**部分核心代码**

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\_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 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);

}