北京邮电大学

《操作系统课程设计》实验报告

**指导教师 李文生**

|  |  |  |
| --- | --- | --- |
| **班级** | **学号** | **姓名** |
| **2020211304** | **2020211346** | **倪玮昊** |
| **2020211304** | **2020211345** | **缪奇志** |
| **2020211304** | **2020211347** | **钱锡锐** |
| **2020211304** | **2020211339** | **葛北鱼** |
| **2020211304** | **2020211349** | **施天勤** |
| **2020211304** | **2020211350** | **石亚行** |
| **2020211304** | **2020211340** | **何正豪** |

**计算机学院**

**2023 年 5 月**

## 1．课程设计题目

编写代码，模拟操作系统的文件管理，内存管理，进程管理，并使得3个模块之间互相联系，形成一个完整的操作系统。

## 2．课程设计目标和要求

根据本次课程设计的相关要求，团队当前的整体工作与实现目标在于实现一个具有三大基本功能的操作系统模拟程序，该程序需要能够模拟现代操作系统所具备的三部分功能：

① 进程管理；② 内存管理；③ 文件管理;

该程序应能够支持在裸机上独立运行，或作为高层应用，能够支持多平台的运行（例如Windows 操作系统环境、OpenEuler 操作系统环境）。除了程序外，团队还应当为本次完整的开发过程配备相应完整的说明性文档。

同时该系统应该具有较为合理的UI交互系统,便于用户的使用.

## 3．需求分析

### 内核

可以对输入的指令进行简单处理,并且传入其他模块解析

### 进程模块

进程模块负责对批处理任务进行解析，使用ProcessControlBlock作为进程唯一存在的标志，并在ProcessControlBlock中存储进程相关信息。之后对内核创建的用户进程实体进行统一的调配与管理， 实现系统核心资源与外设的充分利用， 实现系统核心资源与外设的充分利用，并通过时间片、优先级的机制，使得用户能够以更加灵活的方式与系统进行任务交互。

### 内存管理模块

内存管理模块在该系统中主要负责对系统内存资源的同意调配与管理，Kernel会向内存管理模块提出请求，通过虚拟内存机制，对可执行文件在内存里进行有效的装载与回收

### 设备管理模块

设备管理模块需要完成以下功能：

* 设备的申请
* 设备的分配
* 记录设备安装时间、使用时间、空闲时间等相关时间
* 设备的卸载和安装
* 设备状态更新
* 所有设备信息(如使用设备的进程号、设备种类等)的统计和输出

### UI:

设计可视化系统交互界面，实现用鼠标点击的方式调用系统功能，并使用图形来更直观地表示操作系统各模块的运行状态，包括进程的运行状态、页框的使用状态、文件系统的显示等。重点需要处理的内容是界面交互的逻辑、界面与内核的信息交流以及系统信息的图形化表示。

#### 文件管理模块

提供ls（列出目录文件），cd（切换工作路径），mkdir（创建文件夹），mkf（创建文件），rm（删除文件和文件夹），vi（编辑可编辑文件的内容）等功能。

#### 硬盘管理模块

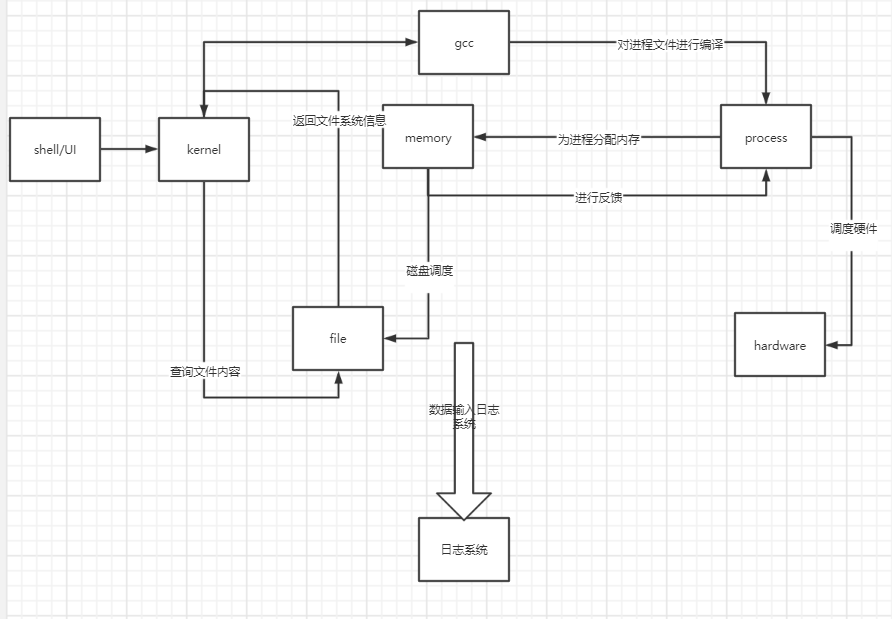
提供使用FIFO,SSTF,SCAN,CSCAN,LOOK,CLOOK六种策略对同一请求序列情况下，计算磁头移动的距离并将的方法。

## 4．开发环境

JAVA语言,用idea进行开发

## 5．总体设计

### 调用逻辑:



shell或者UI通过管道向kernel输入指令,对各个模块进行调用,同时把所以的数据写入日志系统,disk模块通过模拟磁头数组移动来进行一个磁盘调度,hardware用来模拟硬件,gcc通过检查执行文件的格式来决定是否进行编译.

### 具体指令:

| 命令 | 格式 | 功能 |
| --- | --- | --- |
| re | re [command] | 对 re 之后字段的命令中的路径字段采用正则表达式进行解析与替换 |
| man | man [command1] [command2] ... | 展示单条或多条命令的帮助信息，若未携带具体命令，则默认对所有命令进行展示 |
| ls | ls [-a|-l-al][path] | 列出指定路径 path 的内容，-a 选项列出全部内容（包括隐藏文件或目录），-l 选项列出文件或目录的详细信息，使用-al 同时指明以上两个选项。若未提供 path 参数，则默认 path 为当前目录 |
| cd | cd [path] | 修改当前工作目录，若未提供 path 参数，则默认 path 为系统根目录 |
| rm | rm [-r|-f|-rf] path | 删除文件或目录。其中待删除的文件或目录的路径必须提供，-r 选项能够递归地删除目录，-f 选项对应于强制删除功能，使用-rf 同时指明以上两个选项 |
| mkf | mkf path size | 创建具有指定大小的文件 |
| mkdir | mkdir path | 创建目录 |
| dss | dss | 展示系统外存各磁盘块的占用状态 |
| dms | dms | 展示系统物理内存占用状态 |
| exec | exec path | 执行指定路径下的文件，该文件须为可执行文件 |
| ps | ps | 展示当前系统所有进程状态 |
| rs | rs | 展示当前系统所有资源的使用状态 |
| td | td | 整理系统磁盘外部碎片，即“紧凑”操作 |
| kill | kill pid | 强制结束指定进程 |
| exit | exit | 退出 |
| vi | vi [path] | 打开编辑器 创建文件 |
| gcc | gcc [num] path | 编译程序 num为优先级 |
| mc | mc [kind] | 修改内存分配模式 |
| ar | ar [device] | 添加设备(camera or printer) |
| pc | pc 查表模式 | 修改查表模式 |

## 6．详细设计

### FILE(1)

* **public List<String> ls(String dir\_path, String mode, String method)**

1. 列出目录文件：函数通过传入目录路径 dir\_path 实现列出目录文件的功能。
2. 支持相对路径和当前工作目录：如果 dir\_path 参数为空，则默认列出当前工作目录的文件；否则，列出目标目录的文件。
3. 支持不同的显示模式：函数通过 mode 参数控制显示的模式，支持的模式包括 -a（显示所有文件，包括隐藏文件）、-l（以详细列表形式显示文件信息）、-al（同时显示所有文件并以详细列表形式显示文件信息）。
4. 提供正则表达式匹配功能：如果 method 参数为 "get"，则返回符合正则表达式匹配的文件列表，用于实现shell的正则表达式匹配功能。
5. 处理异常情况：函数能够处理路径出错、目标文件不存在或不是文件夹等异常情况，并返回相应的错误信息。
6. 输出结果信息：函数能够将结果信息打印到控制台，并返回结果列表。
7. 文件和文件夹的显示区分：函数能够根据文件和文件夹的不同类型，采用不同的显示方式。文件夹可以高亮显示，可执行文件可以高亮显示为绿色。
8. 隐藏文件的处理：根据模式设置，可以选择是否显示隐藏文件（以 "." 开头的文件）。
9. 目录为空时的处理：如果目录为空，则直接返回空列表。
10. 错误选项的处理：如果 mode 参数不是有效的选项，则返回错误信息。

* **public String cd(String dir\_path)**

1. 支持切换当前工作目录：函数通过传入目录路径 dir\_path 实现对当前工作目录的切换。
2. 支持相对路径和绝对路径：函数能够处理相对路径，并根据当前工作目录的状态进行切换。
3. 支持特殊目录符号的处理：函数能够处理 "."（表示当前目录）、".."（表示上一级目录）等特殊目录符号。
4. 提供异常处理：函数能够处理路径出错、目录不存在或不是目录等异常情况，并返回相应的错误信息。
5. 更新当前工作目录：函数能够根据目录切换的结果更新当前工作目录的路径 current\_working\_path。
6. 输出结果信息：函数能够将结果信息打印到控制台，并返回结果字符串。
7. 未解决异常：函数中有一处注释指出未解决的异常情况，即当路径以多个文件分隔符结尾时的处理方式。

* **public String mkdir(String dir\_path) throws IOException**

1. 创建目录：函数通过传入目录路径 dir\_path 实现创建目录的功能。
2. 支持相对路径和绝对路径：函数能够处理相对路径。如果 dir\_path 参数是相对路径，则在当前工作目录下创建目录。
3. 处理异常情况：函数能够处理路径出错、目标目录已存在等异常情况，并返回相应的错误信息。
4. 创建成功与失败的处理：如果成功创建目录，则返回成功信息；如果目录已存在或创建失败，则返回相应的失败信息。
5. 更新目录结构：如果成功创建目录，函数会更新当前工作目录的目录结构，并将新目录添加到父目录的映射中。
6. 输出结果信息：函数能够将结果信息打印到控制台，并返回结果字符串。
7. 异常处理：函数通过异常处理机制处理文件系统相关的异常，如 I/O 异常。
8. 可选的文件输入记录：如果存在文件输入流 FileManagerWin.fileInput，则在创建目录成功或失败时，将相应的结果写入文件输入流。

* **public String rm(String file\_path, String mode)**

1. 删除文件或目录：函数通过传入文件或目录的路径 file\_path 实现删除功能。
2. 支持相对路径和绝对路径：函数能够处理相对路径和绝对路径。如果 file\_path 参数是相对路径，则在当前工作目录下删除文件或目录；如果是绝对路径，则在根目录下删除文件或目录。
3. 处理异常情况：函数能够处理路径出错、目录不存在、文件只读等异常情况，并返回相应的错误信息。
4. 强制删除：如果传入的 mode 参数以 -r 开头，表示要强制删除目录。在强制删除目录时，函数会递归地删除目录下的所有文件和子目录。
5. 删除文件：如果传入的 mode 参数为空或为 -f，表示要删除文件。函数会判断文件是否存在，并根据文件的可写性进行删除操作。
6. 更新目录结构：如果成功删除文件或目录，函数会更新当前工作目录的目录结构，并从父目录的映射中删除相应的文件或目录。
7. 输出结果信息：函数能够将结果信息打印到控制台，并返回结果字符串。
8. 删除文件块：在删除文件时，函数会调用 delete\_file\_from\_blocks 方法删除文件占据的磁盘块。
9. 异常处理：函数通过异常处理机制处理文件系统相关的异常，如无效的路径异常。
10. 支持 -rf 选项：如果传入的 mode 参数为 -rf，表示要强制递归删除目录，即使目录非空。

* **public String mkf(String file\_path, String file\_type, String size) throws IOException**

1. 创建文件：函数通过传入文件的路径 file\_path、文件类型 file\_type 和大小 size 来创建文件。
2. 文件类型限制：函数限制只能创建普通文件，如果传入的文件类型不是 c 开头，则返回错误信息。
3. 处理异常情况：函数能够处理路径出错和文件已存在的异常情况，并返回相应的错误信息。
4. 构建文件内容：函数使用 JSON 格式构建文件的内容，并包括文件名、文件类型、文件大小和内容列表。
5. 输出 JSON 数据：函数将构建的 JSON 数据打印到控制台。
6. 处理路径：函数通过调用 path\_split 方法将文件路径分割为上层路径 upper\_path 和文件名 basename。
7. 获取当前工作目录：函数通过调用 path2map 方法获取当前工作目录的映射。
8. 检查文件名是否存在：函数检查当前工作目录中是否已存在同名的文件，如果存在则返回错误信息。
9. 填充文件数据块：函数调用 fill\_file\_into\_blocks 方法将文件的 JSON 数据填充到数据块中，并返回是否填充成功的结果。
10. 写入文件：如果填充文件数据块成功，函数将 JSON 数据写入文件，并保存到指定的路径。
11. 更新目录结构：如果成功创建文件，函数会更新当前工作目录的目录结构，并在父目录的映射中添加相应的文件。
12. 输出结果信息：函数能够将结果信息打印到控制台，并返回结果字符串。
13. 处理 IO 异常：函数能够处理文件写入过程中可能发生的 IO 异常，并返回相应的错误信息。
14. 支持传入内容列表：函数接受一个内容列表 contents，并将其添加到文件的 JSON 数据中。

* **public List<String> readContentFromFile(String filePath)**

1. 读取文件内容：函数通过传入文件路径 filePath 来读取文件的内容。
2. 构建完整路径：函数根据根路径 root\_path 和当前工作路径 current\_working\_path，将传入的文件路径 filePath 拼接成完整的文件路径。
3. 异常处理：函数能够处理文件读取过程中可能发生的异常，并打印异常信息。
4. 解析 JSON 文件：函数使用 JSON 解析器（JsonParser）解析指定路径的 JSON 文件。
5. 获取内容数组：函数从解析得到的 JSON 对象中获取名为 "content" 的 JSON 数组（contentArray）。
6. 遍历内容数组：函数遍历内容数组，逐个提取数组元素，并去除双引号，将内容添加到结果列表 result 中。
7. 返回结果：函数返回读取到的内容列表 result。

* **public String vi(String path)**

1. 验证路径权限：函数检查当前工作路径和传入的文件路径是否为特定目录 \\user 的组合，如果是，则返回错误信息。
2. 构建完整路径：函数将根路径 root\_path、当前工作路径 current\_working\_path 和传入的文件路径 path 拼接成完整的文件路径。
3. 判断文件状态：函数通过创建 File 对象，判断文件是否存在、是否为目录，并进行相应的错误处理和返回。
4. 读取文件内容：函数调用 readFile 方法，读取文件的内容并返回。
5. 解析 JSON 数据：函数将文件内容转换为 JSON 对象，以便后续提取 content 字段。
6. 检查文件编辑权限：函数检查 JSON 对象中的 type 字段，如果不包含字符 'x'，则表示文件不可编辑，返回相应的错误信息。
7. 提取内容数组：函数从 JSON 对象中获取名为 content 的 JSON 数组，并进行遍历。
8. 拼接内容字符串：函数将内容数组中的每个元素拼接为一个字符串，用于后续在文本框中显示。
9. 返回结果：函数返回拼接好的内容字符串。

* **public void sv(String path,String fileData) throws IOException**

1. 构建完整路径：函数将根路径 root\_path、当前工作路径 current\_working\_path 和传入的文件路径 path 拼接成完整的文件路径。
2. 读取文件内容：函数调用 readFile 方法，读取文件的内容并将其保存在 text\_content 变量中。
3. 解析 JSON 数据：函数将文件内容转换为 JSON 对象，以便后续提取 content 字段。
4. 处理文件数据：函数接受传入的文件数据 fileData，将其按行分割成字符串数组 text。
5. 转换为 JSON 数组：函数使用 Gson 将字符串数组 text 转换为 JsonArray 对象。
6. 更新 JSON 对象：函数将转换后的 JsonArray 对象放回原来的 JSON 对象的 content 字段，并更新 size 字段为 fileData 的长度。
7. 转换为 JSON 字符串：函数使用 Gson 将更新后的 JSON 对象转换为 JSON 字符串 jsonData。
8. 写入文件：函数调用 writeFile 方法，将更新后的内容字符串 jsonData 写回文件中。
9. 重新分配磁块：函数调用 delete\_file\_from\_blocks 方法，删除原来文件占用的磁块，并调用 fill\_file\_into\_blocks 方法，将更新后的 JSON 对象分配到磁块中。
10. 输出成功信息：函数打印输出 "success" 表示操作成功。

### DISK

* **DiskScheduler类**

1. 类名：DiskScheduler
   * 磁盘调度器，用于处理磁盘读写请求的调度和计算磁头移动距离。
2. 属性：
   * private int initialPosition：初始磁头位置。
   * private int totalDistance：总的磁头移动距离。
   * private int[] requests：磁盘读写请求列表。
   * private int endPosition = 199：磁盘终止位置。
   * private int startPosition = 0：磁盘起始位置。
3. 构造函数：
   * DiskScheduler()：默认构造函数，未实现具体逻辑。
4. 方法：
   * public void Disk\_find(int[] requests)：处理磁盘读写请求的方法。
     + 参数：int[] requests - 磁盘读写请求列表。
     + 首先对请求列表中的每个元素进行取模操作，确保请求在磁盘范围内。
     + 生成一个随机的初始位置（initialPosition）。
     + 调用allResult方法获取所有算法的计算结果。
     + 将计算结果以及初始位置和请求列表写入文件"System/block\_access"。
   * private List allResult()：计算所有算法的磁头移动距离。
     + 返回一个包含所有算法计算结果的列表。
   * private int FIFO()：先进先出（FIFO）算法的磁头移动距离计算。
     + 复制请求列表。
     + 初始化磁头位置为初始位置（initialPosition）。
     + 初始化磁头移动距离为0。
     + 遍历请求队列，按照先进先出的顺序处理请求。
     + 计算当前请求和磁头位置之间的距离，将磁头移动到当前请求的位置，并累计磁头移动距离。
     + 返回磁头移动距离。
   * private int SSTF()：最短寻道时间优先（SSTF）算法的磁头移动距离计算。
     + 复制请求列表。
     + 计算请求总数。
     + 记录已访问的请求。
     + 记录磁头当前位置为初始位置（initialPosition）。
     + 记录磁头移动距离为0。
     + 循环处理所有请求，每次找到与当前位置最近的未访问请求，记录移动距离并标记该请求为已访问。
     + 返回磁头移动距离。
   * private int LOOK()：LOOK算法的磁头移动距离计算。
     + 复制请求列表。
     + 对请求列表进行排序。
     + 记录移动距离和磁头顺序的数组。
     + 初始化磁头位置为初始位置（initialPosition）。
     + 初始化移动距离为0。
     + 使用index变量记录当前位置在排序后的请求列表中的索引。
     + 遍历排序后的请求列表，找到第一个大于等于当前位置的请求，并记录其索引index。
     + 在增大方向上遍历请求列表，记录每个请求与当前位置的距离，并更新当前位置和移动距离。
     + 在减小方向上遍历请求列表，记录每个请求与当前位置的距离，并更新当前位置和移动距离。
     + 将所有移动距离相加，得到总的磁头移动距离。
     + 返回磁头移动距离。
   * private int CLOOK()：循环LOOK算法的磁头移动距离计算。
     + 复制请求列表。
     + 对请求列表进行排序。
     + 记录移动距离和磁头顺序的数组。
     + 初始化磁头位置为初始位置（initialPosition）。
     + 初始化移动距离为0。
     + 使用index变量记录当前位置在排序后的请求列表中的索引。
     + 遍历排序后的请求列表，找到第一个大于等于当前位置的请求，并记录其索引为index。
     + 在增大方向上遍历请求列表，记录每个请求与当前位置的距离，并更新当前位置和移动距离。
     + 在减小方向上遍历请求列表，记录每个请求与当前位置的距离，并更新当前位置和移动距离。
     + 将所有移动距离相加，得到总的磁头移动距离。
     + 返回磁头移动距离。
   * private int SCAN()：扫描（SCAN）算法的磁头移动距离计算。
     + 将请求列表转换为ArrayList。
     + 在ArrayList末尾添加终止位置。
     + 将ArrayList转换为数组。
     + 对数组进行排序。
     + 记录移动距离和磁头顺序的数组。
     + 初始化磁头位置为初始位置（initialPosition）。
     + 初始化移动距离为0。
     + 使用index变量记录当前位置在排序后的请求列表中的索引。
     + 遍历排序后的请求列表，找到第一个大于等于当前位置的请求，并记录其索引为index。
     + 在增大方向上遍历请求列表，记录每个请求与当前位置的距离，并更新当前位置和移动距离。
     + 在减小方向上遍历请求列表，记录每个请求与当前位置的距离，并更新当前位置和移动距离。
     + 将所有移动距离相加，得到总的磁头移动距离。
     + 返回磁头移动距离。

