**《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** | **计科161** | **测试时间** | **2019/3/13** | |
| **指导老师** | **姜海燕教授** | **助教**  **姓名** | **钱峥远** | | **电话**  **QQ** | **18013886150** |
| **姓名**  **学号** | **梁嘉文**  **19216126** | **手机**  **QQ** | **15326099602**  **635229914** | | **申请成绩** | **B** |
| **是否组长** | **否** | **组长**  **手机** | **17761703297** | | **组长**  **申请成绩** | **A+** |
| **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\_location\_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\_location\_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 | | | | | |
| **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000000000000000000000000000000000000000000000000000000000000000011111，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。 | | | | | |
| 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. 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  } | | | | | |
| **功**  **能**  **测**  **试**  **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     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。        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。        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。   1. 释放内存空间     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。     1. 分配虚存空间   Main函数中调用函数申请5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。          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。       1. 释放虚存空间     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。     1. 通过页面修改内存数据             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。            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。    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。      调用函数释放页32。    查看页面数据和内存第32页处数据。    同样重新获取第32页后查看数据。      结果正确。   1. 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。 | | | | | |
| **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。 | | | | | |