目录

Linux2.6进程管理系统的仿真实现

计算机科学与技术专业

指导老师

摘要：本次课程设计仿真实现Linux2.6进程管理系统，以操作系统原理为理论指导，采用面向对象的设计方法进行设计，力争实现各个模块之间低内聚，高耦合。该系统采用的程序设计语言为Java，开发采用的JDK版本为Java SE-15。最终基本完成了硬件系统、存储管理（内存与磁盘）、文件管理、作业与进程管理、设备管理以及可视化界面等基本功能。对于硬件仿真部分，完成了对地址线与数据线、物理块、内存、磁盘、CPU、MMU、时钟和随机数发生器的设计，并对系统资源进行了规划。而在数据结构与基础操作板块，本文在作业控制块、指令集设计、进程控制块、阻塞队列、OS内核原语、固定分配的映射、页表上进行抽象和设计，并完成了文件Innode、i_Inode表以及外部设备表等的规划设计。在程序结构和模块功能实现的过程中，本文设计了系统工作流程（含时钟、CPU、指令执行流程），并对作业调度、进程调度、页面置换的算法进行研究，给出磁盘空闲块的管理方法和外部设备响应算法。此外，本文还针对部分测试用例，对MMU地址变化、缺页中断、内存空间分配、页表生成、作业与进程调度等一系列内容进行测试和分析并得出结论，针对部分技术问题提出解决方案，如超级块设计中采用SuperBlock类完成本地存储以及使用notifyAll方法实现各个线程的通信等。

关键词：操作系统，Linux，Java

1. 实践内容与目标

1.1. 实践内容

本次课程设计仿真实现 Linux2.6 进程管理系统，以操作系统原理为理论指导，采用面向对象的设计方法进行设计，力争实现各个模块之间低内聚，高耦合。该系统采用的程序设计语言为 Java，开发采用的 JDK 版本为 Java SE-15。

1.2. 目标

通过课程所学以及 Linux 系统源码的阅读，我最终基本完成了如下基本功能：硬件系统、存储管理（内存与磁盘）、文件管理、作业管理、进程管理、设备管理以及可视化界面。

1.3. 完成任务

子系统	任务	描述
硬件系统	地址线与数据线	完成了硬件系统的物理结构仿真及其功能仿真，其具体内容详见“硬件仿真设计”。
	物理块	
	内存	
	磁盘	
	时钟	
	CPU	
	MMU	
	随机数发生器	
存储管理	内存空闲区管理	内存空闲 Inode 用位示图法，内存中的空闲物理块用位示图法。分配策略采用固定分配，局部替换。
	磁盘空闲区管理	磁盘空闲 Inode 用位示图法，磁盘中的空闲物理块用成组链接法。
	页表设计	每个进程分配一个页表，页表项数始终为 2.
	虚拟页表机制	给一个进程分配的虚拟空间为 32 块，调入内存的页框为数 2。
	缺页中断	均由 MMU 完成。
	地址变换	
文件管理	超级块	具体内容见“数据结构”部分。
	Inode	
	i_Inode 表	
	系统文件打开表	
	进程文件打开表	
	Inode 高速缓冲	将部分经常使用的 Inode 存储在内存中。
	文件操作	提供打开文件，关闭文件，读文件，写文件操作。

	文件保护	设置 RWX 标志位。
作业管理 进程管理	作业池	作业池由一个个 JCB 组成，JCB 与作业一一对应。
	进程池	进程池存放当创建的 PCB，PCB 与进程一一对应。
	作业初始化	开机时读取初始作业集。
	新建作业	创建随机作业。
	作业调度算法	静态优先级+先来先服务。
	进程调度算法	静态优先级+时间片轮转。 参考五态转换模型。
	进程资源申请	申请内存资源（代码块+数据块）。
	死锁预防	系统资源设置上来预防死锁。
	指令集	表示某个作业需要进行的操作。
设备管理	设备表	一张包括系统所有外设信息的表。
	文件系统调用阻塞队列	具体内容见“数据结构”部分。
	键盘输入阻塞队列	
	屏幕输出阻塞队列	
	打印机阻塞队列	
	摄像头阻塞队列	
	打印机等待阻塞队列	
	阻塞工作线程	具体内容见“模块功能实现”部分。
	阻塞检查唤醒线程	
可视化界面	开机界面	登录，重装系统，查看系统开机过程等功能。
	主界面	程序实时运行信息，新建作业，进入黑盒信息界面等功能。
	黑盒信息界面	内存管理，磁盘管理，文件管理，作业管理，进程管理，设备管理。

2. 系统功能与 API 函数说明

2.1. 硬件系统

类	函数	功能
CPU	void run()	CPU 工作
CPU	void PCBRunning(Memory memory, int pcbRunIndex)	CPU 执行一条指令
Memory	void Initialize(Disk disk)	内存初始化
Disk	void doLoad()	磁盘初始化
Disk	int BlockTransform_Location_To_Index(int cyliner, int track, int sector)	把柱面，磁道，扇区转换为磁盘块号
Disk	int[] BlockTransform_Index_To_Location(int index)	把磁盘块号转换为柱面，磁道，扇区
Computer	void Initlize()	开机

Computer	void rebuild(boolean flag)	重装系统
Clock	void run()	开启时钟线程
Language	String getValue(String key)	将二位十六进制转化为对应字符。
Language	String getKey(String value)	将一个字符转化为二位十六进制。
RandomGenerator	int getRandomOperand(int type)	根据指令类型获得对应的随机操作数
RandomGenerator	int getRandomFromPageTable(PageTable pageTable)	根据页表获得不产生缺页中断的逻辑地址
RandomGenerator	int getRandomFromLogicalSpace()	产生整个逻辑地址空间的逻辑地址
RandomGenerator	int getRandom(int start, int end)	产生[start,end]范围内的随机数
RandomGenerator	int getRandomFromVC()	获得高频率不产生阻塞的指令类型。
RandomGenerator	int getRandomFromFile()	获得高频率文件系统调用类型的指令类
RandomGenerator	int getRandomFromOther()	获得高频率长时间阻塞的指令类型
Block	String INT_TO_HEX(int dec)	十进制转换为十六进制
Block	int HEX_TO_INT(String hex)	十六进制转换为十进制
Block	void writeBack()	写回到 txt 映像文件中
Block	void setDataFromSringArray(int start, int length, String[] stringArray)	修改块中 start 开始长度 length 字的数据
Block	String[] getStringArrayFromData(int start, int length)	获取块中 start 开始长度 length 字的数据
Block	String getDataAtIndex(int Index)	获得块内 index 偏移量的数据
Block	void setDataAtIndex(int Index, String Data)	设置块内 index 偏移量的数据
Block	String[] getAllData()	获得块内所有数据

2.2. 存储管理

类	函数	功能
Memory	void writeToDisk(int indexM, int indexD, Disk disk)	把内存中第 indexM 块内容写会到磁盘第 indexD 中
Memory	void writeFromDisk(int indexM, int indexD, Disk disk)	将磁盘中第 indexD 块内容写到内存 indexM 中
Memory	void allocateIndexBlock(int index)	分配一个 index 号的块
Memory	void reclaimIndexBlock(int index)	回收一个 index 号的块
Memory	void loadAnInode(int index, Inode inode)	装载一个 inode
Disk	void writeFreeBlocksGroupsLink(int number,	写入成组链接的空闲块链

	FreeBlocksGroupsLink link, boolean flag)	
SuperBlock	int getFistFreeBlockIndexInMemory()	获得一个空闲内存块号
SuperBlock	void loadFromDisk(Block block)	从磁盘块中加载超级块
SuperBlock	int[] Inode_IndexToLocation(int index)	将 Inode 的逻辑位置转换为物理位置
SuperBlock	int Inode_LocationToIndex(int x, int y)	将 Inode 的物理位置转换为逻辑位置
SuperBlock	int allocateAFreeBlock(Disk disk)	分配一个磁盘空闲块
SuperBlock	int reclaimAFreeBlock(int number, Disk disk)	回收一个磁盘空闲块
SuperBlock	int[] getFirstFreeInodeLocationInMemory()	获得一个内存中空闲 Inode 的物理位置
SuperBlock	int[] getFirstFreeInodeLocationInDisk()	获得一个磁盘中空闲 Inode 的物理位置
SuperBlock	void reclaimAnInodeInMemory(int index)	回收一个内存的 Inode
SuperBlock	int allocateAnInodeInMemory()	申请一个内存的 Inode
MMU	boolean isMissPage(PCBPool pcbPool, int pcbNumber, int logicalAddress)	判断是否会引发缺页中断
MMU	Address getAddressFromPageTable(PCBPool pcbPool, int pcbNumber, int logicalAddress)	将逻辑地址转换为物理地址
MMU	void MissPage(PCBPool pcbPool, int pcbNumber, int logicalAddress)	缺页中断处理
PageTable	void accessAPage(int page)	访问某个页
PageTable	void writeBackWhenReplace(int index)	写会操作
PageTable	void addAPageItem(int page, int mblock, int dblock, Memory memory, Disk disk)	添加一个页表项
PageTable	boolean deleteAPageItem(int page, Memory memory, Disk disk)	删除一个页表项
PageTable	int getReplaceedPageNumberWithLRU()	确定应被替换的页号
PageTable	void Initlize(int pcbIndex)	初始化

2.3. 文件管理

类	函数	功能
ProcessFileTable	void add(String des, int number)	添加一项进程文件打开表项
ProcessFileTable	void remove()	移除一项进程文件打开表项
OpenFileTable	void addAOpenFile(int number, int pNumber, int ptr)	添加一个表项
OpenFileTable	void removeAOpenFile(int number)	删除一个表项
OpenFileTable	void incPNumberForInode(int number)	使 number 号 inode 的进程使用数加一
OpenFileTable	void decPNumberForInode(int number)	使 number 号 inode 的进程使用数减一
i_InodeTable	void loadFromDisk(Disk disk, SuperBlock sBlock)	加载 i_Inode 表
Inode	void doIitialize(String[] dateInBlockString)	初始化 Inode

Inode	String[] getInodeString()	将 Inode 转换为磁盘存储格式

2.4. 作业管理与进程管理

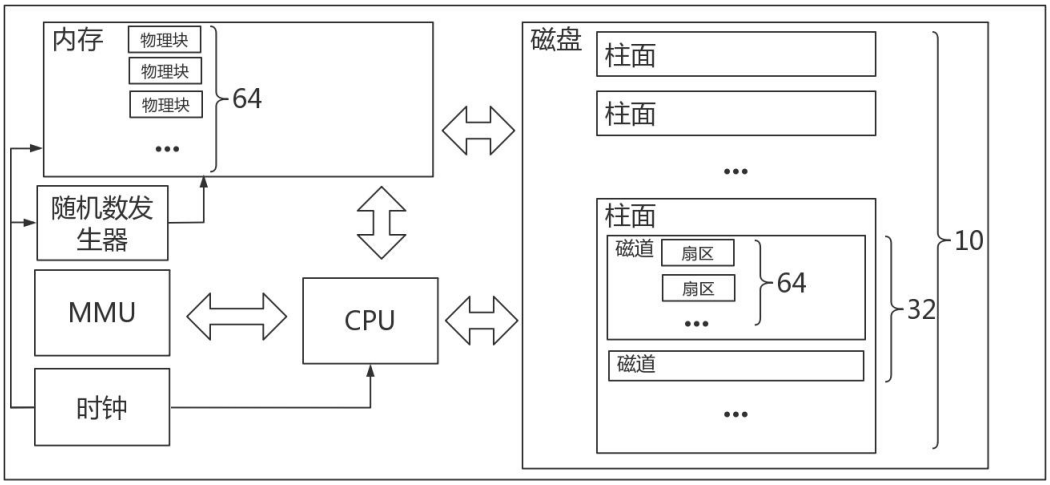
类	函数	功能
JCBPool	void Initialize()	读取 input 文件夹中的内容创建 JCB
JCBPool	int checkReturnIndex()	检查是否有到达时间的作业
JCB	void make(String data)	用本地作业文件初始化 JCB
JCB	void randMake(int type, int p, int length, JTextArea text)	用随机数初始化 JCB
PCB	void createFromJCB(JCB jcb, int pcbIndex)	从 JCB 里创建一个 PCB
Memory	int checkPCBPoolIsNew()	是否有新建态
Memory	int checkPCBPoolIsRunning()	是否有运行态
Memory	int checkPCBPoolIsReady()	是否有就绪态
Memory	int checkTimeIsFull()	是否有时间片满
Memory	int JcbToNew(int jcbIndex)	JCB 创建为新建态
Memory	void NewToReady(int pcbIndex)	新建态转换为就绪态
Memory	void ReadyToRun(int pcbIndex, CPU cpu)	就绪态转换为运行态
Memory	void RunToReady(int pcbIndex, CPU cpu)	运行态转换为就绪态
Memory	void RunToBlock(int pcbIndex, CPU cpu, Instruction instruction)	运行态转换为阻塞态（等待）
Memory	void BlockToReady(int pcbIndex)	阻塞态转换为就绪态（唤醒）
Memory	void ToDead(int pcbIndex)	消亡