### FILE(2):

#### 详细设计

##### 类属性

* private static final String file\_separator = File.separator
* 该属性为系统文件分隔符，用于在文件路径中分隔目录与文件名，根据操作系统而动态提供。
* private static final String root\_path = System.getProperty("user.dir") + file\_separator + "File"
* 该属性表示文件系统的根目录，通过系统属性user.dir获取当前程序所在的目录，并拼接File目录名，作为文件系统的根目录。
* public static String current\_working\_path = file\_separator
* 该属性表示当前工作目录，初始值为文件分隔符，即根目录。
* private int block\_size
* 该属性表示磁盘块的大小，单位为字节。
* private int block\_number
* 该属性表示磁盘块的数量。
* private int tracks
* 该属性表示磁盘磁道的数量。
* private int secs
* 该属性表示磁盘扇区的数量。
* private int[] unfillable\_block = {3, 6, 9, 17}
* 该属性表示不能使用的磁盘块的编号，初始值为{3, 6, 9, 17}。
* private HashMap<String, int[]> block\_dir = new HashMap<>()
* 该属性表示文件路径和磁盘使用情况的映射，初始值为空的HashMap。
* private int[] bitmap
* 该属性表示磁盘块的占用情况，初始值为所有磁盘块均为可用状态。
* private ArrayList<Block> all\_blocks = new ArrayList<>()
* 该属性表示磁盘中所有的块的ArrayList。
* private Map<String, Object> file\_system\_tree
* 该属性表示文件系统树，采用Map形式表示，其中文件名为键，当该文件为文件夹时，其值为一个Map，否则，其值为长度为4的字符串，表示类型 / 读 / 写 / 执行。

##### 类方法

* public FileManager(int block\_size, int tracks, int secs)

该方法为FileManager类的构造函数，初始化block\_size、block\_number、tracks、secs等属性，并调用其他方法初始化磁盘块、设置不能使用的磁盘块、初始化文件系统树等操作。

* public List<String> dss()
* 该函数的作用是获取所有文件块的信息，并返回一个字符串列表，其中每个字符串包含一个文件块的信息。
* 函数首先创建了一个空的字符串列表result，然后对all\_blocks中的每一个块进行遍历，对每一个块获取其文件指针fp以及在磁盘中的位置loc，并将其添加到result列表中。最后，函数返回result。
* public void change\_page(String word) throws IOException
* 该函数的作用是修改System/user文件中的mode字段为指定的字符串word。
* 函数首先读取文件内容，将其转换为JSON对象，然后修改其中的mode字段。接着，函数将更新后的JSON对象转换为JSON字符串，并将其写回到文件中。
* public void change\_mem(String word) throws IOException
* 该函数的作用是修改System/user文件中的mem字段为指定的字符串word。
* 函数的实现与change\_page类似，只是将要修改的字段由mode改为了mem。
* private ArrayList<Block> initBlocks()
* 该函数的作用是初始化文件块。
* 函数首先创建了一个空的块列表blocks，然后对于每一个文件块，创建一个大小为block\_size，位置为cal\_loc(i)的块，并将其添加到blocks中。最后，函数创建了一个大小为block\_number的位图bitmap，并将其中的所有元素设置为1。函数最终返回blocks。
* private int[] cal\_loc(int block\_num)
* 该函数的作用是根据给定的块编号计算该块在磁盘中的位置。
* 函数首先根据block\_num计算出该块所在的磁道号track以及该磁道内的扇区号sec，然后将它们放入一个大小为2的整型数组中，并返回该数组。
* private void setUnfillableBlock()
* 该函数的作用是将所有不可用的文件块的位图标记为0。
* 函数遍历了所有不可用的文件块，将它们在位图中的位置标记为0。
* private void freeUnfillableBlock()
* 该函数的作用是将所有不可用的文件块的位图标记为1。
* private Map<String, Object> init\_file\_system\_tree(String now\_path)

函数参数：

* now\_path：当前递归到的绝对路径

函数返回：

* Map<String, Object>：当前文件夹对应的Map，文件名为键，当该文件为文件夹时，其值为一个Map，否则，其值为长度为4的字符串，表示类型 / 读 / 写 / 执行。

函数作用：

初始化文件系统树。递归遍历当前路径下所有的文件和子目录，将其添加到文件系统树中。使用Map形式存储文件系统树，文件名为键，当该文件为文件夹时，其值为一个Map，否则，其值为长度为4的字符串，表示类型 / 读 / 写 / 执行。

对于每个文件，如果是文件夹，则递归调用该函数。如果是文件，则使用BufferedReader读取文件内容，并使用JSON.parseObject方法将其解析成一个JSON对象，提取type字段并存储在文件系统树中。

* private int fill\_file\_into\_blocks(JSONObject f, String fp, int method)

函数参数：

* f：JSON对象，包含文件信息，如文件大小等。
* fp：文件路径，用于在文件系统块中存储该文件信息。
* method：块分配方法，0：first fit，1：best fit，2：worst fit。

函数返回：

* int：0表示文件填充成功，-1表示没有足够的存储空间。

函数作用：

将文件内容存储到磁盘块中。首先计算文件需要占用多少个磁盘块，并查找可用的连续块（使用find\_free\_blocks方法）。如果没有足够的连续块，则返回-1表示没有足够的存储空间。

然后，将该文件的占用块的信息存储到block\_dir中，包括第一个块的位置、占用块的数量和文件大小。使用all\_blocks和bitmap来存储文件占用的所有块，并将这些块标记为已使用。

* private String bitmap2str(int[] bm)

函数参数：

* bm：整数数组，表示位图。

函数返回：

* String：表示整数数组转换成的字符串。

函数作用：

将整数数组转换成字符串，方便进行匹配。

* find\_free\_blocks(int num, int method)

此函数为主函数，用于调用不同的文件填充算法。它的参数有两个：

* num：需要的连续块数目。
* method：选择使用的文件填充算法，其值必须为0、1或2，分别表示使用first fit、best fit和worst fit算法。

此函数的返回值为int型的连续块的磁盘块号。如果出错则返回-1。

* block\_first\_fit(String goal\_str)

此函数为first fit算法。它的参数是需要的连续块（bitmap形式）的字串。此函数首先将bitmap转换成字符串，然后在其中寻找连续的与goal\_str匹配的子串。如果找到则返回其索引值，否则返回-1。

* block\_best\_fit(String goal\_str)

此函数为best fit算法。它的参数是需要的连续块（bitmap形式）的字串。此函数首先将bitmap划分为一系列连续的空闲块，并记录它们的起始位置和长度。然后对所有空闲块按照长度进行升序排序，寻找第一个满足需要的连续块长度的空闲块并返回其起始位置。如果没有找到合适的空闲块则返回-1。

* block\_worst\_fit(String goal\_str)

此函数为worst fit算法。它的参数是需要的连续块（bitmap形式）的字串。此函数首先将bitmap划分为一系列连续的空闲块，并记录它们的起始位置和长度。然后对所有空闲块按照长度进行降序排序，返回最大的空闲块的起始位置。如果没有找到合适的空闲块则返回-1。

* path2map

该函数实现了将路径转换为嵌套的Map的功能。

输入参数

dir\_path: 字符串类型，表示要转换为Map的路径。

返回值

一个Map类型的变量，表示转换后的文件系统目录树结构。

实现细节

1. 判断输入路径是否为空或以根目录“\”开头，若不是，则将其与当前工作路径合并。
2. 将输入路径按照文件路径分割函数（path\_split）返回的结果，即不包含空目录的目录名列表（dir\_list\_cleaned）进行遍历。
3. 如果目录名为“.”，则跳过，如果为“..”，则从上一级目录中查找文件。
4. 如果目录名为普通目录，则将该目录加入到目录树的嵌套Map中，并将嵌套Map指向该目录的子目录。

* path\_split

该函数实现了将文件路径分割为目录和文件名两部分的功能。

输入参数

path: 字符串类型，表示要分割的文件路径。

返回值

一个字符串数组类型的变量，包含两个元素。第一个元素为文件所在目录的路径，第二个元素为文件名。

实现细节

1. 如果路径最后一个字符是“\”，则去掉该字符。
2. 从路径最后一个“\”位置处将路径分割为两部分，第一部分是文件所在目录，第二部分是文件名。
3. 如果第一部分路径最后一个字符是“\”，则去掉该字符。

* fp2loc

该函数实现了根据文件路径返回其存储block的位置的功能。

输入参数

fp: 字符串类型，表示要查找的文件路径。

返回值

一个由多个int数组组成的List类型的变量，每个int数组代表一个block的位置信息，包括该block所在磁盘的编号、该block在磁盘中的偏移量。

实现细节

1. 判断输入路径是否为相对路径，如果是，则将其与当前工作路径合并。
2. 从文件块所在目录（block\_dir）中获取该文件的存储位置信息，包括第一个block的编号、占用block数量和文件大小。
3. 根据存储位置信息，遍历每个block，并将每个block的位置

* delete\_file\_from\_blocks(String fp)

这个方法是用来删除文件的。它接收一个字符串参数 fp，表示要删除的文件在磁盘上的路径。

在方法内部，首先通过 block\_dir 获取文件在磁盘上的起始位置 start 和长度 length。接下来，它遍历了文件所占用的所有块，即从起始位置开始遍历到起始位置加上文件长度减一的位置，对于每个块，它将该块的空闲空间设置为块大小，文件指针设置为 null，位图上的对应位置设置为 1。

最后，它从 block\_dir 中删除该文件。

* tidy\_disk()

这个方法是用来整理磁盘碎片的。它不接受参数。

在方法内部，它首先创建了一个新的 block\_dir，并将当前 block\_dir 中的所有条目复制到新的 block\_dir 中。接下来，它调用 initBlocks() 方法来创建新的块列表 all\_blocks，以便在整理后的磁盘上重新存储文件数据。

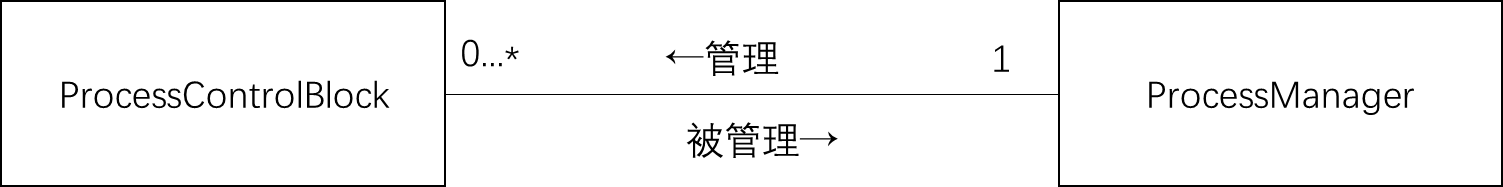
然后，它遍历了新的 block\_dir，对于每个条目，它从中获取文件大小信息，并将其作为字符串 "size:文件大小" 进行解析为 JSON 对象 data。接下来，它调用 fill\_file\_into\_blocks() 方法，将文件数据填充到新的块列表中。

### PROCESS:

本模块负责对批处理任务进行解析，使用ProcessControlBlock作为进程唯一存在的标志，并在ProcessControlBlock中存储进程相关信息。之后对内核创建的用户进程实体进行统一的调配与管理， 实现系统核心资源与外设的充分利用， 实现系统核心资源与外设的充分利用，并通过时间片、优先级的机制，使得用户能够以更加灵活的方式与系统进行任务交互。

#### 结构设计

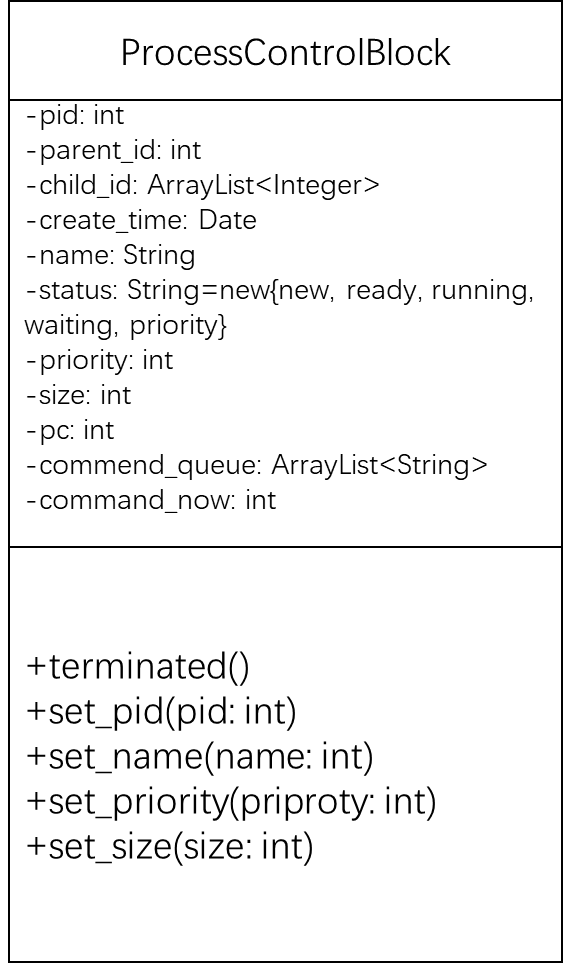
在需求中，构建两个类ProcessControlBlock和ProcessManager，ProcessControlBlock为单个PCB，ProcessManager为PCB管理器，其关系如下：



##### ProcessControlBlock

###### 类图

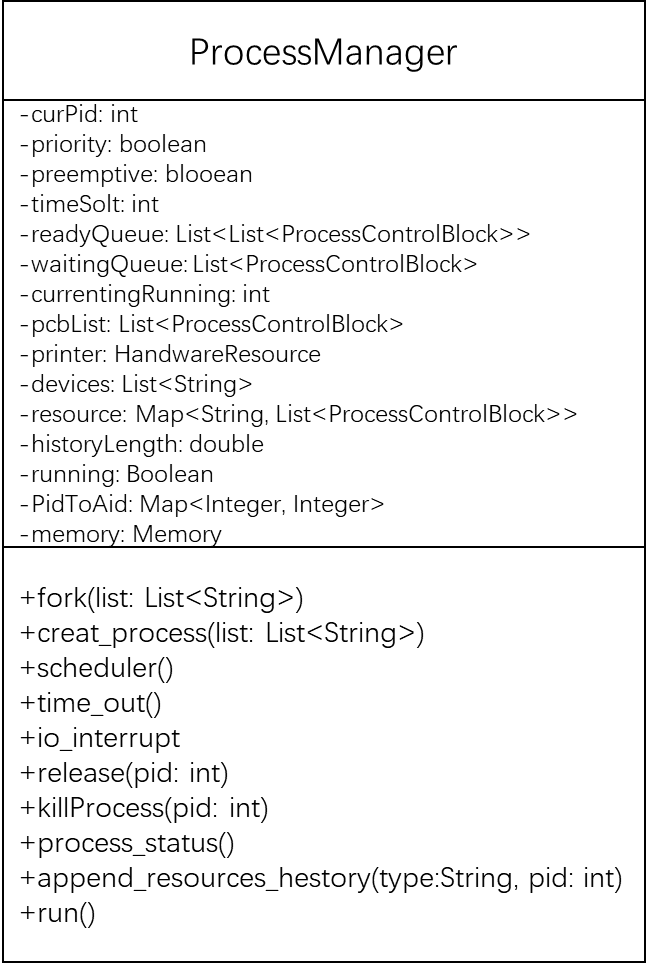
ProcessControlBlock类图如下：



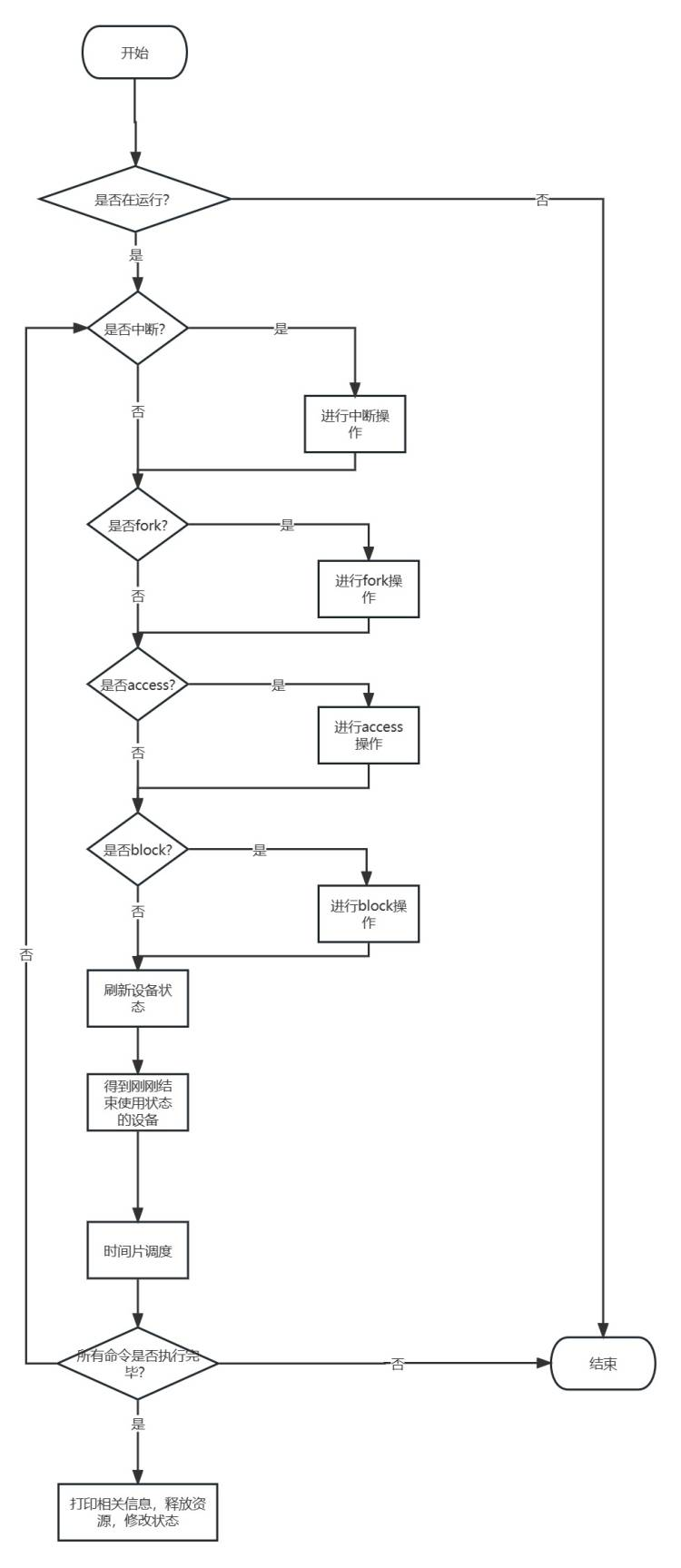
##### ProcessManager

###### 类图

ProcessManager类图如下：



###### 处理逻辑



当开机时就使用另一个进程调用run开始不断循环执行进程，根据时间片轮转的原则切换不同的进程来执行，在进程调度时根据优先级进行进程调度，当需要创建进程时，调用memory类的方法确定是否能够创建进程，在确定内存足够后创建进程，设置状态为ready，加入进程链表以及对应优先级的就绪队列，当需要访问内存时，传入需要访问的地址调用memory的方法进行对内存里该地址的访问。当需要调用其他设备时，首先完成中断请求，把对应的进程的状态设置为waiting，把该进程加入对应的设备的链表。同时在进程执行完一个程序时需要让进程的程序计数器pc加1，最后当程序里的所有指令完毕后就结束该进程，释放该进程占用的资源，把进程的状态设置为terminated，在程序运行中可以监视并且查看各个进程的信息。

