操作系统设计方案

学院:计算机学院

专业:计算机科学与技术

班级:2020211304

组员:倪玮昊,缪奇志,何正豪,石亚行,施天勤,葛北鱼,钱锡锐

一.组员任务

该项目由JAVA语言开发,开发平台为idea

姓名	分工
倪玮昊(组长)	kernel和shell的开发
缪奇志	进程系统开发
何正豪	UI系统设计与开发
石亚行	文件管理系统开发
施天勤	文件管理系统开发
葛北鱼	进程系统开发
钱锡锐	内存管理系统开发

二.设计思路

1.系统概要

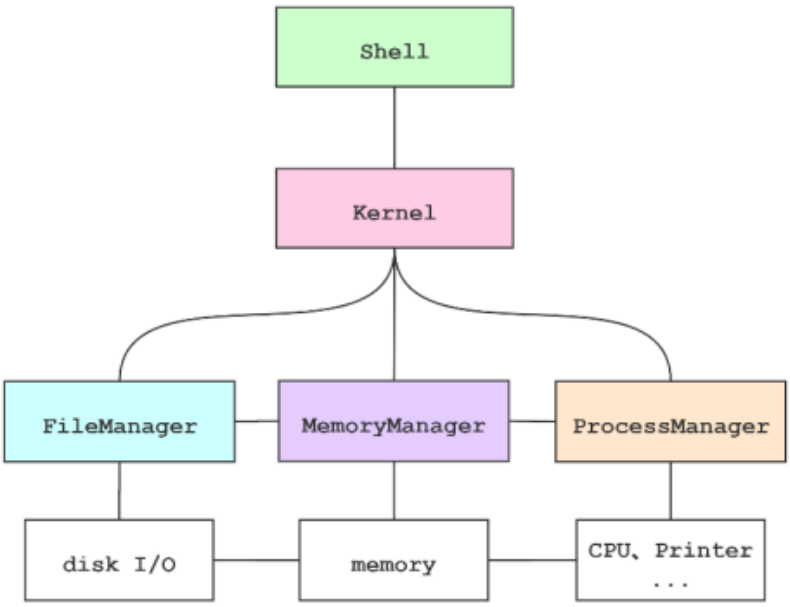
系统有四个不同的模块：Shell、Kernel、File Manager、Memory Manager、Process Manager。其中Shell 是用户与HOMOS沟通的桥梁，其在我们设计的系统中既是一种命令语言的存在，又是第一个在本系统之上运行的默认应用程序。

Shell 提供了一个界面，用户通过这个界面访问操作系统内核的服务；Kernel是MiniOS的核心部分，其实际上又由本身以及剩余的三个模块构成，负责管理系统的进程、内存、设备、文件等，决定着系统的性能和稳定性。

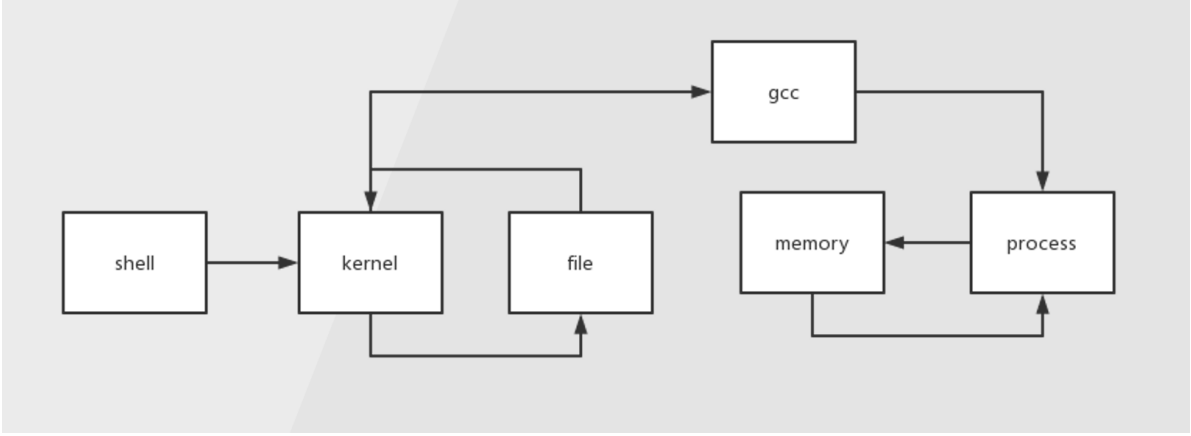
进一步划分，Kernel 实际上能够调用本系统剩余的三大核心模块。

其中 File Manager 负责对系统文件进行文件的逻辑组织和物理组织、文件树的结构和管理。所谓文件管理，就是操作系统中实现文件统一管理的一组软件，亦或是被管理的文件以及为实施文件管理所需要的一些数据结构的总称。在我们设计的系统中，该模块还负责与模拟磁盘设备进行交互，实现文件在物理存储体上的读写，并负责对空闲磁盘块的调度；另外，其具有驱动功能，能够对模拟磁盘设备进行直接控制，接管了与外存交互的所有事务。

对于 Process Manager，由于在HOMOS 中，进程是正在运行的程序实体，也包括这个运行的程序中占据的所有系统资源，比如说 CPU、打印机等。而这一模块能够实现对这些实体对系统核心资源使用的管理，如采用高效的调度算法（如优先级调度算法、高响应比调度算法等）以及通过维护 PCB（进程控制块）结构来实现对各进程实体关键信息的记录，以及使用资源的合理调度。



2.调用过程:



三.模块详细设计

1.KERNEL和SHELL

1.指令设计

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
mon	mon [-o]	开始监控系统资源使用情况，将监视结果以图片的方式实时输出；-o 选项停止监控
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

2.模块交互

shell与kernel进行交换:

使用监听器监听用户输入

```
procedure actionPerformed(e: ActionEvent)
    input := terminalInput.getText()
    terminalInput.setText("")
    terminalOutput.append("$ " + input + "\n")
    if input equals "clear" then
        terminalOutput.setText("")
        return
    end if
end procedure
```

处于实时交互的要求,无法通过调用函数执行命令,所以使用管道通信连接kernel和shell

```
   static PipedInputStream shellOutput = new PipedInputStream();
    static PipedOutputStream shellInput = new PipedOutputStream();
    //管道1,kernel向shell传输数据
    static PipedInputStream shellOutput1 = new PipedInputStream();
    static PipedOutputStream shellInput1 = new PipedOutputStream();
```