2.5. 设备管理

类	函数	功能
DeviceTable	void Initlize()	设备表初始化
ProcessBlock	int getMaxPriorityIndex()	获取优先级最高的
ProcessBlock	void workFile()	文件系统调用（磁盘）工作
ProcessBlock	void workPrinter()	打印机工作
ProcessBlock	void work(int index)	阻塞队列中某进程占用该设备工作
ProcessBlock	int check()	检查是否有已完成的阻塞进程
BlockWorkThread	void run()	阻塞进程占用资源工作线程
BlockAwakeThread	void run()	检查并唤醒阻塞进程线程

3. 硬件仿真设计

3.1. 硬件系统设计思路



3.2. 地址线与数据线

系统的地址线与数据线均为 16 位，即为两字节，一个字。

3.3. 物理块

物理块按字编址，一个物理块的寻址范围为 0~255，即物理块大小为 512B。以下为抽象的出来的类。

Block 类的主要属性	
属性名	含义
cylinderNumber	柱面号
trackNumber	磁道号
sectorNumber	扇区号
memoryNumber	内存块号
data	块内数据，是一个 256 长度的数组

3.4. 内存

内存共有 64 个物理块，本系统对内存进行分区。内存中系统区为 0~15 共 16 块，用户区为 16~63 共 64 块。内存的分配策略为固定分配，局部替换。系统共支持 12 个进程并发执行，并为每个进程分配 3 个空闲内存块，一块不参与页面替换作为代码块，一块参与页面替换作为数据块。

采用虚拟页式存储，每个进程在磁盘里的数据块为 32 块，所以它的的逻辑寻址空间被扩大到 0~8191。

内存分区情况如下。

内存分区情况	
块号	描述
0	存放引导块

1	存放超级块
2~5	操作系统进程
6、7	Inode 块
8~15	系统表
16~27	进程代码块
28~51	进程数据块
52~63	缓冲区

3.5. 磁盘

磁盘共有 10 个柱面，每个柱面有 32 个磁道，每个磁道有 64 个扇区，每个扇区大小为 512B，可以看做一个物理块，同时每一块在 disk 文件夹中又有对应的 txt 文件。由于系统采用虚拟页式存储，所以磁盘中需要预留 12*32 块作为交换区。同时还要为作业内容，代码文件分配存储空间。所以最终 0~2047 作为目录区，2048~20479 作为文件区。

磁盘分区情况如下。

磁盘分区情况	
块号	描述
0	存放引导块
1	存放超级块
2~119	Inode 区
120~127	系统表
128~511	交换区
512~575	作业区
576~1023	进程代码区
1024~2047	预留区域
2048~20479	文件区

3.6. CPU

CPU 类继承自 Thread 类，其中属性有程序计数器，指令寄存器和四个通用寄存器，以及一个标志位 state 来表示是出于用户态还是核心态。CPU 中有一个 run 方法仿真开机后 CPU 的工作。

3.7. MMU

MMU 作为地址变换机构，其方法均为静态方法，主要用于缺页中断和将逻辑地址转换为物理地址。

3.8. 时钟

时钟被抽象为 Clock 类，它继承自 Thread 类，它也有一个 run 方法，用于模拟每一秒钟的时钟中断，每次中断，会使类内计数器加一，而且时钟中断会释放各种锁，唤醒其他锁对应线程，从而使得程序有序进行。

3.9. 随机数发生器

随机数发生器被抽象为 RandomGenerator 类，其中的方法全部是静态方法，用于产生各种随机数，比如新建作业时作业各个属性的随机数，产生指令序列各种指令的随机数，针对不同种类的作业，还可以产生指令类型有一定概率的指令集。

3.10. 系统资源

系统共有 5 种资源，分别是缓冲区（12），键盘（1），屏幕（1），打印机（3），摄像头（1）。由于缓冲区资源远远大于完成读写文件阻塞（2 秒）的时间，所以这里也预防了死锁。

4. 数据结构与基础操作的抽象与设计

4.1. 作业控制块

JCBPool 为作业池的抽象，其中有 JCB 类型的数组来表示各个作业的抽象。JCB 抽象如下。

JCB 类的属性	
属性	**描述**
arrivalTime	作业到达时间
startTime	作业开始时间
finishTime	作业结束时间
jobNumber	作业号
jobName	作业名
codeLength	代码指令长度
priority	作业优先级

其中，在“创建作业”的功能中，可以选择三种作业类型，分别是寻址计算高频、文件读写高频、长时阻塞高频，他们的各种指令及其概率如下。

指令类型	寻址计算高频%	文件读写高频%	长时阻塞高频%
Value	40	10	10
Calculate	40	10	10
Read	5	30	0
Write	5	30	0
Input	3	5	25
Output	3	5	25
Printer	3	5	20
Camera	1	5	10

4.2. 指令集设计

本操作系统设计的所有指令如下。

代号	指令类型	操作数	功能
0	Value	0-5	声明一个变量。产生一个不产生缺页中断的逻辑地址,并进行寻址。操作数表示变量类型:int,short,string, char, long。
1	Calculate	0-3	计算。产生一个覆盖所有逻辑地址的随机数，并进行寻址。操作数表示计算方式: add,sub,mul,div。
2	Read	0-9	读文件。打开并读文件（inode 号为操作数），两秒后关闭文件。
3	Write	0-9	写文件。打开并写文件（inode 号为操作数），两秒后关闭文件。
4	Input	5-15	键盘输入。输入时间由操作数指定。
5	Output	5-15	屏幕输出。输出时间由操作数指定。
6	Printer	0	打印机服务。操作数无意义。
7	Camera	0	摄像头服务。操作数无意义。

4.3. 进程控制块

PCBPool 为作业池的抽象，其中有 PCB 类型的数组来表示各个作业的抽象。PCB 抽象如下。

PCB 类的属性	
属性	描述
startTime	进程开始时间
runTime	进程占用 CPU 时间
finishTime	进程结束时间
processNumber	进程号
processName	进程名
priority	优先级
state	进程状态（五态模型）
codeLength	代码长度
codes	代码数组
timeFlak	已用的时间片
pageTable	进程页表
processFileTable	进程打开文件表
scene	保存的现场

4.4. 阻塞队列

进程会根据指令执行进入不同的阻塞队列中，这里将资源阻塞抽象为同一个类 ProcessBlock，其中保存的为该阻塞队列中的 PCB 阻塞信息，该信息被抽象为 PcbBlock，而打印机资源等待阻塞队列则被像为 WaitingBlock，根据各种阻塞形式的不同，会调用不同的阻塞占用资源工作的方法。

ProcessBlock 将被实例化为 fileBlock，inputBlock，outputBlock，printerBlock

与 cameraBlock 对象。而 WaitingBlock 则被实例化为 waitingPrinter。

以下为关键类的属性。

ProcessBlock 类的属性	
属性	描述
list	PcbBlock 类型的数组

PcbBlock 类的属性	
属性	描述
pcbIndex	PCB 指针，指向进程池中的某个 PCB
priority	优先级
arrivalTime	进入阻塞队列的时间
time	在阻塞队列中经过的时间
blockTime	应当阻塞的时间

通过 blockTime 值的设置，可以实现在一个阻塞队列中，其中里面的进程应阻塞时间不同，即为 input 和 output 指令的阻塞效果。各种阻塞及其时长如下。

阻塞时长	
阻塞	时长
fileBlock（文件系统调用）	2
inputBlock（键盘输入）	operand（自定义）
outputBlock（屏幕输出）	operand（自定义）
printerBlock（打印机服务）	6
cameraBlock（摄像头服务）	20
waitingPrinter（等待打印机）	无限（直到有打印机资源）

4.5. OS 内核原语

4.5.1. 由 JCB 创建 PCB 新建态的原语

```
public void JcbToNew(int jcbIndex)
{
    获取空闲 pool 指针
    if(获取失败)
        创建失败
    else
    {
        新 PCB 初始化
        拷贝 JCB 信息到新 PCB 中
        设置新 PCB 为新建态
        分配内存资源
        删除被创建后的 JCB
    }
```

```
}
```

4.5.2. 新建态转换为就绪态原语

```
public void NewToReady(int pcbIndex)
{
    设置 PCB 状态为就绪态
}
```

4.5.3. 就绪态转换为运行态原语

```
public void ReadyToRun(int pcbIndex)
{
    设置 PCB 状态为运行态
    设置 PCB 初始已使用时间片为 0
    恢复现场
}
```

4.5.4. 运行态转化为就绪态原语

```
public void RunToReady(int pcbIndex)
{
    设置 PCB 状态位就绪态
    保护现场
}
```

4.5.5. 运行态转换为阻塞态原语

```
public void RunToBlock(int pcbIndex, Instruction instruction)
{
    设置 PCB 状态为阻塞态
    保护现场
    switch(指令类型)
    {
        case Read:
        {
            打开文件
            读文件
            进程加入到 fileBlock 阻塞队列中
        }
        case Write:
        {
            打开文件
            写文件
            进程加入到 fileBlock 阻塞队列中
        }
        case Input:
            进程加入到 inputBlock 阻塞队列中
        case Output:
            进程加入到 outputBlock 阻塞队列中
        case Printer:
        {
```

```
                if(无打印机资源)
                    进程加入到 printerWaiting 阻塞队列中
                else
                    进程加入到 printerBlock 阻塞队列中
            }
            case Camera:
                进程加入到 cameraBlock 阻塞队列中
        }
    }
```

4.5.6. 阻塞态转换为就绪态原语

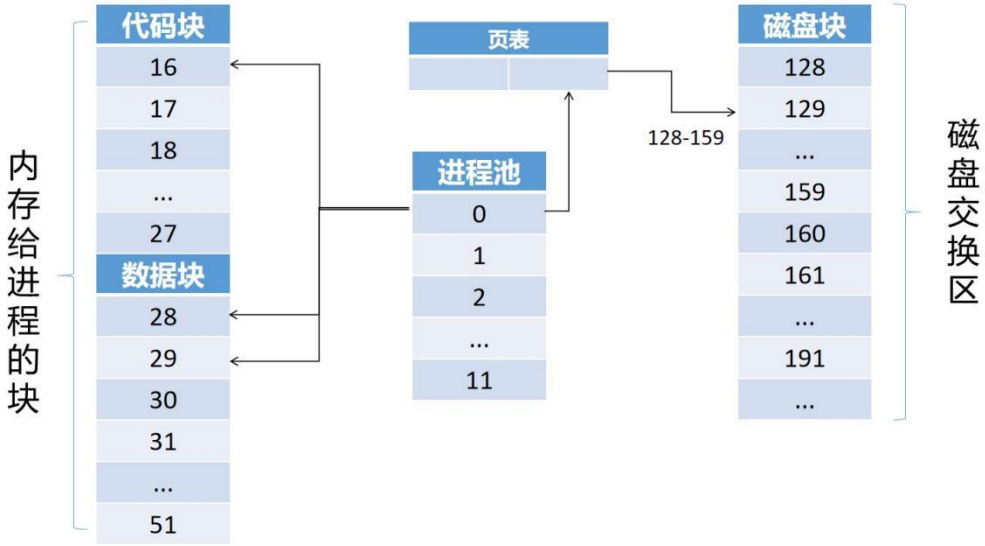
```
    public void BlockToReady(int pcbIndex)
    {
        进程离开阻塞队列
        相应资源数 + 1
        设置 PCB 状态为就绪态
    }
```

