需求分析

新增及修改Shell指令 finished

exec

目前情况：exec <filename>

目前功能：创建一个进程执行可执行文件 <filename>，filename 参数即为新建的PCB代码段

目前问题：新建的PCB的优先级默认为0（即最高优先级）

修改目标：exec <filename> <priority>

修改需求：新建PCB时使其优先级初始化为 <priority>

```
case "exec":
    if (parts.length == 3) {
        String filename = parts[1];
        int priority;
        try {
            priority = Integer.parseInt(parts[2]);
            // 检查文件是否存在
            String fileContent = fileDiskManagement.readFileData(filename);
            if (!fileContent.equals("-1")) {
                // 文件存在，创建进程
                process_management.Scheduler.getInstance().createProcess(filename, priority);
            } else {
                System.out.println("文件 " + filename + " 不存在或无法读取");
            }
        } catch (NumberFormatException e) {
            System.out.println("优先级必须是整数");
        }
    } else {
        System.out.println("Usage: exec <filename> <priority>");
    }
    break;
```

addevc

新增指令：addevc <deviceName> <deviceId>

备注：deviceName只用于前端展示，是IODevice的一个属性，其余交互仍使用deviceId作为唯一标识

```
case "addevc":
    if (parts.length == 3) {
        String deviceName = parts[1];
        try {
            int deviceId = Integer.parseInt(parts[2]);
            deviceManager.addDevice(deviceId, deviceName);
        } catch (NumberFormatException e) {
            System.out.println("设备ID必须是整数");
        }
    } else {
        System.out.println("Usage: addevc <deviceName> <deviceId>");
    }
    break;
```

rmdevc

新增指令 rmdevc <deviceId>

```
case "rmdevc":
    if (parts.length == 2) {
        try {
            int deviceId = Integer.parseInt(parts[1]);
            deviceManager.removeDevice(deviceId);
        } catch (NumberFormatException e) {
            System.out.println("设备ID必须是整数");
        }
    } else {
        System.out.println("Usage: rmdevc <deviceId>");
    }
    break;
```

kill

新增指令：kill <pid>

```
case "kill":
    if (parts.length == 2) {
        try {
            int pid = Integer.parseInt(parts[1]);
            process_management.Scheduler.getInstance().terminateProcess(PCB.getPCB(pid));
        } catch (NumberFormatException e) {
            System.out.println("进程ID必须是整数");
        }
    } else {
        System.out.println("Usage: kill <pid>");
    }
    break;
```

新增及修改机器指令 finished

修改C

目前问题：假设程序时间片为200ms，在执行 C 1000 时 CPU 会 sleep 1000ms 再去处理能够触发进程调度的时钟中断，导致该进程无法被抢占。

修改目标：执行200ms后会被抢占，但是保存剩余的没执行完的时间，本例中剩余800ms，会在下一次被调度进CPU时继续执行剩余的时长，直到执行完才能执行下一条机器指令。

修改R/W

目前问题：执行文件读写时进程处于运行态。

修改目标：执行文件读写时使进程阻塞，进入阻塞队列，完成读写后回到就绪队列。

修改Q

Q指令目前对PCB在各个层面上的资源释放仍然存在问题，需要进一步排查。

新增M

M <byte> 申请使用的内存大小

新增MW/MR

MW/MR <logicAddress> <bytes> 写入指定长度随机数据或者读取指定长度数据

进程调度算法 finished

主要相关函数：

public PCB getNextProcess()

private PCB selectProcessFromQueue(BlockingQueue queue, SchedulingPolicy policy)

目前问题：进程调度算法混杂。

需求：根据一个全局变量，确定系统当前使用的唯一一种进程调度算法。

备注：MLFQ的最低优先级队列直接采用FCFS属于该算法实现的一部分，不算混杂算法。

PCB新增属性 finished

暂定名称：originalInstruction和remainInstruction

originalInstruction：保存当前PCB正在执行的指令，用于前端展示

remainInstruction：与机器指令C相关，保存被抢占后剩余的执行时间

例：时间片为200ms的PCB执行C 1000时，200ms后被抢占，此时originalInstruction为C 1000，remainInstruction为C 800