#### 主要任务

##### 指令设计

| 批处理指令 | 格式 | 说明 |
| --- | --- | --- |
| fork | fork | 本指令用于在当前进程的运行基础上创建 进程，需要调度 CPU 部件来执行，执行时间为 1 个单位，为原子性操作 |
| access | access int(address) | 本指令用于模拟进程访问其所占用逻辑空间中的任一地址（读或写），执行时间为 1 个单位，为原子性操作 |
| cpu | cpu int(time) | 本指令用于模拟程序请求计算任务，需要调度 CPU 部件来执行，执行时间为 time 个单位，不需要连续执行 |
| printer | printer int(time) | 本指令用于模拟程序请求打印任务，需要调度 Printer 部件来执行，执行时间为 time 个单位，需要连续执行 |
| block | block int[] | 本指令用于模拟寻道算法，后续是一串数据表示磁道的位置 |

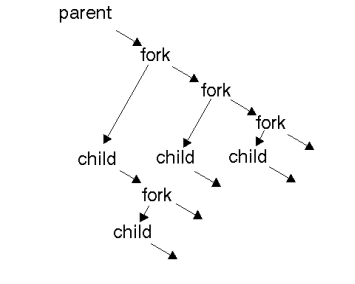
#### 进程创建

**creat\_process:系统自己的程序创建进程**  
public void creat\_process(String name,String priority,String size,List list)

初始化设置好进程名称、进程优先级、进程的大小、进程的指令队列

**fork：进程内创建子进程**  
public void fork(List list)

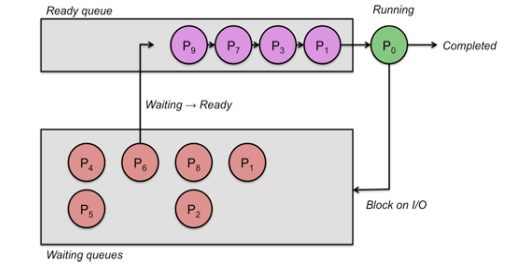
继承父进程的各种参数，程序计数器加一。



#### 进程调度算法

**进程调度算法：public void schduler()**

按照优先级，从ready队列里面找到第一个进程，修改currentrunning，把该pcb从ready队列里面移除。



#### 设备中断处理

**中断管理：public void io\_interrput()**

修改进程状态为waiting，加入到waiting队列。

**释放资源：public void release(int pid)**

把进程从waiting队列里面移除，重新对waiting队列的进程分配设备资源。

#### 终止进程

**杀死进程：public void killProcess(int pid)**

找到进程，释放所占用的内存和资源。设置进程状态为termainted。

### KERNEL:

通过解析命令的格式去跳转到不同的模块进行执行,逻辑较为简单,通过管道或者shell与UI通信.

### GCC:

通过检查代码格式来判断是否符合格式其中，check 方法用于检验字符串列表的格式是否符合要求，readFile 方法用于读取文件内容，writeFile 方法用于将内容写入文件，compile 方法用于编译操作。如果可以编译成功则赋予优先级且修改文件类型

伪代码逻辑如下:

if str.length == 1:
  
 if str[0] equals "fork":
  
 # Do nothing
  
 else:
  
 return false
  
else if str.length == 2:
  
 if str[0] equals "cpu" or str[0] equals "access" or str[0] equals "printer":
  
 if str[1] matches "[0-9]+":
  
 # Do nothing
  
 else:
  
 return false
  
 else if str[0] equals "block" and str.length <= 100:
  
 str1 = str[1].split(" ")
  
 for j from 0 to str1.length:
  
 if str1[j] matches "[0-9]+":
  
 # Do nothing
  
 else:
  
 return false
  
else:
  
 return false

### MEMORY:

##### 连续分配

核心是 ConMemoryManager，它负责内存的分配和释放。它维护了两个列表：freeList用于存储空闲内存块，allocatedList 用于存储已分配的内存块。

在内存分配时，它会遍历 freeList中的每个内存块，找到第一个大小大于等于进程大小的内存块，将其分配给该进程，并从 freeList 中移除该内存块并加入到 allocatedList 中。如果分配后该内存块还有多余的空间，则将其分割为两个内存块，其中一个已被分配，另一个加入到 freeList 中。在释放内存时，它会将该进程占用的内存块从 allocatedList 中移除，并将其与相邻的空闲内存块合并为一个更大的空闲内存块，并将其加入到 freeList 中。

维护了一些辅助变量，如 usedSize 和 freeSize 来记录已使用和空闲内存的大小，以及 usedRate 来记录已使用内存的百分比。在内存分配和释放时更新这些变量，并在更新后使用 JavaFX 更新 UI 上的内存使用情况。

此外，它还提供了一个 ConProcess 类来表示进程，它包含了进程的名称、大小、所在内存块和进程 ID，以及一个 MemoryBlock 类来表示内存块，它包含了内存块的起始地址、结束地址、大小、是否空闲以及分配给哪个进程。

在使用该类时，可以通过 allocate 方法为进程分配内存，如果分配成功则返回 true，否则返回 false。在进程执行结束后，可以通过 deallocate 方法释放该进程占用的内存。可以通过 getFreeList 和 getAllocatedList 方法获取当前空闲内存块和已分配的内存块列表。

##### 页式分配

该模块提供了两种页面置换算法来处理页面故障，包括先进先出（FIFO）和最近最少使用（LRU）算法。程序还跟踪页面访问时间、页面故障和页面替换等信息。

核心是PageMemoryManager类，该类有多个字段来跟踪与内存相关的信息，包括页面帧大小、页面帧数、虚拟内存数、页面表大小、页面表数、访问时间、页面故障、页面替换、页面替换算法、页面表、进程映射和使用率。

PageMemoryManager的构造函数有五个参数：pageFrameSize、pageFrameNum、pageTableSize、pageTableNum和pageReplaceAlgorithm。pageFrameSize表示每个页面帧的大小，pageFrameNum表示物理内存中的页面帧总数，pageTableSize表示一个页表的大小，pageTableNum表示虚拟内存中的页表总数，pageReplaceAlgorithm表示用于页面替换的算法。根据pageReplaceAlgorithm的值，构造函数将初始化FIFO队列或LRU队列。

createProcess方法创建一个新进程并为其分配内存。它有三个参数：pid、name和size。pid是进程ID，name是进程名称，size是进程的大小（以字节为单位）。该方法使用allocateMemory方法为进程分配内存。如果内存分配成功，则将新进程添加到processMap哈希映射中。否则，该方法返回false。

allocateMemory方法为新进程分配内存。它有一个参数：process，代表需要分配内存的进程。该方法计算进程所需的页面数，然后在pageTable对象中搜索空闲的页表条目。如果找到足够的空闲条目，则该方法更新页表条目并返回true。否则，该方法返回false。

accessProcess方法在进程尝试访问内存地址时被调用。它有两个参数：process，代表试图访问内存的进程，address，代表正在访问的内存地址。该方法首先计算地址的页号和偏移量。然后在pageTable对象中查找相应于页号的页表条目。如果页表条目存在，则该方法更新条目的访问时间、访问次数和适当的页面替换队列，并返回1。如果页表条目不存在，则该方法增加页面故障计数器并检查是否有可用的空闲页面帧。如果有，则该方法更新页表条目，分配一个空闲页面帧，并更新适当的页面替换队列。如果没有可用的空闲页面帧，则该方法调用适当的页面替换算法选择要替换的页面。然后该方法更新页表条目，分配所选页面帧，并更新适当的页面替换队列。

该程序还定义了一个PageProcess类，该类有多个字段来跟踪与进程相关的信息，包括名称、大小和pid。PageMemoryManager类可以使用allocateMemory方法创建新进程并为其分配内存。

该程序使用FifoQueue和LruQueue类来实现页面置换算法。当发生页面故障时，PageMemoryManager类会使用页面替换算法并将页面的索引推入相应的队列中。

##### 控制

有一个类名为Memory，这个类是一个内存管理器，它的作用是在程序运行时管理系统的内存分配和释放，以及进程的访问和操作。它包含了一些方法来创建进程、访问进程、释放进程和展示所有内存等功能，可以通过这些方法来控制和管理程序中的内存使用。

##### 数据结构(队列)

有一个队列类Queue及其两个子类FifoQueue和LruQueue，它们都是继承自Queue类。Queue类是一个基本的队列类，实现了队列的基本操作，如push、isEmpty、getQueue、size和getTail等。这些类主要用于管理系统中的页面帧，通过FIFO（先进先出）和LRU（最近最少使用）算法控制页面置换，并且记录页面帧的使用情况到文件中。

FifoQueue是先进先出（FIFO）队列，它重写了Queue类中的push方法，实现了一种保留固定大小的队列，并在队列已满时，将最老的元素删除，把新元素添加到队列头部的功能。同时，每次插入新元素时，将当前队列的内容写入一个文件中，记录内存中每个页面的使用情况。FifoQueue还向Memory类的pageFrameList添加了一个UsingFrameBar对象，用于在内存中绘制当前页面的使用情况。

LruQueue是最近最少使用（LRU）队列，它也重写了Queue类中的push方法，实现了保留固定大小的队列，当队列已满时，将最近最少使用的元素删除，并把新元素添加到队列头部的功能。与FifoQueue类似，每次插入新元素时，也将当前队列的内容写入一个文件中，并向Memory类的pageFrameList添加一个UsingFrameBar对象。

这两个队列类的作用是模拟操作系统中的内存管理。它们用于管理内存中的页面，FifoQueue使用FIFO算法，LruQueue使用LRU算法，根据不同的算法，管理内存中的页面。FifoQueue和LruQueue都继承自Queue类，因此可以复用Queue类中的代码和方法，避免了代码冗余。同时，它们都与Memory类和UsingFrameBar类有关联，可以向Memory类的pageFrameList添加新的页面，并为每个页面绘制UsingFrameBar对象，方便操作系统管理内存。

##### 页框监视

核心是PageInspectWin类，继承了Win类。PageInspectWin类是一个窗口界面，用于显示系统中当前的页框状态。在构造函数中，使用了JavaFX的控件来创建了一个表格视图，用于显示页框的信息，包括每个页框的编号和页框内存储的页面信息。页面信息是通过绑定UsingFrameBar类中的getPage()方法返回值来实现的。

在这段代码中，还可以看到使用了SystemCore.UsingFrameBar类。UsingFrameBar类是对Memory中每个页框的封装，包含了页框编号、页框内存储的页面信息等属性。此外，该类中还使用了JavaFX中的SimpleObjectProperty类和TableView类，用于显示表格视图。

### UI:

#### 1.Controller

##### 1.1功能逻辑

* 管理程序窗口的新建、隐藏和删除，当用户对程序窗口进行操作时，指令会传至Controller并由Controller来执行对窗口的新建或者移除。
* 管理UI与Kernel间的通信，Controller通过管道持续监测和接受来自Kernel的信息，UI只能直接向Controller发送指令，并由UI根据指令内容进行处理后通过管道传输给Kernel。

##### 1.2主要方法

* public void newWin(String name): 创建新窗口并添加到桌面。
* public void newText(Win textWin, String name): 创建新的文本窗口并添加到桌面。
* public void closeWin(String name): 关闭指定名称的窗口。
* public Boolean isOpen(String name): 判断指定名称的窗口是否已打开。
* public void setVisible(String name): 设置指定名称的窗口可见。
* public void communicate(String command): 向Shell发送命令并处理输出。
* public void run(): 实现Runnable接口的run方法，处理Shell输出的数据。

#### 2.Desktop

##### 2.1功能逻辑

* 新建程序窗口，左侧面板用于摆放基本系统图标，当用户点击左侧面板的系统图标后或是文件编辑框等窗口需要显示时，由Controller实例化程序窗口，Desktop再将实例化的窗口加入到中心面板中进行显示。
* 删除窗口，当要关闭某个程序窗口时，将该窗口从中心面板移除。

##### 2.2主要方法

* public Pane getBase(): 返回程序窗口基础面板。
* public void addWin(Win win): 添加窗口到中心面板。
* public void deleteWin(Win win, String name): 从中心面板删除指定窗口，并根据窗口名称重置应用程序图标样式。

#### 3.Win

##### 3.1功能逻辑

* 模拟程序窗口边框，提供窗口的最小化、关闭以及拖动功能。
* 提供窗口主界面，其他窗口在继承Win类后在主界面显示组件。

##### 3.2主要方法

* public Win(Controller controller, String name, int width, int height): 构造函数，创建Win对象，传入controller、窗口名称以及窗口尺寸。
* public Pane getPane(): 获取窗口的主界面。
* public String getName(): 获取窗口名称。

#### 4.TerminalWin

##### 4.1功能逻辑

* 用户在TerminalWin输入命令，TerminalWin会对输入的命令进行拆解并传入Controller进行进一步处理以及命令传输。
* 显示由Controller接收到的来自Kernel的信息。
* 提供历史指令的功能，用F1和F2按键来切换历史指令。

##### 4.2主要方法

* private void recordHistory(String command): 记录命令历史。
* protected static void setText(String text): 设置终端文本区域的文本内容。

#### 5.FileManagerWin

##### 5.1功能逻辑

* 文件系统的显示，文件管理器窗口通过监控一个存有文件基本信息的文件树实现对文件的显示和更新。
* 文件\文件夹的删除，当用户选择删除某一文件夹时，生成相应的rm命令，并通过Controller向Kernel发送命令。
* 文件\文件夹的新建，当用户选择新建文件\文件夹时，通知Controller生成一个命名窗口，当命名结束后通过Controller向Kernel发送相应的指令。
* 文件的编辑，当用户选择编辑某文件时，通知Controller生成一个编辑窗口并向Kernel传输相关指令，文件系统在读取了对应文件后通过管道将文件内容传送给文件编辑窗口。
* 可执行文件的运行，当用户选择执行某文件时，通过Controller向Kernel发送相关执行指令。

##### 5.2主要方法

* private void buildTree(ObservableMap<FileInfo, Object> map, FileTreeItem parentItem)：根据文件树内容，递归地构建文件表。
* public class FileTreeItem extends TreeItem：自定义的文件表节点类，继承自TreeItem。该类包含了文件信息FileInfo。
* private class TextWin extends Win：自定义的文本窗口类，继承自Win。用于创建命名窗口和编辑窗口。

#### 6.FileEditWin

##### 6.1功能逻辑

* 获取从文件系统传递的文件内容，在文件编辑窗口中进行显示。
* 保存编辑后的文件，窗口在创建时保存了对应文件的路径，在用户选择保存文件时通过Controller发送相应的保存命令。

##### 6.2主要方法

* setText(String text)：设置文本编辑框中的文本内容。

#### 7.TaskManager

##### 7.1功能逻辑

* 通过监控进程系统中的pcbList结构，将当前的进程信息实时显示在表格中。
* 通过监控内存系统中的usedRate变量，将当前内存的使用率实时显示在窗口中。
* 提供终止进程的按钮，当用户选择终止某一进程时，获取选定进程的pid，并通过Controller将相应的命令发送给Kernel。

##### 7.2主要方法

* public static void updateMemory(Float usedRate)：设置显示内存使用率的label对象。

#### 8.DeviceWin

##### 8.1功能逻辑

* 通过监控硬件管理系统的hardwares结构，将系统硬件的信息实时显示在表格中。

#### 9.PageInspectWin

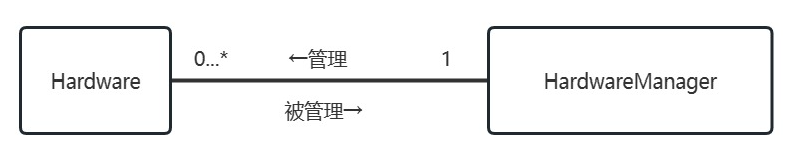
##### 9.1功能逻辑

* 通过监控内存系统的*pageFrameList*结构，将页框使用信息实时显示在表格中。

### HARDARE

#### 需求结构

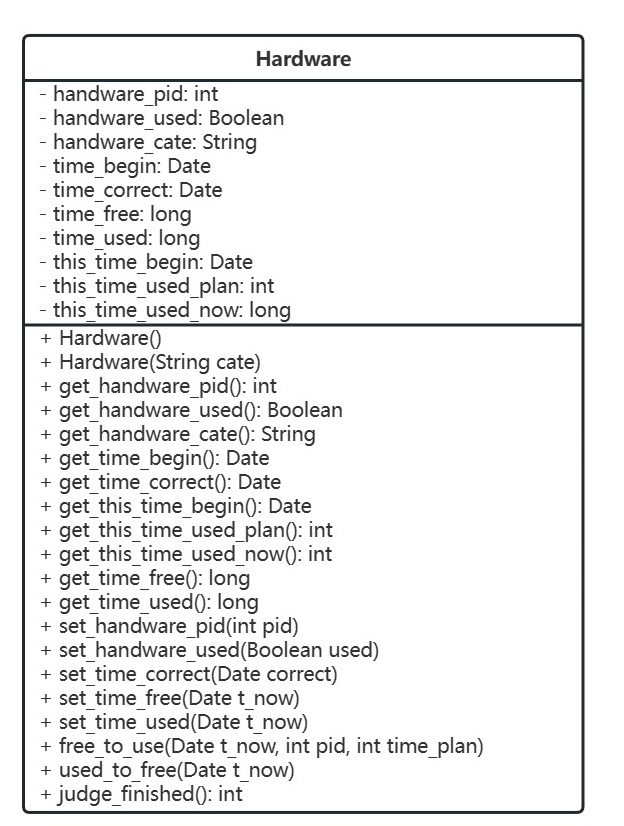
根据使用需求，构建两个类Hardware和HardwareManager，Hardware为单个硬件，HardwareManager为硬件管理器，控制所有硬件，其关系如下：