4.5.7. 转换为终止态原语

```
    public void ToDead(int pcbIndex)
    {
        设置 PCB 状态为终止态
        设置终止时间
        计算进程 CPU 占有率
        移除 PCB
        回收内存
    }
```

4.6. 固定分配的映射

下图展示了系统运行期间，内存分配给进程的块，进程池，磁盘中块的映射关系。



可以看到，进程池第 0 位置上，系统给他分配的内存块为 16（代码块）和 28,29（数据块），磁盘上的交换区部分为 128-159 共 32 块。进程池其他位置上则以此类推。

4.7. 页表

系统为每个进程分配一张页表，其数据结构如下。

PageTable 类的属性	
属性	描述
pageNumber	逻辑页号和 LRU 标志位
memoryBlockNumber	内存物理块号
diskBlockNumber	外存物理块号
isModify	是否修改

4.8. 文件 Inode

单个文件和 Inode 一一对应，文件 Inode 被抽象为 Inode 类，其属性值如下。

Inode 类的属性	
属性	描述
Number	Inode 号
fileName	文件名
fileType	文件类型
filePermission_R	读权限
filePermission_W	写权限
filePermission_X	执行权限
createTime	创建时间
modifyTime	修改时间
linkNum	硬链接数（未应用）
quoteNum	内存引用数（未应用）
blockNum	物理块数（文件大小）
directAddressList	是一个长度为 8 的数组，直接地址
primaryReferenceAddress	一级间接地址
secondaryReferenceAddress	二级间接地址

4.9. i_Inode 表

i_Inode 表在系统开机时就已经装入内存，其中存储的事系统的全部 Inode 索引信息。它被抽象为 i_InodeTable 类，其属性值如下。

i_InodeTable 类的属性	
属性	描述
fileName	文件名
inodeIndex	Inode 在磁盘中的索引值

inodeNumber	Inode 号
isLoadMemory	是否装入内存
isDir	是否是文件
next	如果为文件，其下属含有的文件 Inode 号

4.10. 系统文件打开表

为了方便快速访问 Inode，系统会将常用的 Inode 装入内存 Inode 区中，所以需要一张系统表来管理它们，即被抽象为 OpenFileTable 类。

OpenFileTable 类的属性	
属性	**描述**
inodeNumber	inode 号
inodePtr	在内存中的 inode 指针
processNumber	打开此文件的进程数
isModify	是否修改

4.11. 进程文件打开表

系统为每个进程设置一张打开文件表，同时规定一个进程只能打开一个文件，将其抽象为 ProcessFileTable 类，其属性值为系统分配的文件描述符 descriptor，和打开文件的 inode 号 inodeNumber。

4.12. 外部设备表

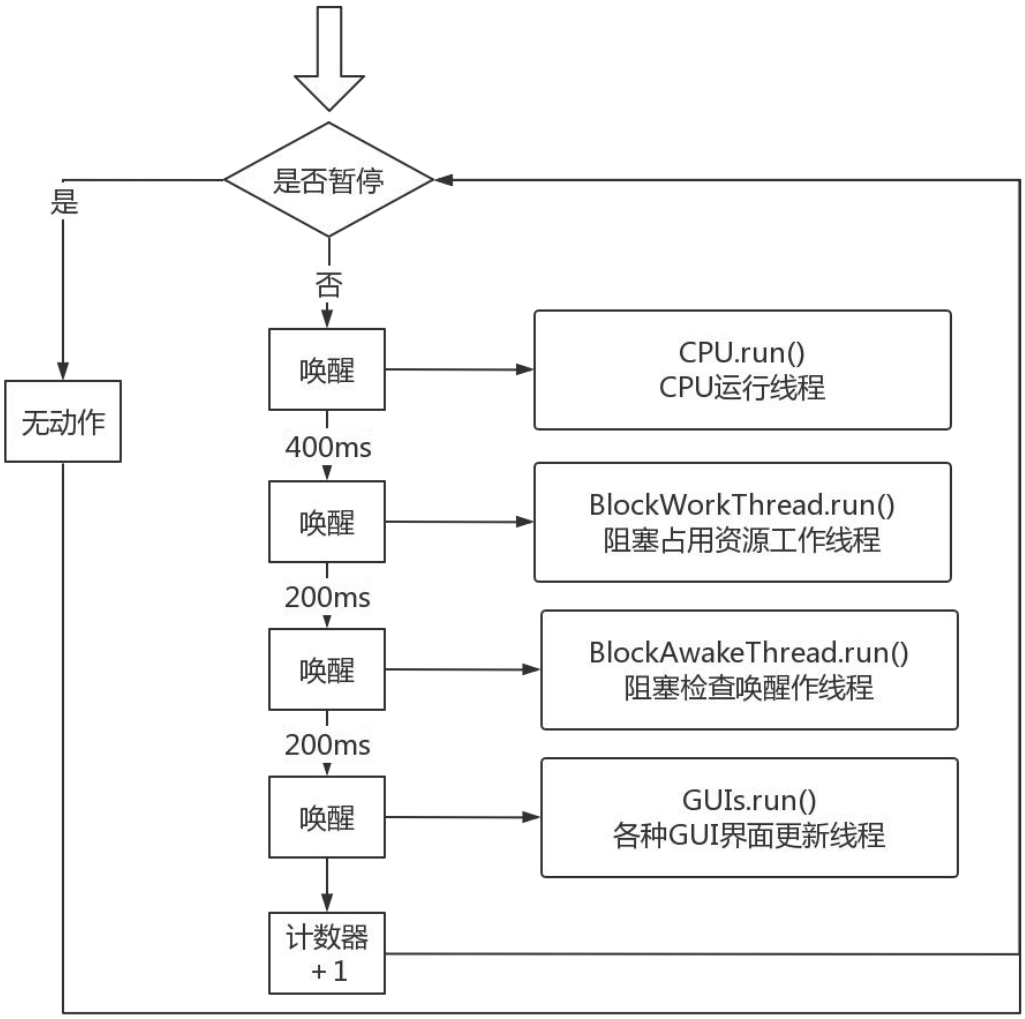
系统有一张设备表，用来进行设备管理，将其抽象为 DeviceTable 类，其属性值如下，其中设备号隐含为表项号。

DeviceTable 类的属性	
属性	**描述**
deviceName	设备名
pAddress	设备驱动程序入口地址
time	设备工作时间

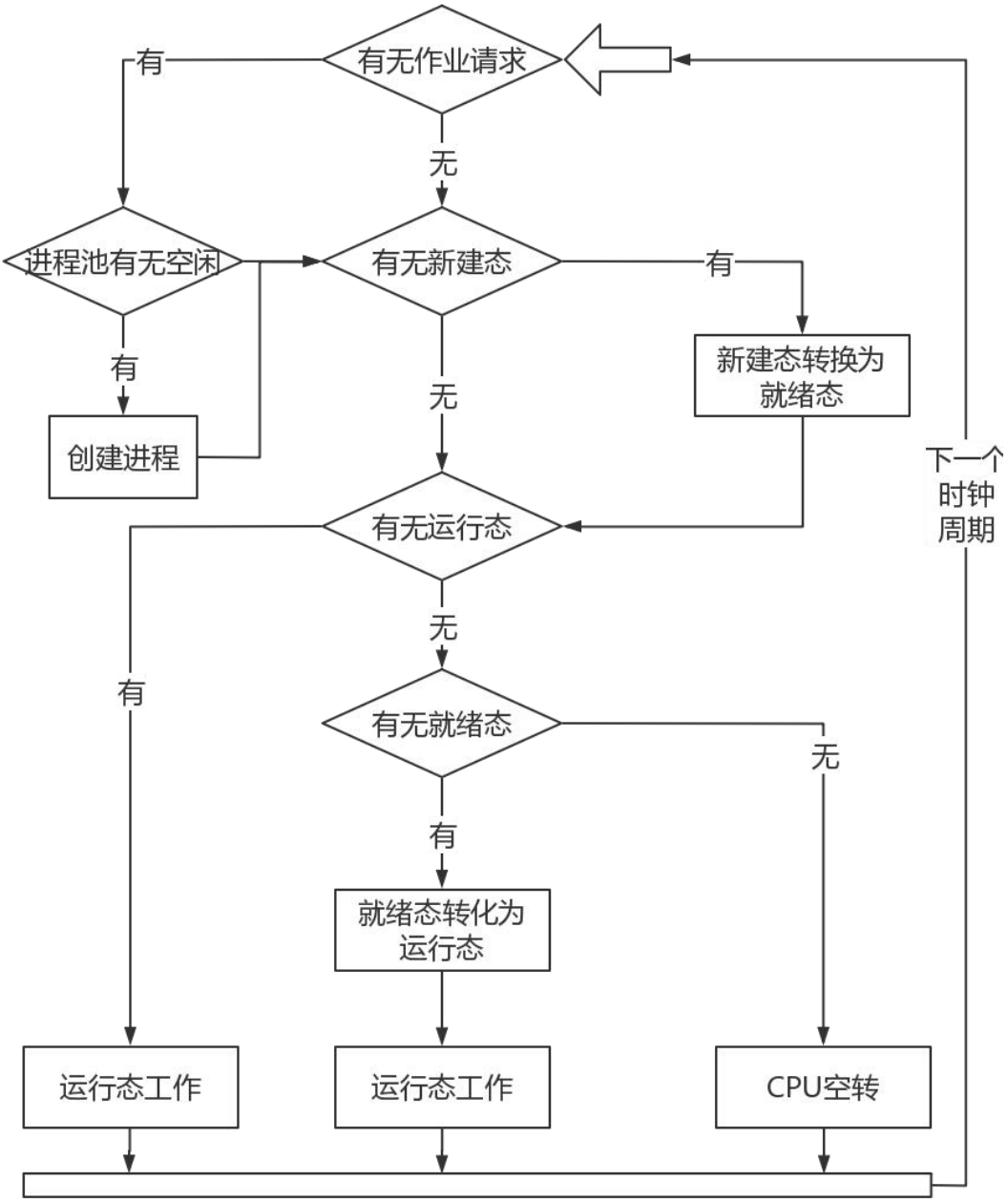
5. 程序结构及模块功能的实现

5.1. 系统工作流程

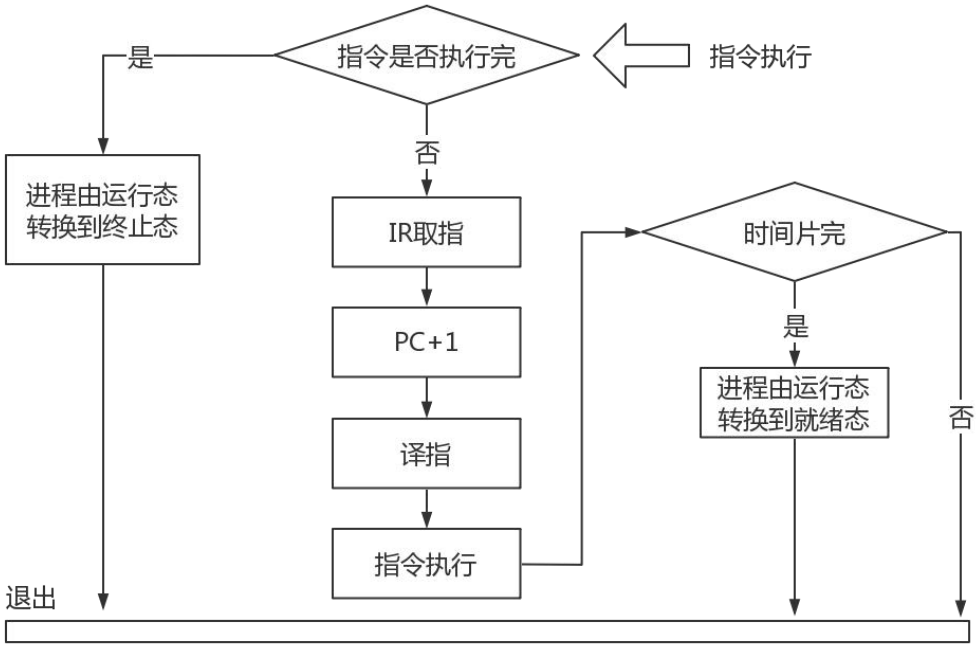
5.1.1. 时钟工作流程


是否暂停
是
否
无动作
唤醒
CPU.run()
CPU运行线程
400ms
唤醒
BlockWorkThread.run()
阻塞占用资源工作线程
200ms
唤醒
BlockAwakeThread.run()
阻塞检查唤醒作线程
200ms
唤醒
GUIs.run()
各种GUI界面更新线程
计数器
+1