使用多线程做到一边传递用户输入的系统,一边接收kernel发来分反馈信息

```
procedure run()
    try
        bytesRead := 0
        buffer := new byte[1024]
        while ((bytesRead = kernelOutput1.read(buffer)) != -1) do
            data := new String(buffer, 0, bytesRead)
            //打印data
            print(data)
            terminalOutput.append(data + "\n")
            // Process the data as needed
        end while
    catch (IOException e)
        e.printStackTrace()
    end try
end procedure
```

3.kernel设计逻辑

通过指令格式进行简单分析,并且报错或者执行

```
procedure deal_ls(word: String) throws IOException
    words := word.split(" ")
    list := null
    if words.length equals 1 then
        //调用ls模块
        list := fileManager.ls("", "-a", method)
    else if words.length equals 2 then
        if words[1] equals "-l" then
            //调用ls -l模块
            list := fileManager.ls("", "-l", method)
        else if words[1] equals "-a" then
            //调用ls -a模块
            list := fileManager.ls("", "-a", method)
```

```
        else if words[1] equals "-al" then
            //调用ls -al模块
            list := fileManager.ls("", "-al", method)
        else
            //调用ls path模块
            list := fileManager.ls(words[1], "-a", method)
        end if
    else if words.length equals 3 then
        if words[1] equals "-l" then
            //调用ls -l -a模块
            list := fileManager.ls(words[2], "-l", method)
        else if words[1] equals "-a" then
            //调用ls -a -l模块
            list := fileManager.ls(words[2], "-a", method)
        else if words[1] equals "-al" then
            //调用ls -al模块
            list := fileManager.ls(words[2], "-al", method)
        else
            shellInput1.write("ls:第二,三个参数格式错误".getBytes())
            shellInput1.flush()
        end if
    else
        shellInput1.write("ls:参数过多".getBytes())
        shellInput1.flush()
    end if
    //打印list
    printList(list)
end procedure
```

2.GCC

对执行文件进行检查,如果符合格式则赋予优先级修改文件性质

执行前:

```
{"size":"2000","name":"test","type":"crwx","content":["fork","cpu 10","access 10",
```

执行后:

```
{"size":"2000","name":"test","type":"erwx","priority":"2","content":["fork","cpu 1
```

3.进程管理

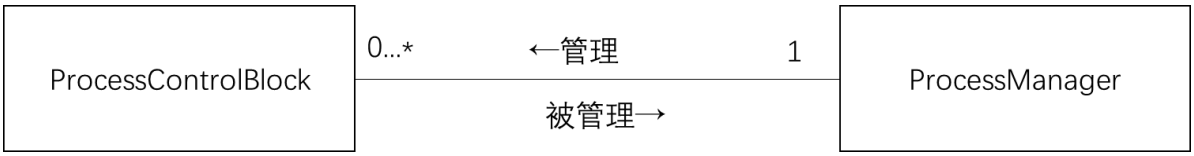
1.需求分析

1.1需求概述

设计对应的批处理指令，完成对于批处理任务的解析与执行，使用PCB作为进程唯一存在的标志，并在PCB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，并通过时间片、优先级等机制，使得用户能够以更加灵活的方式与系统进行任务交互。

