**陈扬核心代码**

**一、死锁检测**

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;

}

**二、进程管理**

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

}

}

**三、内核变量**

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