

**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**

|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | **班级** | **学号** | **备注** |
| **李祥宇** | **2020211305** | **2020211375** | **组长** |
| **李润杰** | **2020211304** | **2020211374** |  |
| **孟宇航** | **2020211305** | **2020211377** |  |
| **覃韦唯** | **2020211305** | **2020211381** |  |
| **马天成** | **2020211305** | **2020211376** |  |
| **锁佳鑫** | **2020211305** | **2020211380** |  |

**计算机学院**

**2023 年 5 月**

**目录**

[1．课程设计题目 3](#_Toc134738784)

[2．课程设计目标和要求 3](#_Toc134738785)

[3．需求分析 3](#_Toc134738786)

[4．开发环境 4](#_Toc134738787)

[5．总体设计 5](#_Toc134738788)

[6．详细设计 5](#_Toc134738789)

[6.1 OS-shell 5](#_Toc134738790)

[6.2 OS-kernel 6](#_Toc134738791)

[6.2.1类设计 6](#_Toc134738792)

[6.1.2 服务设计 7](#_Toc134738793)

[6.3 Process-manager 8](#_Toc134738794)

[6.3.1 基本模块分析 8](#_Toc134738795)

[6.3.2 模块&服务逻辑详细设计 9](#_Toc134738796)

[6.3.3 用户接口设计 14](#_Toc134738797)

[6.4 Memory-manager 14](#_Toc134738798)

[6.4.1 类设计 14](#_Toc134738799)

[6.4.2 服务设计 16](#_Toc134738800)

[6.5 File-manager 20](#_Toc134738801)

[6.5.1 主要模块设计思路 20](#_Toc134738802)

[6.5.2主要实现功能 21](#_Toc134738803)

[6.5.3对外主要接口 23](#_Toc134738804)

[6.6 I/O device 23](#_Toc134738805)

[6.6.1 类设计 24](#_Toc134738806)

[6.6.2 服务设计 26](#_Toc134738807)

[6.7 UI设计 29](#_Toc134738808)

[7．程序清单 29](#_Toc134738809)

[8．测试报告 29](#_Toc134738810)

[8.1 模块测试 29](#_Toc134738811)

[8.1.1 Process\_manager本地模块测试 29](#_Toc134738812)

[8.1.2 Process\_manager本地模块测试 30](#_Toc134738813)

[8.1.3 Process\_manager本地模块测试 30](#_Toc134738814)

[8.1.4 Memory\_manager本地模块测试 34](#_Toc134738815)

[8.1.5 File\_manager本地模块测试 35](#_Toc134738816)

[8.1.6 Device本地模块测试 37](#_Toc134738817)

[8.2 总体测试 41](#_Toc134738818)

[9．课程设计总结 43](#_Toc134738819)

[9.1 设计过程中遇到或存在的主要问题及解决方案 43](#_Toc134738820)

[9.2 改进建议 44](#_Toc134738821)

[9.3 体会/收获 45](#_Toc134738822)

# 1．课程设计题目

设计并实现一个具有操作系统基本功能的软件。

# 2．课程设计目标和要求

目标：需要设计一个具有操作系统基本功能的软件。

基本功能主要包括以下几个方面：

1. 进程管理功能：进程创建（new）、进程调度（scheduling）、进程阻塞（block）、进程唤醒（wakeup）、进程同步（synchronous）等。
2. 内存管理功能：进程存储空间的分配和回收，空闲空间的管理等。
3. 文件系统：目录/文件的创建和删除、空间分配和回收
4. 设备管理：设备的申请、分配、使用、释放等
5. UI界面
6. 中断机制

# 3．需求分析

我们首先进行的是操作系统的整体需求分析，经过小组讨论，我们从以下六个方面进行分析。

1. **功能需求分析**：确定模拟操作系统需要支持哪些模块，包括进程管理、内存管理、文件管理、设备管理等。对于每个模块，需要进一步分析其具体的子功能和实现方法。模拟操作系统的功能应该与实际操作系统的功能相似或相同。
2. **用户界面需求分析**：模拟操作系统需要有一个易于使用的用户界面，以便用户能够轻松地操作模拟操作系统。用户界面应该包括菜单、按钮、输入框等常见的控件，并且应该有相应的交互反馈。
3. **可维护性需求分析**：确定模拟操作系统需要具备的可维护性指标，包括易于调试、修改和扩展等。需要考虑到系统的设计和实现，以确保系统的可维护性。
4. **成本需求分析**：确定模拟操作系统的开发、维护和使用成本。需要考虑到系统的开发周期、开发团队的规模和技能、硬件和软件成本等因素。
5. **学习需求分析**：模拟操作系统应该有一定的学习价值，可以帮助用户了解操作系统的工作原理、内部结构、算法等。需要考虑到不同用户的学习需求和程度，以提高用户的学习效果和兴趣。

在进行了上述的综合分析之后，我们确定了模块划分：分别包括**shell，kernel，process\_manager，memory\_manager，file\_manager，device\_manager**六个具体要实现的模块功能，其中shell是人机交互的媒介，用户的指令通过shell进行获取和识别，在识别了用户的指令之后，进入到kernel中进行实现。Kernel作为操作系统的内核部分是整个程序的核心，但是更进一步的分析可以知道，kernel本身是对剩余模块的统一调度和管理，通过协调不同模块之间的关系，实现对系统进程，内存，文件，设备等的统一管理，保证系统的稳定运行。

进一步考虑，file\_manager主要负责的是对文件进行逻辑组织，通过目录这一结构实现文件系统的有序管理，除了文件的逻辑管理，在本程序中，存取文件时进行的内存交互也是在该模块中进行实现，但是，为了实现生产者消费者这一问题，我们在系统中假设了一个缓冲区buffer，该缓冲区的作用就是文件系统进行生产，即从磁盘中取出文件加载到buffer中，内存系统进行消费，即从buffer中取出内容加载到内存中。

对于process\_manager来说，进程是一个正在运行的程序实例。每个进程都有自己的地址空间、内存和资源需求，以及一组线程和相关的状态信息。process\_manager需要对进程进行管理，以确保系统资源的合理分配和使用。

上述为简单的对主要模块进行描述，下面是各模块的详细功能。

|  |  |
| --- | --- |
| **模块名称** | **主要功能** |
| Shell | Shell是操作系统中的一个命令行解释器，它是用户与操作系统之间的接口，用户可以通过Shell输入命令来控制和操作计算机系统。总的来说为用户提供了一个方便、灵活和强大的操作系统接口，用户可以通过Shell完成各种系统管理和任务处理操作，提高工作效率和准确性。 |
| Kernel | Kernel（内核）是操作系统的核心部分，是操作系统中最基本的部分之一。它通常是由操作系统内置的一个软件模块，用于管理操作系统的所有硬件和软件资源，并提供各种系统服务和功能。 |
| Process\_manager | 进程管理所具有的主要功能为创建进程，为每个进程按照需求分配对应的设备资源，对创建出来的多个进程采用一定的调度算法进行进程调度，以实现单个CPU下进程的有序进行，并在抢占时对被抢占进程及时进行进程阻塞，在时机合适时及时将阻塞进程唤醒，在进程执行结束或用户强制关闭进程时及时回收进程所占用的内存和设备资源。总体来说就是管理整个进程的生命周期以及设备资源的分配情况 |
| File\_manager | 文件系统是操作系统中的一个组成部分，它的主要作用是为用户和应用程序提供一个统一的接口来管理和存储文件，以及提供文件的访问和共享功能。这里主要是通过对FCB的管理，构建树型目录结构，来实现对文件的逻辑组织。初次之外，也能够实现与磁盘之间的读写操作，负责对磁盘的访问。 |
| Memory\_manager | 内存模块主要实现的共嗯那个是响应来自内核的内存分配和释放请求，对内存资源进行统一管理，除此之外，虚拟内存机制的实现也是由该模块负责，包括逻辑地址与物理地址之间的转换，选择合适的虚拟页的替换算法等。 |
| Device\_manager | 设备管理是操作系统中非常重要的一部分，它提供了设备的发现、配置、驱动程序管理、分配和释放、状态监控和错误处理、性能优化等功能，保证了设备与操作系统的兼容性、稳定性和可靠性，提高了系统的效率和响应速度。 |

# 4．开发环境

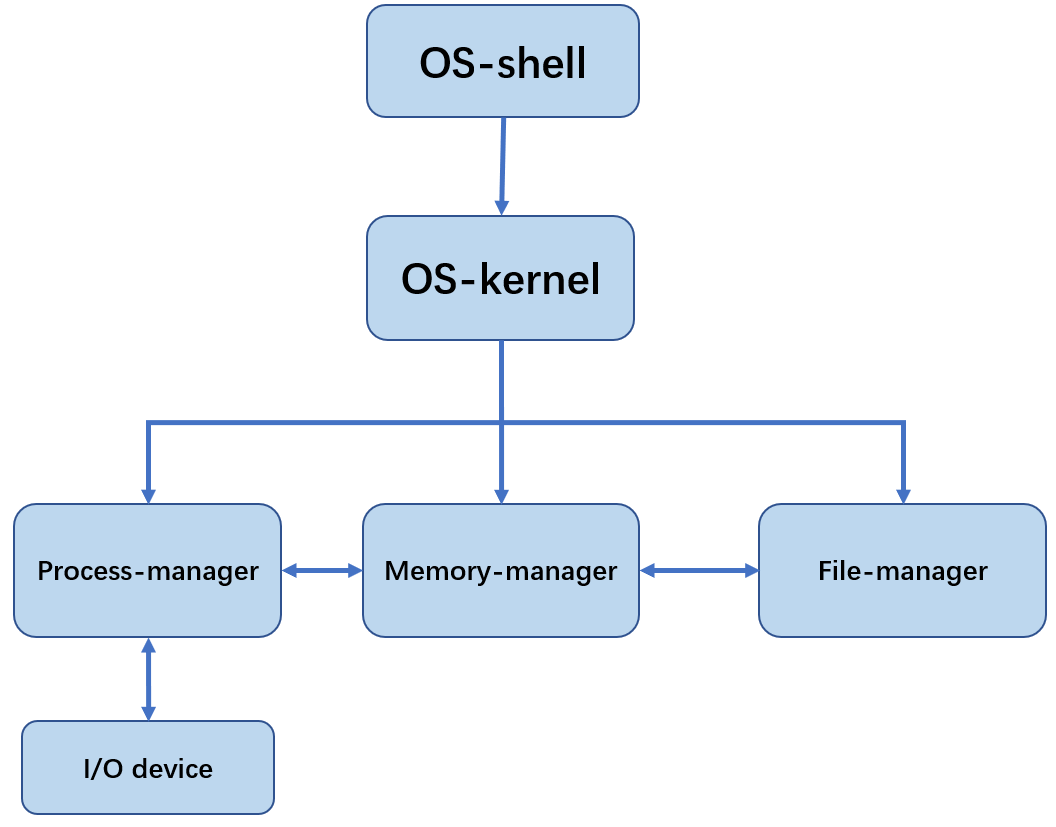
该程序的开发环境为python3.8+PyQt5

# 5．总体设计

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五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五部分功能：