1.2需求结构

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：

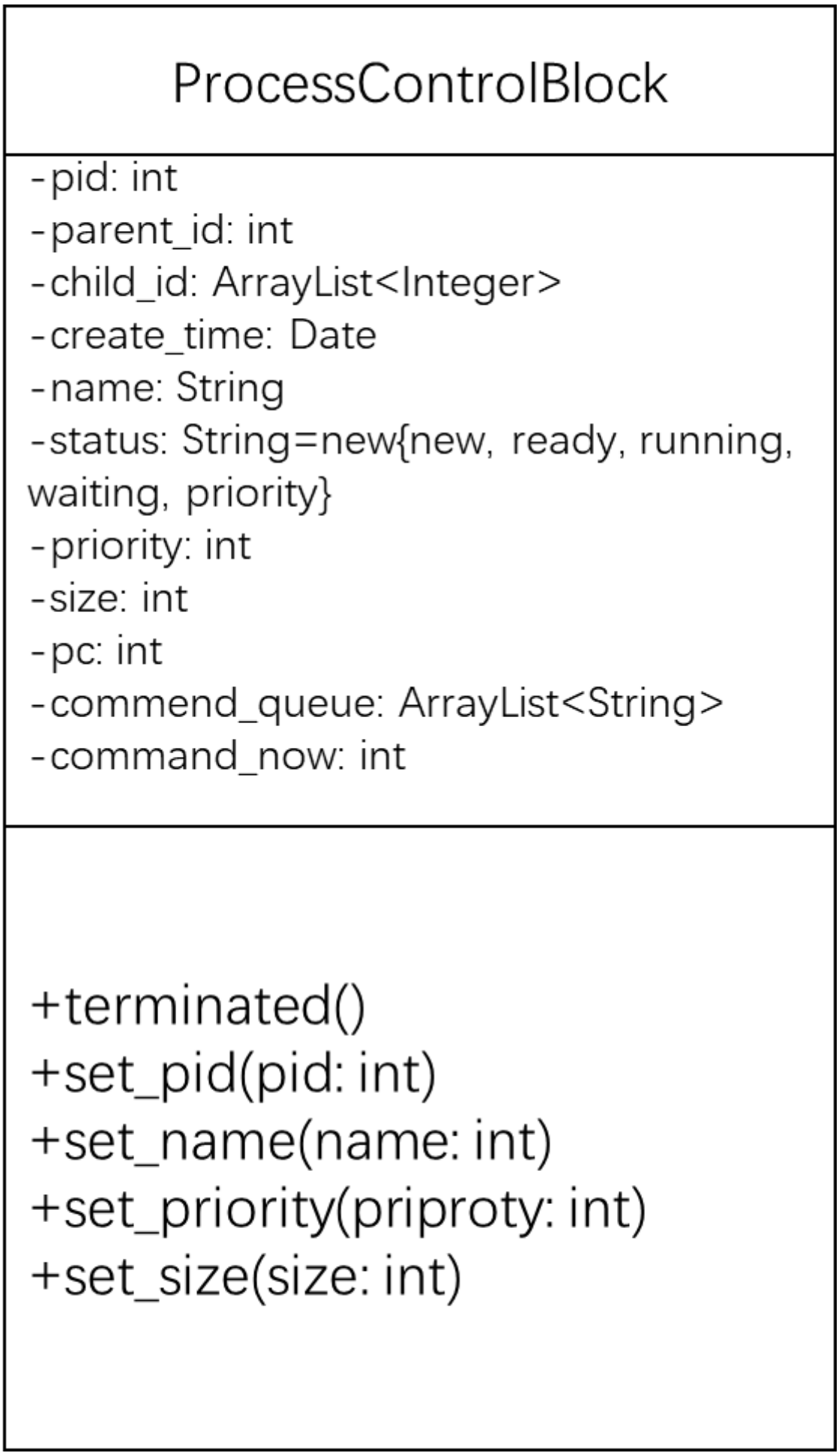


2.结构设计

2.1ProcessControlBlock类

2.1.1类图

`ProcessControlBlock`类图如下：


ProcessControlBlock
-pid: int
-parent_id: int
-child_id: ArrayList<Integer>
-create_time: Date
-name: String
-status: String=new{new, ready, running, waiting, priority}
-priority: int
-size: int
-pc: int
-commend_queue: ArrayList<String>
-command_now: int
+terminated()
+set_pid(pid: int)
+set_name(name: int)
+set_priority(priproty: int)
+set_size(size: int)

2.1.2拥有属性

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
```

2.1.3提供操作

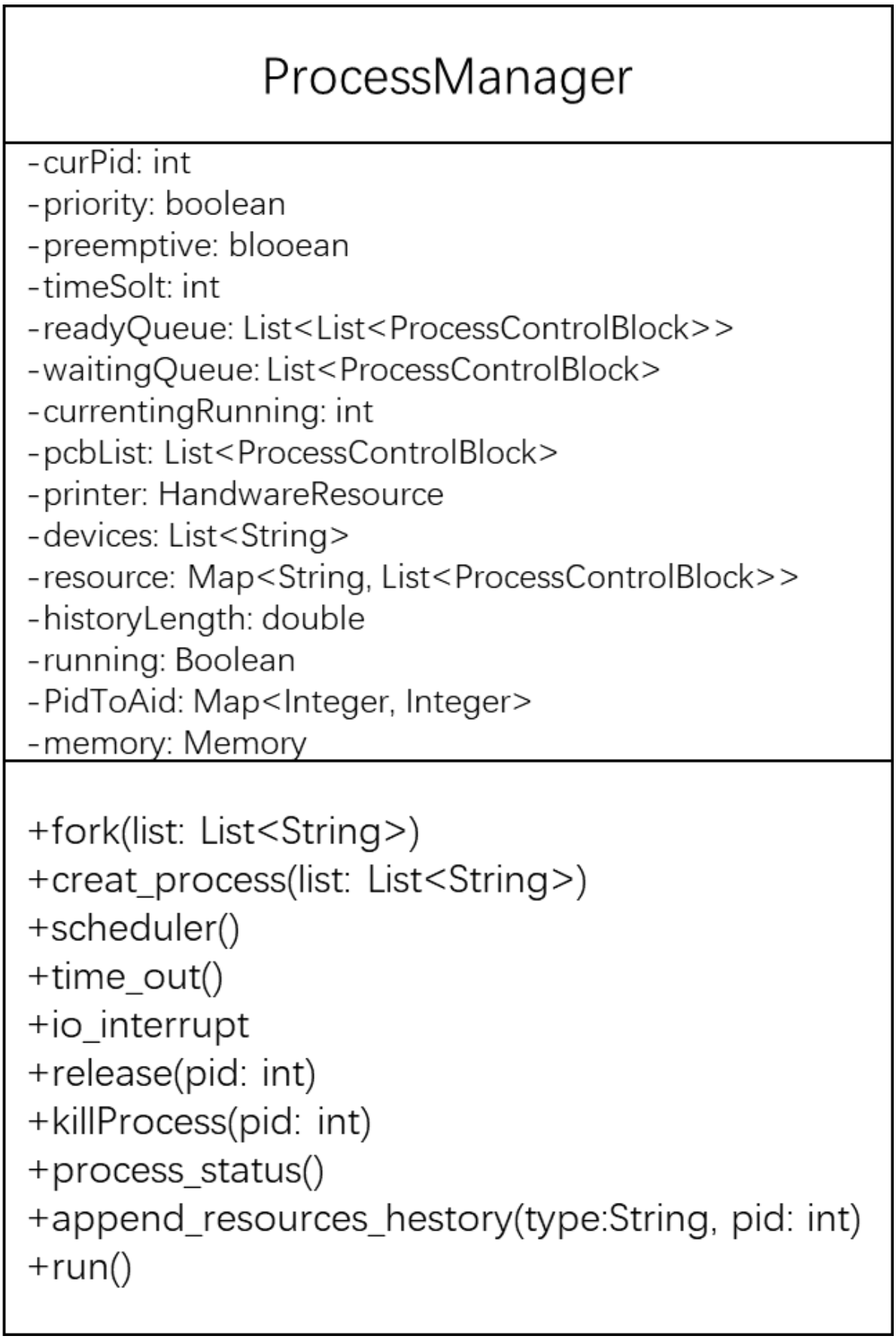
`ProcessControlBlock`类提供如下操作：