5.1.2. CPU 工作流程


有无作业请求
有
无
进程池有无空闲
有无新建态
有
新建态转换为
就绪态
有
创建进程
无
下一个
时钟
周期
有无运行态
无
有
有无就绪态
无
有
就绪态转化为
运行态
运行态工作
运行态工作
CPU空转

5.1.3. 指令执行流程



5.2. 作业调度算法

作业调度采用静态优先级+先来先服务的算法。即系统会优先扫描到达时间，若出现多个作业到达的情况，则选中优先级最高的作业进行调度，若仍然有多个作业到达并且他们优先级相同，则会选取最先到达的作业，若它们到达时间仍然相同，则在它们中间按照检索顺序选取。

其伪代码如下：

```
public int checkReturnIndex()
{
    int index = -1;
    int time =  当前系统时间;
    int tempP = 0;
    for(遍历进程池)
    {
        if(当前时间  >=  到达时间)
            if(优先级  > tempP)
            {
                index = i;
                tempP =  优先级;
            }
    }
    return index;
}
```

5.3. 进程调度算法

进程调度采用非抢占式静态优先级＋时间片轮转算法。即从就绪队列中选取

进程到运行态时总是选择优先级最高的，若多个优先级相同，则按照进程池顺序索引选取，若进程在运行期间没有出现阻塞状态，则会在时间片用完后有运行态转换为就绪态，参与下一轮调度。本系统设置的时间片为 4。

5.4. 页面置换算法

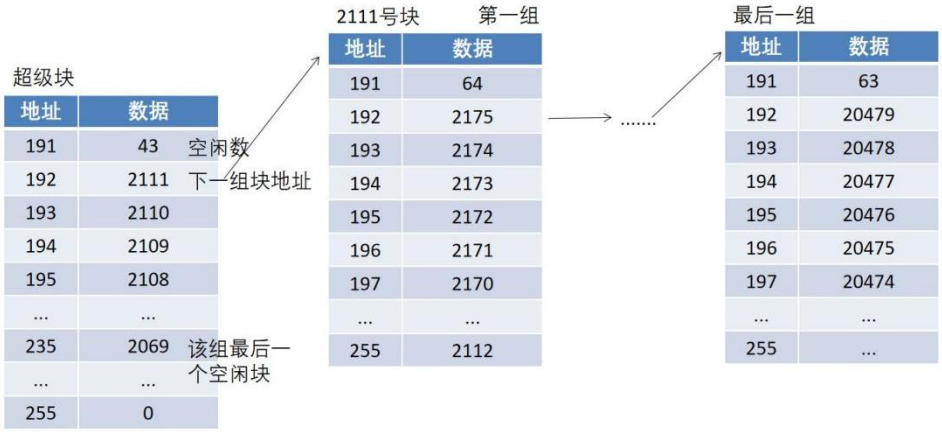
页面置换算法采用 LRU 算法，它根据数据的历史访问记录来进行淘汰数据，其核心思想是：如果一个数据在最近一段时间没有被访问到，那么在将来它被访问的可能性也很小。也就是说，当限定的空间已存满数据时，应当把最久没有被访问到的数据淘汰。

本系统 LRU 算法的实现将在后文“技术问题”中描述。

5.5. 磁盘空闲块管理方法

磁盘中的文件区的空闲块管理为成组链接法。我将空闲块分成若干组，每 64 个空闲块为一组（最后一组为 63 个），即一个磁道为一组，每组的第一空闲块登记了下一组空闲块的物理盘块号和空闲块总数。如果一个组的第二个空闲块号等于 20479，则有特殊的含义，意味着该组是最后一组，即无下一个空闲块。

第一组空闲块数据存储在超级块中。本系统的成组链接描述如下。



本系统中的成组链接法的算法原理以及实现将在后文“技术问题”中描述。

5.6. 外部设备响应算法

前文说到，系统中一共有六种阻塞队列，分别是 fileBlock，inputBlock，outputBlock，printerBlock，cameraBlock 以及 printerWaiting。它们根据功能有如下不同的响应策略。

（1）fileBlock。队列中的阻塞进程全部占用资源工作。

（2）inputBlock，outputBlock，cameraBlock。选取优先级高的进程处理。

（3）printerBlock。队列中的阻塞进程全部占用资源处理。队列位置上的进程在顺序索引对应的打印机上工作。

6. 测试与分析

6.1. 测试用例

6.1.1. 作业集

测试用例的初始作业集共 6 条作业，用于如下测试，具体内容如下：

ArrivalTime	Name	Codes	Priority
5	Job_0	14	9
5	Job_1	10	5
7	Job_2	10	5
8	Job_3	15	5
9	Job_4	10	5
15	Job_5	5	6
20	Job_6	5	1

初始作业集设计原则，0 号作业为高频寻址作业类型，1 号与 0 号作业检验静态优先级的作业调度算法，1,2,3,4 号作业有一段连续的打印机请求服务，用来检查当打印机资源空时，打印机等待阻塞的情况，5,6 号作业为长时间阻塞的作业类型。

6.1.2. 代码

作业 0 代码文件

Type	Operand	Type	Operrand
0	1	1	2
0	2	1	3
0	3	0	5
0	4	0	2
0	5	2	4
1	0	3	6
1	1	4	12

作业 1 代码文件

Type	Operand	Type	Operand
2	3	6	0
3	5	6	0
2	4	1	0
6	0	2	7
6	0	5	8

作业 2 代码文件

Type	Operand	Type	Operand
2	3	6	0
3	5	6	0
2	4	6	0
0	1	1	0
6	0	2	7

作业 3 代码文件

Type	Operand	Type	Operand
2	3	6	0
3	5	1	0
2	4	2	7
0	1	5	8
1	3	1	2
6	0	0	2
6	0	0	3
6	0		

作业 4 代码文件

Type	Operand	Type	Operand
2	3	0	2
3	5	6	0
2	4	6	0
0	1	6	0
1	3	0	2

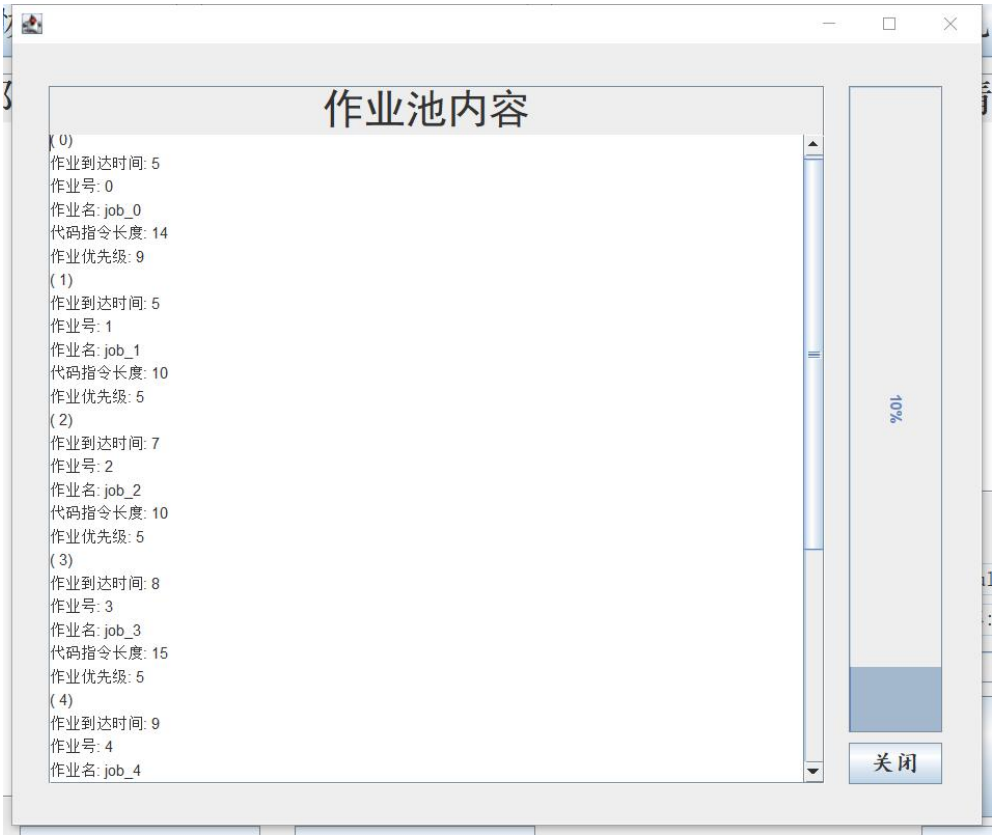
作业 5 代码文件

Type	Operand	Type	Operand
2	3	5	15
3	4	6	0
4	15		

作业 6 代码文件

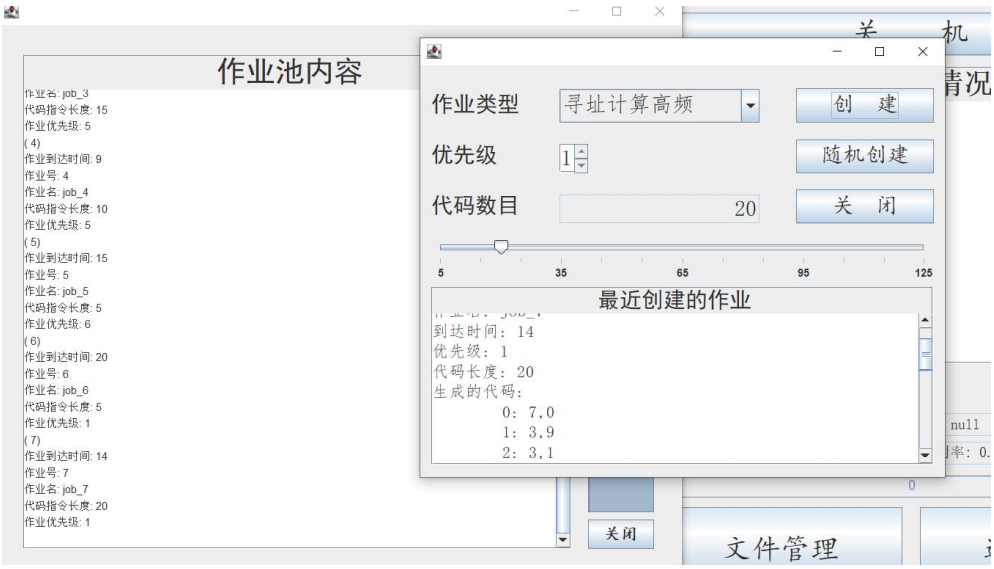
Type	Operand	Type	Operand
6	0	6	0
7	0	0	1
7	0		