1. **进程管理**
2. **内存管理**
3. **文件管理**
4. **I/O管理**
5. **UI设计**

在分析了具体的课程设计要求之后，我们通过讨论后认为通过模拟的方式进行实现，即在已有的操作系统为底层架构，在其之上实现较为简便的操作系统模拟程序，该模拟程序能够实现操作系统本身具有的五大基本功能，并且不同模块功能之间，能够相互协作，共同实现相应的操作系统的功能。具体的目标设计如下图所示。



# 6．详细设计

## 6.1 OS-shell

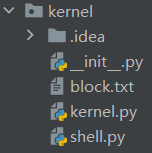
Shell是操作系统中的一个命令行解释器，它是用户与操作系统之间的接口，用户可以通过Shell输入命令来控制和操作计算机系统。总的来说为用户提供了一个方便、灵活和强大的操作系统接口，用户可以通过Shell完成各种系统管理和任务处理操作，提高工作效率和准确性。

由于shell和kernel部分紧密相连且都是由一名同学做的，所以shell部分内容写在了下面的kernel部分。

## 6.2 OS-kernel

Kernel作为操作系统的内核部分是整个程序的核心， 更是对剩余模块的统一调度和管理，通过协调不同模块之间的关系，实现对系统进程，内存，文件，设备等的统一管理，保证系统的稳定运行。

### 6.2.1类设计



#### 6.2.1.1类设计及相应调用关系

|  |  |  |  |
| --- | --- | --- | --- |
| 类名/中文名 | 功能 | 调用 | 被调用 |
| **report\_error**  显示错误信息 | 用户输入指令不存在时提供错误信息 |  | 1. run模块 |
| **display\_command\_description**  存储指令信息 | 将所有可用指令以及其用法存储在一个列表中，可以用来识别用户输入的指令是否合法以及在用户输入对应指令（help）时打印指令用法 | 1.report\_error  模块 | 1. run模块 |
| **run**  指令执行模块 | Kernel的核心部分，用来接收shell中用户输入的指令并逐条执行 | Kernel中所有模块以及shell、process、memory、file、device模块 |  |
| **print\_system\_info**  打印系统信息 | 打印系统信息，包括系统的名称和现在的时间 |  | 1.run模块 |
| **get\_split\_command**  获取用户输入的指令 | 获取用户输入的指令， |  | 1.run模块 |

#### 6.1.1.2类内部主要数据结构

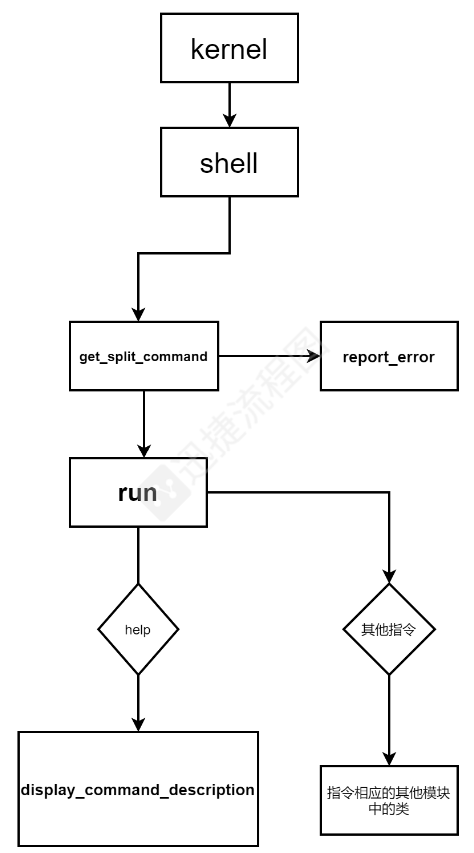
* **command\_description**

**使用列表实现，存储所有可执行指令和他们的用法。**

* **command\_split\_list**

使用字符数组，存储用户输入的所有指令

#### 6.1.1.3 类图



### 6.1.2 服务设计

#### 6.1.2.1 对外提供的服务

**对用户提供的服务**

* **get\_split\_command(self, cwd)**

接收并存储用户输入的指令，两指令间用分号隔开，指令间的字符串以空格或tab隔开

* **run**

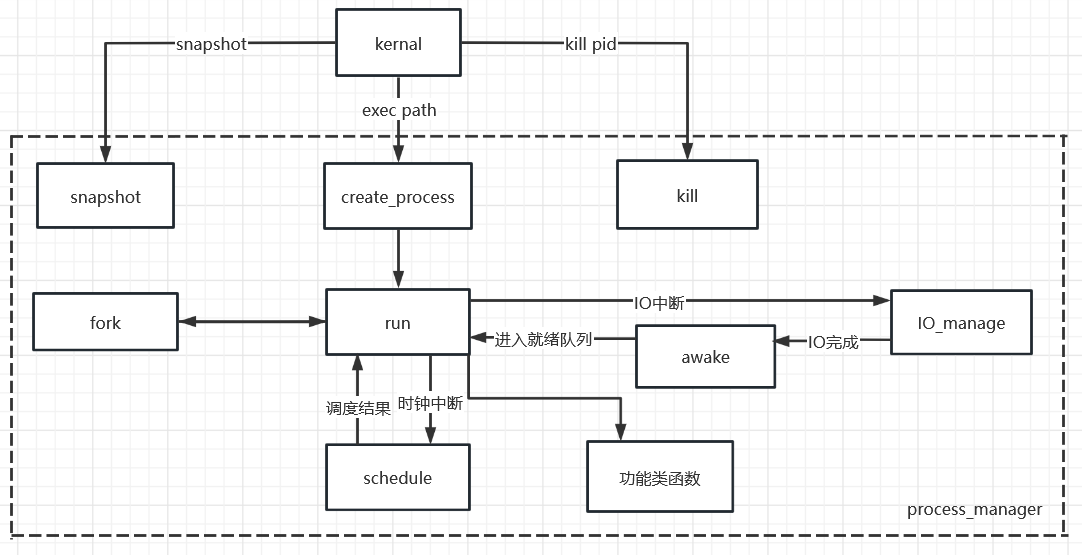
判断用户输入的指令是否正确，如果正确则逐条执行

面向用户指令设计：

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | | 功能 | 主要应用模块 |
| ls [选项] [目录名] | | 列出目录内容 | 文件模块 |
| cd [目录名] | | 更改当前目录 |
| mkdir [目录名] | | 创建新目录 |
| mkf·[文件名] | | 创建新文件 |
| rm [选项] [文件或目录名] | | 删除文件或目录 |
| vi·[文件名] | | 编辑文件 |
| cat [文件名] | | 显示文件内容 |
| save· | | 保存更改的所有文件信息 |
| dseek[选项] | | 寻道模拟 |
| exec path | 创建进程并执行用户所指定的文件路径的文件内容 | | 进程模块 |
| snapshot | 向用户展示各并发进程的状态，大致包括：进程的pid、优先级、创建时间和进程当前时刻所处的状态。 | |
| test | 创建一个最高优先级的进程来测试进程的调度，与exec格式相同 | |
| kill pid | 强行终止一个用户指定pid的未完成的进程，并将其所占用的内存和设备全部释放 | |
| device show | 显示设备管理器的信息（输入输出队列以，现有实例设备以及相关信息） | | 设备模块 |
| display[选项] | 展示系统虚拟内存和物理内存的占用状态 | | 内存模块 |
| help | 展示所有指令用法，可以通过help [cmd1] [cmd2] ...的形式精确查找一条指令的用法 | | Kernel＆shell模块 |
| exit | 退出模拟操作系统MyOS | |

## 6.3 Process-manager

### 6.3.1 基本模块分析



* **create\_process模块**：负责创建一个进程并根据所选择的进程调度算法将新创建的进程放入就绪队列中，先判断进程PCB队列是否已经满了，再判断内存中是否有足够的空间来分配进程，将新创建的进程加入就绪队列后，还要提醒系统有新进程加入，以便在执行schedule模块时判断是否需要调度相应的进程。
* **fork模块**：对于fork指令的处理，在创建时也需要判断PCB队列是否已经满了，内存中是否有足够的内存空间，在满足条件的情况下创建一个父进程pid为当前进程pid，代码段继承自父进程的子进程，用子进程PCB中的pc来记录子进程代码段的执行位置。
* **schedule模块**：调度算法模块，根据当前设定的调度算法对处在就绪队列中的进程进行调度（时钟中断触发），具体的实现细节在详细设计中。
* **I/O\_manage模块**：对于模拟的输出设备进行管理。
* **awake模块**：对于完成IO操作的进程，I/O\_manage模块会产生I/O中断并调用awake模块来唤醒waiting队列中的进程。
* **run模块**：整个进程管理的核心模块，主要功能是根据进程文件内容执行相应的命令，对每一条命令进行处理，每隔固定的时钟需要调用schedule进行进程调度，如果发生了调度就执行新进程内的指令。（简单来说是一个指令处理器）
* **kill模块**：强制终止一个进程，对于处在waiting队列和设备队列中的进程也会进行强制终止。
* **snapshot模块**：输出当前PCB队列中的所有进程的详细信息，同时，将目前所有正在执行的进程的CPU使用时间和IO使用时间以图像的形式绘制出来，相当于对整个系统的当前状态进行一个快照。
* **功能类函数模块**：主要为了实现其他模块的功能而封装的一些函数：判断就绪队列是否为空函数、添加新进程函数、结束进程函数。

### 6.3.2 模块&服务逻辑详细设计

#### 6.3.2.1 PCB结构设计

（不包括\_\_str\_\_、\_\_lt\_\_、\_\_eq\_\_方法）进程PCB的主要属性有：

1. 进程的父进程pid（由操作系统创建的为0，fork创建的则为父进程的pid）。
2. 当前进程的pid。
3. 进程的优先级（值越小优先级越高）。
4. 进程的程序计数器pc（用于记录进程文件内的指令执行到哪一行）。
5. 进程的创建时间。
6. 进程的大小。
7. 进程的类型（是由操作系统创建还是fork创建）。
8. 进程状态（ready、running、waiting）。
9. 进程文件的总行数（进程文件的大小 / 指令所占字节）（指令长度为定长）。