```
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
```

2.2ProcessManager类

2.2.1类图

`ProcessManager`类图如下：



2.2.2拥有属性

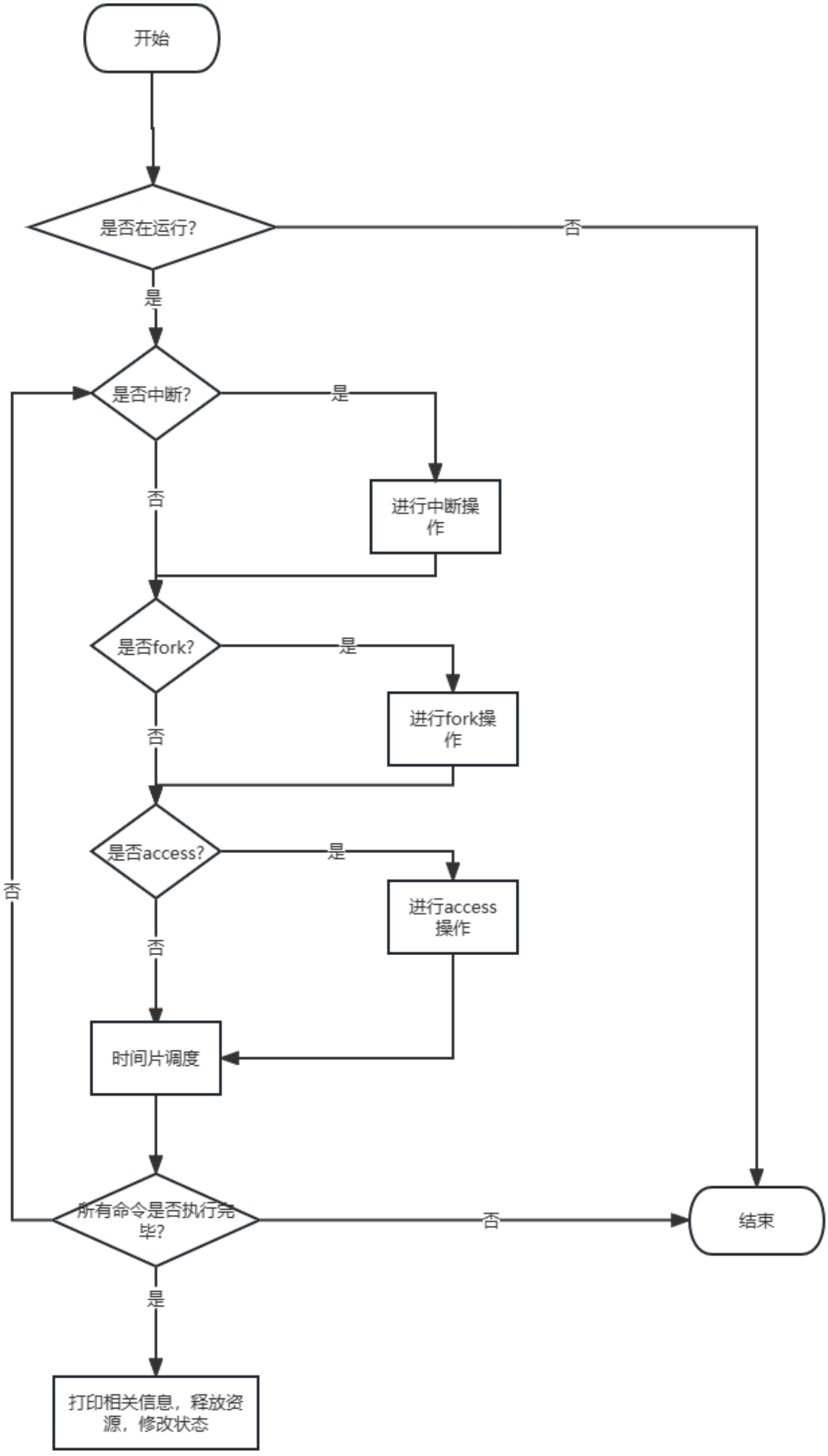
```
public class ProcessManager {
    private int curPid;             //当前分配的进程的pid
    private boolean priority;       //是否采用优先级调度算法
    private boolean preemptive;     //是否允许抢占
    private int timeSlot;           //时间片
```

```
    private List<List<ProcessControlBlock>> readyQueue;     //就绪队列
    private List<ProcessControlBlock> waitingQueue;         //等待队列
    private int currentRunning;
    private List<ProcessControlBlock> pcbList;
    private HardwareResource printer;
    private List<String> devices; //含有的设备
    private Map<String, List<ProcessControlBlock>> resourcesHistory;
    private double historyLength;
    private boolean running;        //时候有进程在执行
    private Map<Integer, Integer> pidToAid;//虚拟地址对应实际地址
    private Memory memory;
    public List<ProcessControlBlock> getProcess() {
        //遍历pcbList,打印状态,name,pid
        for (int i = 0; i < pcbList.size(); i++) {
            System.out.println("进程名字:" + pcbList.get(i).name + " 进程状态:" +
pcbList.get(i).status + " 进程pid:" + pcbList.get(i).pid);
        }
        return pcbList;
    }

    //memoryManager是内存管理器对象，用于管理进程的内存分配和释放。
    public ProcessManager(Memory memory) {
        this.curPid = 0;
        this.priority = true;
        this.preemptive = false;
        this.timeSlot = 0;
        //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 = new ArrayList<ProcessControlBlock>();
        //表示打印机资源
        //********this.printer = new HardwareResource(printerNum);******//
        //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<>());
        this.historyLength = 14.0;
        //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;
    }
```

2.2.3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，需要调用到文件模块
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()
```

3.处理逻辑


开始
是否在运行?
否
是
是否中断?
是
进行中断操作
否
是否fork?
是
进行fork操作
否
是否access?
是
进行access操作
否
否
时间片调度
所有命令是否执行完毕?
否
结束
是
打印相关信息，释放资源，修改状态

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terminal，在程序运行中可以监视并且查看各个进程的信息。

4.UI

1.需求分析

1.1需求概述

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。重点需要处理的内容是界面交互的逻辑、界面向的内核的信息发送以及系统信息的图形化表示。

1.2需求结构

设置三个类desktop、win和controller，desktop为UI主舞台，用于放置软件图标以及程序窗口；win为窗口类，模拟生成软件窗口；controller为界面逻辑模块，用于处理界面交互中的逻辑功能。三者与内核的关系如下：

2.结构设计

2.1Desktop类

2.1.1界面

2.1.2界面描述

界面元素	元素说明
顶部菜单栏	用于放置系统时间和关机按钮
桌布	用于显示桌面背景
系统应用栏	用于放置系统应用图标
应用图标	用于生成对应应用窗口，同时通过图标边框变化表示应用是否运行

2.1.3功能详细说明

Desktop主要功能是作为其他组件显示的舞台，以及处理用户的初级操作指令。用户可以在桌面查看时间、关闭程序以及打开应用。

2.1.4类图

2.1.5主要方法

public Pane getBase();//获取主舞台

public void addWin(Win win)；//向主舞台添加新窗口

public void deleteWin(Win win,String name)；//从主舞台删除指定窗口

2.2Win类

2.2.1界面

·TerminalWin

·TaskManager

·FileManager

·DeviceWin

2.2.2界面描述

界面元素	元素说明
窗口名称	显示当前窗口的名称
最小化按钮	隐藏窗口但不关闭
最大化按钮	将窗口铺满桌面
关闭按钮	关闭窗口
中央画布	窗口中央部分，用于提供给不同窗口的构建方法进行对应组件的显示