#### 结构设计

##### Hardware类

Hardware类图如下：



拥有属性解释如下：

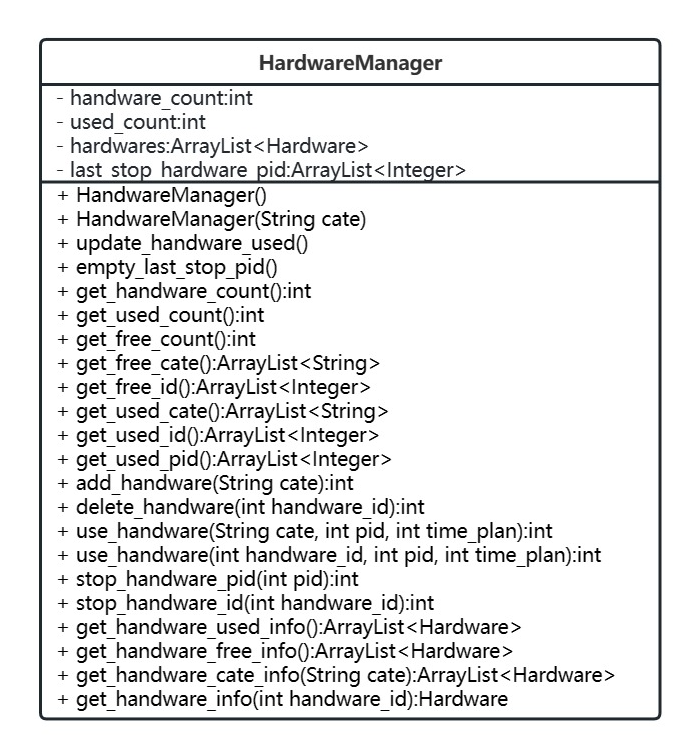
protected int handware\_pid;//记录硬件对应的进程号，如果目前硬件未被使用，设为-1  
 protected Boolean handware\_used;//记录目前硬件是否被使用，被使用为TRUE，未被使用为FALSE  
 protected String handware\_cate;//设备种类  
 protected Date time\_begin;//所有硬件安装的时间  
 protected Date time\_correct;//上次硬件的空闲/使用的时间被修改的时间  
 protected long time\_free;//硬件所有空闲的时间，单位ms，当被调用或者修改的时候进行更新  
 protected long time\_used;//硬件所有使用的时间，单位ms，当被调用或者修改的时候进行更新  
 protected Date this\_time\_begin;//本次硬件开始使用的时间  
 protected int this\_time\_used\_plan;//本次硬件计划被使用的时间，单位ms  
 protected long this\_time\_used\_now;//本次硬件已经被使用的时间，单位ms

提供操作解释如下：

* Hardware()：默认构造函数
* Hardware(String cate)：带参数的构造函数
* get\_handware\_pid()：获取硬件对应的进程号
* get\_handware\_used()：获取目前硬件是否被使用
* get\_handware\_cate()：获取硬件种类
* get\_time\_begin()：获取所有硬件安装的时间
* get\_time\_correct()：获取上次硬件的空闲/使用的时间被修改的时间
* get\_this\_time\_begin()：获取本次硬件开始使用的时间
* get\_this\_time\_used\_plan()：获取本次硬件计划被使用的时间
* get\_this\_time\_used\_now()：获取本次硬件已经被使用的时间
* set\_handware\_pid(int pid)：设置硬件对应的进程号
* set\_handware\_used(Boolean used)：设置目前硬件是否被使用
* set\_time\_correct(Date correct)：设置上次硬件的空闲/使用的时间被修改的时间
* set\_time\_free(Date t\_now)：设置硬件所有空闲的时间
* set\_time\_used(Date t\_now)：设置硬件所有使用的时间
* free\_to\_use(Date t\_now, int pid, int time\_plan)：设备从空闲到正在使用
* used\_to\_free(Date t\_now)：设备从正在使用到空闲
* judge\_finished()：判断该设备是否已经运行完成

##### HardwareManager类

Hardware类图如下：



拥有属性解释如下：

protected int handware\_count;//硬件总个数  
 protected int used\_count;//被使用的硬件个数  
 protected ArrayList<Hardware> hardwares;//所使用的硬件列表  
 protected ArrayList<Integer> last\_stop\_hardware\_pid;//这次循环停止的硬件对应的pid

提供操作解释如下：

* HandwareManager()：默认构造函数
* HandwareManager(String cate)：构造函数，指定第一个设备的种类
* update\_handware\_used()：更新设备的使用情况
* empty\_last\_stop\_pid()：清空上次停止的硬件对应的pid列表
* get\_handware\_count()：得到当前目前的硬件个数
* get\_used\_count()：得到正在被使用的硬件个数
* get\_free\_count()：得到正在空闲的硬件个数
* get\_free\_cate()：得到所有的现在未被使用的硬件设备的种类
* get\_free\_id()：得到所有的现在未被使用的硬件设备的设备编号
* get\_used\_cate()：得到所有的现在正在使用的硬件设备的种类
* get\_used\_id()：得到所有的现在正在被使用的硬件设备的设备编号
* get\_used\_pid()：得到所有的现在正在被使用的硬件设备的进程pid
* add\_handware(String cate)：添加某种类的硬件，并返回添加后的硬件个数
* delete\_handware(int handware\_id)：删除某硬件
* use\_handware(String cate, int pid, int time\_plan)：给某个进程pid使用某个种类的硬件，并传入预计使用的时间(单位ms)
* use\_handware(int handware\_id, int pid, int time\_plan)：给某个进程pid使用某个硬件编号的硬件，并传入预计使用的时间(单位ms)
* stop\_handware\_pid(int pid)：给某个进程pid对应的硬件停止执行
* stop\_handware\_id(int handware\_id)：给某个硬件停止执行
* get\_handware\_used\_info()：得到正在运行的硬件的相关信息
* get\_handware\_free\_info()：得到空闲的硬件的相关信息
* get\_handware\_cate\_info(String cate)：得到某个种类的硬件的相关信息
* get\_handware\_info(int handware\_id)：得到某个硬件的相关信息

#### 与其它模块的关系

进程模块在进行设备调用时，首先进行中断操作，之后使用设备管理相关方法进行设备的调用；前端通过调用设备管理相关方法，将设备相关信息展示到前端。

## 程序清单

## 7．程序清单

### FILE:

构建文件树与分配磁盘核心代码如下

private Map<String, Object> init\_file\_system\_tree(String now\_path) {
  
 // now\_path是当前递归到的绝对路径
  
 /\* 文件树采用Map形式，文件名为键，
  
 \* 当该文件为文件夹时，其值为一个Map，
  
 \* 否则，其值为长度为4的字符串，表示类型 / 读 / 写 / 执行。\*/
  
 File folder = new File(now\_path);
  
 File[] file\_list = folder.listFiles();
  
 Map<String, Object> part\_of\_tree = new HashMap<String, Object>(); // 当前文件夹对应的Map
  
 for (File file : file\_list) {
  
 String file\_path = file.getAbsolutePath();
  
 if (file.isDirectory()) { // 文件夹为键，其值为Map
  
 part\_of\_tree.put(file.getName(), init\_file\_system\_tree(file\_path));
  
 } else {
  
 try (BufferedReader reader = new BufferedReader(new FileReader(file\_path))) {
  
 StringBuilder stringBuilder = new StringBuilder();
  
 String line;
  
 while ((line = reader.readLine()) != null) {
  
 stringBuilder.append(line);
  
 }
  
 String fileContent = stringBuilder.toString();
  
 JSONObject data = JSON.parseObject(fileContent); // 读取json字符串，便于之后提取type
  
 part\_of\_tree.put(file.getName(), data.getString("type"));
  
 if (this.fill\_file\_into\_blocks(
  
 data, file\_path.substring(this.root\_path.length()), 0) == -1) { // 将此文件的信息存于外存块中
  
 // 没有足够的存储空间
  
 Diary.println("block storage error: No Enough Initial Space");
  
 }
  
 } catch (IOException e) {
  
 // 处理读取或解析 JSON 异常
  
 Diary.println("error: Json exception");
  
 }
  
 }
  
 }
  
 return part\_of\_tree;
  
 }
  
  
 private int fill\_file\_into\_blocks(JSONObject f, String fp, int method) {
  
 int num = Integer.parseInt(f.getString("size")) / block\_size;
  
 int occupy = Integer.parseInt(f.getString("size")) % block\_size;
  
 if (occupy == 0) {
  
 num = num - 1;//如果正好是磁块的整数个，则不需考虑多余的余数
  
 }
  
 int first\_free\_block = find\_free\_blocks(num + 1, method);
  
 if (first\_free\_block == -1) {
  
 return -1;
  
 }
  
 int free = block\_size - occupy;
  
 // block分配信息存在dir中
  
 block\_dir.put(fp, new int[]{first\_free\_block, num + 1, Integer.parseInt(f.getString("size"))});
  
  
 int count = first\_free\_block;
  
 for (int i = 0; i < num + 1; i++) {
  
 if (i == num) {
  
 // 最后一块可能有碎片
  
 all\_blocks.get(count).set\_free\_space(free);
  
 } else {
  
 //否则该block被全部占满
  
 all\_blocks.get(count).set\_free\_space(0);
  
 }
  
 bitmap[count] = 0;
  
 all\_blocks.get(count).set\_fp(fp);
  
 count++;
  
 }
  
 return 0;
  
 }

### PROCESS:

#### ProcessControlBlock

ProcessControlBlock拥有属性如下所示：

int pid; //进程标识号  
int parent\_pid;//父进程id，若没有父进程的话则设为-1  
ArrayList<Integer> child\_pid = new ArrayList<>();//子进程id  
Date create\_time = new Date();//创建时间  
String name;//进程名字  
String status;//状态，共5种，为new, ready, running, waiting, terminated  
int priority;//优先级，数值越大，优先级越高  
int size;//进程大小，需要为进程分配内存  
int pc;//程序计数器  
ArrayList<String> commend\_queue = new ArrayList<>();//执行队列(commend\_queue)：记录该进程仍需要执行的批处理指令  
int commend\_now;//此时运行到的指令号，初始为0

提供操作

ProcessControlBlock类提供如下操作：

void terminated();//该进程已完成，变为terminated状态  
void set\_pid(int pid);//更改pid  
void set\_name(String name);//更改name  
void set\_priority(int priority);//更改优先级  
void set\_size(int size)//更改size

#### ProcessManager

拥有属性

//ProcessManager初始化  
 public ProcessManager(Memory memory, HardwareManager hardwareManager) {  
 this.curPid = 0;  
 this.priority = true;  
 this.preemptive = false;  
 this.timeSlot = 1000;  
 //一个包含三个空列表的列表，表示三个不同优先级的就绪队列。  
 this.readyQueue = new ArrayList<>();  
 for (int i = 0; i < 3; i++) {  
 this.readyQueue.add(new ArrayList<>());  
 }  
 this.waitingQueue = new ArrayList<>();  
 //现在正在运行的内存的pid，初始没有置为-1  
 this.currentRunning = -1;  
 this.pcbList = FXCollections.observableArrayList();  
 //表示设备管理资源  
 this.hardwareManager = hardwareManager;  
 //设备列表，初始先默认加入cpu，打印机两个设备  
 this.devices = new ArrayList<>();  
 this.devices.add("cpu");  
 this.devices.add("printer");  
 //表示每个设备的历史记录  
 this.resourcesHistory = new HashMap<>();  
 this.resourcesHistory.put("cpu", new ArrayList<>());  
 this.resourcesHistory.put("printer", new ArrayList<>());  
 //时间戳  
 this.historyLength = 14;  
 //初始没有正在进行运行的进程置为false  
 this.running = false;  
 this.pidToAid = new HashMap<>();  
 this.memory = memory;  
 this.sock=0;  
 }

###### 提供方法

//进程内创建子进程  
public void fork(List<String> list)  
  
//系统自己的程序创建进程  
public void creat\_process(String name,String priority,String size,List<String> list)  
   
//进程调度算法  
public void schduler()  
   
//时间片管理  
public void time\_out()  
   
//中断管理  
public void io\_interrput()  
   
//释放资源  
public void release(int pid)  
   
//杀死进程  
public void killProcess(int pid)  
   
//显示进程状态  
public void process\_status()  
   
//记录设备使用历史  
public void append\_resources\_hestory(String type ,int pid)  
   
//启动进程管理器  
public void run()

### KERNEL:

通过解析一段指令的不同位置来调用不同的函数

private static void deal\_pc(String word) throws IOException{
  
 String[] words = word.split(" ");
  
 if(words.length==2 &&(words[1].equals("LRU")||words[1].equals("FIFO"))){
  
 fileManager.change\_page(words[1]);
  
  
 }
  
 else{
  
 try {
  
 write("pc:错误");
  
 } catch (IOException e) {
  
 e.printStackTrace();
  
 }
  
 }
  
 }

调度函数有:

if(words[0].equals("ls")){
  
 deal\_ls(word);
  
  
  
 }
  
 else if(words[0].equals("cd")){
  
 deal\_cd(word);
  
 }
  
 else if(words[0].equals("mkdir")){
  
 deal\_mkdir(word);
  
 }
  
  
 else if(words[0].equals("rm")){
  
 deal\_rm(word);
  
 }
  
 else if(words[0].equals("dss")){
  
 deal\_dss(word);
  
 }
  
 else if(words[0].equals("exec")){
  
 deal\_exec(word);
  
 }
  
 else if(words[0].equals("dms")){
  
 deal\_dms(word);
  
 }
  
  
 else if(words[0].equals("td")){
  
 deal\_td(word);
  
 }
  
 else if(words[0].equals("mkf")){
  
 deal\_mkf(word);
  
 }
  
 else if(words[0].equals("kill")){
  
 deal\_kill(word);
  
 }
  
 else if(words[0].equals("ps")){
  
 deal\_ps(word);
  
 }
  
 else if(words[0].equals("rs")){
  
 deal\_rs(word);
  
 }
  
 else if(words[0].equals("man")){
  
 deal\_man(word);
  
 }
  
 else if(words[0].equals("re")){
  
 deal\_re(word);
  
 }
  
 else if (words[0].equals("vi")){
  
 deal\_vi(word);
  
 }
  
 else if (words[0].equals("gcc")){
  
 deal\_gcc(word);
  
 }
  
 else if(words[0].equals("sv")){
  
 deal\_sv(word);
  
 }
  
 else if(words[0].equals("jr")){
  
 deal\_jr(word);
  
 }
  
 else if (words[0].equals("mc")){
  
 deal\_mc(word);
  
 }
  
 else if (words[0].equals("ar")){
  
 deal\_ar(word);
  
  
 }
  
 else if (words[0].equals("pc")){
  
 deal\_pc(word);
  
  
 }
  
  
 else{
  
 write("命令不存在");
  
  
 }

### UI:

#### 程序清单

构建文件树与分配磁盘核心代码如下

private Map<String, Object> init\_file\_system\_tree(String now\_path) {
  
 // now\_path是当前递归到的绝对路径
  
 /\* 文件树采用Map形式，文件名为键，
  
 \* 当该文件为文件夹时，其值为一个Map，
  
 \* 否则，其值为长度为4的字符串，表示类型 / 读 / 写 / 执行。\*/
  
 File folder = new File(now\_path);
  
 File[] file\_list = folder.listFiles();
  
 Map<String, Object> part\_of\_tree = new HashMap<String, Object>(); // 当前文件夹对应的Map
  
 for (File file : file\_list) {
  
 String file\_path = file.getAbsolutePath();
  
 if (file.isDirectory()) { // 文件夹为键，其值为Map
  
 part\_of\_tree.put(file.getName(), init\_file\_system\_tree(file\_path));
  
 } else {
  
 try (BufferedReader reader = new BufferedReader(new FileReader(file\_path))) {
  
 StringBuilder stringBuilder = new StringBuilder();
  
 String line;
  
 while ((line = reader.readLine()) != null) {
  
 stringBuilder.append(line);
  
 }
  
 String fileContent = stringBuilder.toString();
  
 JSONObject data = JSON.parseObject(fileContent); // 读取json字符串，便于之后提取type
  
 part\_of\_tree.put(file.getName(), data.getString("type"));
  
 if (this.fill\_file\_into\_blocks(
  
 data, file\_path.substring(this.root\_path.length()), 0) == -1) { // 将此文件的信息存于外存块中
  
 // 没有足够的存储空间
  
 Diary.println("block storage error: No Enough Initial Space");
  
 }
  
 } catch (IOException e) {
  
 // 处理读取或解析 JSON 异常
  
 Diary.println("error: Json exception");
  
 }
  
 }
  
 }
  
 return part\_of\_tree;
  
 }
  
  
 private int fill\_file\_into\_blocks(JSONObject f, String fp, int method) {
  
 int num = Integer.parseInt(f.getString("size")) / block\_size;
  
 int occupy = Integer.parseInt(f.getString("size")) % block\_size;
  
 if (occupy == 0) {
  
 num = num - 1;//如果正好是磁块的整数个，则不需考虑多余的余数
  
 }
  
 int first\_free\_block = find\_free\_blocks(num + 1, method);
  
 if (first\_free\_block == -1) {
  
 return -1;
  
 }
  
 int free = block\_size - occupy;
  
 // block分配信息存在dir中
  
 block\_dir.put(fp, new int[]{first\_free\_block, num + 1, Integer.parseInt(f.getString("size"))});
  
  
 int count = first\_free\_block;
  
 for (int i = 0; i < num + 1; i++) {
  
 if (i == num) {
  
 // 最后一块可能有碎片
  
 all\_blocks.get(count).set\_free\_space(free);
  
 } else {
  
 //否则该block被全部占满
  
 all\_blocks.get(count).set\_free\_space(0);
  
 }
  
 bitmap[count] = 0;
  
 all\_blocks.get(count).set\_fp(fp);
  
 count++;
  
 }
  
 return 0;
  
 }

### HARDWARE:

Hardware类部分代码（全部代码详见源程序，此处只列举关键代码）：

public class Hardware {  
 protected int handware\_pid;//记录硬件对应的进程号，如果目前硬件未被使用，设为-1  
 protected Boolean handware\_used;//记录目前硬件是否被使用，被使用为TRUE，未被使用为FALSE  
 protected String handware\_cate;//设备种类  
 protected Date time\_begin;//所有硬件安装的时间  
 protected Date time\_correct;//上次硬件的空闲/使用的时间被修改的时间  
 protected long time\_free;//硬件所有空闲的时间，单位ms，当被调用或者修改的时候进行更新  
 protected long time\_used;//硬件所有使用的时间，单位ms，当被调用或者修改的时候进行更新  
 protected Date this\_time\_begin;//本次硬件开始使用的时间  
 protected int this\_time\_used\_plan;//本次硬件计划被使用的时间，单位ms  
 protected long this\_time\_used\_now;//本次硬件已经被使用的时间，单位ms  
  