#### 6.3.2.2 进程管理结构设计

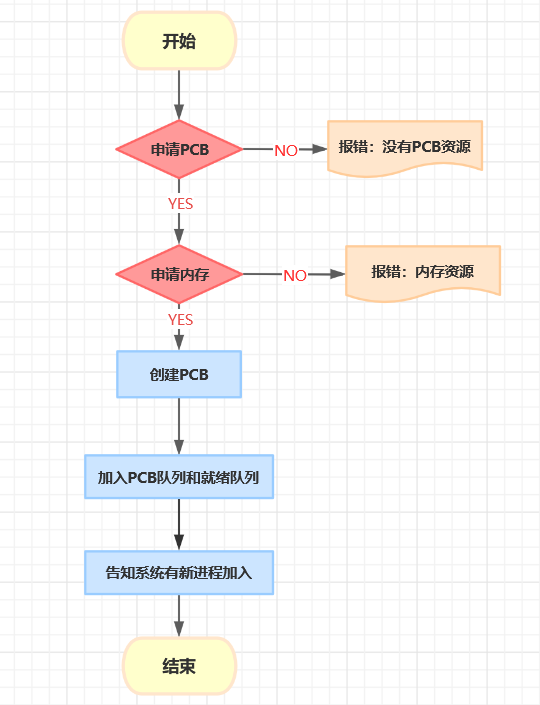
进程管理的主要属性有：

1. 当前分配的pid号（只用于区分不同进程，确保进程号不会重复分配，并不属于进程资源）。
2. 进程的就绪队列（不同的调度算法就绪队列的数据结构不同，从就绪队列获取元素的方式不同，但实际意义相同，是一个存储已经就绪的进程PCB的队列）。
3. 进程调度算法（优先级抢占、优先级非抢占、FCFS、RR）。
4. 当前正在运行的进程cur\_process（有进程处于running状态则为对应进程的PCB，否则为默认PCB，pc=-1）。
5. 进程的PCB队列（PCB属于进程资源，有数量限制）、进程PCB队列的最大数量。
6. 标记是否有新进程加入的标志位new\_process（schedule模块根据标志位调度进程）。
7. 进程等待队列。
8. 记录进程CPU指令情况的字典（主要用于记录进程对CPU的剩余使用时间，用于cpu time指令处理）。
9. 操作系统的系统时间（每产生一个时钟中断系统时间加一）。
10. 记录进程CPU使用时间（列表）和IO使用时间（字典）（用于绘制Gantt图）。

#### 6.3.2.3 进程创建和fork原语设计

进程创建和fork模块实际上是类似的，只不过fork中子进程的一些属性是继承自父进程的，而进程创建中的属性是系统分配的，二者都涉及到申请PCB资源和内存资源的过程，这里就主要说明进程创建部分。

创建进程首先要看PCB队列是否已满，是否有PCB资源可供分配，没有会向用户产生报错信息，如果有PCB资源分配，接下来就要根据进程文件的路径申请内存资源，如果没有足够的内存资源也会产生报错信息，有足够的内存空间才会为进程创建PCB，为了更加真实的模拟实际情况，进程的优先级通过随机数产生，创建好PCB后会将其加入到PCB队列和就绪队列中，并将当前进程的状态置为”ready”，最后还要告知整个系统有新进程加入，以便在下一个时钟周期时，schedule模块在进行进程调度时会考虑新加入的进程。



#### 6.3.2.4 进程的调度算法

对于进程调度算法，我们的最终设想是设计多种调度算法，以option选项的方式供用户选择，调度算法有FCFS、抢占式优先级调度、非抢占式优先级调度和轮询算法。由于操作系统是中断驱动的，因此，每过一个特定的时间周期就会产生一个时钟中断事件，此时就会调用schedule函数去判断是否需要进行调度，new\_process是用来判断在这一个时钟周期内是否有新进程加入到就绪队列中的标志位，每次发生进程调度时，还需要记录上一个进程CPU结束时间和下一个进程CPU开始时间（第一次调度只记录开始信息）以便绘制进程的Gantt图。

对于抢占式优先级调度算法来说，schedule会在有新进程在一个时钟周期内加入到就绪队列中时，去判断**当前就绪队列中**最高优先级进程的优先级是否比**当前正在执行的进程**的优先级还高，如果比当前正在执行的进程优先级要高则发生抢占，修改当前进程的状态并进行上下文保存，然后开始执行新进程的指令。如果没有变化则正在执行的进程不变。

对于轮询算法来说，schedule就是按进程在就绪队列中的顺序轮流进行调度，每过一个时钟周期就修改当前进程的状态并进行上下文保存，如果没执行完则直接放到就绪队列的末尾，然后开始执行新进程的指令。

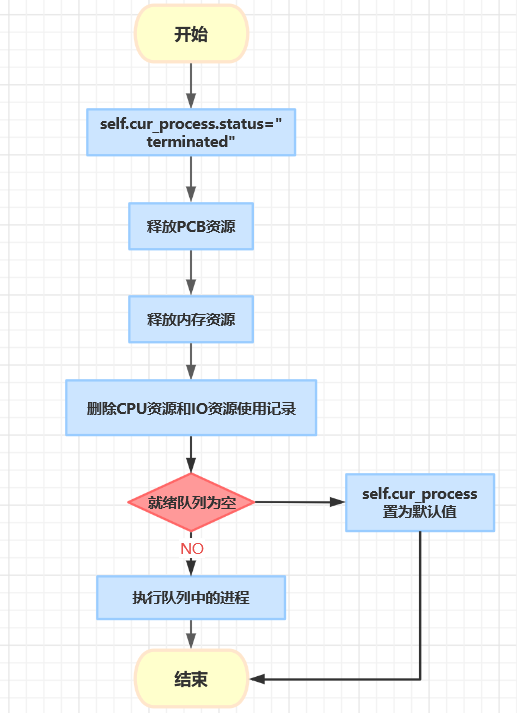
对于FCFS和优先级非抢占算法来说，由于这两个算法都不涉及到抢占，因此只有调入第一个进程时会通过schedule调度，后面进程执行完切换到下一进程时就可以通过end指令来进行自动调度。

#### 6.3.2.5 进程的阻塞、唤醒、结束

进程阻塞主要按照对应的算法规则进行实现，发生抢占导致进程阻塞时，会将进程状态改为ready，然后添加进就绪队列中；产生IO中断而导致进程阻塞时，会将进程状态改为waiting，然后添加进waiting队列中。无论是进程调度阻塞还是IO中断阻塞，进程的程序计数器pc会记录当前指令进行到什么位置。

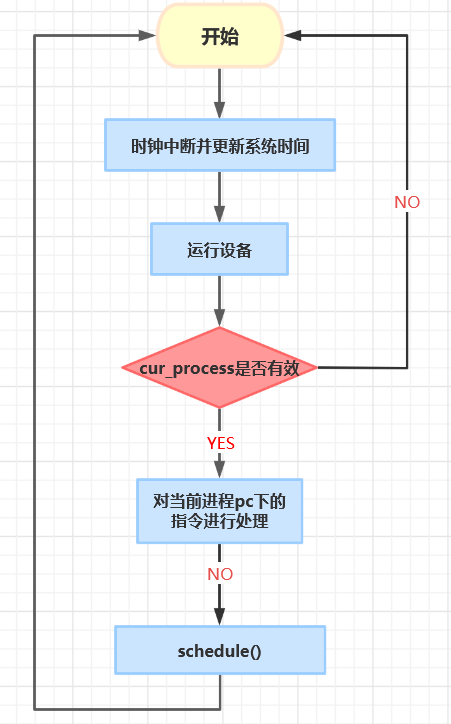
进程唤醒主要通过PCB中的程序计数器pc来进行，每次进程唤醒后都会从pc处开始执行。对于IO唤醒来说，会先将IO中断结束的进程从waiting队列移到ready队列中，并告知系统有新进程加入到ready队列中，然后就由schedule模块进行正常的调度算法。（awake模块）

进程结束或撤销时首先会先修改进程的状态，然后将进程从PCB队列中删除，释放PCB资源，然后根据进程的pid来释放对应的内存空间，删除其CPU资源和IO资源使用记录。完成进程的结束工作后需要将下一个就绪的进程调入到CPU中并修改状态为running，如果就绪队列中没有进程的话记录当前正在执行的进程的变量cur\_process就设置为默认值。下面是进程结束部分的流程图：



#### 6.3.2.6 run模块的结构设计

run模块实际上是一个不断循环的过程，除了每次循环产生一个时钟中断外，其核心功能是对当前正在处理的进程cur\_process进行指令的逐条解析和执行（先判断是什么指令再进行处理），通过cur\_process内的pc是否为-1来判断当前是否有进程正在执行，在run模块对当前指令分析完毕后，会调用schedule函数来判断当前是否会发生优先级抢占以便及时切换当前正在执行的进程，同时，schedule函数可以保证在就绪队列中有进程的前提下cur\_process始终不为默认值。



#### 6.3.2.6 snapshot模块的结构设计

这部分首先是打印进程队列中所有进程的状态情况，为了节省PCB资源，对于已经terminated的进程我们会将其从PCB队列中删除，因此PCB队列中的进程一定是处于running、ready、waiting状态的进程，我们只需要遍历整个PCB队列，逐一打印进程信息即可。

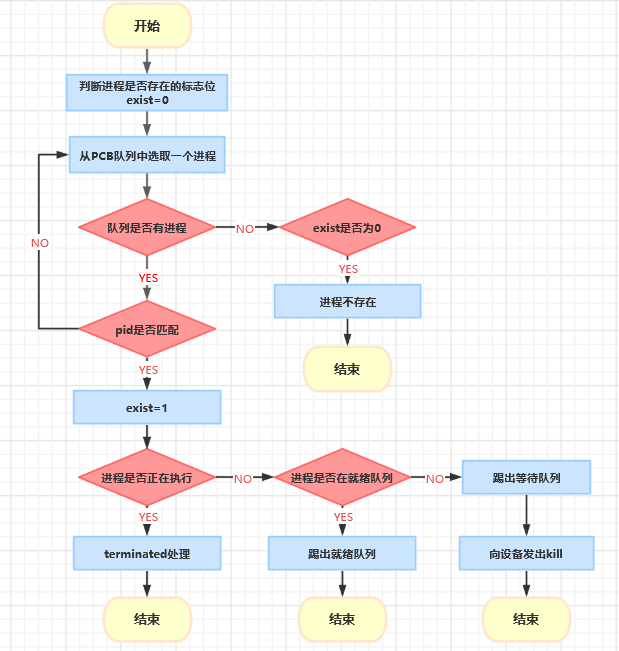
第二部分则是绘制PCB队列中每个进程的CPU使用时间和IO使用时间。IO使用时间我设计了一个字典，字典的key是进程pid，value是一个大列表，包含多个小列表，每个小列表的第一个元素是IO起始时间，第二个元素是IO终止时间，由于一个进程可能会调用多次IO，因此设计了这种二维数组的方法。

对于CPU使用时间，由于只有一个CPU，因此一个进程的CPU使用时间必然是连续的，从起始时间开始，终止时间结束，中间不可能有其它进程的起始时间或终止时间，因此CPU使用时间的设计也是一个大列表包含众多小列表。小列表以两个为单位，第一个小列表的第一个值为进程的pid，第二个值为进程的开始时间，第二个小列表的第一个值也为进程的pid，第二个值为进程的结束时间。

有了每个进程的CPU使用时间和IO使用时间记录情况，剩下的就是利用matplotlib进行绘图。对于调用snapshot时某一进程正在使用CPU和某一进程正在使用IO的情况：根据我的设计思路，此时CPU使用时间的记录信息内有单数个小列表（因为当前进程正在运行，只有开始时间，没有结束时间），我们会人为添加一条[当前进程pid, 当前时间]（以当前时间为结束信息的记录）到CPU使用时间记录信息中，确保绘图时有偶数个记录信息。对于IO记录也是同理，就是将当前时间作为结束时间添加到记录信息中，在绘制完图片后还要去掉我们人工添加的信息。具体的绘图结果和呈现方式详见单元测试部分。

#### 6.3.2.7 kill模块的结构设计

kill模块的设计逻辑也比较简单，首先要遍历整个PCB队列寻找目标进程，如果是正在执行的进程则进行terminated处理；如果进程处在ready队列中则释放掉PCB、内存资源和记录信息后需要将其从ready队列中删除；如果进程处在waiting队列中则释放掉PCB、内存资源和记录信息后需要将其从waiting队列中删除，同时，还要向设备管理部分发出信号，设备那边也要kill掉该进程。