2.2.3功能详细说明

Win类是一个父类，本身只实现了模拟软件窗口的最大化、最小化、关闭以及拖动功能，窗口中央部分则是空白。而TerminalWin等详细窗口类都继承Win类，并从Win类直接继承窗口中央的Pane，并在其中组装自己的UI组件。

2.2.4类图

2.2.5主要方法

·Win

public Pane getPane();//获取窗口中央Pane

public String getName()；//获取窗口名字

·DeviceWin

private void refreshDevices();//更新设备状态

2.3Controller类

2.3.1功能详细说明

Controller负责处理UI组件的交互逻辑，主要有两个作用。一是组装和销毁窗口，二是将用户操作转换为对应指令传递给内核。

2.3.2类图

2.3.3主要方法

private void newWin(String name)；//根据窗口名字实例化对应的窗口，并将窗口加入winHashMap中，并将窗口显示在Dessktop中。

public void closeWin(String name)；//在Desktop中删除对应窗口，并将窗口从winHashMap中移除。

public Boolean isOpen(String name)；//判断对应名称的窗口是否已经打开。

public void setVisible(String name)；//将对应的最小化状态的窗口重新出现在Desktop中。

3.处理逻辑

3.1新建窗口

当用户点击桌面上的应用图标，Desktop将调用Controller的newWin(String name)方法，并传递窗口名称。Controller根据传入的窗口名称，实例化对应的窗口，并将该窗口加入winHashMap中以方便管理当前已经打开的窗口。最后Controller再调用Desktop的addWin(Win win)方法将该窗口显示在Desktop上。

3.2关闭窗口

当用户点击关闭窗口时，Win调用Controller的closeWin(String name)方法，Controller首先调用Desktop的deleteWin(Win win,String name)方法，将对应的窗口从Desktop中移除，再将该窗口从winHashMap中移除，没有了引用的窗口会被Java的垃圾回收机制回收。

3.3内核命令处理

当用户在应用窗口进行某些操作时，比如在Terminal中输入命令、在文件管理窗口操作文件或是在任务管理器中终止某个进程，此时通过事件监听机制将上述操作转换为对应的内核指令，并由Controller传递给内核。

3.4对内核信息的监控

使用属性绑定进行监控,以设备管理为例，首先创建一个静态类DeviceInfo,该类包括一个设备所拥有的所有信息，设备管理器类将所有的设备信息都加入ObservableList中，而后创建一个TableView，该表格类只接受DeviceInfo类型的输入，而后进行属性绑定。

```
        //创建一个名为nameColumn的列，该列接受的输入类型为DeviceInfo,但显示的内容类型为
String
        TableColumn<DeviceInfo, String> nameColumn = new TableColumn<>("Name");
        //将该列与ObservableList中DeviceInfo的name属性绑定
        nameColumn.setCellValueFactory(param -> new SimpleObjectProperty<>
(param.getValue().getName()));
```

在绑定后，若该静态类的属性值发生了更改，则表格中显示的属性值也会对应变化。

5.文件管理

1.文件系统类图

2.FileManager

3.拥有属性

```
    private static final String file_separator = File.separator;//根据操作系统，a动
态的提供分隔符
    private static final String root_path = System.getProperty("user.dir") +
file_separator + "File"; // 当前程序所在目录+分隔符+文件名

    private String current_working_path = file_separator;

    private int block_size;//磁盘块的大小
    private int block_number;//磁盘块的数量
    private int tracks;//磁道数
    private int secs;//扇区数

    private int[] unfillable_block = {3, 6, 9, 17};// 不能使用的编号
    private HashMap<String, int[]> block_dir = new HashMap<>();//文件路径和磁盘使用
情况的映射
    private int[] bitmap;//磁盘块的占用情况
    private ArrayList<Block> all_blocks = new ArrayList<>();// 磁盘中所有的块的list

    private Map<String, Object> file_system_tree;// 文件系统树
```

4.文件树处理逻辑

private Map<String, Object> init_file_system_tree(String now_path)

参数为：now_path是当前递归到的绝对路径，返回一个Map类的文件树

根据传入的绝对路径创建一个File对象，并调用listFiles()方法获取该文件夹下的所有子文件和子文件夹。接着，代码遍历文件列表，对于每个文件，判断它是文件夹还是文件，如果是文件夹则递归调用init_file_system_tree()方法，将返回的Map对象作为当前文件夹对应的键值。如果是文件，则读取文件内容，并解析其中的JSON字符串，提取"type"字段作为该文件的类型，然后将文件名作为键，类型字符串作为值，存入当前文件夹对应的Map对象。最后，将该Map对象返回。同时在该过程中为文件分配磁盘空间。

磁盘分配方法在该方法实现：private int fill_file_into_blocks(JSONObject f, String fp, int method)
参数f为填充文件的内容，格式为JSONObject；fp为文件名；method决定分配磁盘算法，0为first fit算法，1为best fit算法，2为worst fit算法。

5.模块API

File Manager提供了如下API接口

- `public List<String> ls(String dir_path, String mode, String method)`：列出目标路径下的文件。dir_path，字符串类型，目标路径，支持相对或绝对路径，当目标路径不是目录时，仅列出该目标路径的文件信息。
    1. 首先检查目标路径参数dir_path，如果为空，则默认为当前工作目录current_working_path。
    2. 将相对路径转换为绝对路径，并将路径解析为各级目录名称，通过逐级访问文件树字典，找到目标路径对应字典元素的值。
    3. 如果目标路径对应的值是一个文件，则仅输出该文件的信息，否则，遍历目录下所有文件并按情况输出：
    - 如果文件是一个目录，以绿色输出。
    - 如果文件是一个隐藏文件，不输出。
    - 如果文件是一个可执行文件，以绿色输出。
    - 否则，按默认方式输出。
    4. 如果mode参数为"-l"，则输出详细信息，如果mode参数为"-a"，则输出隐藏文件。
    5. 如果method参数为"get"，则返回目录下所有文件列表，用于实现shell的正则表达式匹配功能，如果method参数为"print"，则将结果输出到控制台并返回结果列表。