加载到系统的信息显示

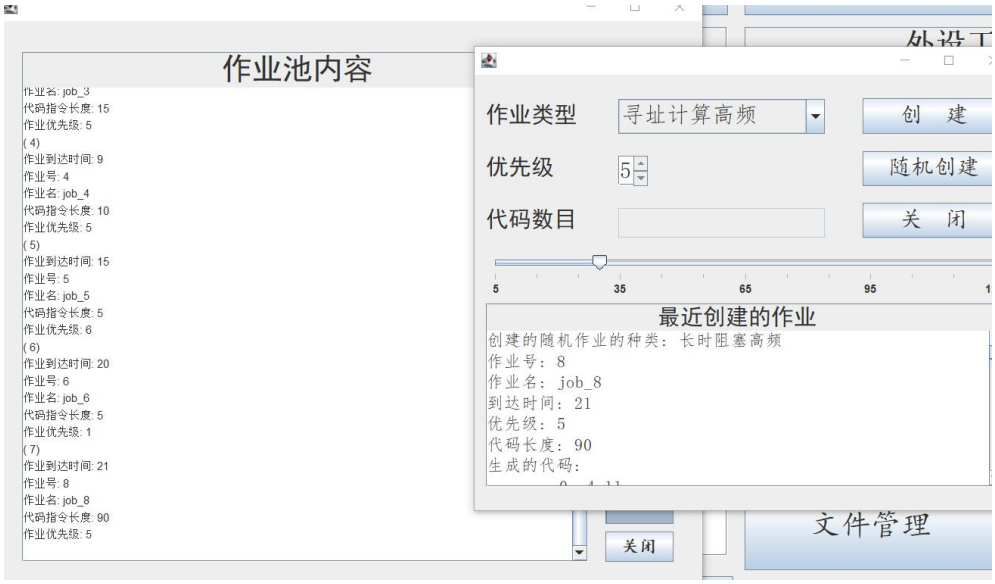


6.2. 并发作业请求（随机作业产生）

半随机模式，设置“寻址计算高频”作业类型，优先级为 1，代码长度为 20。作业池中出现新个作业，创建作业界面中出现创建的作业的全部信息。



全随机模式，直接点击随即创建，作业池中出现新的作业，创建作业界面中出现创建的作业的全部信息。



6.3. MMU 地址变化

以系统执行作业 0 的第一条指令为例。

```
5: 0 号PCB新建态 -> 就绪态
5: 0 号PCB就绪态 -> 运行态
5: 0 号进程执行一条指令
        ●PC: 1 Type: value Operand: short
        ●逻辑页号: 0, 内存块号: 28, 外存块号: 128, 偏移地址: 230
        ●PCBIndex: 0, 访问地址(230 -> 7398)
6: 开始运行 1 号作业
6: 1 号PCB被创建为新建态
        ●进程名(job_1), 代码数(10), 优先级(5)
6: 1 号PCB新建态 -> 就绪态
6: 0 号进程执行一条指令
```

预分配时将 0,1 号页面调入内存，而 value 类型的指令是产生不引发缺页中断的随机地址，这里可以看出访问地址为 230（0 页），而 0 号进程为第一个被调入内存中的作业，其分配的内存块空间为 16（代码块），28，29（数据块）块，所以将其地址变换为：内存块号 * 256 + 偏移地址 = 7398。

6.4. MMU 缺页中断

以系统执行作业一的某一条指令为例。

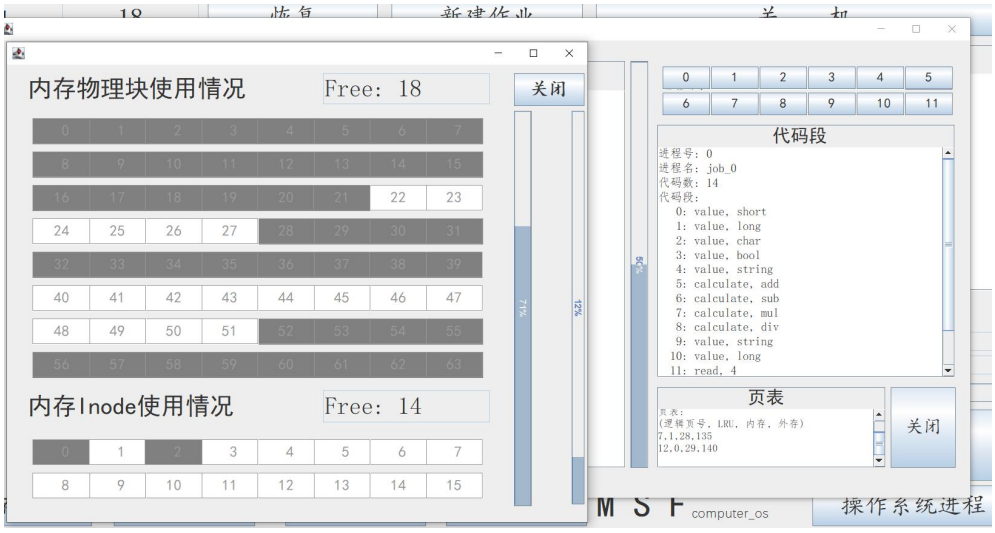
```
9: 0 号进程执行一条指令
        ●PC: 5 Type: value Operand: string
        ●逻辑页号: 1, 内存块号: 29, 外存块号: 129, 偏移地址: 28
        ●PCBIndex: 0, 访问地址(284 -> 7452)
10: 0 号进程执行一条指令
        ●PC: 6 Type: calculate Operand: add
        ●(0, 28, 128) -> (27, 28, 155)
        ●逻辑页号: 27, 内存块号: 28, 外存块号: 155, 偏移地址: 184
        ●PCBIndex: 0, 访问地址(7096 -> 7352)
11: 0 号进程执行一条指令
        ●PC: 7 Type: calculate Operand: sub
```

10 时钟时间时执行 calculate 指令，该指令为产生一个进程逻辑地址空间（0~8191）的随机数以供寻址，大概率引发缺页中断，可以观察到，上一条指令访问了 1 号页面，所以在缺页中断时选择替换 0 号页面（LRU 算法），而要访问

的逻辑地址为 7096，其逻辑页号为 7096/256=27，所以 0 号替换为 27 号页面，之后的内存地址变换与 MMU 地址变化所述一致。

6.5. 内存空间分配

某时刻系统进程数为 6 的分配。



内存物理块 16~21 被分配给 6 个进程的代码块（每个进程一块），28~39 被分配给 6 个进程的数据块（每个进程 2 块），同时该时刻还有一次文件读写指令，加上操作系统常置打开主目录，共分配了两个 Inode。