### 6.3.3 用户接口设计

**指令集**

1. **fork**: 模拟进程通过fork原语创建另一个进程，除了ppid、pid外其余都与父进程相同，开始的指令位置与当前父进程的位置相同，时长为1个时钟周期。
2. **cpu time**: 模拟使用cpu，时长为time（整型）个时钟周期。
3. **read filename**: 从给定文件名中读取文件的内容，时长为1个时钟周期。
4. **output info**：设备I/O指令，将information信息加入到设备队列中，时长与设备队列的长度有关。
5. **end**: 进程结束，进行进程资源的释放工作和执行下一个进程的准备工作，时长为1个时钟周期。

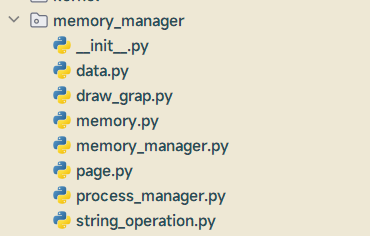
**用户指令**

* **exec path**：创建进程并执行用户所指定的文件路径的文件内容。
* **snapshot**：向用户展示各并发进程的状态，大致包括：进程的pid、优先级、创建时间和进程当前时刻所处的状态，并且将目前所有正在执行的进程的CPU使用时间和IO使用时间以图像的形式绘制出来。
* **kill pid**：强行终止一个用户指定pid的未完成的进程，并将其所占用的内存和设备资源全部释放。

## 6.4 Memory-manager

本模块实现对系统内存资源的统一调配与管理，使用请求分页系统实现对内存的分配及释放，实现了虚拟内存。页大小以及总内存大小是可配置的。当收到进程的内存请求时，为其实现单独的虚拟内存管理，这以类的方式实现。请求页表以字典的形式实现其用户程序虚拟内存与物理内存的映射以及记录有效位。针对页表的操作（插入新页/删除页/修改有效位/查询页）则作为这个类的方法。

### 6.4.1 类设计



#### 6.4.1.1 类设计及相应调用关系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 文件名/中文名 | 类名/中文名 | 功能 | 调用 | 被调用 |
| **Memory\_manager**  内存管理模块 | **MemoryManager**  内存管理器 | 负责整个内存模块的调度，实现进程的内存分配，访问和释放功能；负责管理虚拟内存和物理内存，是与磁盘和进程模块连接的接口。 | 1.内存模块  2.进程内存管理模块  3.绘图模块  4.字符串操作模块 | 1.外部进程模块  2.外部文件模块 |
| **Process\_manager**  进程内存管理模块 | **ProcessManager**  进程内存管理器 | 管理每个进程的页表，页表储存着每页的物理内存地址，虚拟内存地址，有效位等信息。以及维护LRU队列，当进程访存时，返回给上层需要换入换出的页号。 |  | 1.内存管理模块 |
| **memory**  内存模块 | **Memory**  内存条 | 代表“内存”，每个实例即是一整块的内存空间，其根据上层模块发出的信号给出空闲的内存空间并；或是释放指定的内存空间；或是访问。 | 1. 内存页模块 | 1. 内存管理模块 |
| **Page**  内存页模块 | **Page**  内存页 | 代表内存中的每一页，存有Data | 1.内存数据模块 | 1.内存模块 |
| **Data**  内存数据模块 | **Data**  数据流 | 实际上的数据信息 |  | 1.内存页模块 |
| **Draw\_grap**  绘图模块 | **无** | 绘制内存信息 |  |  |

#### 6.4.1.2 类内部主要数据结构

* **LRU队列**

每个进程一个。**使用列表实现，下标越大则代表访问时间越靠后。**

* **页表**

每个进程一个。**使用列表实现，每一页的表项有[虚存页号， 物理内存页号，有效位， 是否被修改]。**

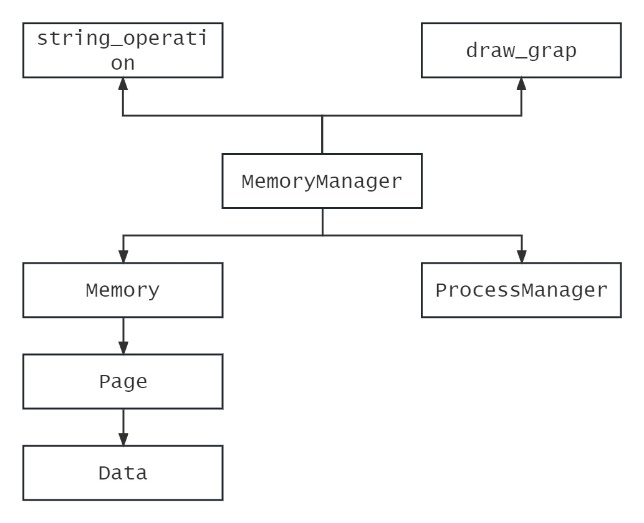
* **文件字典**

每个进程一个。使用字典实现。File[fileName] = [首地址，长度，该文件在磁盘中块号，是否为代码段]。进程每读入一个文件，就写入一项，并依据首地址和长度将该文件写入内存中。

* **Data**

存储在内存页中，每个页有一个Data字典，键为该数据首地址，值为Data对象。Data对象有两个属性，其一为文件内容，其二为文件长度。

#### 6.4.1.3 类图



### 6.4.2 服务设计

#### 6.4.2.1 对外提供的服务

* 对进程系统提供的服务

1. **Access(pid, address, len)**：访存，pid为进程id，地址为该进程空间内地址。
2. **freeAllMem(pid)**：释放该进程所有内存空间。
3. **alloc(pid, path)**：进程给出代码段路径，内存发出请求文件的申请，获取到代码后申请对应的内存空间，并写入代码段，返回给进程状态码。
4. **alloc\_fork(fpid, cpid)**：主进程fork子进程，复制该进程的页表，文件字典，依据文件字典为子进程内存空间依次写入相同数据。

* 对文件系统提供的服务

1. **read\_file(pid, path, data)**：提供给文件系统调用，主要用于vi打开文件。
2. **out\_file(pid, path)**：主要用于vi修改完成后，写回文件系统。

#### 6.4.2.2 外部交互数据类型设计

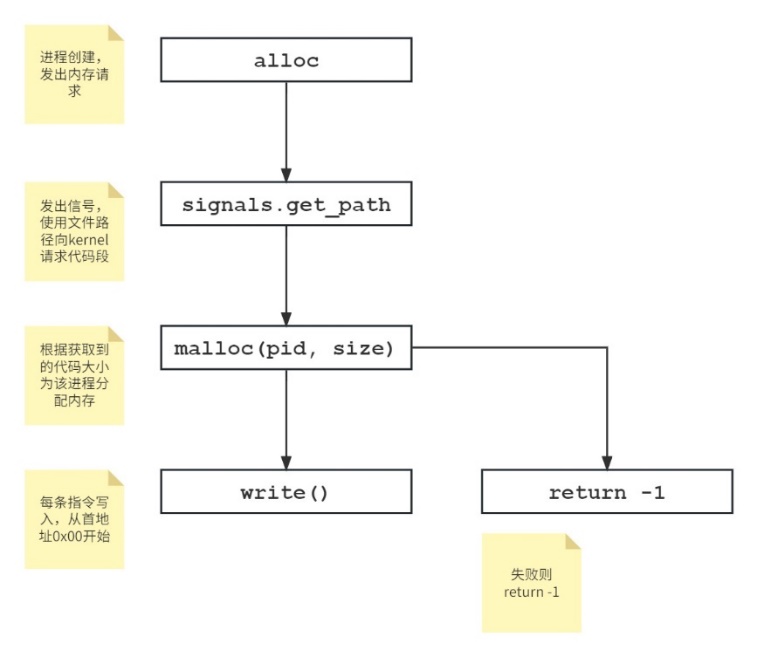
主要与文件交互时需要传输较多数据。

1. 当文件传入内存信息时，数据结构为数组，[文件内容，文件大小，文件所在块号]
2. 当内存写回文件时，数据结构为数组，[文件路径，文件内容]
3. 文件内容和文件路径均为字符串
4. 当进程从内存读指令或者读文件时，传回对应的代码或文件内容，为字符串

#### 6.4.2.3 对外接口调用流程图

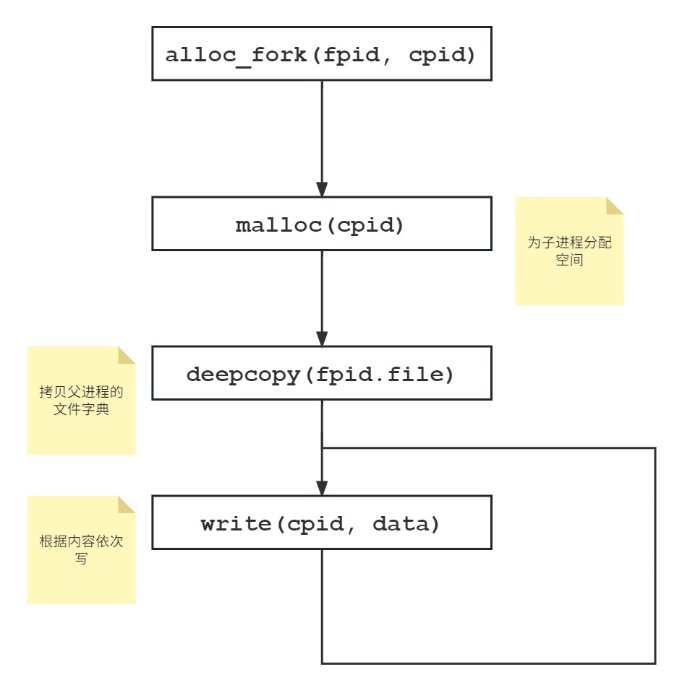
* ALLOC

进程创建时首先调用alloc函数，内存模块收到指令后发出信号，获取到代码段，并检查是否有空余内存，如有，则分配，并将代码段写入内存。如果未获取到代码段或者剩余空间不足则返回-1。

****

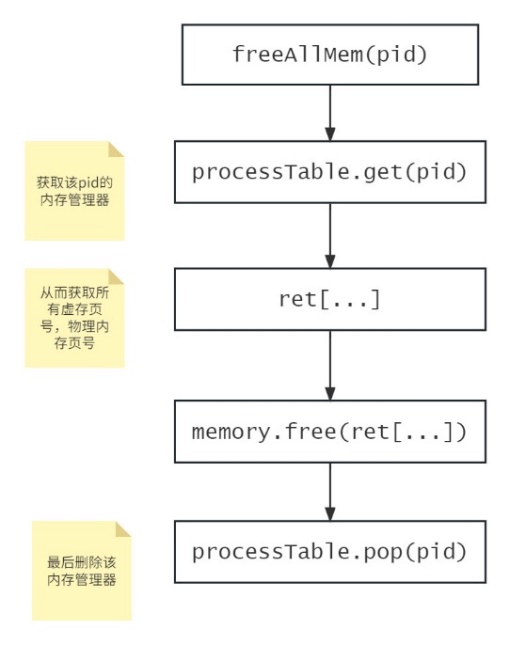
* ALLOC\_FORK

首先分配和父进程一样的内存空间大小，然后使用copy库中的deepcopy函数拷贝父进程的文件字典。遍历字典，读出每一个在父进程中的文件，包括代码段。然后依次写入子进程内存空间的相同位置。