总体思路是先处理路径，然后遍历目录下的所有文件，根据情况输出文件信息。需要注意的是，函数要能够处理各种异常情况，例如目标路径不存在、路径格式不正确、目标路径不是一个目录等等。cd、mkdir、mkf、rm的实现思路与 ls大致都相同，先处理路径，再扎到map，再对map进行操作。下文不再详细分析他们的实现思路。

- `public String cd(String dir_path)`：改变当前工作目录。dir_path，字符串类型，目标目录的路径，支持相对或绝对路径。当前工作目录以一个字符串变量记录在File Manager类中，cd对其进行修改。
- `public String mkdir(String dir_path)`：新建文件夹。 dir_path为字符串，即目标新建目录的路径，本系统支持相对或绝对路径。
- `public String mkf(String file_path, String file_type, String size)`：创建文件。file_path，字符串类型，创建文件的路径，支持相对或绝对路径。 file_type，size，字符串类型，指定文件的类型、大小。
- `public String rm(String file_path, String mode)`：删除文件。 file_path，字符串类型，创建文件的路径，支持相对或绝对路径。 mode，字符串类型 rm的功能模式：为空时表示删可读的文件，‘-r’删空文件夹，‘-f’强制删文件，‘-rf’强制删文件夹。‘-rf’的实现思路：递归地对该目录进行rm操作，当文件是非空目录时 ，进入其中；当文件是空目录时 ，‘-r’删除；当文件是普通文件时，‘-f’删除。

- `public String vi(String path)`：编辑文件。 path，字符串类型，创建文件的路径，支持相对或绝对路径。该函数实现的功能是在终端中打开指定文件并允许用户编辑文件内容，支持保存、不保存或取消操作。

  实现思路如下：

  1. 根据给定的相对或绝对路径，拼接出完整的文件路径。
  2. 判断文件是否存在且不是目录，如果是目录则返回错误信息。
  3. 读取文件内容并将其转换为 JSON 对象。
  4. 判断文件是否可编辑，如果不可编辑则返回错误信息。
  5. 将 JSON 对象中的 content 字段转换为字符串数组，并将其拼接为一个字符串用于在文本框中显示。
  6. 弹出编辑窗口，让用户编辑文件内容并选择保存、不保存或取消。
  7. 如果用户选择保存，则将文本框中的内容写回文件，并进行磁块等的重新分配。
  8. 如果用户选择取消，则返回相应信息。
  9. 如果出现异常则返回错误信息。
- `public Map<String, String> getFileInfo(String fileName) throws IOException`:fileName，字符串类型，目标文件的路径。返回值包含文件名、文件大小和文件类型（如果类型包含“e”，则还会包含优先级）的Map类型变量，throws IOException 读取文件时可能发生的IO异常。这个函数用于获取一个文件的信息，并将这些信息以Map的形式返回。如果文件类型包含“e”，即可执行文件，则还需获取文件的优先级信息。

  实现思路如下：

  1. 构造文件路径，并读取文件内容到字符串变量jsonStr中。
  2. 将jsonStr字符串转换为JSON对象。
  3. 从JSON对象中获取文件名、文件大小和文件类型等信息。
  4. 如果文件类型包含“e”，则还需获取优先级信息，将这些信息存储在Map变量中，并返回该Map变量。
  5. 如果文件类型不包含“e”，则直接返回null。
- `public List<String> readContentFromFile(String filePath)`:fileName，字符串类型，目标文件的路径。这个函数用于获取一个文件的内容，并将这些信息以Map的形式返回。实现思路同上。

6.内存管理总体设计

连续存储方式和页式存储方式

连续存储表类

1.任务概述

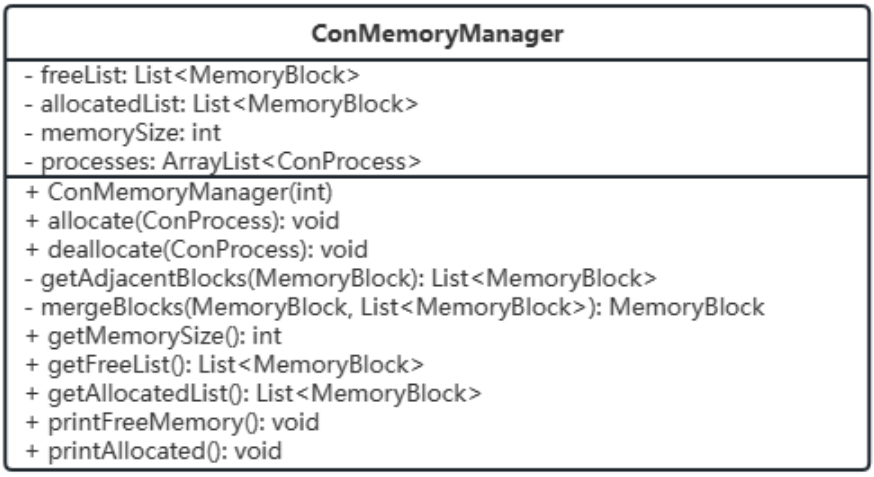
内存管理模块在该系统中主要负责对系统内存资源的同意调配与管理，Kernel会向内存管理模块提出请求，通过虚拟内存机制，对可执行文件在内存里进行有效的装载与回收。并且，内存管理模块能够分别选择模拟连续分配式存储管理与分页式存储管理。