 //设备从空闲到正在使用  
 public void free\_to\_use(Date t\_now, int pid, int time\_plan)  
 {  
 this\_time\_used\_plan = time\_plan;  
 this\_time\_begin = t\_now;  
 this\_time\_used\_now = 0;  
  
 handware\_pid = pid;  
 handware\_used = true;  
 long time\_diff = t\_now.getTime() - time\_correct.getTime();  
 time\_free += time\_diff;  
 time\_correct = t\_now;  
 }  
  
 //设备从正在使用到空闲  
 public void used\_to\_free(Date t\_now)  
 {  
 this\_time\_used\_now = 0;  
 this\_time\_used\_plan = 0;  
  
 handware\_pid = -1;  
 handware\_used = false;  
 long time\_diff = t\_now.getTime() - time\_correct.getTime();  
 time\_used += time\_diff;  
 time\_correct = t\_now;  
 }  
  
 //判断该设备是否已经运行完成，若运行完成，返回1，该设备停止；否则返回-1，表示还未运行完成  
 public int judge\_finished()  
 {  
 int plan\_time = get\_this\_time\_used\_plan();  
 int used\_time = get\_this\_time\_used\_now();  
 if(plan\_time > used\_time)//预计时间大于已经运行的时间，则还没有完成  
 return -1;  
   
 return 1;  
 }  
}

HardwareManager类部分代码（全部代码详见源程序，此处只列举关键代码）：

public class HandwareManager {
  
 protected int handware\_count;//硬件总个数
  
 protected int used\_count;//被使用的硬件个数
  
 protected ArrayList<Hardware> hardwares;//所使用的硬件列表
  
 protected ArrayList<Integer> last\_stop\_hardware\_pid;//这次循环停止的硬件对应的pid
  
  
 //更新设备的使用情况
  
 public void update\_handware\_used()
  
 {
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 //judge\_finished返回1，该设备停止；否则返回-1，表示还未运行完成
  
 if(hardwares.get(i).get\_handware\_used() == true && hardwares.get(i).judge\_finished() == 1)
  
 {
  
 //把pid记下来
  
 int stop\_pid = hardwares.get(i).get\_handware\_pid();
  
 last\_stop\_hardware\_pid.add(stop\_pid);
  
 //把这个硬件变为空闲状态
  
 Date t\_now = new Date();
  
 hardwares.get(i).used\_to\_free(t\_now);
  
 used\_count--;
  
 }
  
 }
  
 }
  
  
 //得到所有的现在未被使用的硬件设备的种类
  
 public ArrayList<String> get\_free\_cate()
  
 {
  
 update\_handware\_used();//更新目前设备的使用情况
  
 ArrayList<String> free\_cate = new ArrayList<>();
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 if(hardwares.get(i).handware\_used == false && !free\_cate.contains(hardwares.get(i).handware\_cate))
  
 {
  
 free\_cate.add(hardwares.get(i).handware\_cate);
  
 }
  
 }
  
 return free\_cate;
  
 }
  
  
 //得到所有的现在未被使用的硬件设备的设备编号
  
 public ArrayList<Integer> get\_free\_id()
  
 {
  
 update\_handware\_used();//更新目前设备的使用情况
  
 ArrayList<Integer> free\_id = new ArrayList<>();
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 if(hardwares.get(i).handware\_used == false)
  
 {
  
 free\_id.add(i);
  
 }
  
 }
  
 return free\_id;
  
 }
  
  
 //得到所有的现在正在使用的硬件设备的种类
  
 public ArrayList<String> get\_used\_cate()
  
 {
  
 update\_handware\_used();//更新目前设备的使用情况
  
 ArrayList<String> free\_cate = new ArrayList<>();
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 if(hardwares.get(i).handware\_used == true && !free\_cate.contains(hardwares.get(i).handware\_cate))
  
 {
  
 free\_cate.add(hardwares.get(i).handware\_cate);
  
 }
  
 }
  
 return free\_cate;
  
 }
  
  
 //得到所有的现在正在被使用的硬件设备的设备编号
  
 public ArrayList<Integer> get\_used\_id()
  
 {
  
 update\_handware\_used();//更新目前设备的使用情况
  
 ArrayList<Integer> free\_id = new ArrayList<>();
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 if(hardwares.get(i).handware\_used == true)
  
 {
  
 free\_id.add(i);
  
 }
  
 }
  
 return free\_id;
  
 }
  
  
 //得到所有的现在正在被使用的硬件设备的进程pid
  
 public ArrayList<Integer> get\_used\_pid()
  
 {
  
 update\_handware\_used();//更新目前设备的使用情况
  
 ArrayList<Integer> free\_id = new ArrayList<>();
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 if(hardwares.get(i).handware\_used == true)
  
 {
  
 free\_id.add(hardwares.get(i).get\_handware\_pid());
  
 }
  
 }
  
 return free\_id;
  
 }
  
  
 //添加某种类的硬件，并返回添加后的硬件个数
  
 public int add\_handware(String cate)
  
 {
  
 handware\_count++;
  
 Hardware hardware = new Hardware(cate);
  
 hardwares.add(hardware);
  
 return handware\_count;
  
 }
  
  
 //删除某硬件，若成功，返回删除后的硬件个数（从0开始）；
  
 //若失败（若该硬件还有任务，即对应handware\_used的值为TRUE），返回-1
  
 //若失败（没有该硬件编号），返回-2
  
 //注意handware\_id是从0开始的
  
 public int delete\_handware(int handware\_id)
  
 {
  
 if(handware\_id + 1 > handware\_count)
  
 return -2;
  
 if(hardwares.get(handware\_id).get\_handware\_used() == true)
  
 return -1;
  
  
 handware\_count--;
  
 hardwares.remove(handware\_id);
  
 return handware\_count;
  
 }
  
  
 //给某个进程pid使用某个种类的硬件，并传入预计使用的时间(单位ms)，如果该硬件被其它进程占用，则使用失败，返回-1，否则使用成功返回1
  
 public int use\_handware(String cate, int pid, int time\_plan)
  
 {
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 if(hardwares.get(i).handware\_used == false && hardwares.get(i).handware\_cate == cate)
  
 {
  
 Date t\_now = new Date();
  
 hardwares.get(i).free\_to\_use(t\_now, pid, time\_plan);
  
 used\_count++;
  
 return 1;
  
 }
  
 }
  
  
 return -1;
  
 }
  
  
 //给某个进程pid使用某个硬件编号的硬件，并传入预计使用的时间(单位ms)，若失败（没有该硬件编号），返回-2;如果该硬件被其它进程占用，则使用失败，返回-1，否则使用成功返回1
  
 public int use\_handware(int handware\_id, int pid, int time\_plan)
  
 {
  
 if(handware\_id + 1 > handware\_count)
  
 return -2;
  
 if(hardwares.get(handware\_id).handware\_used == true)
  
 return -1;
  
  
 Date t\_now = new Date();
  
 hardwares.get(handware\_id).free\_to\_use(t\_now, pid, time\_plan);
  
 used\_count++;
  
 return 1;
  
 }
  
  
 //给某个进程pid对应的硬件停止执行，若该进程有对应的硬件返回1，停止执行成功，返回1，否则返回-1
  
 public int stop\_handware\_pid(int pid)
  
 {
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 if(hardwares.get(i).handware\_pid == pid)
  
 {
  
 Date t\_now = new Date();
  
 hardwares.get(i).used\_to\_free(t\_now);
  
 used\_count--;
  
 return 1;
  
 }
  
 }
  
  
 return -1;
  
 }
  
  
 //给某个硬件停止执行，若有对应的硬件返回1，停止执行成功，返回1，否则返回-1
  
 public int stop\_handware\_id(int handware\_id)
  
 {
  
 if(handware\_id + 1 > handware\_count)
  
 return -1;
  
  
 Date t\_now = new Date();
  
 hardwares.get(handware\_id).used\_to\_free(t\_now);
  
 used\_count--;
  
 return 1;
  
 }
  
  
 //得到正在运行的硬件的相关信息
  
 public ArrayList<Hardware> get\_handware\_used\_info()
  
 {
  
 ArrayList<Hardware> hardwares\_used = new ArrayList<>();//所使用的硬件列表
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 if(hardwares.get(i).handware\_used == true)
  
 {
  
 hardwares\_used.add(hardwares.get(i));
  
 }
  
 }
  
 return hardwares\_used;
  
 }
  
  
 //得到空闲的硬件的相关信息
  
 public ArrayList<Hardware> get\_handware\_free\_info()
  
 {
  
 ArrayList<Hardware> hardwares\_free = new ArrayList<>();//空闲的硬件列表
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 if(hardwares.get(i).handware\_used == false)
  
 {
  
 hardwares\_free.add(hardwares.get(i));
  
 }
  
 }
  
 return hardwares\_free;
  
 }
  
  
 //得到某个种类的硬件的相关信息
  
 public ArrayList<Hardware> get\_handware\_cate\_info(String cate)
  
 {
  
 ArrayList<Hardware> hardwares\_free = new ArrayList<>();//空闲的硬件列表
  
 for(int i = 0; i < handware\_count; i++)
  
 {
  
 if(hardwares.get(i).handware\_cate == cate)
  
 {
  
 hardwares\_free.add(hardwares.get(i));
  
 }
  
 }
  
 return hardwares\_free;
  
 }
  
  
}

### MEMORY:

ConMemoryManager

拥有属性：

// freeList为空闲内存块链表，allocatedList为已分配内存块链表  
// memorySize为内存总大小，freeSize为空闲内存大小  
// usedSize为已分配内存大小  
// usedRate为内存利用率  
// ProcessMap为进程pid与进程的映射  
// accessTimes为访问次数，accessSuccessTimes为访问成功次数  
private List<MemoryBlock> freeList;  
private List<MemoryBlock> allocatedList;  
private int memorySize;  
private HashMap<Integer, ConProcess> ProcessMap = new HashMap<Integer, ConProcess>();  
private int freeSize;  
private int usedSize = 0;  
private float usedRate;  
static int accessTimes = 0;  
private int accessSuccessTimes = 0;

提供方法：

// 进程申请内存  
public boolean allocate(ConProcess p)  
// 进程释放内存  
public void deallocate(ConProcess p)  
// 找到相邻的内存块  
private List<MemoryBlock> getAdjacentBlocks(MemoryBlock block)  
// 合并相邻的内存块  
private MemoryBlock mergeBlocks(MemoryBlock block, List<MemoryBlock> adjacentBlocks)  
// 得到内存利用率  
public List<String> getFreeList()  
// 得到空闲内存的List  
public List<String> getFreeList()  
// 得到已分配内存的List  
public List<String> getAllocatedList()  
// 得到内存使用情况  
public List<String> getUseInformation()  
// 创建进程  
public boolean createProcess(int pid,String name,int size)  
// 进程访问内存  
public int accessProcess(int pid, int address)  
// 获得进程  
public ConProcess getProcess(int pid)

PageMemoryManager

拥有属性：

// Memory：里面包括物理内存与虚拟内存  
// pageFrameSize：页框大小，表示一个页框的大小。  
// pageFrameNum：页框数量，表示物理内存中的页框数量。  
// virtualMemoryNum：虚拟内存数量，表示虚拟内存的页数。  
// pageTableSize：页表大小，表示一个页目录中的页表大小。  
// accessTimes：访问次数，表示访问的总次数，包括成功和失败的。  
// pageTableNum：页表数量，表示总页表数量。  
// pageFault：缺页次数，表示缺页的次数。  
// pageReplace：页面置换次数，表示页面置换的次数。  
// pageReplaceAlgorithm：页面置换算法，表示页面置换的算法。  
// pageTable：页表，表示页表。  
// fifoQueue：FIFO队列，表示FIFO队列。  
// lruQueue：LRU队列，表示LRU队列。  
// usedRate：内存利用率，表示内存利用率。  
// processMap：进程映射，表示进程映射。  
// freePageFrameNum：空闲页框数量，表示空闲页框的数量。  
// freePageNum：空闲页数量，表示空闲页的数量。  
private MemoryPage memory;  
private int pageFrameSize;  
private int pageFrameNum;  
private int pageTableSize;  
static int accessTimes = 0;  
private int pageTableNum;  
private int pageFault = 0;  
private int pageReplace = 0;  
private int freePageFrameNum;  
private int freePageNum;  
String pageReplaceAlgorithm;  
private HashMap<Integer, PageProcess> processMap = new HashMap<Integer, PageProcess>();  
private PageTable pageTable;  
private FifoQueue fifoQueue;  
private LruQueue lruQueue;  
private Float usedRate;

提供方法：

// 创建进程  
public boolean createProcess  
// 分配内存(策略)  
public boolean allocateMemory(PageProcess process)  
// 访问进程  
public int accessProcess(PageProcess process, int address)  
// 获取内存使用情况  
public List<String> getBriefUsage()  
// 删除进程，并释放对应的内存  
public boolean deleteProcess(int pid)

## 8．测试报告

### HARDWARE:

#### 单一模块测试

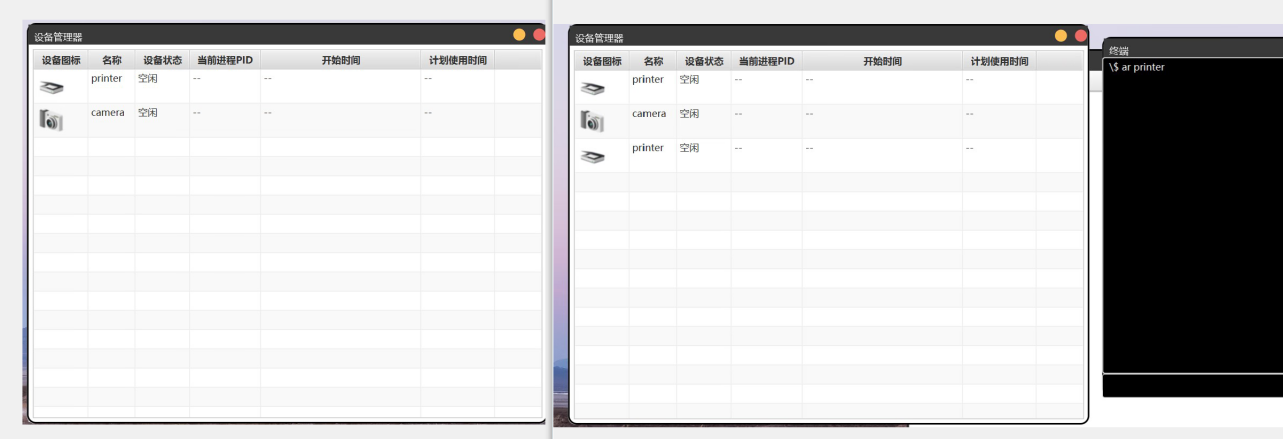
如下，通过进行相关指令，并输出相关信息，可以看出，单一模块测试成功。

测试说明及输出如下：

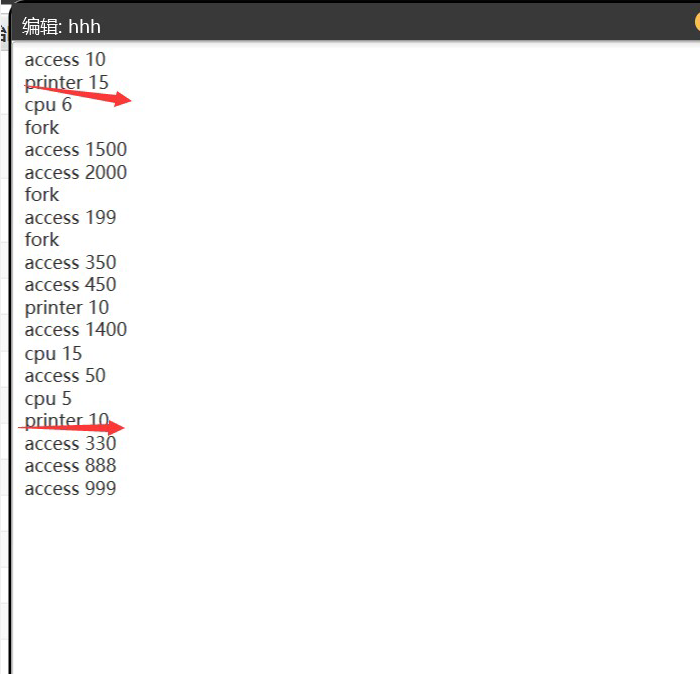
new一个Hardware类，应该内置一台打印机  
------------------  
id:0  
pid:-1  
used:false   
cate:printer   
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:0  
time\_used:0  
this\_time\_used\_plan:0  
this\_time\_used\_now:0  
------------------  
进程1使用打印机Printer  
1  
------------------  
id:0  
pid:1  
used:true  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023   
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:67  
time\_used:0  
this\_time\_used\_plan:1000  
this\_time\_used\_now:0  
------------------  
total: 1  
used: 1  
free: 0  
------------------  
id:0  
pid:1  
used:true  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:67  
time\_used:3  
this\_time\_used\_plan:1000  
this\_time\_used\_now:3  
------------------  
进程2使用打印机Printer，但因为此时进程1没使用完，所以应该使用失败  
-1  
total: 1  
used: 1  
free: 0  
------------------  
id:0  
pid:1  
used:true  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:67  
time\_used:7  
this\_time\_used\_plan:1000  
this\_time\_used\_now:7  
------------------  
添加一台打印机  
2  
total: 2  
used: 1  
free: 1  
------------------  
id:0  
pid:1  
used:true  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:67  
time\_used:10  
this\_time\_used\_plan:1000  
this\_time\_used\_now:10  
------------------  
------------------  
id:1  
pid:0  
used:false  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:0  
time\_used:0  
this\_time\_used\_plan:0  
this\_time\_used\_now:0  
------------------  
进程2使用打印机，应该使用成功  
1  
total: 2  
used: 2  
free: 0  
------------------  
id:0  
pid:1  
used:true  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:67  
time\_used:14  
this\_time\_used\_plan:1000  
this\_time\_used\_now:14  
------------------  
------------------  
id:1  
pid:2  
used:true  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:5  
time\_used:0  
this\_time\_used\_plan:1000  
this\_time\_used\_now:0  
------------------  
停止进程2  
1  
total: 2  
used: 1  
free: 1  
------------------  
id:0  
pid:1  
used:true  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:67  
time\_used:20  
this\_time\_used\_plan:1000  
this\_time\_used\_now:20  
------------------  
------------------  
id:1  
pid:-1  
used:false  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:5  
time\_used:5  
this\_time\_used\_plan:0  
this\_time\_used\_now:0  
------------------  
输出现在空闲的种类，应该是打印机  
[printer]  
total: 2  
used: 1  
free: 1  
------------------  
id:0  
pid:1  
used:true  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:67  
time\_used:25  
this\_time\_used\_plan:1000  
this\_time\_used\_now:25  
------------------  
------------------  
id:1  
pid:-1  
used:false  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:5  
time\_used:5  
this\_time\_used\_plan:0  
this\_time\_used\_now:0  
------------------  
添加一个摄像头  
3  
total: 3  
used: 1  
free: 2  
------------------  
id:0  
pid:1  
used:true  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:67  
time\_used:31  
this\_time\_used\_plan:1000  
this\_time\_used\_now:31  
------------------  
------------------  
id:1  
pid:-1  
used:false  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:5  
time\_used:5  
this\_time\_used\_plan:0  
this\_time\_used\_now:0  
------------------  
------------------  
id:2  
pid:0  
used:false  
cate:carema  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:0  
time\_used:0  
this\_time\_used\_plan:0  
this\_time\_used\_now:0  
------------------  
输出现在空闲的种类，应该是打印机和摄像头  
[printer, carema]  
total: 3  
used: 1  
free: 2  
------------------  
id:0  
pid:1  
used:true  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:67  
time\_used:39  
this\_time\_used\_plan:1000  
this\_time\_used\_now:39  
------------------  
------------------  
id:1  
pid:-1  
used:false  
cate:printer  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:5  
time\_used:5  
this\_time\_used\_plan:0  
this\_time\_used\_now:0  
------------------  
------------------  
id:2  
pid:0  
used:false  
cate:carema  
time\_begin:Wed May 10 21:34:14 CST 2023  
time\_correct:Wed May 10 21:34:14 CST 2023  
time\_free:0  
time\_used:0  
this\_time\_used\_plan:0  
this\_time\_used\_now:0