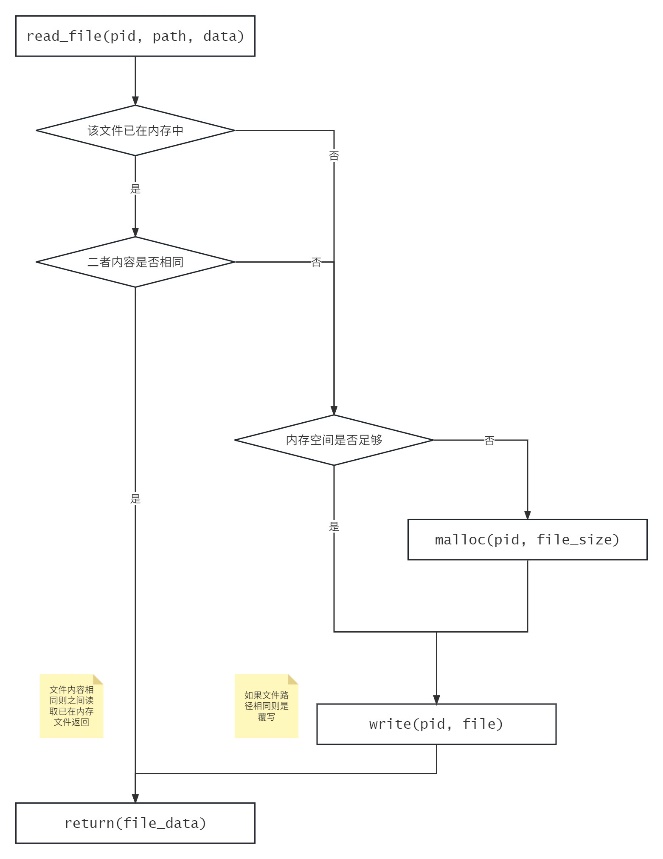
* FREEALLMEM

先获取该pid的内存管理器，在其页表中可以找到所有其使用的物理内存和虚拟内存页号，将其组合成数组返回，调用memory.free方法依次清空被占用的页，再在字典中删除该pid对应的管理器即可。



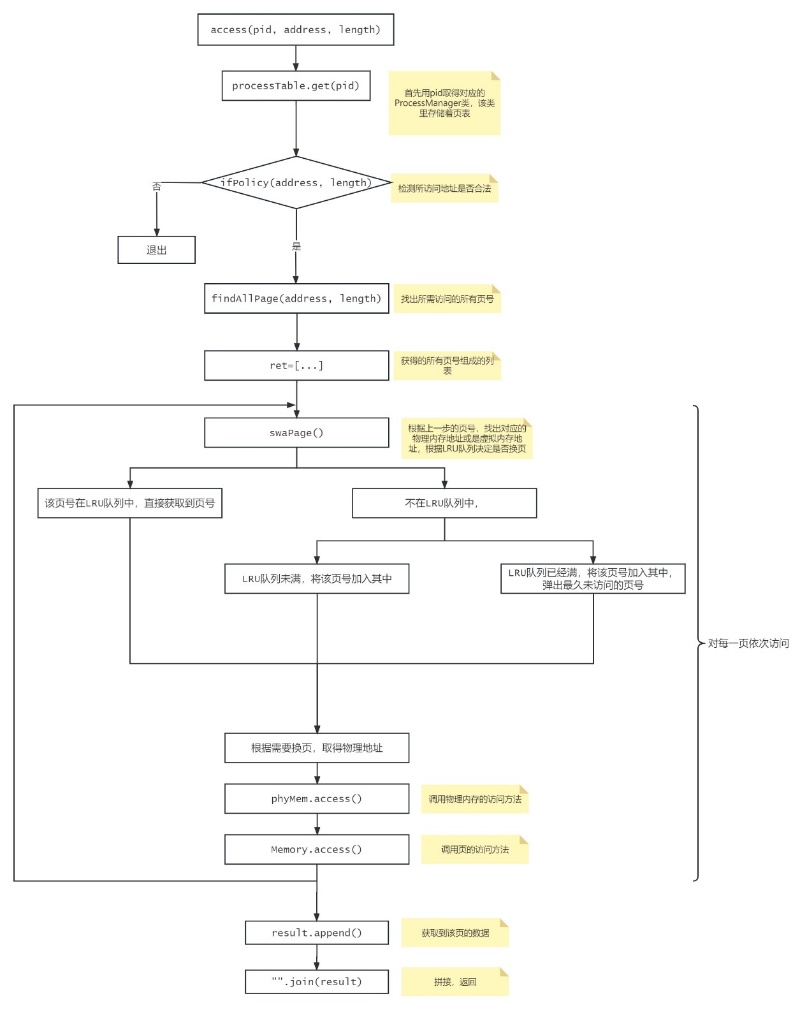
* READ\_FILE

当获取到读文件请求时，先在该pid的内存管理器中的文件字典寻找该文件，如果找到则读出，判断其与传入的文件内容是否相同，如果相同，之间返回即可，不必重复写。如果不相同，则覆写原文件。如若找不到，则新写入该文件。



* **ACCESS**

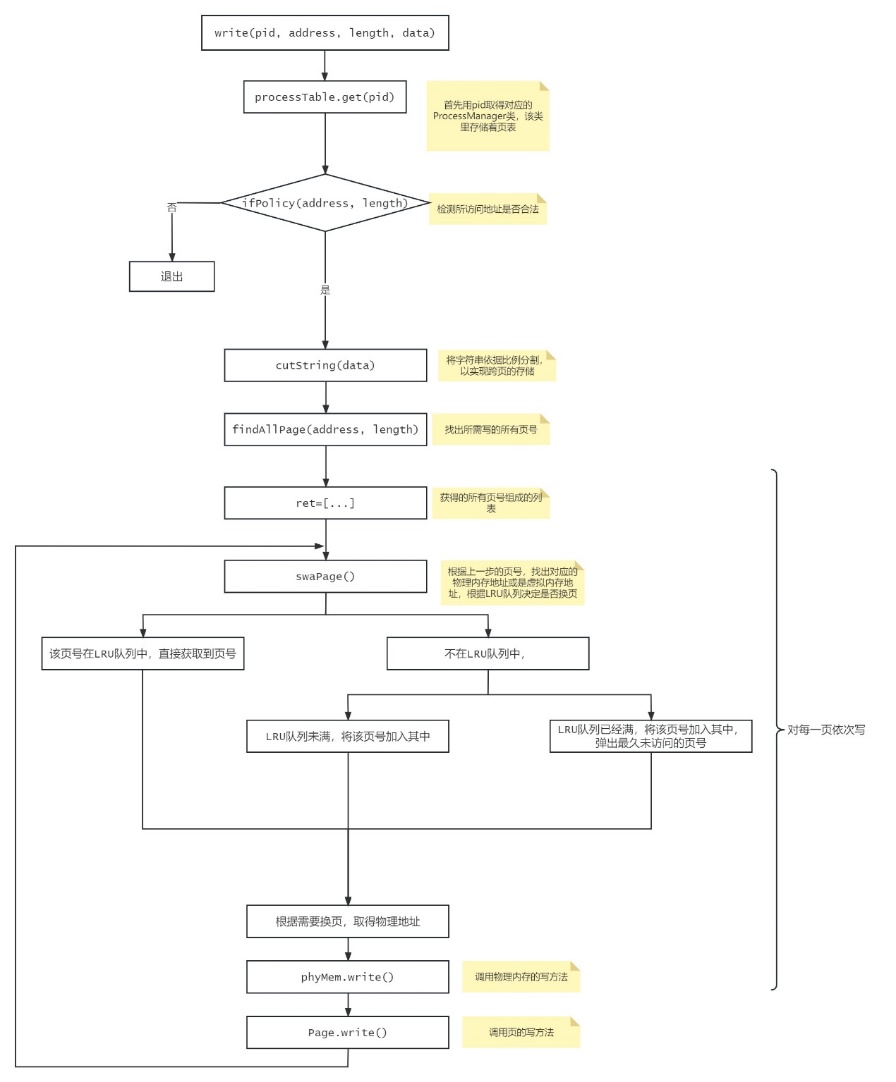
流程较为复杂，需要先判断访问地址是否合法，合法后再将其调入物理内存，这其中涉及到LRU队列，物理内存是否充足，以及被换出的页是否被修改的问题。当其最终被换入后，调用物理内容的access访问。

****

#### 6.4.2.4 内部重要方法流程图

* WRITE

和ACCESS相似，不再赘述。



## 6.5 File-manager

### 6.5.1 主要模块设计思路

**目录结构**

采用树形目录结构，可以明显提高对目录的检索速度和文件系统的性能，利用字典树的形式，来实现树形目录结构。一个文件对应字典中的一个元素。元素的键对应文件名，元素的值保存的则是文件的FCB。这里的FCB包含着文件在磁盘中的位置，设计参考UNIX系统，即FCB初步设计包含以下的属性：文件名、文件的索引节点。这样即可减小FCB的大小，可使查找文件的平均启动磁盘次数减少，节省系统开销。

而FCB本身是存储在一个文件中，每次在进行目录初始化的时候就是对该FCB文件进行读取，该文件是存储在磁盘当中。

这里我们的目录就是通过字典进行存储，其中如果创建的是子目录，那么存储的仍然是一个字典，如果创建的是一个文件，那么就是存储该文件对应的FCB。

**磁盘**

使用一个固定大小的文件进行磁盘模拟，模拟的磁盘块用来存储一下内容：

1. 磁盘的整体信息，包括磁道数，扇区数，数据块数量，数据块大小等磁盘的基本信息
2. 目录逻辑结构的存储，文件的FCB等信息
3. 文件内的数据存储

在本操作系统中，我们假定磁盘只有一个分区，对于每个数据块，我们通过两个属性进行定义，rack和secs这两个属性来确定数据块的位置，其中track模拟的是磁盘中的磁道号，secs模拟的是磁盘中的扇区号，（track，secs）这一向量可以唯一确定一个数据块，地址的转换模拟则是通过将文件所在的块号转换为对应的磁道号和扇区号来进行的。文件系统中包括的属性有块的总数，空闲块的数量和空闲块的位置，目录结构以及各个文件的位置。

**磁盘类代码包含磁盘的磁道数和扇区数，通过这两个数据计算得出整体的磁盘块数，对于磁盘块具体实现，我们设计了Block类，代表磁盘块，一个Block类对象代表一个磁盘块。同时磁盘采用位示图法管理块是否分配，通过freemap这个二维数组进行磁盘是否分配的标记。磁盘块包含磁盘块的大小和文件项，初始设计文件都为空，若存入文件将文件项替代即可**

