的功能便是指令的运行。在该测试中，将测试程序对于指令的运行模拟，并分析其正确性。  点击“载入作业”，从文件中读取作业，此处使用了“/测试输入”文件夹下的样例1进行测试，程序的运行结果如下：            将程序的运行结果输出保存到文件，文件内容如下：    该作业中全部为普通类型指令，即不存在系统调用，因此不会进入等待队列。当进程运行到时间片到时，则进行进入就绪态，进行重新调度。根据进程的优先级不同，调入优先级最高的进程进行运行。  接着，开始测试系统调用的情况。  选择“/测试输入”文件夹下的测试样例2文件进行导入，查看运行结果。结果如图所示：    可以看到，当执行到系统调用指令时，CPU直接进入等待状态。在下一次进行低级调度时，CPU直接会检测到当前的等待队列中有进程且还没有能够达到出等待态的条件，这时，CPU直接放弃低级调度，继续等待等待队列中的指令执行完成。  测试PV操作的情况，同样可以使用“/测试输入”文件夹中的对应文件进行输入，输出的结果，可以反映出程序试图P和试图V信号量时的具体信息。通过这些测试输入，便可以体现出PV操作的同步互斥的状态，如图：    测试死锁状态检测功能，同样可以导入对应的测试样例，检查输出文件：    可以看到，程序自动检测到了死锁，并调用了该进程的撤销原语进行撤销。当该死锁进程被撤销后，其他的进程便可以继续开始运行，直至所有进程运行结束并退出。 | | | | | |
| **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。 | | | | | |