修改涉及部分：PCB.java

```
private String originalInstruction; // 当前正在执行的指令
private String remainInstruction;   // 被抢占后剩余的执行时间的执行指令
```

```
public String getOriginalInstruction() {
    return originalInstruction;
}

public void setOriginalInstruction(String originalInstruction) {
    this.originalInstruction = originalInstruction;
}

public String getRemainInstruction() {
    return remainInstruction;
}

public void setRemainInstruction(String remainInstruction) {
    this.remainInstruction = remainInstruction;
}
```

PCBlist finished

新增一个系统数据结构PCBlist，里面存有所有活跃的PCB，用于前端便利的获取各个进程的状态信息。

修改涉及部分：

PCB.java

```
// 存储所有活跃PCB的线程安全列表
private static final List<PCB> activePCBs = Collections.synchronizedList(new ArrayList<>());
```

```
// 获取所有活跃PCB的列表
public static List<PCB> getActivePCBs() {
    return Collections.unmodifiableList(activePCBs);
}

// 从活跃PCB列表中移除PCB
public void removePCB() {
    activePCBs.remove(this);
}

// 将PCB添加到活跃PCB列表中
public void addPCB() {
    activePCBs.add(this);
}
```

Scheduler.java

在`createProcess`方法中新建好PCB之后就将新建的PCB加入`PCBlist`

```
// 当进程结束时，清理相关资源
public void terminateProcess(PCB pcb) {
    if (pcb != null) {
        schedulerLock.lock();
        try {
            System.out.println("调度器: 终止进程 " + pcb.getPid());

            // 从等待时间映射中移除
            waitingTimeMap.remove(pcb.getPid());

            // 释放页表和内存资源
            memory_management.PageTableArea.getInstance().removePageTable(pcb.getPid());

            // 释放PID
            PIDBitmap.getInstance().freePID(pcb.getPid());

            // 如果进程持有任何文件锁，释放它们
            file_disk_management.FileLockManager.getInstance().releaseAllLocks(pcb.getPid());

            // 从活跃PCB列表中移除
            pcb.removePCB();

            System.out.println("进程 " + pcb.getPid() + " 已终止，所有资源已释放");
        } finally {
            schedulerLock.unlock();
        }
    }
}
```

在`terminateProcess`方法中释放资源的时候将`PCBlist`中对应PCB删除

```java
public PCB createProcess(String filename) {
    // 使用PIDBitmap分配PID
    int pid = PIDBitmap.getInstance().allocatePID();

    // 检查是否成功分配PID
    if (pid == PIDBitmap.EMPTY_PID) {
        System.out.println("无法创建新进程: PID资源已耗尽");
        return null;
    }

    String fileContent = fileSystem.readFileData(filename);
    if (fileContent.equals("-1")) {
        System.out.println("无法读取可执行文件: " + filename);
        return null;
    }

    int codeSize = fileContent.length();
    int[] diskAddressBlock = fileSystem.getFileDiskBlock(filename);

    // 所有新进程都拥有最高优先级(0)
    int highestPriority = 0;
    PCB pcb = new PCB(pid, codeSize, diskAddressBlock, highestPriority);
    pcb.setState(ProcessState.READY);
    pcb.setExecutedFile(filename);

    // 设置时间片
    pcb.setTimeSlice(TIME_SLICE_BY_PRIORITY[highestPriority]);

    // 将新进程添加到最高优先级的就绪队列
    readyQueues.get(highestPriority).add(pcb);
    System.out.println("创建进程 " + pid + ", 执行文件: " + filename + ", 优先级为 " + highestPriority);

    // 将新进程添加到活跃PCB列表中
    pcb.addPCB();

    // 创建进程后立即尝试调度
    schedule();
```

中断 finished

目前问题：没有将所有的中断处理程序集成在一个中断向量表中统一调用，导致中断模块很分散

需求：创建一个interruptHandler（名称暂定）类，把所有中断处理函数放到这个类里，原有调用位置要先经过interruptHandler（名称暂定）类提供的接口再调用中断处理函数，相当于包装了一层。

增加了`interruptHandler`类,同时集成了其他中断处理函数

关于设备中断：目前设备IO完成后，直接调用scheduler类的接口把进程放回就绪队列，这个过程要单独写一个中断处理函数来进行操作。

增加了`deviceInterrupt`类，同时也封装进了`interruptHandler`类

系统启动引导程序

系统启动时先用一个程序通过询问用户的方式在前端交互确定系统的一些可变常量参数。

测试

对各个模块进行单元测试