**文件**

1. name 表示文件名，文件名应保持同目录下唯一性。文件名可以由除了 / \ : \* " < > | ？ 这九个字符以外的其他字符组成。特别：当文件名以“.”开头时，该文件被视为隐藏文件
2. type 表示文件类型
3. size 表示文件所占磁盘空间大小，单位为 byte
4. prmis 表示访问权限，以字符串形式存储，计划设置权限rwe（r为读权限，w为写权限，e为执行权限）
5. ctime 创建时间，记录首次创建文件时间
6. mtime 修改时间，记录最后一次修改文件时间
7. content 为模拟文件的内容

**文件控制系统**

文件控制系统主要是对整个文件系统的变量进行管理，并且提供相应的外部接口函数。首先读取存储于磁盘文件中的字典树，生成目录字典，之后初始化当前目录和当前路径。

### 6.5.2主要实现功能

* **ls(mode='')**

描述：列出目录下文件。根据不同模式，有不同效果。mode为模式参数，根据参数实现指定功能。

mode相关参数：

1. -a 显示所有文件及目录 (. 开头的隐藏文件也会列出)
2. -d 只列出目录（不递归列出目录内的文件）。
3. -l 以长格式显示文件和目录信息，包括权限、所有者、大小、创建时间等
4. 为空默认显示非隐藏文件和目录

代码设计思想：根据当前路径获取当前目录字典，展示目录字典中的key值，即展示文件和目录名。对于具体的信息，就需要通过目录字典存储的fcb，读出文件块位置，进入磁盘块中读取文件的具体信息，至于目录信息，存在‘\目录名’的fcb记载目录的信息

* **cd(path='')**

描述：改变当前工作目录。其中path传入参数 可为绝对路径或相对路径。若目录名称省略，则回到根目录。. 则是表示目前所在的目录， .. 则表示目前目录位置的上一层目录，无参数则会回到根目录。

代码设计思想：根据文件系统的设计的当前目录字典和当前路径，若相对寻址，则从当前的目录字典开始往下寻找，同时改变当前路径；若绝对寻址，则从根目录出发，一层一层寻找目标路径。

* **mkdir(name,mode=’’,prmis='rwe')**

描述：创建目录。mode代表创建模式,默认为空，直接创建，若已存在，则不会再次创建； prmis为用户权限 ‘r‘ 为读权限，’w‘ 为写权限，’e‘ 为执行权限，默认创建者拥有全部权限。

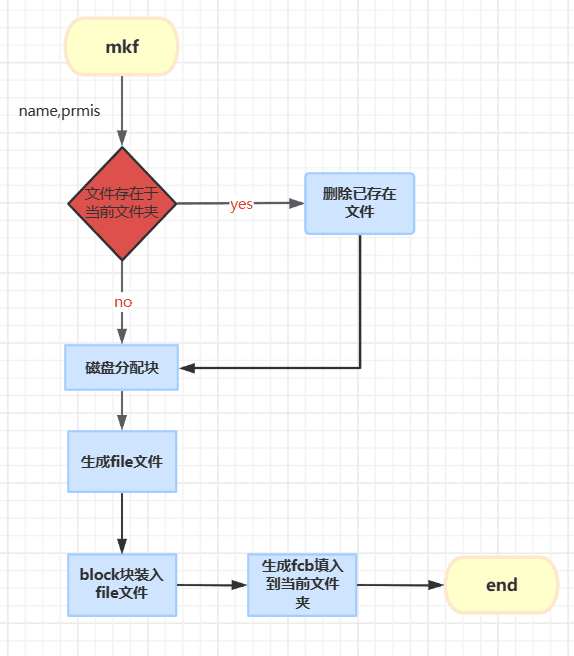
mode相关参数：为空则确保目录名称存在，不存在的就建一个。-t 创建目录，若目录已存在替换它

代码设计思想：在目录字典中创建一个新的字典，同时创建以’\字典名’为名的fcb，同时向磁盘申请一个块存储目录的信息，磁盘块实例化一个文件对象，更新文件的创建时间，将分配的块号填充在fcb的loc地址中。

* **mkf(name,prmis='rwe')**

描述：创建文件。name为文件名（应包含文件类型，不加文件类型默认为‘.file’ ）；prmis为用户权限 ‘r‘为读权限，’w‘为写权限，’e‘为执行权限，默认创建者拥有全部权限。

代码设计思想：在当前目录下创建一个fcb项，向磁盘申请一个空闲块，磁盘块实例化一个文件对象，更新文件的创建时间，将块号记录在fcb的loc中。



* **vi(name)**

描述：编辑文件。使用本系统内置编辑器打开文件，可以修改文件内容。name为文件名。

代码设计思想：根据文件名在当前目录下找到文件fcb，根据fcb记录的文件位置，从磁盘块中读出文件，用ui界面打开展示文件内容，将用户进行的修改以字符串形式返回并存储。根据此次修改，计算文件大小，根据文件大小判断是否需要增加块还是减少块。并将文件内容写入对应的块。

* **cat(name)**

描述：查看指定文件。查看文件内容，将文件内容展示在命令行界面name为文件名。

代码设计思想：根据文件名在当前目录下找到文件fcb，根据fcb记录的文件位置，从磁盘块中读出文件，直接打印文件内容

* **rm(name,mode='')**

描述：删除文件或目录。mode为权限模式，默认只能删除文件和空目录。

mode相关参数：-f 若删除目录，删除目录内全部内容；若删除文件，直接删除

代码设计思想：若删除的文件，则根据文件名在当前目录下找到文件fcb，根据fcb记录的文件位置，读出磁盘块中的文件所占有的所有块，将占用块的文件内容全部删除并释放占用的磁盘块，最后删除此fcb；若删除文件夹，则需递归删除所有文件和文件夹，删除文件夹同时要删除文件夹的fcb信息，删除文件部分则和前面删除文件部分相同。

* **imt\_disk\_seek(method='SSTF')**

描述：磁盘寻道算法模拟，由于文件块的分配是按照连续索引分配，所以不会出现碎片，且受限于创建的文件数量，很难访问到更深的磁道，因此我们打算采用模拟的形式，展现我们的磁盘寻道算法。

method 为寻道方式，默认为SSTF，可选参数为FCFS，SSTF，C\_LOOK，LOOK，SCAN，C\_SCAN

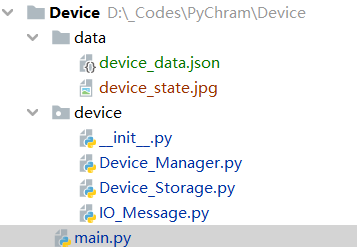
代码设计思想：首先生成随机数序列作为需要访问的磁道，第一个数为开始磁道，之后根据具体算法具体实现，例如，FCFS方法，根据算法内容排序随机数序列，计算从开始到最终总的移动距离，调用函数生成图像

### 6.5.3对外主要接口

|  |  |
| --- | --- |
| 函数名称 | 主要功能 |
| Vi | 打开文件编辑页 |
| Ls | 展示当前文件夹内容 |
| Mkf | 创建文件 |
| Mkdir | 创建文件夹 |
| Rm | 删除文件 |
| Path\_to\_file | 获取文件内容 |

## 6.6 I/O device

### 6.6.1 类设计



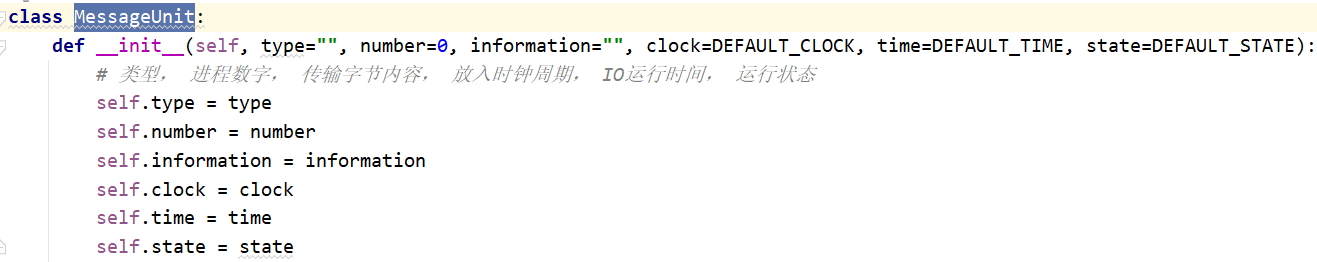
#### 6.6.1.1 类设计及相应调用关系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 文件名/中文名 | 类名/中文名 | 功能 | 调用 | 被调用 |
| **IO\_Message**  信息单元模块 | **MessageUnit**  IO信息单元 | 定义CPU和设备管理的通信数据类型；同时设计为方便设备管理进行调度和使用的数据类型。 |  | 1. 外部CPU模块；2. 信息存储模块；3. 设备管理模块 |
| **Device\_Storage**  信息存储模块 | **MsgStorage**  IO信息存储器 | 固定大小的IO信息单元存储管理器。设备管理实例会初始化Task和Send两个IO信息存储实例，方便接收和发送IO信息单元。 | 1. 信息单元模块 | 1. 设备管理模块； |
| **Device\_Manager**  设备管理模块 | **DeviceManager**  设备管理器 | 服务核心，外部CPU模块实例化的对象。其封装了所有的对外功能：1. 加入任务；2. 删除任务；3. 获取任务执行结果；4. 查看设备运行状态。 | 1. 信息单元模块；2. 信息存储模块 | 1. 外部CPU模块 |

#### 6.6.1.2类详解

* **MessageUnit**

该类用于交互数据的规整，没有任何方法，只有数据域：

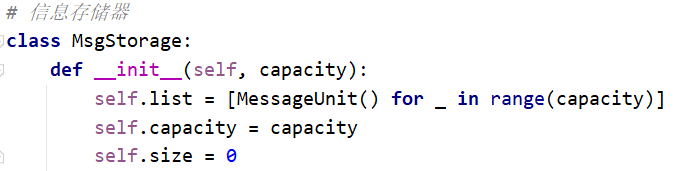


（图6-？：IO信息单元数据域）

1. **type**：信息单元类型，初始值为空””，赋值后合法值为”input”/”output”；
2. **number**：CPU发来的进行标号，唯一不重复；
3. **information**：该任务的信息域，类型为字符串；
4. **clock**：该任务需要执行的clock数，初始值为0，通过informaion字段的属性进行设置；
5. **time**：记录该任务在设备管理中运行的时钟数（等待时钟+执行时钟）
6. **state**：记录运行信息，调度后值变成设备信息；被杀死后变成”Has been killed“