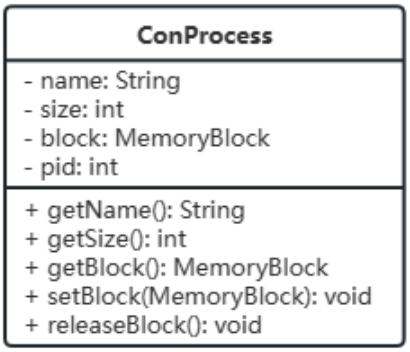
2.结构设计

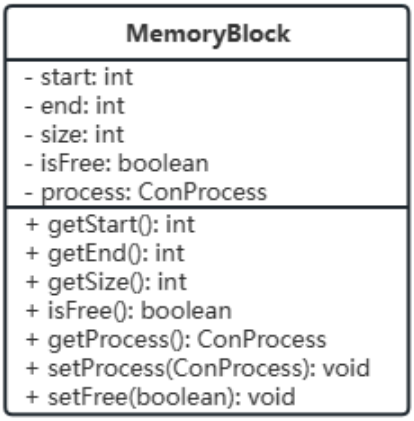
2.1类

2.1.1 连续分配式存储管理

类图







1. ConProcess 和 MemoryBlock 之间是一个关联关系，即 ConProcess 对象关联着一个 MemoryBlock 对象，MemoryBlock 对象也关联着一个 ConProcess 对象。
2. ConMemoryManager 和 MemoryBlock 之间是一个聚合关系，即 ConMemoryManager 对象包含了多个 MemoryBlock 对象，但这些 MemoryBlock 对象可以独立存在。
3. ConMemoryManager 和 ConProcess 之间是一个聚合关系，即 ConMemoryManager 对象包含了多个 ConProcess 对象，但这些 ConProcess 对象可以独立存在。

conMemoryManager类

拥有属性

```
private List<MemoryBlock> freeList;
private List<MemoryBlock> allocatedList;
private int memorySize;
private ArrayList<ConProcess> processes = new ArrayList<>();
```

提供操作

```
public ConMemoryManager(int size);  //
public void allocate(ConProcess p);
public void deallocate(ConProcess p);
private List<MemoryBlock> getAdjacentBlocks(MemoryBlock block);
private MemoryBlock mergeBlocks(MemoryBlock block, List<MemoryBlock>
adjacentBlocks)
public int getMemorySize()
public List<MemoryBlock> getFreeList()
public List<MemoryBlock> getAllocatedList()
public boolean access(int pid, int address)
```

ConProcess类

拥有属性

```
private String name; // 进程名称
private int size; // 进程大小
private MemoryBlock block; // 进程所在内存块
private int pid; // 进程id
```

提供操作

```
public ConProcess(String name, int size, int pid)
public String getName()
public int getSize()
public MemoryBlock getBlock()
public void setBlock(MemoryBlock block)
public void releaseBlock()
```

MemoryBlock类

拥有属性

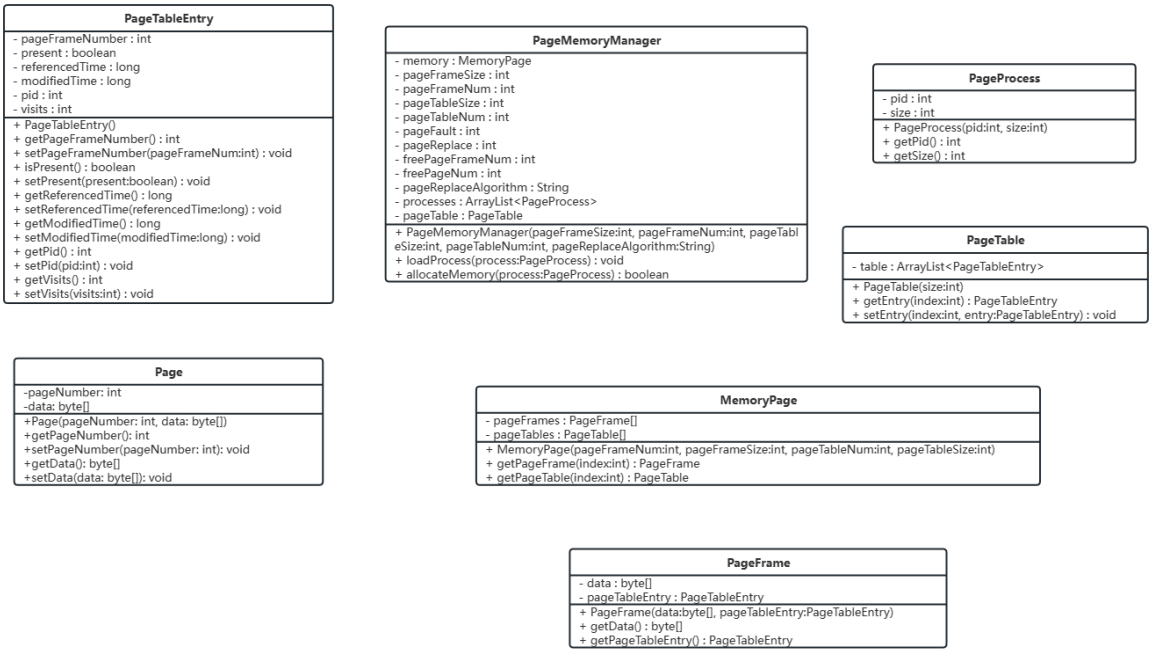
```
private int start; // 内存块起始地址
private int end; // 内存块结束地址
private int size; // 内存块大小
private boolean isFree; // 内存块是否空闲
private ConProcess process; // 内存块所分配的进程
```

提供操作

```
public MemoryBlock(int start, int size)
public int getStart()
public int getEnd()
public int getSize()
public boolean isFree()
public ConProcess getProcess()
public void setProcess(ConProcess process)
public void setFree(boolean free)
```

2.1.2分页式存储管理

类图



1.PageTableEntry 和 PageTable： PageTableEntry 属于 PageTable，PageTable 包含多个 PageTableEntry。这是一种聚合关系（aggregation）。

2.PageMemoryManager 和 MemoryPage：PageMemoryManager 对象包含 MemoryPage 对象。这也是一种聚合关系（aggregation）。

3.PageMemoryManager 和 PageTable：PageMemoryManager 对象包含 PageTable 对象。这是一种组合关系（composition）。

4.PageMemoryManager 和 PageProcess：PageMemoryManager 对象包含多个 PageProcess 对象。这是一种聚合关系（aggregation）。

5.PageTableEntry 和 PageProcess：PageTableEntry 对象与 PageProcess 对象之间没有直接的关系，它们之间通过 PageTable 以及 PageMemoryManager 进行了关联。

PageTableEntry类

拥有属性

```
private int pageFrameNumber;
private boolean present;
private long referencedTime;
private long modifiedTime;
private int pid;
private int visits;
```

提供操作

```
public PageTableEntry();
public int getPageFrameNumber();
public void setPageFrameNumber(int pageFrameNumber);
public boolean isPresent();
public void setPresent(boolean present);
public long getReferencedTime();
public void setReferencedTime(long referencedTime);
public long getModifiedTime();
public void setModifiedTime(long modifiedTime);
public int getPid();
public void setPid(int pid);
public int getVisits();
public void setVisits(int visits);
```