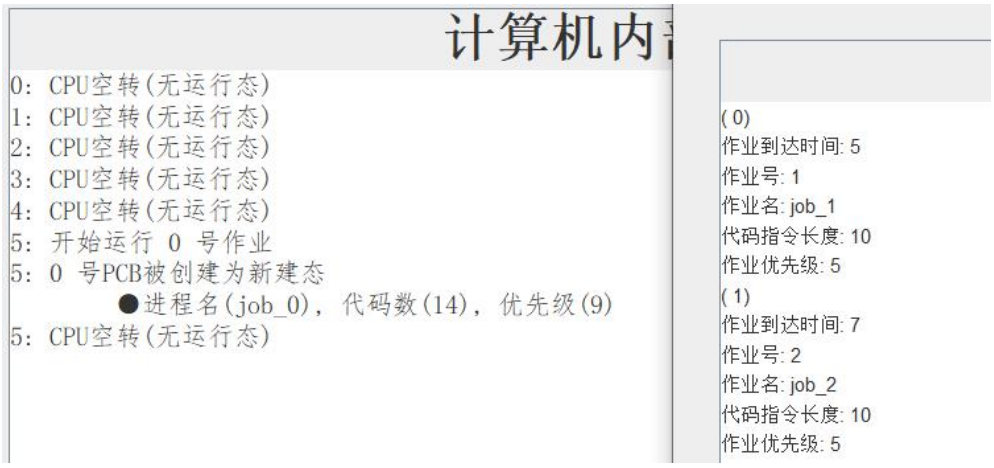
6.6. 页表生成

每个进程被创建时会分配 0,1 逻辑页号。

之后随着寻址访问更新页表，比如寻址访问频繁的 0 号作业。

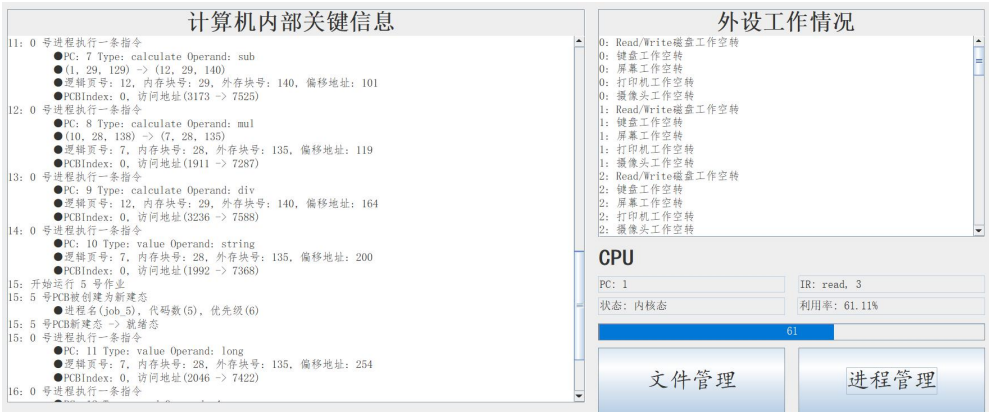
6.7. 作业调度

根据测试用例，可以得到 0 号作业最先调度，其优先级在已到达的时间里最高。



6.8. 进程调度

全部进程调度信息会在计算机内部关键信息中显示。



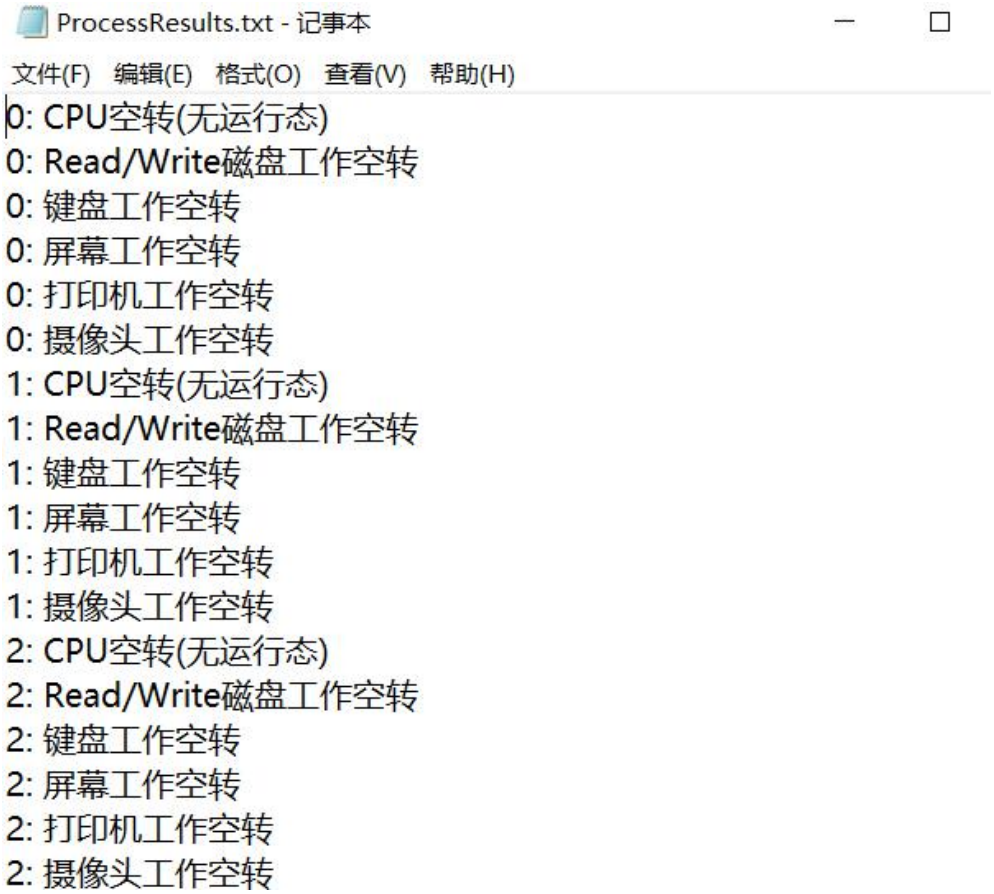
6.9. 进程基础信息查看

可以查看五态的所有进程信息，阻塞态会额外包括保护现场信息。在界面右边可以点击进程池的索引号，来查看该索引号对应的 PCB 中存储的进程基础信

息，包括其代码和页表。中间进度条会显示进程池的使用情况。

6.10. 程序运行调度输出 ProcessResults.txt 文件内容

全部运行信息保存在此文件中。



6.11. 界面程序运行信息显示

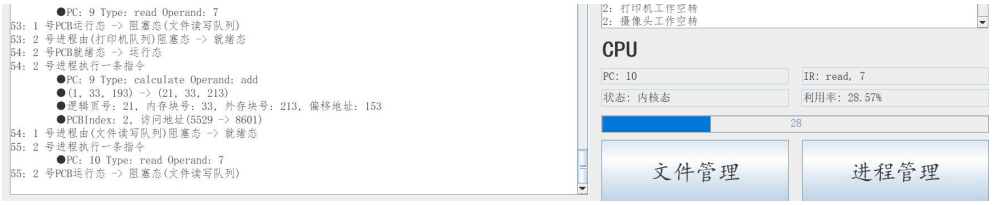
进程调度信息和设备使用情况，此外还会显示 CPU 利用率，CPU 当前内容。

6.12. 文件 Inode 信息的查看

文件管理中可以点击磁盘中的 inode 号来查看其具体内容。

6.13. 系统打开文件表

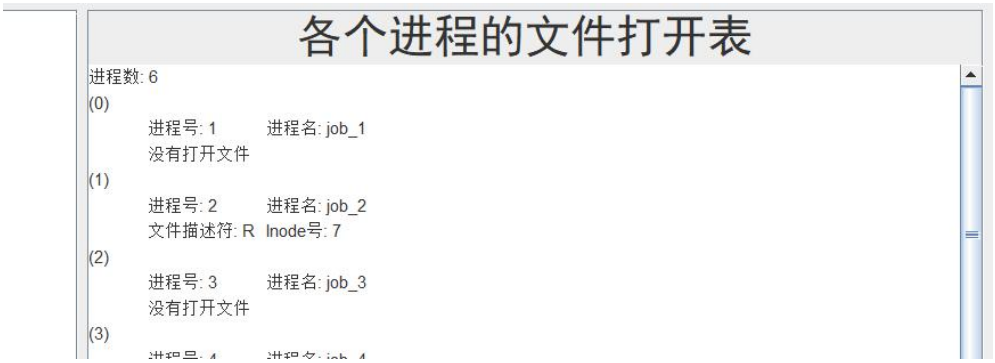
55 时钟时，执行 2 号进程的 read 指令。



“文件管理”中查看系统文件打开表以及相应的 Inode 调入内存中的分配情况。

6.14. 进程打开文件表

还是在 55 时钟刻，2 号进程打开文件表

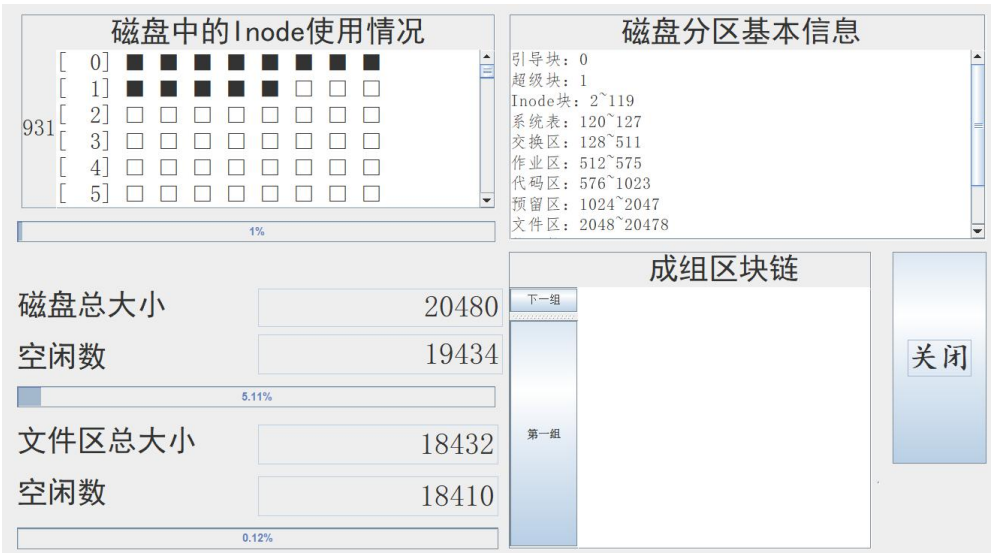


6.15. 文件系统的目录结构

“文件管理”中可以查看文件系统的目录结构。

6.16. 磁盘存储信息

显示磁盘 Inode 的位示图，以及使用和分区情况。



6.17. 成组链接法

第一组空闲区块链。

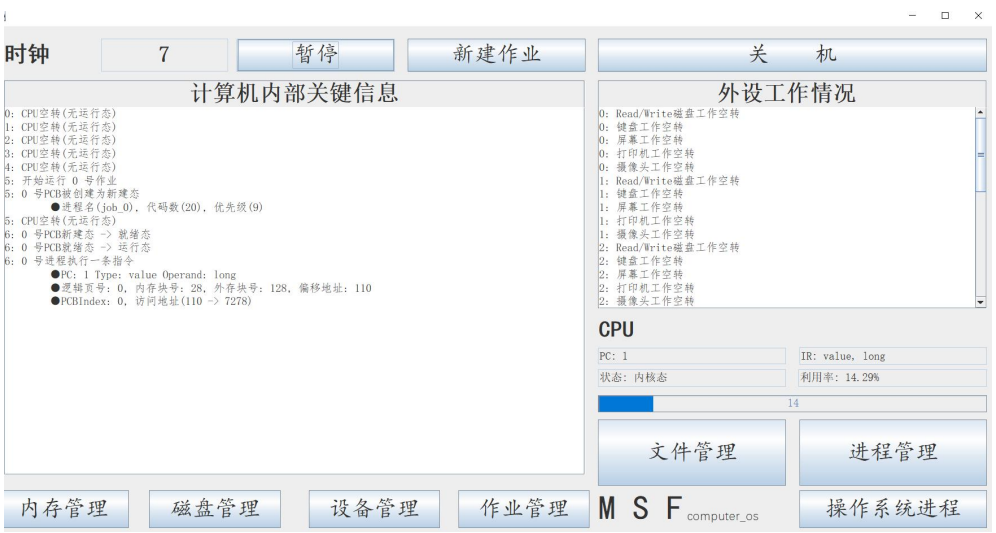
搜索到下一组。

6.18. 设备管理

在“设备管理”中，会显示所有外设的有效工作时间比，还有各种资源的剩余数目。同时还有外部设备表。

6.19. 时钟暂停

暂停和恢复按钮

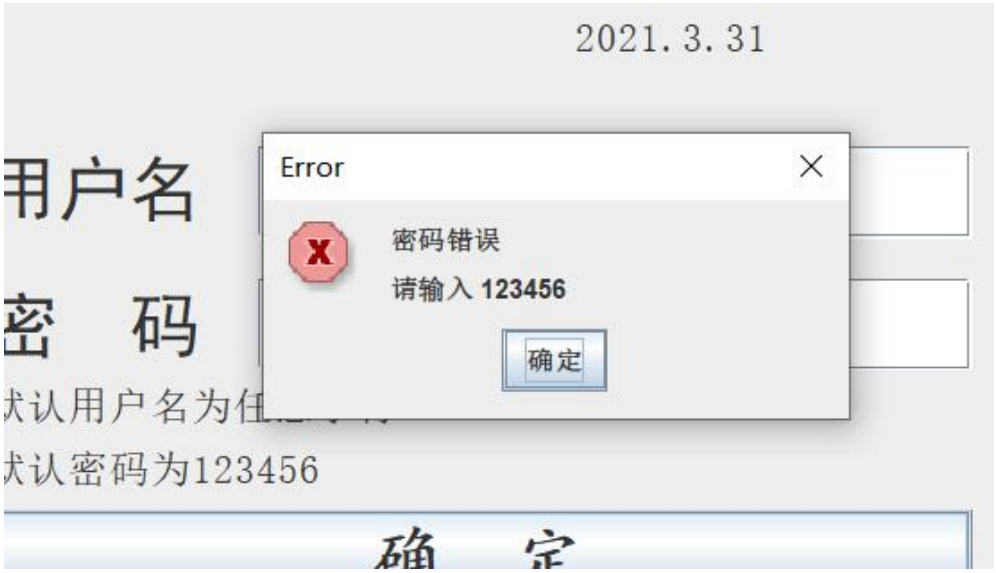


6.20. GUI 开机功能

开机未完成

开机完成（按钮变化）

密码错误



密码正确点击确定后即可切换到主界面。

7. 技术问题及解决方案

7.1. 设计中的问题

7.1.1. 超级块的设计及存储

超级块的设计和本地存储是一大难题，如何合理分配其中的信息，将位示图存储在本地 txt 中也是一个问题，为此我设计了如下 SuperBlock 类。

SuperBlock 类的主要属性	
属性名	含义
freeBloInMemory	内存中的空闲块数
freeBlockInMemoryMap	内存空闲块位示图
freeInodeInMemory	内存空闲 Inode 数
freeInodeInMemoryMap	内存 Inode 位示图
freeInodeInDisk	磁盘空闲 Inode 数
freeInodeInDiskMap	磁盘 Inode 位示图

freeBlocksGroupsLink	第一组空闲区块

这是超级块在本体 txt 文件中的存储格式。

SuperBlock 本地存储	
地址	信息
0	内存空闲物理块数
1，2,3,4	内存空闲块位示图
5	内存空闲 Inode 数
6	内存空闲 Inode 位示图
7	磁盘空闲 Inode 数
8-70	磁盘空闲 Inode 位示图
...	
191	第一组空闲区块数
192-255	第一组空闲区块号

7.1.2. Inode 的设计及存储

这是 Inode 在本体 txt 文件中的存储格式。共占 32B，所以一个块中可以存储 8 个 Inode。

Inode 本地存储	
地址	信息
0	Inode 号
1	文件类型
2	读权限
3	写权限
4	使用权限
5-9	创建日期
10-14	修改日期
15,16	无效位
17	文件大小
18-25	8 个直接地址
26	一级间接地址
27	二级间接地址

7.2. 代码实现中的问题

7.2.1. 多线程通信

由于各个 GUI，各个类之间复杂的关系，既要求 GUI 实时更新，又要要求各个线程之间的资源不能混乱，所以这对各个线程之间的通信有很高的要求。由此我使用锁。除了时钟线程之外的线程在死循环运行中，首先获得对应的锁，之后第一步操作是使该线程在这个锁上等待（使用了 java 的 synchronized 关键字和 wait 方法），而当时钟线程运行时，会根据“时钟工作流程”所示的顺序依次在其他线程对应锁上唤醒等待的线程（使用了 notifyAll 方法），由此实现了各个

线程的通信。

7.2.2. 类之间的通信

各个类中的方法需要频繁的使用 Memory 和 Disk 类的信息，而 CPU 等线程的 run 方法不能传参，为了解决这个问题，我创建了 Computer 类，在里面的 main 方法开启所有线程，并且其成员属性皆为共有静态属性。由此各个类的访问便方便了许多。

7.2.3. LRU 算法

由于页框数为 2，则若使用 LRU 算法，数据结构会方便很多，标志位 0 表示上次之前有访问或从未访问，1 表示上次就被访问，所以每次替换时，只需要找到记录为 0 的页表项即可。

8. 实践心得

通过本次实验，我深刻理解了 Linux 操作系统内核原理，理解了各种有效算法，包括 LRU，时间片轮转等等，同时也提高了我面向对象编程的能力。在设计中，我深刻意识到必须在一开始就做好各种规划和设计，并在实现中逐步细化。同时要尽量与他人讨论，从而进一步改进自己的设计，在全部方案基本没有问题时，采用面向对象的设计方法，逐步细化各个模块及其功能，最终完成全部内容。

在设计中，由于我在文件系统方面设计不是很完善，导致在后期实现以及其他子系统匹配中不得不修改或者删减功能，才能完成最终系统的构造，由此可见，我的设计能力还需进一步提高。

参考文献

[1] 刘建臣.多层面“把脉”Linux 运行状态[J].网络安全和信息化,2021(03):98-101.

[2] 姜绍萍.基于 RTLinux/Linux 的计算机联锁系统可靠性分析[J].自动化与仪器仪表,2021(02):65-68.