* **MsgStorage**

该存储结构用来管理上一个IO信息单元，本质上是一个固定大小的IO信息单元列表。这里主要有以下管理方法和功能调用接口：



（图6-？：信息存储器存储对象）

1. 基本的增删改查
2. 根据index减小对应任务时钟数
3. 根据index增加对应任务运行时间
4. 根据index修改任务的state

* **DeviceManager**

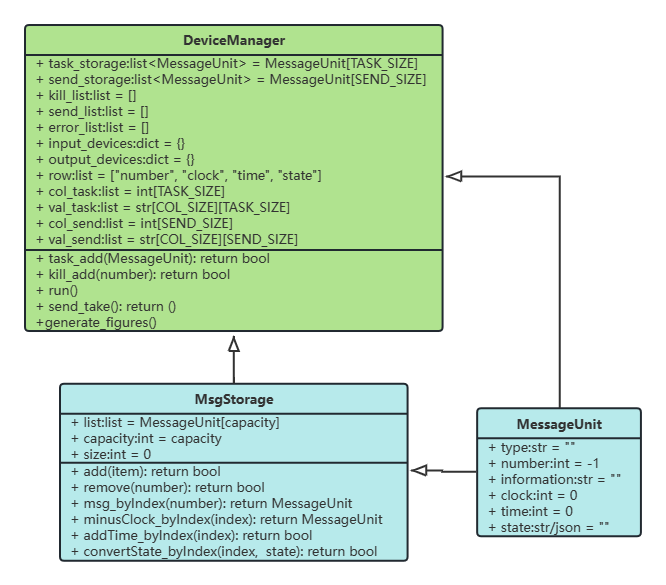
该类是暴露给外部调用的核心对象。其中用到了一些数据结构，**内部变量**如下：

1. task\_storage：接收任务队列
2. send\_storage：发送任务队列
3. kill\_list：该时钟周期下发送的杀死任务队列
4. send\_list：该时钟周期下发送的反馈信息队列（取出send）
5. error\_list：该时钟周期下的报错信息

对外部暴露主要可用**接口**如下：

1. task\_add：放入一个任务
2. kill\_add：杀死一个任务
3. run：设备执行调用
4. send\_take：获取执行信息
5. look\_state：查看当前设备使用状态
6. generate\_figures：生成设备状态图

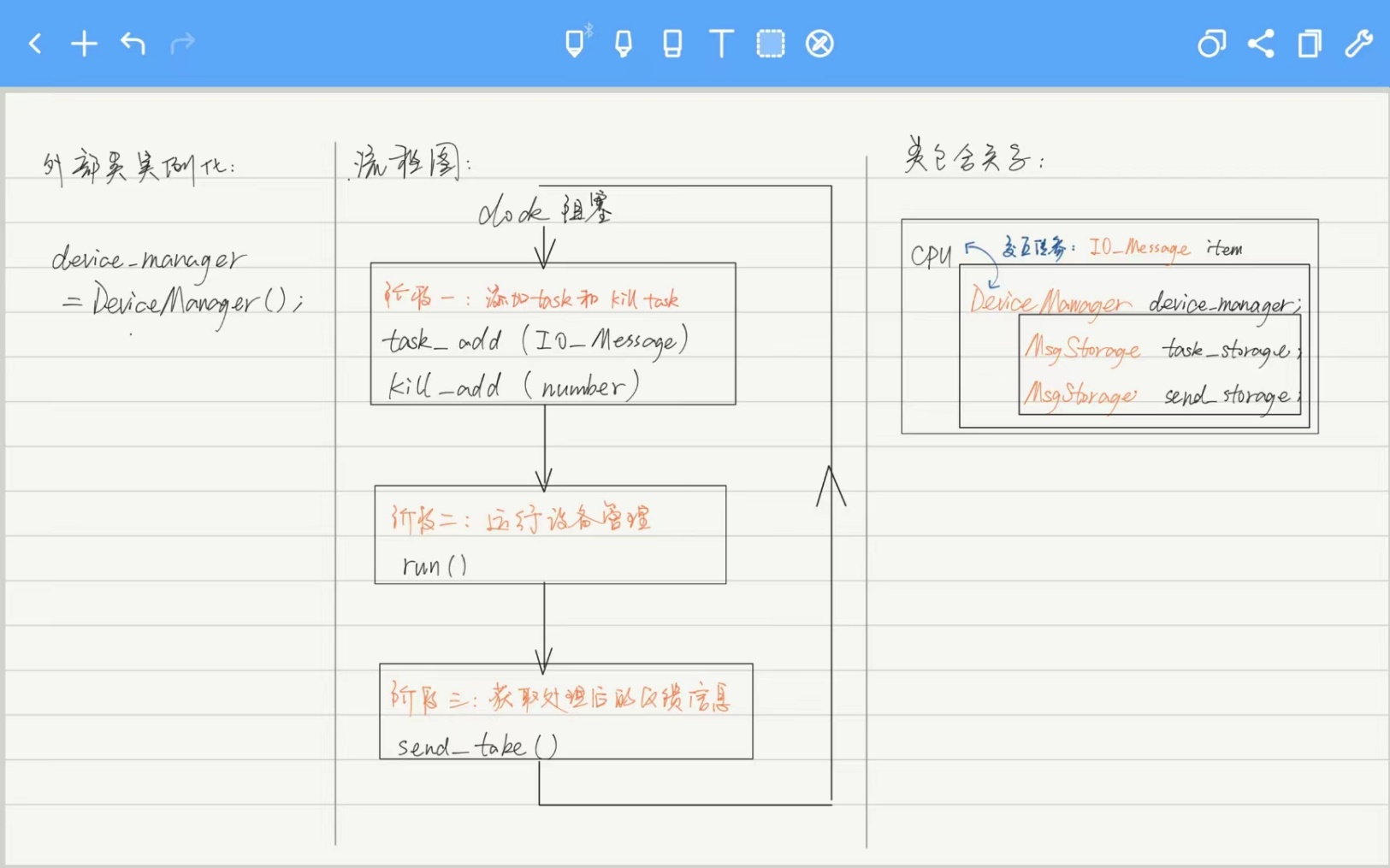
#### 6.6.1.3 类图



### 6.6.2 服务设计

#### 6.6.2.1 外部交互数据类型设计

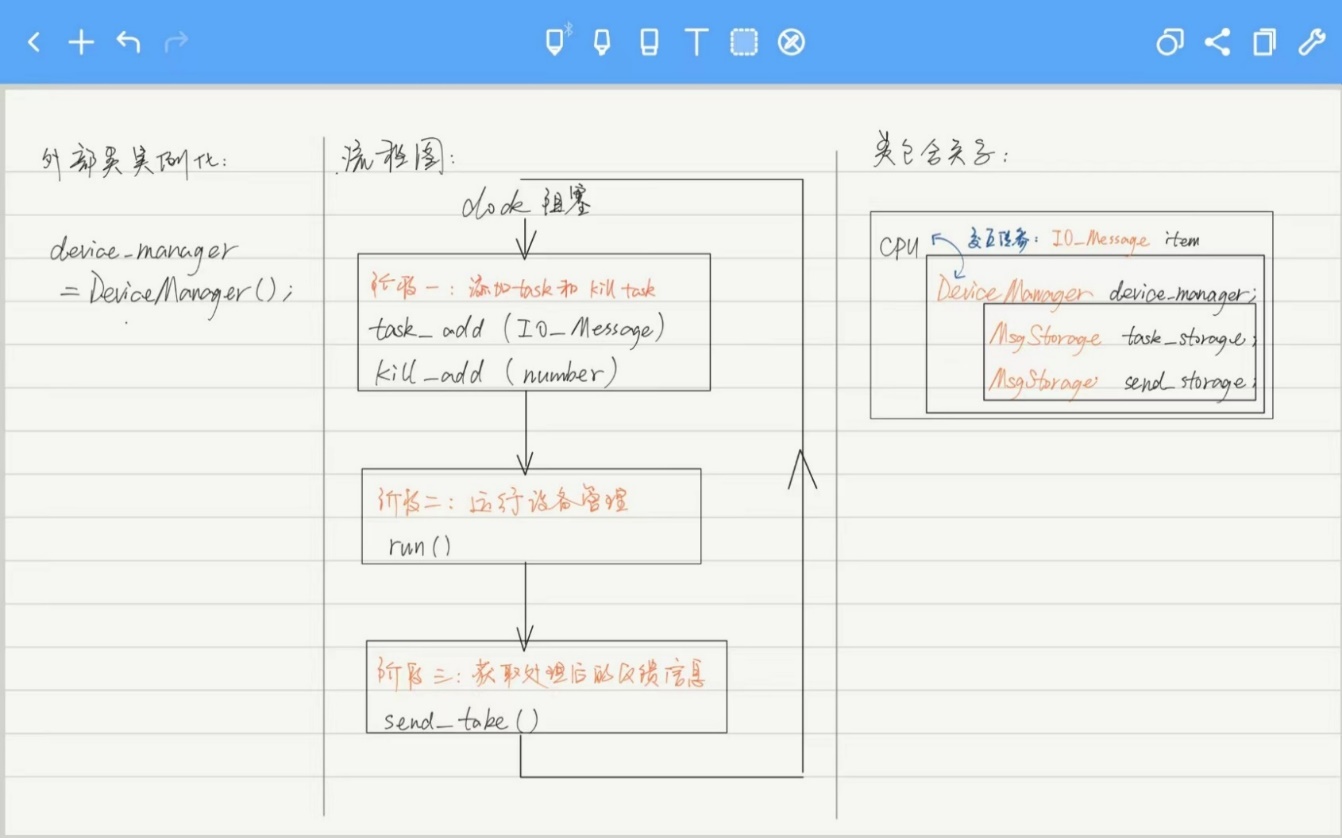
CPU模块实例化后，会生成一个DeviceManager的对象。在此对象里，也会生成自己定义的类：MsgStorage，两个分别对应发送来的人物列表和需要反馈的信息列表。在其中我们更是设计了一个通用交互类IO\_Message，规整化整个传参和反馈数据。MsgStorage实质上就是管理这种信息的一个数组列表。



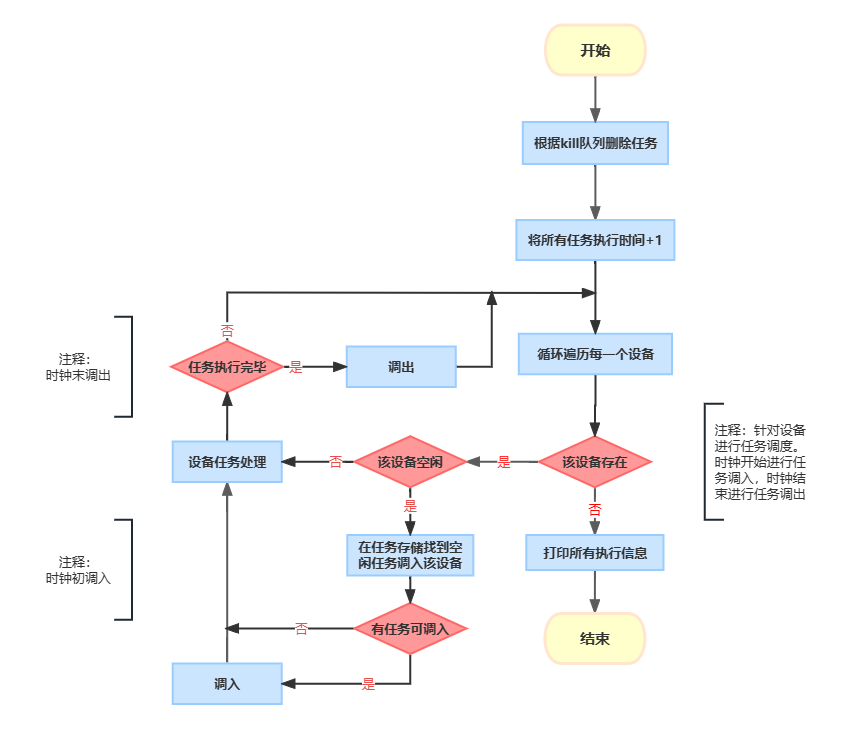
#### 6.6.2.2 对外（对CPU）接口调用流程图

因为我们需要模拟时钟驱动，所以我们的设计是三个阶段：

* 添加task和删除task。这两个函数在第一阶段可以任由CPU调用
* 该时钟下运行
* 提取出运行结果（包含反馈信息和错误信息）



#### 6.6.2.3 Run函数流程图



#### 6.6.2.4 服务细节设计

* 设备调度

首先想的是先来先服务。但其中的难点在于：如何和CPU进行对接，以及如何实现任务的添加和删除。不考虑删除时，就是最简单的任务调度，先来先服务。但如果加上删除功能，那意味着会有插队。所以在考量过后我们确定了算法。鉴于kill的次数较少，我们能采用最后向前覆盖来进行任务的删除。这样的话是类似想来先服务但是具有一定的不稳定性。在测试程序后没有问题，全部都能成功调度，饥饿的情况也基本不会出现，因为删的时候都是将后边的任务往前面调，任务只会往前调，总有一个调到前面的机会。