#### 集成到系统中的测试

添加一个设备，原本有一个打印机和一个摄像机，使用ar printer指令，添加一台打印机，左边添加前，后面添加后：

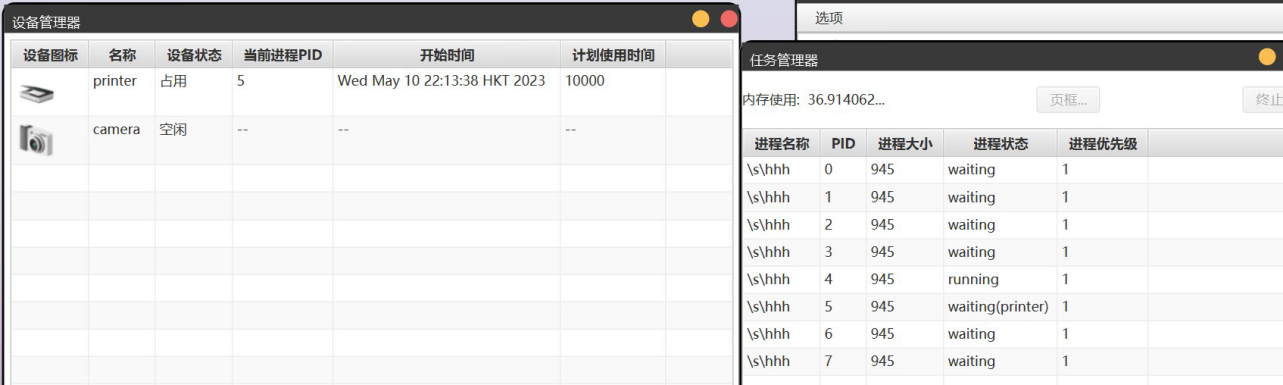


之后测试打印机使用，使用命令集合如下，标红的是与硬件有关的代码，其中注意，这里后面的15是打印机使用时间，单位是秒，而后面设备管理器显示的计划使用时间单位是ms：



当运行到相关代码时，会显示打印机从空闲到占用，如下，上面是空闲时，下面是运行时：





### FILE:

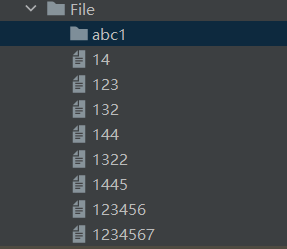
有如下的代码进行测试，在构建不同的文件树情况下，调用如下方法测试执行情况。

public void test\_bitmap() {  
 for (int i = 0; i < bitmap.length; i++)  
 System.out.print(bitmap[i] + " ");  
 System.out.println();  
 for (int[] value : block\_dir.values()) {  
 for (int i = 0; i < value.length; i++) {  
 System.out.print(value[i] + " ");  
 }  
 }  
 }  
  
 public void test\_filetree() {  
 System.out.println(file\_system\_tree);  
 }  
  
 public void test\_blocks() {  
 System.out.println("当前block\_dir为：");  
 for (Map.Entry<String, int[]> entry : block\_dir.entrySet()) {  
 System.out.print(entry.getKey() + ":");  
 for (int i : entry.getValue()) {  
 System.out.print(i + " ");  
 }  
 System.out.println();  
 }  
 System.out.println("当前all\_blocks为：");  
 for (Block block : all\_blocks) {  
 System.out.print(block.get\_fp() + ": ");  
 for (int i : block.get\_loc()) {  
 System.out.print(i + " ");  
 }  
 System.out.println();  
 }  
 }  
  
 public void test\_fp2loc(String fp) {  
 List<int[]> temp = fp2loc(fp);  
 for (int[] t : temp) {  
 for (int i : t) {  
 System.out.print(i + " ");  
 }  
 System.out.println();  
 }  
 }

例如，对于block的测试，有如下结果

当前block\_dir为：  
\test\abc.json:0 1 37   
\test\_dir1\test\_dir2\test\_dir3\test3.json:4 2 1088   
\test\_dir1\test\_dir2\test1.json:2 1 1024   
\test2.json:1 1 888   
当前all\_blocks为：  
\test\abc.json: 0 0   
\test2.json: 0 1   
\test\_dir1\test\_dir2\test1.json: 0 2   
null: 0 3   
\test\_dir1\test\_dir2\test\_dir3\test3.json: 0 4   
\test\_dir1\test\_dir2\test\_dir3\test3.json: 1 0   
null: 1 1   
null: 1 2   
null: 1 3   
null: 1 4   
null: 2 0   
null: 2 1   
null: 2 2   
null: 2 3   
null: 2 4   
null: 3 0   
null: 3 1   
null: 3 2   
null: 3 3   
null: 3 4   
null: 4 0   
null: 4 1   
null: 4 2   
null: 4 3   
null: 4 4

#### 测试用例



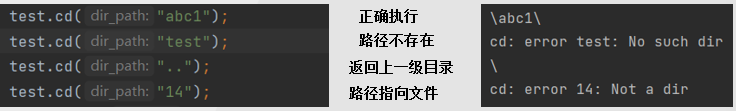
public class Test {  
 public static void main(String[] args) {  
  
 FileManager test = new FileManager(1024,5,5);  
 test.ls(null,null,"print");  
 test.ls(null,"-l","print");  
 test.cd("abc1");  
 test.cd("test");  
 test.cd("..");  
 test.cd("14");  
 test.mkdir("abc1\\test");  
 test.cd("abc1\\test");  
 test.mkf("test1","crwx","10");  
 System.out.println(test.vi("test1"));  
 System.out.println(test.vi("test1"));  
 test.cd("");  
 test.rm("abc1\\test","-rf");  
 for(String content: test.readContentFromFile("1234567"))  
 System.out.println(content);  
 System.out.println(test.vi("123456"));  
  
 int[] requests = {139, 117, 59, 173, 60, 57, 20, 5, 74};  
 DiskScheduler diskScheduler = new DiskScheduler(40, requests);  
 int totalDistance = diskScheduler.FIFO();  
 System.out.println("FIFO策略下磁盘臂移动距离为：" + totalDistance);  
 System.out.println("SSTF策略下磁盘臂移动距离为：" + diskScheduler.SSTF());  
 System.out.println("LOOK调度算法下的磁盘移动距离为：" + diskScheduler.LOOK());  
 System.out.println("C-LOOK调度算法下的磁盘移动距离为：" + diskScheduler.CLOOK());  
 System.out.println("SCAN调度算法下的磁盘移动距离为：" + diskScheduler.SCAN());  
 System.out.println("CSCAN调度算法下的磁盘移动距离为：" + diskScheduler.CSCAN());  
 }  
}

#### 测试结果

ls函数测试结果

[132, 144, 123, abc1, 14, 1445, 123456, 1322, 1234567]  
[crwx 132, crwx 144, crwx 123, d--- abc1, crwx 14, cr-- 1445, cr-- 123456, cr-- 1322, crwx 1234567]

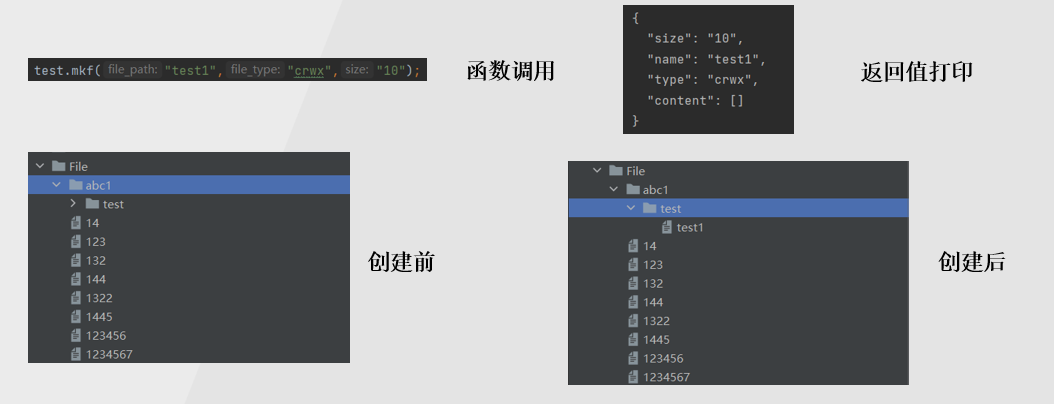
cd函数测试结果



mkdir函数测试结果



mkf函数测试结果



vi函数测试结果



rm测试结果



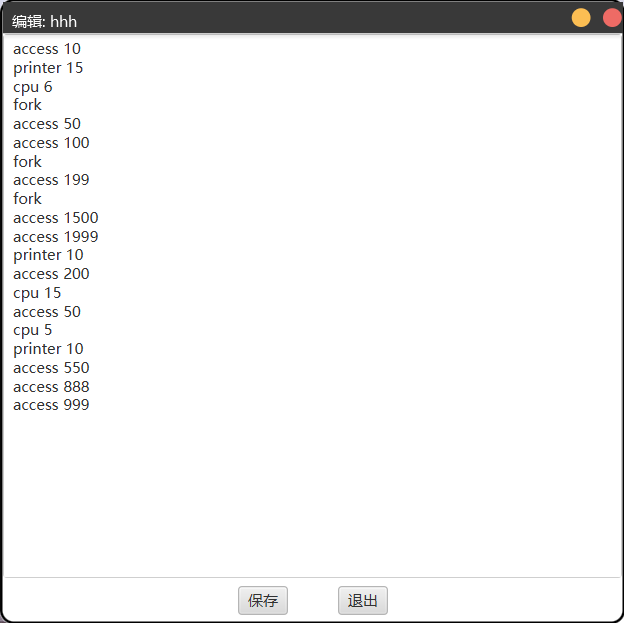
磁盘调度测试结果

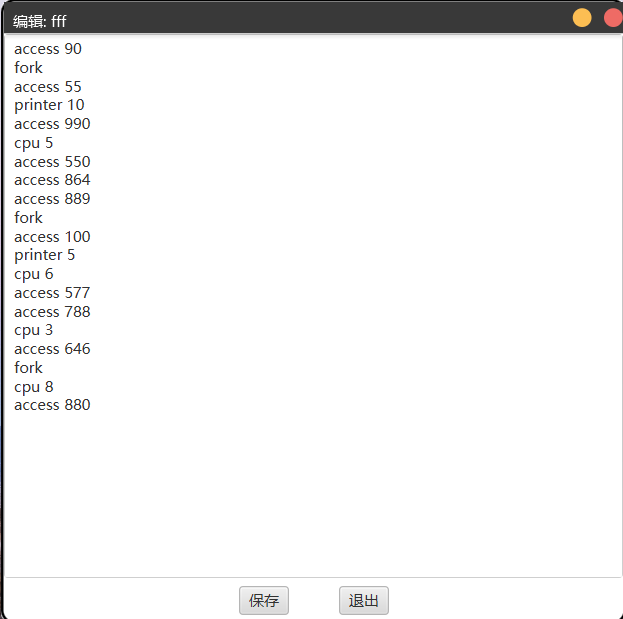
FIFO策略下磁盘臂移动距离为：530  
SSTF策略下磁盘臂移动距离为：301  
LOOK调度算法下的磁盘移动距离为：301  
C-LOOK调度算法下的磁盘移动距离为：316  
SCAN调度算法下的磁盘移动距离为：353  
CSCAN调度算法下的磁盘移动距离为：378