Page类

拥有属性

```
private int pageNumber; // 页面号
private byte[] data; // 存储的数据
```

提供操作

```
public int getPageNumber();
public void setPageNumber(int pageNumber);
public byte[] getData();
public void setData(byte[] data);
```

PageFrame类

拥有属性

```
private int frameNumber; //   页框号
private int pageFrameSize; // 页面大小
private boolean isFree = true; // 是否空闲
private byte[] data; // 存储的数据
```

提供操作

```
public int getFrameNumber();
public boolean setData(byte[] data);
public byte[] getData();
public boolean isFree();
public void setFree(boolean state);
```

PageMemoryManager类

拥有属性

```
private MemoryPage memory;
private int pageFrameSize;
private int pageFrameNum;
private int pageTableSize;
private int pageTableNum;
private int pageFault = 0;
private int pageReplace = 0;
private int freePageFrameNum;
private int freePageNum;
private String pageReplaceAlgorithm;
private ArrayList<PageProcess> processes =new ArrayList<>();
private PageTable pageTable;
```

提供操作

```
public PageMemoryManager(int pageFrameSize, int pageFrameNum, int pageTableSize,
int pageTableNum, String pageReplaceAlgorithm);
public void loadProcess(PageProcess process);
public boolean allocateMemory(PageProcess process);
public boolean accessMemory(PageProcess process, int address);
public int pageReplaceAlgorithm();
public void releaseMemory(PageProcess process);
public void showProcess();
public void showMemory();
```

PageProcess类

拥有属性

```
private String name;
private int pid;
private int size;   // 进程大小
private int[] pageDirectory;
```

提供操作

```
public PageProcess(String name, int pid, int size);
public String getName();
public int[] getPageDirectory();
public void setPageDirectory(int[] pageDirectory);
public int getPid();
public int getSize();
```

PageTable类

拥有属性

```
private PageTableEntry[] entries; // 存储页表项的数组
```

提供操作

```
public PageTable(int numEntries);
public PageTableEntry getEntry(int index);
public void setEntry(int index, PageTableEntry entry);
```

MemoryPage类

拥有属性

```
private int pageFrameSize;
private int pageTableNum;
private int pageTableSize;
private PageFrame[] pageFrames;
private Page[] pages;
```

提供操作

```
public MemoryPage(int numFrames, int pageFrameSize, int pageTableNum, int
pageTableSize);
public PageFrame getPageFrame(int index);
public int findFreePageFrame();
public void freeFrame(int index);
public void useFrame(int index);
```

3.模块逻辑

3.1连续分配式存储管理

整个程序的运行逻辑是，先创建一个 `ConMemoryManager` 对象，然后依次创建若干个 `ConProcess` 对象，再将这些进程加入到 `ConMemoryManager` 中，模拟内存的分配和释放过程，并观察空闲内存块和已分配内存块的情况。

ConMemoryManager 的构造函数初始化了一个空闲内存块列表 freeList，用于存储所有空闲内存块，同时初始化了一个已分配内存块列表 allocatedList，用于存储所有已经分配给进程的内存块。ConMemoryManager 的 allocate() 方法实现了内存的分配，当有一个进程需要分配内存时，ConMemoryManager 将 freeList 中的内存块按照指定算法（此处是首次适应算法）依次遍历，找到第一个空闲内存块，将其划分为两部分，一部分分配给进程，另一部分仍然作为空闲内存块。ConMemoryManager 的 deallocate() 方法实现了内存的回收，当一个进程需要释放其占用的内存块时，ConMemoryManager 将其对应的已分配内存块从 allocatedList 中删除，并将相邻的空闲内存块合并成一个更大的空闲内存块，加入到 freeList 中。ConMemoryManager 还提供了一些用于获取内存信息的方法，比如 getFreeList() 和 getAllocatedList() 用于获取 freeList 和 allocatedList 列表，以及 printFreeMemory() 和 printAllocated() 用于打印空闲内存块和已分配内存块的详细信息。

3.2分页式存储管理

整个系统的运行逻辑如下：

1. 初始化一个PageMemoryManager对象，设置物理内存的大小、页框的大小和数量、虚拟内存的大小和数量、页表的大小和数量、页面置换算法等参数。
2. 调用loadProcess方法将一个进程装载到物理内存中。如果物理内存不足，会报错，否则输出进程ID和他分配的内存信息。
3. 当一个进程需要访问一个虚拟页面时，系统会先检查该页面是否已存在于物理内存中。如果存在，则更新该页表项的访问时间戳和访问次数，并返回该页面的物理地址；如果不存在，则发生缺页中

断，将该页面从虚拟内存中加载到物理内存中，更新该页表项的存在位、页框号、进程id、访问时间戳和访问次数，并返回该页面的物理地址。如果物理内存已满，则需要执行页面置换算法，选择一个页面进行替换。

4. 当一个进程需要释放一个虚拟页面时，系统会将该页面从物理内存中移除，并更新该页表项的存在位和页框号。如果该页面已被修改，则需要将其写回到虚拟内存中。如果该页面所在的进程仍然在运行，则需要更新该进程的页表。

4.管理模式与算法的选择

1.动态分区分配（BF、FF、WF）

Best-fit算法，当Kernel申请存储区时，内存管理模块会选中一个满足要求的最小的空闲区分配给Kernel

First-fit算法，当Kernel申请存储区时，内存管理模块会选中一个最靠前的空闲区分配给Kernel

Worst-fit算法，当Kernel申请存储区时，内存管理模块会选中一个满足要求的最大的空闲区分配给Kernel

2.页式分配（FIFO、LRU、OPT）

采用“按需分页”，只有当进程的某一页被访问时，才会将这一页调入物理内存中。存储一个页表，标志位初始值为-1，即未被引用过

FIFO，先进先出算法，当Kernel申请存储区，总是淘汰最先进入内存的页面,即选择在内存中驻留时间最久的页面予以淘汰

LRU，最久未使用算法，当Kernel申请存储区，以过去预测未来，选择之前最长时间未使用的页面置换

OPT，最佳页面置换算法，在每次需要替换页面时，选择最长时间内不会被使用的页面进行替换。