[3] 李志炜,田秀云.Linux 服务器自动化部署和管理[J].机电工程技术,2021,50(02):171-173.

[4] 刘京义.实例详析 Linux 文件同步机制[J].网络安全和信息化,2021(02):110-113.

[5] 马玉州.信息网络安全监察专业 Linux 操作系统课程实验设计[J].现代计算机,2021(03):72-75.

[6] 曹玉红,陈思羽.基于深度学习的操作系统多维度安全保护架构[J].工业技术创新,2021,08(01):90-95.

[7] 任卓亚.浅谈计算机网络操作系统的种类与安全[J].数码世界,2021(02):50-51.

附件 1：程序文件及结构说明

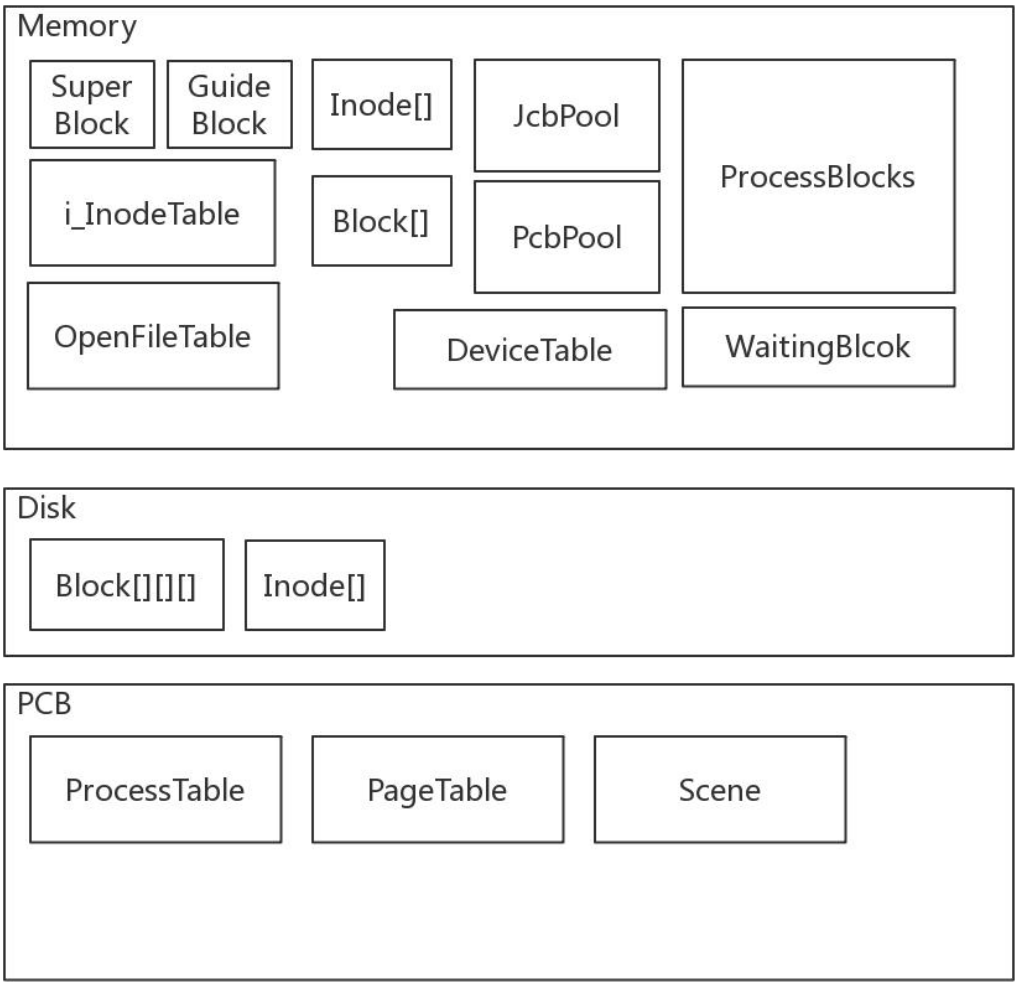
1. 主目录

2. input 目录

作业请求文件：19218101-jobs-input.txt
作业代码文件：作业号.txt

3. output 目录
   程序运行结果：ProcessResults.txt
4. disk 目录
   磁盘本地映像文件

附件 2：类图和顺序图说明



附件 3：带注释的部分核心代码

1. CPU 的 run 方法

```
// 线程 run
    public void run()
    {
        try {
            while(true)
            {
                toKernel();
```

```
synchronized (MyLock.ClockIsChanged)
{
    MyLock.ClockIsChanged.wait();

    int jcbIndex = Computer.memory.checkJCBPool();
    // JCB 池中有需要创建为进程的 JCB
    if(jcbIndex >= 0)
    {
        toKernel();
        int back = Computer.memory.JcbToNew(jcbIndex);
        if(back==0)
        {
            //  如果进程池满，则检查有无新建态
            int pcbNewIndex = Computer.memory.checkPCBPoolIsNew();
            // PCB 池中有新建态
            if(pcbNewIndex >= 0)
            {
                toKernel();
                Computer.memory.NewToReady(pcbNewIndex);
            }
        }
    }
    int pcbNewIndex = Computer.memory.checkPCBPoolIsNew();
    // PCB 池中有新建态
    if(pcbNewIndex >= 0)
    {
        toKernel();
        Computer.memory.NewToReady(pcbNewIndex);
    }
    // JCB 池中没有需要创建为进程的 JCB
//  else
//  {
//      int pcbNewIndex = Computer.memory.checkPCBPoolIsNew();
//      // PCB 池中有新建态
//      if(pcbNewIndex >= 0)
//      {
//          toKernel();
//          Computer.memory.NewToReady(pcbNewIndex);
//      }
//  }
    int pcbRunIndex = Computer.memory.checkPCBPoolIsRunning();
    // PCB  池中没有运行态
    if(pcbRunIndex == -1)
    {
```

```
                                int pcbReadyIndex = Computer.memory.checkPCBPoolIsReady();
                                // PCB 中有就绪态
                                if(pcbReadyIndex >= 0)
                                {
                                    toKernel();
                                    Computer.memory.ReadyToRun(pcbReadyIndex, this);
                                    // 对刚转化为运行态的进程运行
                                    PCBRunning(Computer.memory, pcbReadyIndex);
                                }
                                // 没有就绪态，空转
                                else
                                {
                                    String temp = String.format("%d: CPU 空转(无运行态)\n",
Clock.getClock());
                                    System.out.print(temp);
                                    Computer.RunInfo += temp;
                                    //System.out.println(Clock.getClock() + ": " + "CPU 空转(无运行态)");
                                }
                            }
                            // 有运行态则执行
                            else
                            {
                                PCBRunning(Computer.memory, pcbRunIndex);
                            }
                        }
                    }
                } catch (Exception e) {
                    // TODO: handle exception
                    e.printStackTrace();
                }
            }
            // 指令执行函数
            public void PCBRunning(Memory memory, int pcbRunIndex)
            {
                String temp;
                // 检查是否执行完，若执行完则进程消亡
                if(pc.getPCIndex() >= memory.pcbPool.pool[pcbRunIndex].getCodeLength())
                {
                    // 进程执行完毕
                    memory.ToDead(pcbRunIndex);
                    return;
                }
                Instruction instruction = getInstruction_PC(memory);
                ir.setIR(instruction);
```

```
            pc.inc();
            switch(instruction.getType())
            {
                case Instruction.VALUE:
                {
                    toUser();

                    temp = String.format("%d: %d  号进程执行一条指令\n", Clock.getClock(),
memory.pcbPool.pool[pcbRunIndex].getProcessNumber());
                    System.out.print(temp);
                    Computer.RunInfo += temp;

                    temp = instruction.get(pc.getPCIndex());
                    System.out.print(temp);
                    Computer.RunInfo += temp;

                    memory.pcbPool.pool[pcbRunIndex].incRunTime();
                    // 产生随机地址
                    int pcbRunNumber = Computer.memory.pcbPool.pool[pcbRunIndex].getProcessNumber();
                    int logic =
RandomGenerator.getRandomFromPageTable(Computer.memory.pcbPool.pool[pcbRunIndex].getPageTable());
                    // 访问地址
                    if(MMU.isMissPage(Computer.memory.pcbPool, pcbRunNumber, logic))
                        // 缺页中断
                        MMU.MissPage(Computer.memory.pcbPool, pcbRunNumber, logic);
                    // 地址变换并访问
                    Address address = MMU.getAddressFromPageTable(Computer.memory.pcbPool,
pcbRunNumber, logic);
                    // CPUTime
                    Time++;

                    temp = String.format("\t●PCBIndex: %d,  访问地址(%d -> %d)\n", pcbRunIndex, logic,
address.get());
                    System.out.print(temp);
                    Computer.RunInfo += temp;

                    break;
                }
                case Instruction.CALCULATE:
                {
                    toUser();

                    temp = String.format("%d: %d  号进程执行一条指令\n", Clock.getClock(),
memory.pcbPool.pool[pcbRunIndex].getProcessNumber());
```

```
                        System.out.print(temp);
                        Computer.RunInfo += temp;

                        temp = instruction.get(pc.getPCIndex());
                        System.out.print(temp);
                        Computer.RunInfo += temp;

//                      System.out.println(Clock.getClock() + " : 进程" +
memory.pcbPool.pool[pcbRunIndex].getProcessNumber() + "执行一条指令");
//                      instruction.show(pc.getPCIndex());
                        memory.pcbPool.pool[pcbRunIndex].incRunTime();
                        // 产生随机地址
                        int pcbRunNumber = Computer.memory.pcbPool.pool[pcbRunIndex].getProcessNumber();
                        int logic = RandomGenerator.getRandomFromLogicalSpace();
                        // 访问地址
                        if(MMU.isMissPage(Computer.memory.pcbPool, pcbRunNumber, logic))
                            // 缺页中断
                            MMU.MissPage(Computer.memory.pcbPool, pcbRunNumber, logic);
                        // 地址变换并访问
                        Address address = MMU.getAddressFromPageTable(Computer.memory.pcbPool,
pcbRunNumber, logic);
                        // CPUTime
                        Time++;

                        temp = String.format("\t●PCBIndex: %d, 访问地址(%d -> %d)\n", pcbRunIndex, logic,
address.get());
                        System.out.print(temp);
                        Computer.RunInfo += temp;


                        //System.out.printf("\t●访问地址(%d -> %d)\n", logic, address.get());
                        break;
                    }
                    case Instruction.READ, Instruction.WRITE, Instruction.INPUT, Instruction.OUTPUT,
Instruction.PRINTER, Instruction.CAMERA:
                    {
                        toKernel();

                        temp = String.format("%d: %d 号进程执行一条指令\n", Clock.getClock(),
memory.pcbPool.pool[pcbRunIndex].getProcessNumber());
                        System.out.print(temp);
                        Computer.RunInfo += temp;

                        temp = instruction.get(pc.getPCIndex());
```

```
                    System.out.print(temp);
                    Computer.RunInfo += temp;

//                  System.out.println(Clock.getClock() + " : 进程 " +
memory.pcbPool.pool[pcbRunIndex].getProcessNumber() + " 执行一条指令");
//                  instruction.show(pc.getPCIndex());
                    memory.pcbPool.pool[pcbRunIndex].incRunTime();
                    memory.RunToBlock(pcbRunIndex, this, instruction);
                    break;
                }
                default:
                {
                    System.err.println("出现了未知的指令类型");
                    System.exit(-1);
                }
            }
            // 检查运行态时间片是否完以进行替换
            int timeFullIndex = memory.checkTimeIsFull();
            // 时间片完
            if(timeFullIndex != -1)
            {
                toKernel();
                memory.RunToReady(timeFullIndex, this);
                // 选取新的就绪态装入
                int newRunningIndex = memory.checkPCBPoolIsReady();
                memory.ReadyToRun(newRunningIndex, this);
            }
        }
```