### PROCESS:

**测试功能包括进程调度、进程管理、进程模块与其他模块交互。**

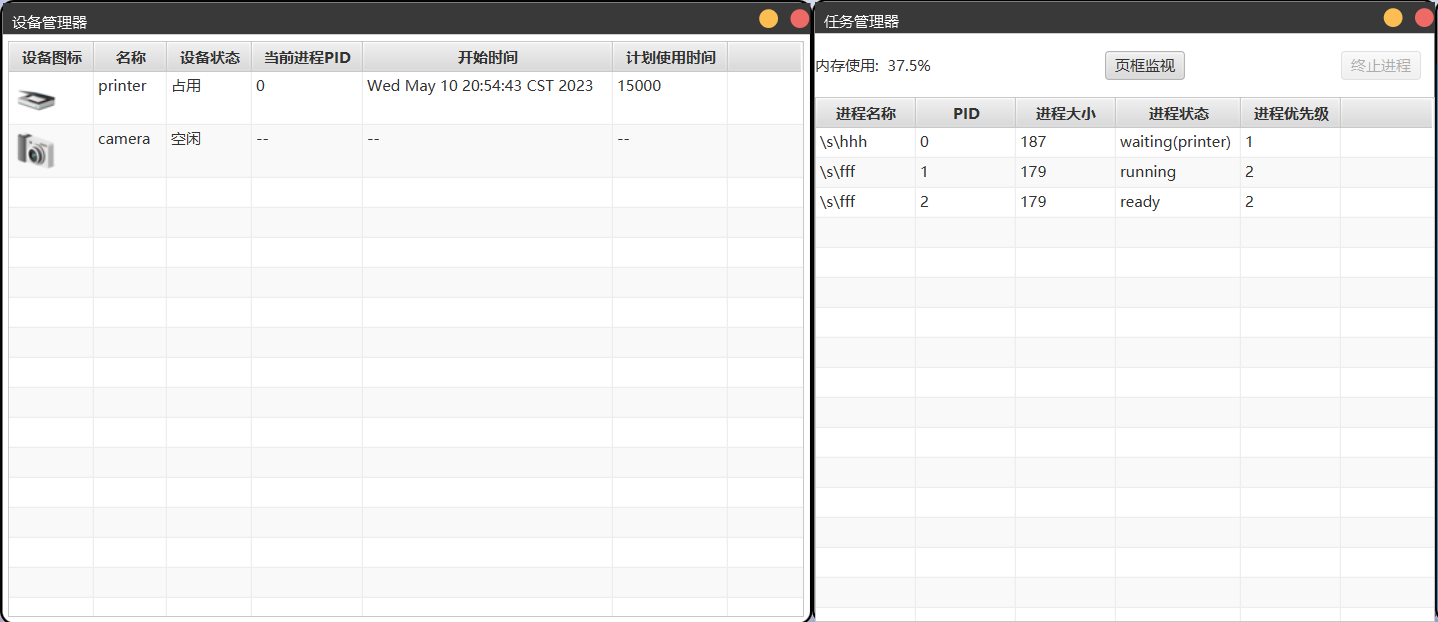
**测试用例：**

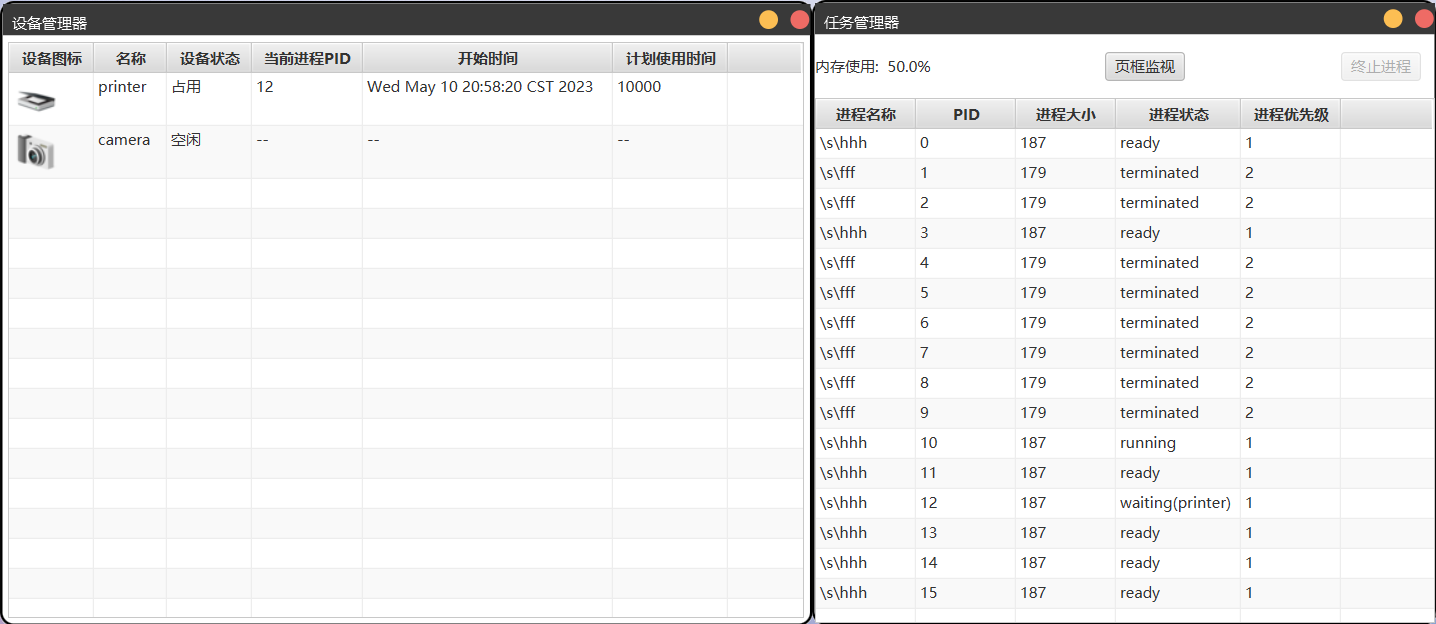


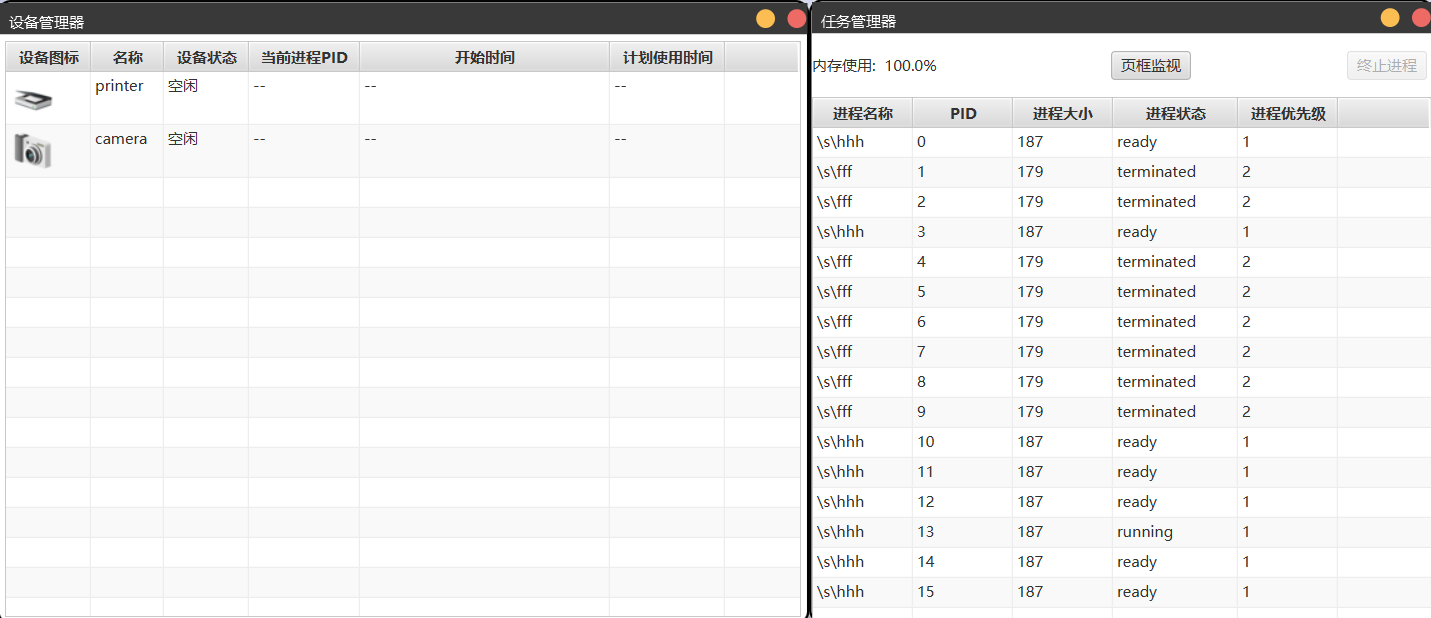


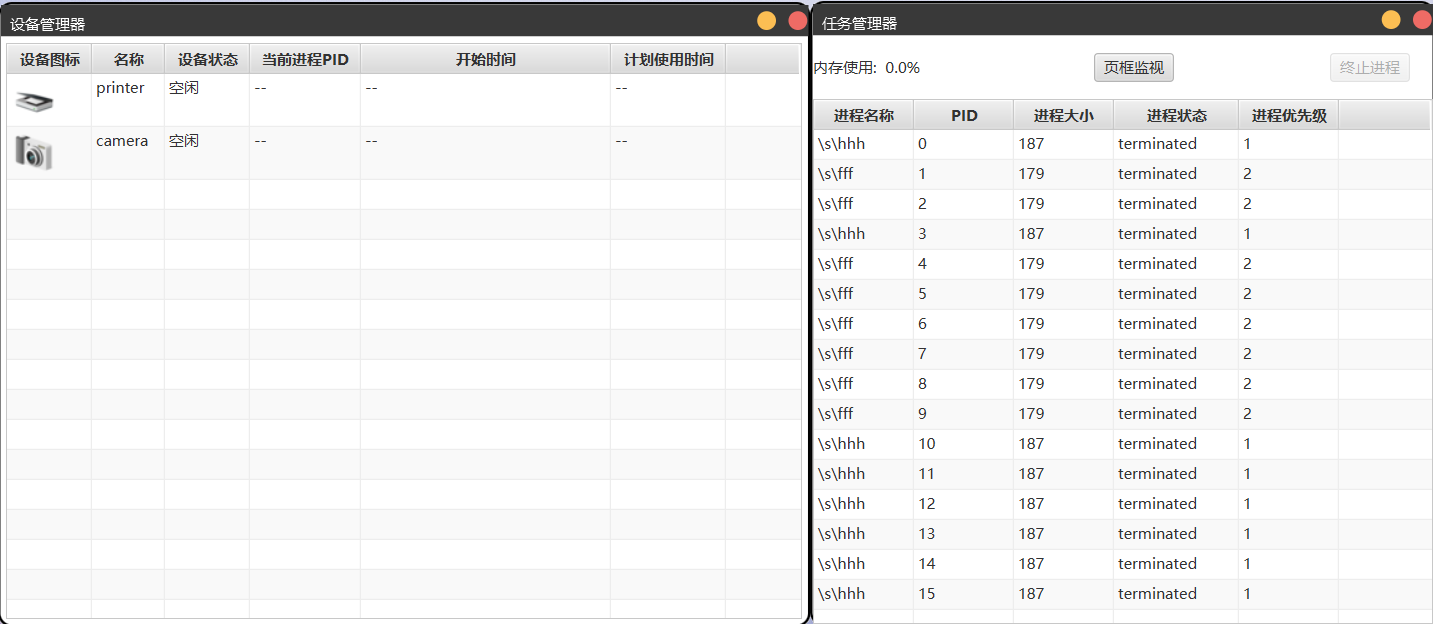
其中hhh文件优先级为1，fff优先级为2，数字越大优先级越高。

**测试结果：**









符合测试的预期结果，一共创建了16个进程，期间最多只有一个进程能是running状态，能够根据设备资源的空闲情况正确分配进程，各个进程在队列间准确潜移，内存实时显示占用百分比。能够成功实现要求的功能。

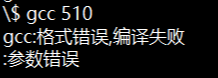
### GCC:

fig:

fig:

编译成功以后修改文件类型和名称

如果不符合格式则无法修改文件性质



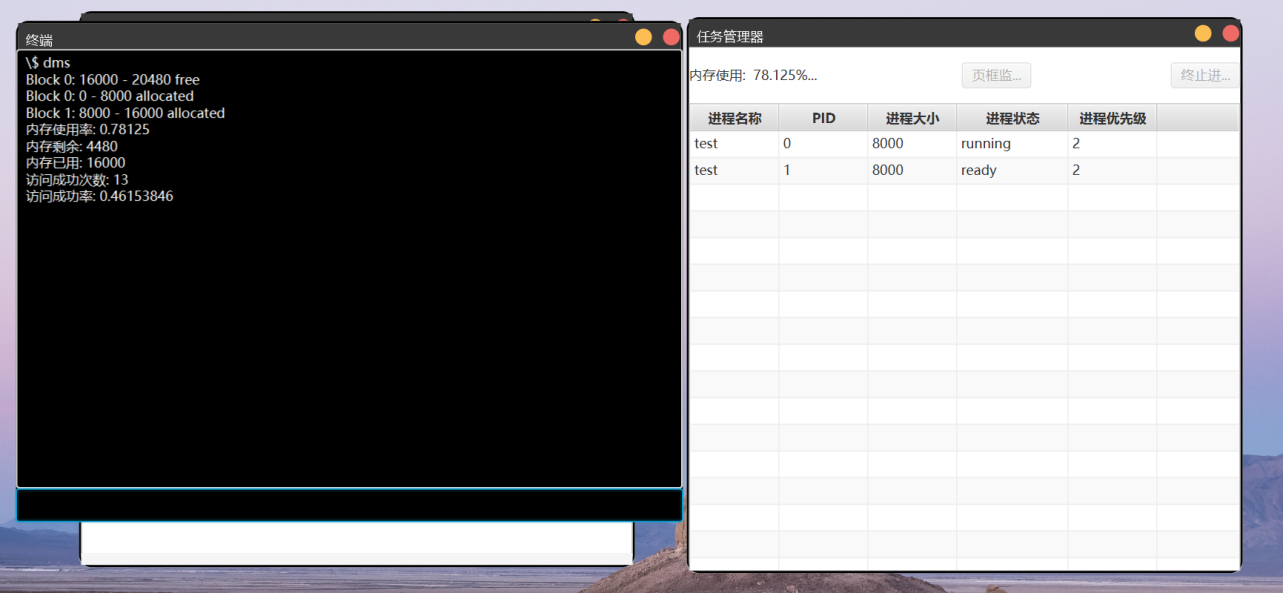
### MEMORY:

##### 测试的功能

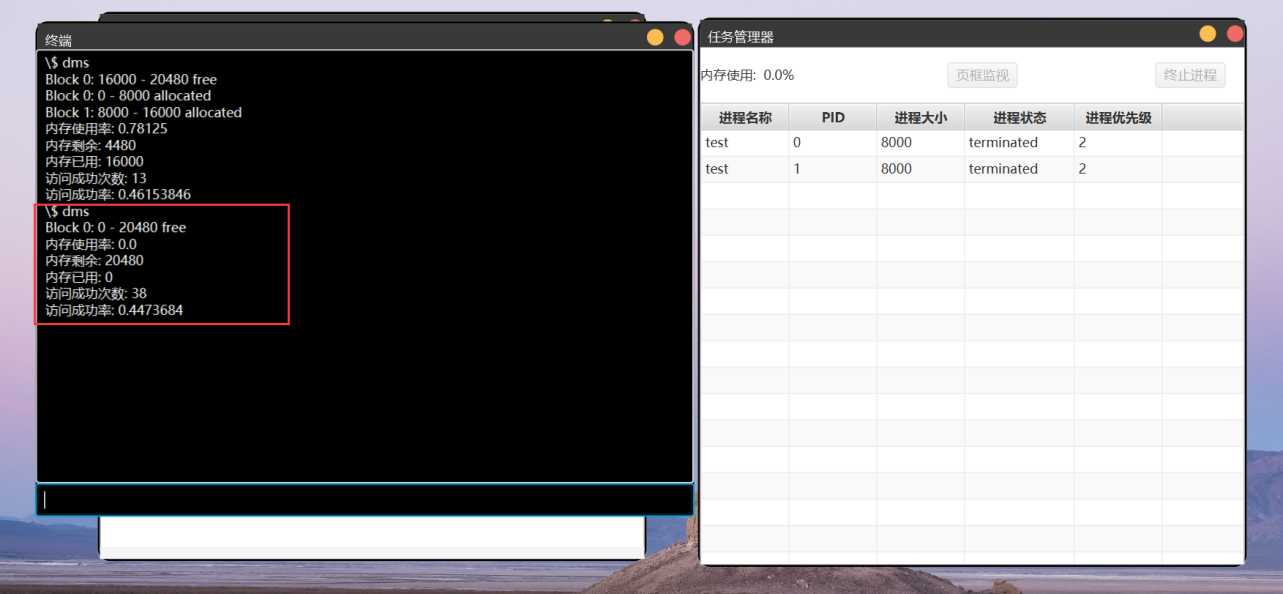
我们分别测试连续分配和页式分配模式，测试申请内存，访问内存，释放内存，查看内存使用信息等。测试文件名为“内存演示”。

###### 连续分配：

我们可以运行两次这个程序，发现两个进程都会申请8000的内存，并且还能用"dms"指令来查看内存使用信息



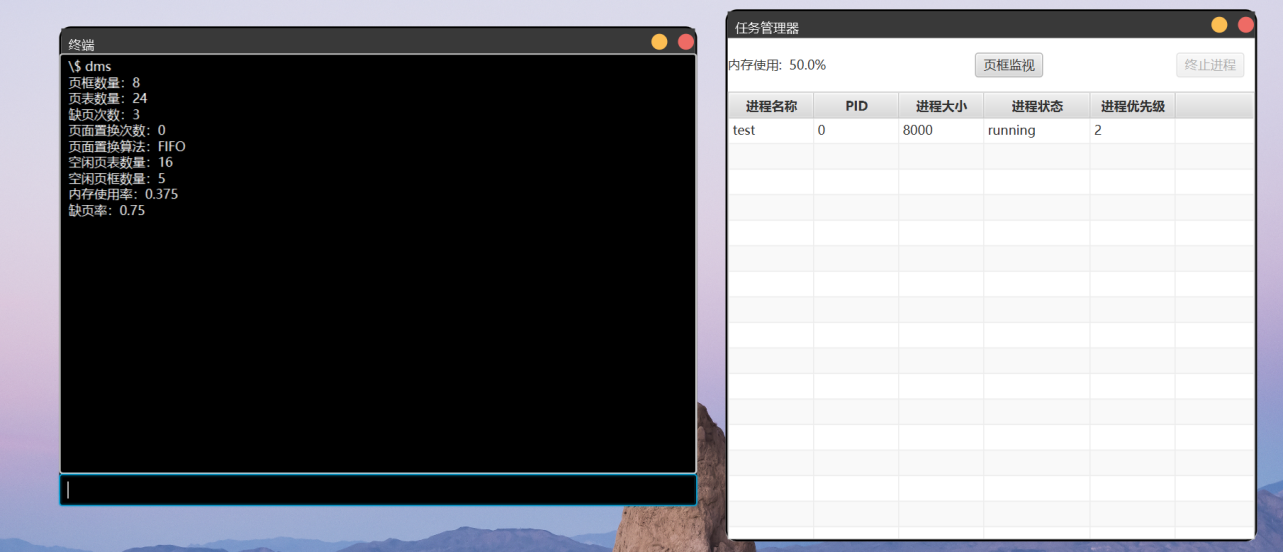
等到程序运行完毕后，又会释放内存



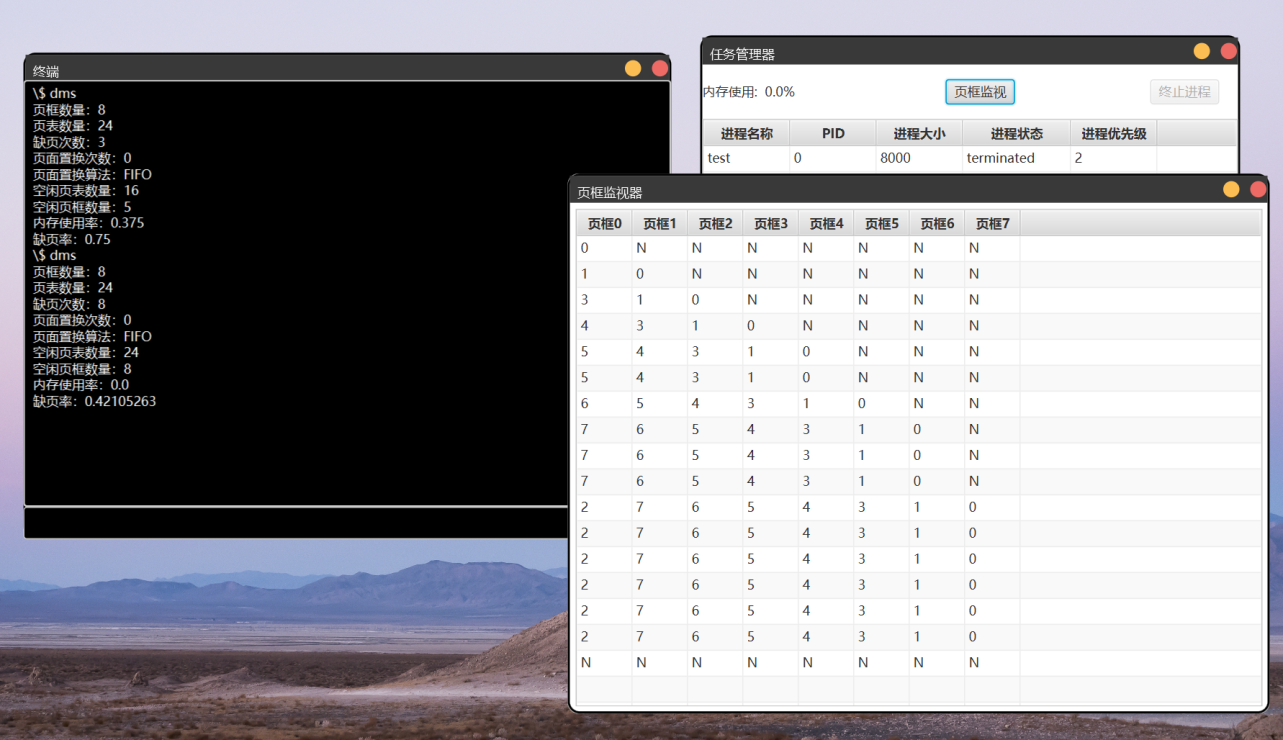
###### 页式分配：

FIFO算法：

首先运行测试程序，可以看到程序正在运行，并且内存使用率还会实时变化，也能通过指令查看具体内存使用情况

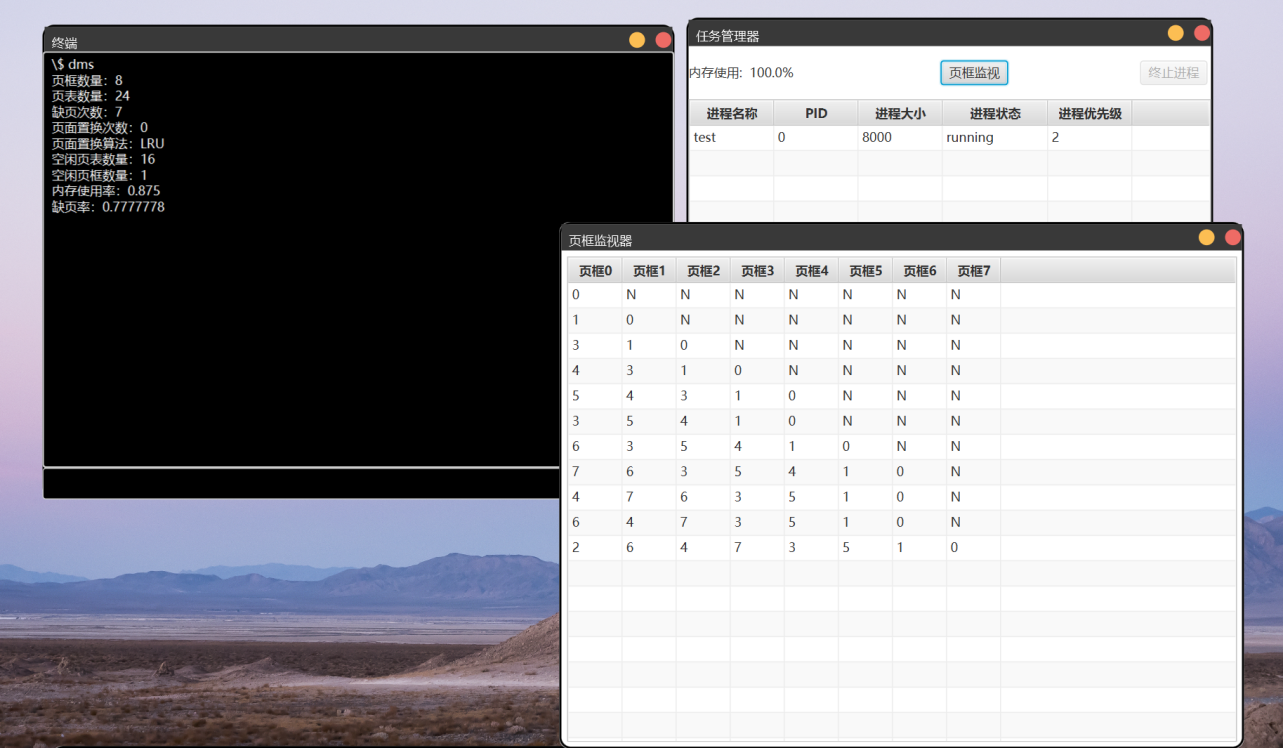


可以在页框监视页面查看具体的页面置换算法细节，这里是FIFO算法

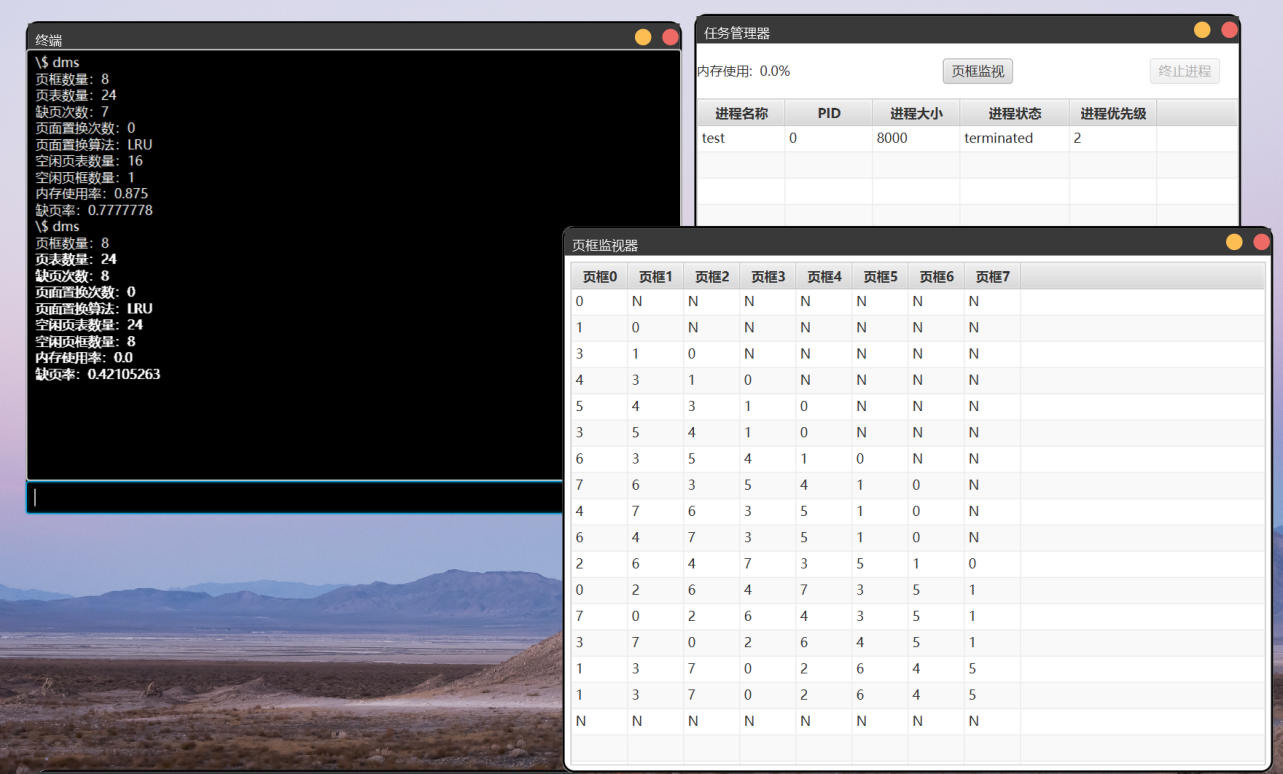


LRU算法：

首先运行测试程序，这里的内存使用率也会实时变化，也能通过指令查看具体内存使用情况



可以在页框监视页面查看具体的页面置换算法细节，这里是LRU算法，页框变化与上述的FIFO不同



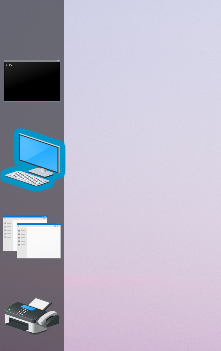
### UI:

程序窗口的实际效果在其他部分的报告中会得到充分体现，这里只测试UI部分的基础功能以及对bug的规避功能。

#### 1.窗口的最小化

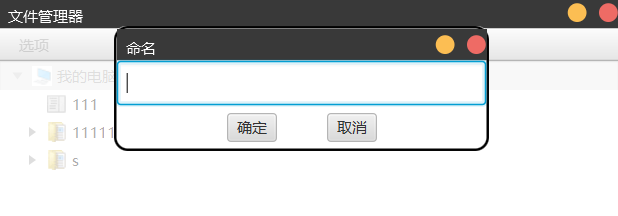


在对应程序窗口打开后，左侧面板会对对应图标边框进行处理。



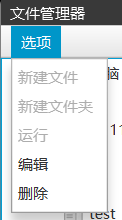
在最小化窗口后，图标边框依旧是高亮状态，此时再次点击图标，窗口就会出现。

#### 2.限制文件\文件夹命名时的操作

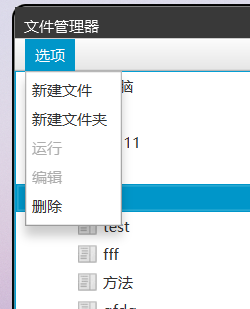


在选择新建文件\文件夹时，会将文件管理器设置为disable的状态，防止用户在进行命名时用户将新建文件的父目录删除从而引起的错误。当命名结束或是选择取消后，文件管理器恢复正常。

#### 3.限制用户操作



当用户选择的是文件时，则禁止新建选项。

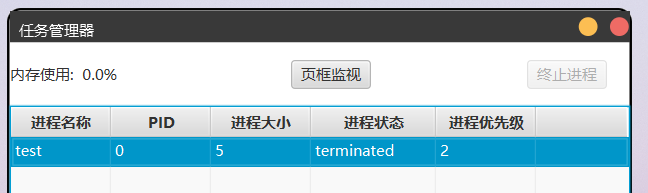


当用户选择的是文件夹时，则启用新建选项，禁止运行和编辑选项。



当用户选择的是可执行文件时，则启用运行选项。

#### 4.终止进程检测



为了便于观察运行结构，进程结束后并不会被移除显示。当进程处于终止状态时，终止进程按钮属于禁用状态。

## 9．课程设计总结

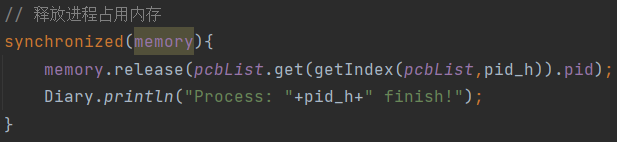
### 缪奇志:

#### 问题1

没有对memory进行互斥访问导致调用内存方法时出错

解决方法：

对调用内存的方法时设置互斥锁，保证同一时间只有一个进程能够修改内存的数据。互斥锁示例如下：



#### 问题2

程序计数器算法出错导致进程终结状态判断异常

解决方法：

确定好什么时候执行完指令内容，什么时候就让对应进程的程序计数器加一，严格按照这个规则，fork指令只有在创建成功后可以跳转到下一条指令、access指令在访问成功后跳转到下一条指令、cpu指令在结束运行时间后跳转到下一条指令、printer指令只有程序成功使用设备结束才能跳转到下一条指令、block指令在磁盘寻道算法成功运行结束后跳转到下一条指令。

#### 改进建议

1.对于access指令给内存一个相对地址，后续能够实现直观表现出哪个进程在访问哪个内存位置，实现实时对内存模块进行更改并实现监视的可视化。

2.进程模块和文件模块几乎没有交互，可以在进程端再加入代码，实现对文件的读写，注意实现时要对文件的读写操作加上互斥锁，保证同一时间只能有一个进程能对文件进行读写。

3.定时检查已经终止的进程，定期释放他们还在占用的所有资源。

#### 心得体会

在本项目中，我主要负责进程管理模块的基础功能实现并与最后其他模块的对接工作， 我将我的工作内容总结如下：

1. 进程调度：经过前期的算法性能测试以及算法实现难度综合考虑，最终协商讨论决定采用时间片和优先级的方法来实现进程调度。
2. 进程状态管理：跟踪每个进程的状态，如ready、waiting和running状态等，能准确地修改进程状态，进程的程序计数器等参数，保持对进程的准确维护与监视。
3. 中断：涉及到时钟中断，硬件 I/O 中断等功能,在程序运行到需要 I/O 请求的部分时，会从 running 状态跳出，等待 I/O 执行。
4. 与各模块交互：在创建进程以及需要访问内存时涉及到与内存模块的交互，出现设备请求时能正确与设备模块进行交互以及与UI和kernel的准确交互。

经过了本次课程设计，我对操作系统底层原理有深入的了解，对进程的各种调度算法和数据结构的创建有更深刻的掌握，积累了良好的软件工程实践经验。

在刚开始设计数据结构和算法设计花费了一定的时间，尤其是如何体现出各个状态的改变以及状态队列的迁移，经过和同组成员的协商最终确定下来使用list来存储各个进程，同时也设计对应的ready队列和waiting队列，出于优先级的考虑，对于ready队列实现不同的优先级有不同的队列，最终在成功实现进程的调度，加深了我对于调度算法的理解。在涉及到内存方面函数的引用以及在杀死一个进程后清除所在队列和所占资源，最终采取锁的方式确定同一时间只能有一个进程能执行访问内存的操作，这让我对于进程的同步性有了更深刻的理解。最后在与各个模块衔接时，我深刻的体会到了编码规范以及具备系统化的思维和设计能力的重要性，也学会如何更好的进行团队合作完成任务。

### 钱锡锐:

##### 设计过程中遇到或存在的主要问题及解决方案

主要问题有，虚拟内存与物理内存不太好模拟，不太好用类去描述他们的关系，页式存储结构很复杂，有虚拟页、物理页、页表项、页表入口等，构思设计就用了比较长的时间。页面置换算法难以直接体现给用户，所有我们在UI上添加了页框监视器来实时监视内存，达到用户直观查看内存分配的目的。

##### 改进建议

1）我们的每一个进程所分配的内存页面或者内存块虽然存储到了数据结构中，但是用户并不能直观的看到进程对于内存的使用情况，这里可以新加一些UI来具体地展示；

2）进程不能访问自己大小以外的数据，我们应该给每一页标记，让进程可能可以访问除了自己以外的其他地址空间；

3）内存应该分配一定数量的物理页框给某个进程，而不是让一个进程去试图抢占所有的物理页框。

##### 体会/收获

我负责内存的分配及管理模块，一开始感觉这个模块无非是申请释放存储，虚实地址转换，页面替换算法，可能还会有内部或外部碎片的处理情况，这些理论知识在之前都有细致的学习并且也做过相应的理论题目，但是只有在把书上的数据结构及算法一点点的用代码实现时，才发现页表结构的设计是一个十分繁复的过程，包括虚拟存储的分配以及释放，其中如果有哪一步不够小心都会很容易出现错误，也是在自己实际的使用代码进行内存的分配与管理时，才理解了内部碎片与外部碎片对内存利用率的影响，理解了处理内存碎片的必要性。当然，为了实现页框监视和可以类似 Windows 的资源管理器一样实时展示内存情况，我也学会了新的绘图方式，最后，将自己的模块整合进整个组的项目， 并且可以正确的运行及查看结果时，有种无法形容的愉悦感与自豪感，在这次课程设计之后，操作系统对于我而言虽然依旧复杂且难以实现，但却不再是那么神秘与无法想象了。

其实，内存管理模块需要改进的地方有很多，比如页表的数据结构可以更优化等等，学无止境，虽然我这个项目的经历终止了，但是以后还会持续地学习OS相关的知识。

在这个项目中，真正的体会到了小组分工与团队协作。每一个人的努力都值得被肯定，在这里也要感谢我的组员们，与我一起努力了这么久，做了这一个小小的“操作系统”，缺少了任何一个人这个项目都是不完整的。

### 倪玮昊:

#### 遇到问题与解决:

无法从shell向kernel传输数据,因为kernel创造了shell,所以此处使用了管道通信完成回应

#### 体会:

我写的kernel模块并不难,也没有遇到什么问题,但是作为组长,我花了很多时间去研究了整合代码和调度模块,最终当我收齐了所有同学的代码进行整合的时候,debug的量很大,所以我感受到了作为组长需要指定清晰的规划,使用github对组员代码进行一些较为清晰的管理,并且要在精神上鼓舞组员(因为有些组员自我驱动力不足),总之在这次课设上,相比技术方面的提升,我更体会到了作为一位开发者小组的组长,我要去怎么做,这项经历对我未来的工作帮助很大.

### 何正豪:

#### 1.设计过程中遇到的问题

**信息实时监控的实现问题：**

早期的方案是在被监控的数据结构在被更改时直接更改UI内容，但由于UI组件与其他系统组件相互独立，互相并没拥有引用，因此只能使用静态函数，同时就诞生了大量的静态变量，导致修改UI内容的过程繁杂并且安全性低。除此之外，由于JavaFX的特性，非UI线程的线程不能正常修改UI的组件，因此每次修改都需要新建一个JavaFX提供的线程来进行更改。后来通过学习JavaFx的属性绑定内容，将相关的数据结构更换为JavaFX提供的实现了监听接口的数据结构，这样就不用处理大量的静态变量，而只需要监听相应的数据结构，在触发信号后使用默认的或者是自定义的方法来对UI进行更改。

##### 系统当前操作目录的问题：

由于系统当前的操作目录作为私有变量存在于文件管理系统中，当用户通过文件管理器窗口对文件进行操作时，只能直接获取到文件名称，无法正常传递文件路径，而系统当前的操作目录也并不会实时进行转换，因此文件的操作基本都不能正常进行。后续自定义了一个TreeItem类，用于记录文件的同时记录文件的绝对路径，并在文件管理窗口添加了一个变量，每当用户点击文件\文件夹时，记录该文件\文件夹的路径，这样一来则成功解决了UI的文件操作问题。

#### 2.程序的不足和改进建议

* 信息的展示仍不够直观。例如文件系统，文件管理窗口没有直观地展示文件的大小、类型等信息，页框监视器通过数字表格展示的页框状态变化也显得不够直观。可以考虑采用更直观的图像方式来绘制相关信息。
* 代码的复用性尚有不足。Win类是我对复用性的一次尝试，因此我在设计新的程序窗口时不需要从头开始设计，因此UI部分本质只有Desktop、Controller和Win类。但Controller的复用性却较差，当设计了新的程序窗口时，必须在Controller类中进行相应的修改后才能进行正常显示，因此可以尝试对类进行更加精确地抽象，减少开发时的工作量。
* 代码的性能较差。UI线程往往不能处理过于复杂的逻辑处理，但此次UI线程里存在着不少的逻辑判断，并且还有部分读写管道的地方，导致UI在进行某些操作的时候响应速度较慢。因此可以尝试简化逻辑处理，优化与kernel的交互方式，以增强UI的性能。

#### 3.体会和收获

此次课程设计团队人数是我经历过的最多的，同时也是我第一次真正意义上和组员相互独立地开发自己的部分，并通过其他系统提供的接口来接入和测试UI模块。由于多人开发存在的频繁的程序更改问题，我们使用了GitHub来对程序进行版本管理，有效保证了开发的有序进行。在开发后期进行测试时，组员密切沟通，积极处理出现的问题，最终得到了一份完整的程序。对于我来说，这是一次收获颇丰的团队开发经历，让我对团队开发中的进度管理、人员管理等方面都有了实际的认识和体验。

除此之外，在此次开发中我是首次使用JavaFX进行界面开发，由于JavaFX的资料相对较少，在开发初期困难重重。在学习了相关课程以及官方文档后，我又阅读了相关代码，才得以成功使用JavaFx进行界面的开发。在开发的过程中，又逐步了解和应用了javaFX特有的一些类及其方法。在这个过程中，我的学习能力得到了增强，同时也适应了在短时间内学习并使用新技术。

最后最重要的是，我对操作系统有了更深入的认识。以前只是从理论上认识操作系统，最多是通过课程实验来了解操作系统的某些重要机制。而此次实验是让我们在理解操作系统的前提下复现操作系统的诸多机制，加深了我对操作系统的认知。

### 葛北鱼:

#### 设计过程中遇到或存在的主要问题及解决方案

* ProcessControlBlock类与ProcessManager类耦合性较强，导致初始ProcessControlBlock类进行一部分进程控制的操作无法使用。
* 解决方案：对该部分代码进行重构，确定在run()方法中实现主要进程的相关操作。
* ProcessManager类需要刚刚结束的硬件对应的pid，然而原逻辑是单个硬件自己控制自己是否结束自己的任务，到时间后自动pid设为-1，使得HandwareManager类无法返回。
* 解决方案：由HandwareManager类控制是否结束某个硬件的任务，从而可以返回给ProcessManager类刚刚结束的硬件对应的pid。

#### 改进建议

硬件模块还可以添加硬件优先级等功能。

#### 体会/收获

本次课设我和缪奇志同学一起参与了进程模块的工作，主要负责参与设备管理模块的构建和PCB结构的构建。在最开始时，本来我和缪同学准备一个人写PCB类，一个人写ProcessManager类，结构最后发现，由于这两个类耦合程度过高，导致我原本写的PCB的很多接口都无法使用，最后在我们讨论后，只能重写，因此我体会到了事先交流的重要性。

在本次的课设中，我们七个人成功构建了HOMOS操作系统，并设计了logo加入到开机页面中。“HOMO”与“鸿蒙”发音相似，加上"S"，又有了操作系统“OS”部分，表达了我们对国产操作系统的信心与致敬；此外，“HOMO”是“人属”的英文单词，在当今人工智能告诉发展的时代，使用ChatGPT等工具似乎也可以构建一个操作系统，然而，我们为自己的操作系统注入了属于“人”的浪漫。在一次次的讨论和共同debug中，“代码”并不再是“机器”的专利，我们将自己的时间和劳动融入其中，构建了这个充满着“人”的HOMOS操作系统。

此外，不出意外的话，操作系统课设应该是我大学本科生涯中最后一门课设了。这是目前为止组队人数最多的一次课设，也是让我感受了整个软件开发流程的一次课设。与正常的理论课和实验课不同，课程设计更多是让我们开发软件，更能提升我们的代码实践能力。那么，请允许我在此向所有的课程设计道个别：再见了、HOMOS；再见了、所有的课程设计:)

### 石亚行:

#### 体会

通过操作系统课程设计，进一步认知了操作系统中各模块的接口与配合，熟知了文件系统的模拟实现。在与组员的精诚合作中，严肃活泼，团结奋进，求实创新，深刻诠释了homos集团的企业理念和文化，对我未来的学业和不远的将来的工作，产生了极大的帮助和启发。

### 施天勤:

#### 遇到问题及解决方案

在编写ls、cd等设计文件树操作过程中，对于文件树的操作有时较为繁琐，通过设置临时量进行记录解决该问题。

#### 改进建议

对于cd，ls等函数虽已实现对绝对路径的操作，但方式较为繁琐，本质是通过多次对相对路径的操作以达到相同的效果。

对于磁盘调度方面，仅实现了对柱面的模拟，可以增加对扇区的模拟等以提高真实性。

#### 心得体会

经过了本次课程设计，我对磁盘调度相关的算法有了更深刻的理解，对于文件相关的编程掌握的也更加熟练。在利用map创建文件树时，对于树中内容的查找与遍历操作的代码实践也让我收获良多。