2. 进程调度原语

```
/**
     * 进程调度原语
     * */
    // JCB 创建为新建态
    public int JcbToNew(int jcbIndex)
    {
        String clock = Clock.getClock() + ": ";
        String tempS = "";

//      System.out.println(clock + "开始运行 " +
Computer.memory.jcbPool.pool.get(jcbIndex).getJobNumber() + " 号作业");
        int freeIndexInPbcPool = pcbPool.getFirstFreeIndexInPoolMap();
        if(freeIndexInPbcPool == -1)
        {
            tempS = String.format("%sPCBPool 已满，无法创建新的 PCB\n", clock);
```

```
                System.out.print(tempS);
                Computer.RunInfo += tempS;
                //System.out.println(Clock.getClock() + ": " + "PCBPool 已满，无法创建新的 JCB");
                return 0;
            }

            tempS = String.format("%s 开始运行 %d 号作业\n", clock,
Computer.memory.jcbPool.pool.get(jcbIndex).getJobNumber());
            System.out.print(tempS);
            Computer.RunInfo += tempS;

            // 新建 PCB
            // 读代码，分配内存及其它初始化
            PCB temp = new PCB();
            temp.createFromJCB(jcbPool.getJCBAtIndex(jcbIndex), freeIndexInPbcPool);
            pcbPool.add(temp);

            tempS = String.format("%s%d 号 PCB 被创建为新建态\n", clock, temp.getProcessNumber());
            tempS += String.format("\t●进程名(%s), 代码数(%s), 优先级(%d)\n", temp.getProcessName(),
temp.getCodeLength(), temp.getPriority());
            System.out.print(tempS);
            Computer.RunInfo += tempS;

//          System.out.println(clock + temp.getProcessNumber() + " 号 PCB 被创建为新建态");
//          System.out.printf("\t●进程名(%s), 代码数(%s), 优先级(%d)\n", temp.getProcessName(),
temp.getCodeLength(), temp.getPriority());
            // 修改内存位示图
            allocateIndexBlock(CODE_BLOCK_START_FOR_A_PROCESS[freeIndexInPbcPool]);
            allocateIndexBlock(DATA_BLOCK_START_FOR_A_PROCESS[freeIndexInPbcPool]);
            allocateIndexBlock(DATA_BLOCK_START_FOR_A_PROCESS[freeIndexInPbcPool]+1);
            // 删除 JCB
            Computer.memory.jcbPool.remove(jcbIndex);
            return 1;
        }
        // PCB 新建态变换为就绪态
        public void NewToReady(int pcbIndex)
        {
            String clock = Clock.getClock() + ": ";
            int pcbNumber = Computer.memory.pcbPool.pool[pcbIndex].getProcessNumber();

            String temp = String.format("%s%d 号 PCB 新建态 -> 就绪态\n", clock, pcbNumber);
            System.out.print(temp);
            Computer.RunInfo += temp;
//          System.out.println(clock + pcbNumber + " 号 PCB 新建态 -> 就绪态");
```

```
            pcbPool.setPcbOnNewToReady(pcbIndex);
        }
        // PCB 就绪态到运行态
        public void ReadyToRun(int pcbIndex, CPU cpu)
        {
            String clock = Clock.getClock() + ": ";
            int pcbNumber = Computer.memory.pcbPool.pool[pcbIndex].getProcessNumber();

            String temp = String.format("%s%d 号 PCB 就绪态 -> 运行态\n", clock, pcbNumber);
            System.out.print(temp);
            Computer.RunInfo += temp;
//          System.out.println(clock + pcbNumber + " 号 PCB 就绪态 -> 运行态");

            pcbPool.pool[pcbIndex].setState(PCB.RUNNING);
            pcbPool.pool[pcbIndex].setTimeFlak(0);
            // 恢复现场
            cpu.Recover(pcbPool.pool[pcbIndex]);
        }
        // PCB 运行态到就绪态
        public void RunToReady(int pcbIndex, CPU cpu)
        {
            String clock = Clock.getClock() + ": ";
            int pcbNumber = Computer.memory.pcbPool.pool[pcbIndex].getProcessNumber();

            String temp = String.format("%s%d 号 PCB 运行态 -> 就绪态\n", clock, pcbNumber);
            System.out.print(temp);
            Computer.RunInfo += temp;
//          System.out.println(clock + pcbNumber + " 号 PCB 运行态 -> 就绪态");

            pcbPool.pool[pcbIndex].setState(PCB.READY);
            pcbPool.pool[pcbIndex].setTimeFlak(0);
            // 保护现场
            pcbPool.pool[pcbIndex].scene.set(cpu.getPCIndex(), cpu.state, cpu.getIR());
        }
        // PCB 运行态到阻塞态
        public void RunToBlock(int pcbIndex, CPU cpu, Instruction instruction)
        {
            String clock = Clock.getClock() + ": ";
            String tempS = "";
            int pcbNumber = Computer.memory.pcbPool.pool[pcbIndex].getProcessNumber();
            pcbPool.pool[pcbIndex].setState(PCB.BLOCK);
            pcbPool.pool[pcbIndex].setTimeFlak(0);
            // 保护现场
```

```
            pcbPool.pool[pcbIndex].scene.set(cpu.getPCIndex(), cpu.state, cpu.getIR());
            // 进入阻塞队列
            switch(instruction.getType())
            {
                case Instruction.READ, Instruction.WRITE:
                {
                    tempS = String.format("%s%d 号 PCB 运行态 -> 阻塞态(文件读写队列)\n", clock,
pcbNumber);
                    System.out.print(tempS);
                    Computer.RunInfo += tempS;
//                  System.out.println(clock + pcbNumber + " 号 PCB 运行态 -> 阻塞态(文件读写队列)");
                    PcbBlock temp = new PcbBlock(pcbIndex, pcbPool.pool[pcbIndex].getPriority(),
Instruction.BLOCK_TIME[instruction.getType()], Clock.getClock());
                    fileBlock.add(temp);
                    fileNum--;
                    // 系统打开文件表
                    int fileInodeNumber = instruction.getOperand();
                    if(openFileTable.isIn(fileInodeNumber)!=-1)
                    {
                        // 在系统表中则进程数加一
                        openFileTable.incPNumberForInode(fileInodeNumber);
                    }
                    else
                    {
                        // 不在系统表中则加入
                        int[] freeLocation = superBlock.getFirstFreeInodeLocationInMemory();
                        int freeIndex = SuperBlock.Inode_LocationToIndex(freeLocation[0],
freeLocation[1]);
                        superBlock.allocateAnInodeInMemory();
                        openFileTable.addAOpenFile(fileInodeNumber, 1, freeIndex);
                    }
                    // 进程打开文件表
                    if(instruction.getType() == Instruction.READ)
                        pcbPool.pool[pcbIndex].processFileTable.add("R", fileInodeNumber);
                    else
                    {
                        pcbPool.pool[pcbIndex].processFileTable.add("W", fileInodeNumber);
                        openFileTable.modify(fileInodeNumber);
                    }
                    break;
                }
                case Instruction.OUTPUT:
                {
                    tempS = String.format("%s%d 号 PCB 运行态 -> 阻塞态(屏幕输出队列)\n", clock,
```

```
pcbNumber);
                    System.out.print(tempS);
                    Computer.RunInfo += tempS;
//                  System.out.println(clock + pcbNumber + " 号 PCB 运行态 -> 阻塞态(屏幕输出队列)");
                    PcbBlock temp = new PcbBlock(pcbIndex, pcbPool.pool[pcbIndex].getPriority(),
instruction.getOperand(), Clock.getClock());
                    outputBlock.add(temp);
                    screenNum--;
                    break;
                }
                case Instruction.INPUT:
                {
                    tempS = String.format("%s%d 号 PCB 运行态 -> 阻塞态(键盘输入队列)\n", clock,
pcbNumber);
                    System.out.print(tempS);
                    Computer.RunInfo += tempS;
//                  System.out.println(clock + pcbNumber + " 号 PCB 运行态 -> 阻塞态(键盘输入队列)");
                    PcbBlock temp = new PcbBlock(pcbIndex, pcbPool.pool[pcbIndex].getPriority(),
instruction.getOperand(), Clock.getClock());
                    inputBlock.add(temp);
                    keyboardNum--;
                    break;
                }
                case Instruction.PRINTER:
                {
                    PcbBlock temp = new PcbBlock(pcbIndex, pcbPool.pool[pcbIndex].getPriority(),
Instruction.BLOCK_TIME[instruction.getType()], Clock.getClock());
                    if(printerNum <= 0)
                    {
                        // 打印机资源不足
                        tempS = String.format("%s%d 号 PCB 运行态 -> 阻塞态(打印机等待队列)\n",
clock, pcbNumber);
                        System.out.print(tempS);
                        Computer.RunInfo += tempS;
//                      System.out.println(clock + pcbNumber + " 号 PCB 运行态 -> 阻塞态(打印机等待
队列)");
                        printerWaiting.add(temp);
                    }
                    else
                    {
                        // 打印机资源充足
                        tempS = String.format("%s%d 号 PCB 运行态 -> 阻塞态(打印机队列)\n", clock,
pcbNumber);
                        System.out.print(tempS);
```

```
                    Computer.RunInfo += tempS;
//                  System.out.println(clock + pcbNumber + " 号 PCB 运行态 -> 阻塞态(打印机队列)");
                    printerBlock.add(temp);
                    printerNum--;
                }
                break;
            }
            case Instruction.CAMERA:
            {
                tempS = String.format("%s%d 号 PCB 运行态 -> 阻塞态(摄像机队列)\n", clock, pcbNumber);
                System.out.print(tempS);
                Computer.RunInfo += tempS;
//              System.out.println(clock + pcbNumber + " 号 PCB 运行态 -> 阻塞态(摄像机队列)");
                PcbBlock temp = new PcbBlock(pcbIndex, pcbPool.pool[pcbIndex].getPriority(), Instruction.BLOCK_TIME[instruction.getType()], Clock.getClock());
                cameraBlock.add(temp);
                cameraNum--;
                break;
            }
        }
    }
    // PCB 阻塞态到就绪态
    public void BlockToReady(int pcbIndex)
    {
        pcbPool.pool[pcbIndex].setState(PCB.READY);
        pcbPool.pool[pcbIndex].setTimeFlak(0);
    }
    // PCB 消亡
    public void ToDead(int pcbIndex)
    {
        String clock = Clock.getClock() + ": ";
        String temp = "";
        pcbPool.pool[pcbIndex].setFinishTime(Clock.getClock());
        int pcbNumber = Computer.memory.pcbPool.pool[pcbIndex].getProcessNumber();

        temp = String.format("%s%d 号进程运行态 -> 终止态\n", clock, pcbNumber);
        System.out.print(temp);
        Computer.RunInfo += temp;
//      System.out.println(clock + pcbNumber + " 号进程运行态 -> 终止态");

        int start = pcbPool.pool[pcbIndex].getStartTime();
        int finish = pcbPool.pool[pcbIndex].getFinishTime();
```

```
            int running = pcbPool.pool[pcbIndex].getRunTime();
            double miu = (double)running / (double)(finish-start);

            temp = String.format("\t●持续(%d-%d), 运行(%d), 在 CPU 运行比率(%.4f)\n", start, finish, running,
miu);
            System.out.print(temp);
            Computer.RunInfo += temp;
//          System.out.printf("\t●持续(%d-%d), 运行(%d), 有效比(%4f)\n", start, finish, running, miu);

            // PCB 消亡
            pcbPool.number--;
            pcbPool.pool[pcbIndex].setFinishTime(Clock.getClock());
            pcbPool.pool[pcbIndex].setState(PCB.DEAD);
            pcbPool.poolMap[pcbIndex] = false;
            Computer.FINISH_NUM++;
            // 更新 GUI
            Computer.processGUI.deadText.append("("+Computer.FINISH_NUM+")\n");
            Computer.processGUI.deadText.append(pcbPool.pool[pcbIndex].getDeadString());
            Computer.processGUI.deadText.append(String.format("在 CPU 运行比率: %.4f\n", miu));
            // 内存释放
            reclaimIndexBlock(CODE_BLOCK_START_FOR_A_PROCESS[pcbIndex]);
            reclaimIndexBlock(DATA_BLOCK_START_FOR_A_PROCESS[pcbIndex]);
            reclaimIndexBlock(DATA_BLOCK_START_FOR_A_PROCESS[pcbIndex]+1);
        }
```