* IO信息处理

我们这里采用动态clock设计：根据信息的长度进行clock数初始化。比如我设置一个时钟最多输出10个字符，那么长指令可能就要用很多个时钟进行任务输出。不仅如此，这样也更能显示出在kill时的效果，直接输出从中间截断，可以显式的看懂操作系统执行的效果。

* 根据各种功能一套求设计数据域处理

time主要记录该任务在设备管理调度中呆的时间。这个主要是针对CPU来说的，因为CPU的处理会给一个初始time时钟数，我在返回的时候会给当前的时钟数，方便CPU画出图。

state主要记录当前任务的执行信息。在被调度后就是设备信息。被杀死后就是提示信息“Has been killed”。

* 核心处理业务

最后是我们如何进行设备调度。我们首先要记录很多接收到外界的信息。比如加入的任务，删除的任务序列。所以我们在时钟驱动的时候，首先进行任务的删除，再进行任务的调度。任务调度的设计可以参考我们上面的流程图。我们很明确的将任务调入放在设备执行前，任务调出放在设备执行后。

## 6.7 UI设计

该程序中UI部分主要是通过使用pyqt来实现的，整体上UI包括主操作界面，文件操作界面，初始化操作界面以及监视界面，其中，主操作界面包括输入对话框，信息展示对话框，以及文件和监视界面菜单栏，主界面的功能主要是进行信息的展示，命令的输入以及相关状态的监视。点击对应的菜单栏选项可以跳转到文件资源管理器中，这里的设计方案主要是参照windows下的文件资源管理器页面进行设计的，并且这里设计了路径的跳转和页面的返回功能，同时能够实现点击时打开对应的文件，能够展示文件内容并且进行编辑。

监视部分则设计了三个选项，包括内存使用，进程状态，设备状态，通过与对应的时间进行连接，在单击对应的按钮时会有相应的状态图进行输出，以此来展示此时操作系统的相应状态是什么样子的。

初始化部分则是在主界面产生之前进行弹出的，在主界面弹出之前，将设备与进程调度算法进行初始化选择，之后进行界面的创建。

界面与后端的逻辑连接则主要通过控件的点击事件和自定义信号量进行连接，其中有关按钮点击的操作，都通过使用相应控件的connect函数进行连接，而当不同模块之间需要相互通信时，为了避免模块之间重复实例化，我们采用定义信号量来解决，将信号量以参数的方式进行传递，在需要调用函数的地方进行信号量的发射，已达到执行相应操作的目的。

# 7．程序清单

# 8．测试报告

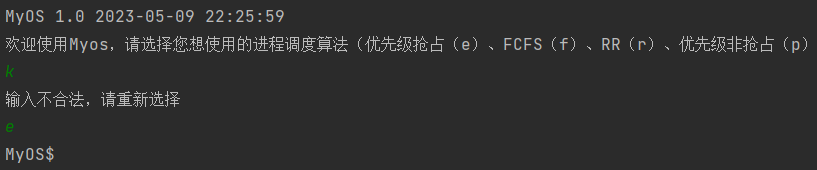
## 8.1 模块测试

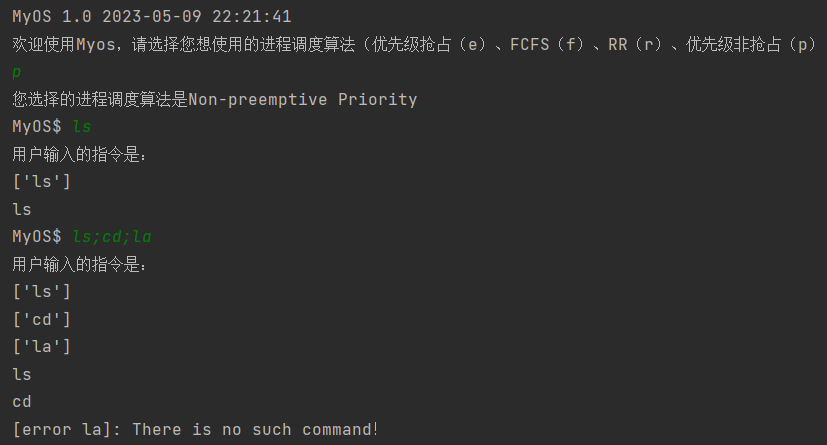
### 8.1.1 Process\_manager本地模块测试

Shell本地模块测试在kernel中一并进行。

### 8.1.2 Process\_manager本地模块测试

Kernel本地模块主要测试kernel和shell能不能正确接受并存储用户输入的指令并正确执行，指令执行结果用输出指令的名称代替





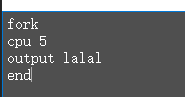
Shell接收以及kernel执行的指令均正确，当用户输入错误时也会给用户反馈

### 8.1.3 Process\_manager本地模块测试

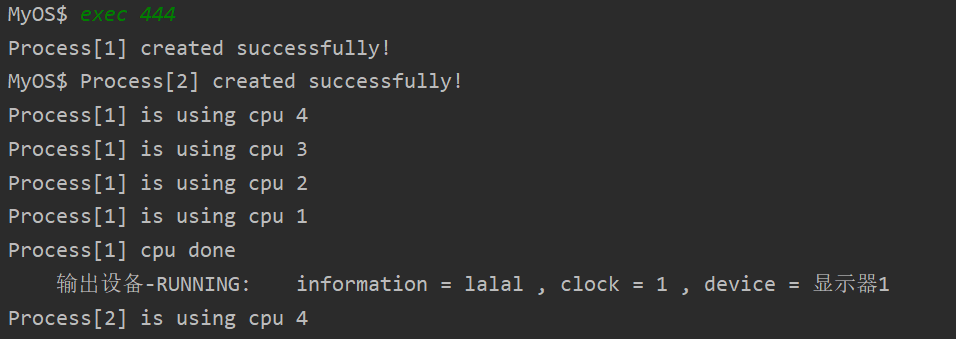
对于进程管理的测试，首先需要通过kernal模块提供的exec执行指令来执行具体的文件内容，文件内有多条指令。在测试时先执行多个文件，创建出多个进程，通过进程调度算法，同一进程在不同的时刻也会有不同的状态，我们可以通过kernal模块提供的snapshot指令来对任一时刻各进程的状态进行监测，来测试进程是否创建成功，是否按照正确的调度算法进行调度，在调度过程中是否有进程的阻塞（running->waiting/ready）与唤醒过程（ready->running）。以此来测试整个进程管理模块。

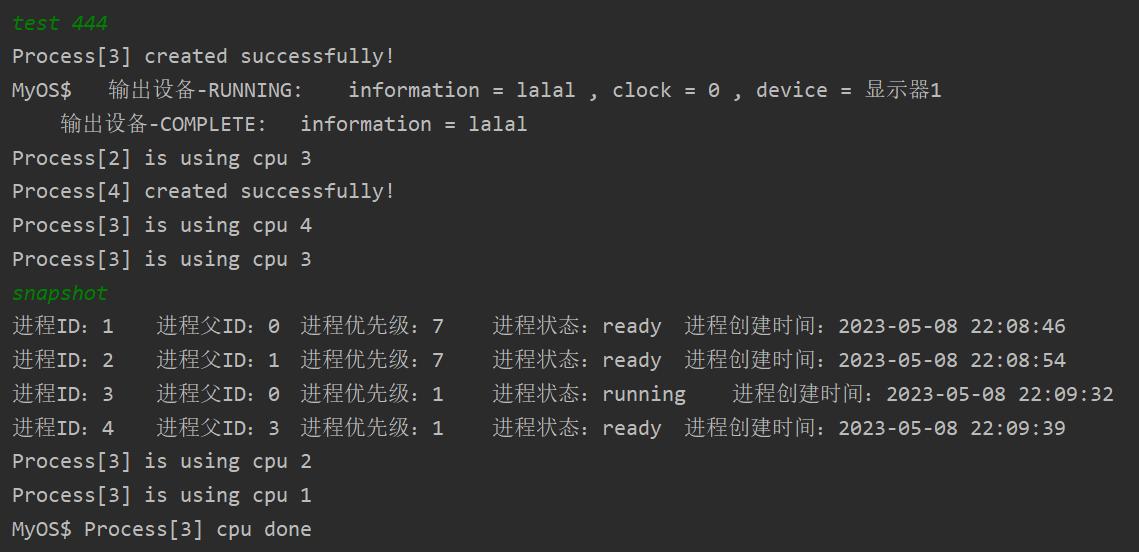
* 正确情况

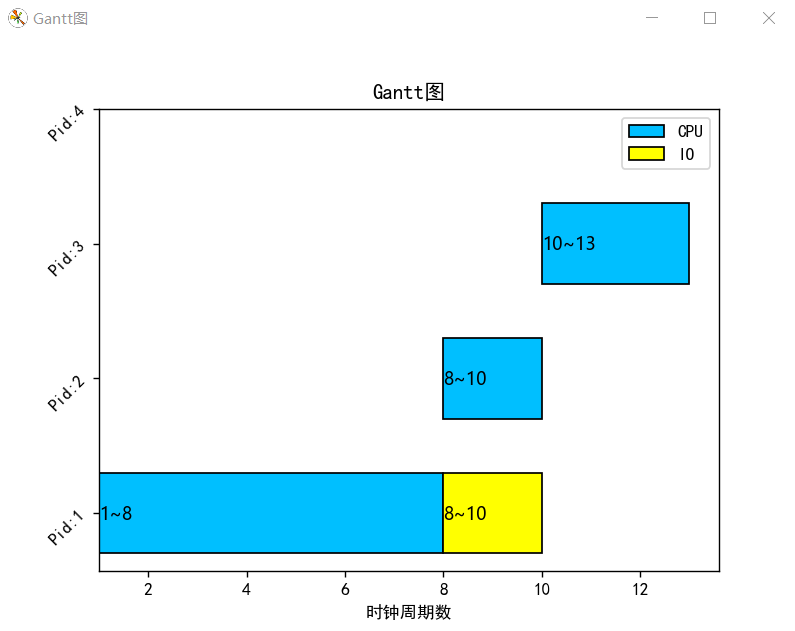
测试文件内容：（文件名：444；调度算法：优先级抢占）

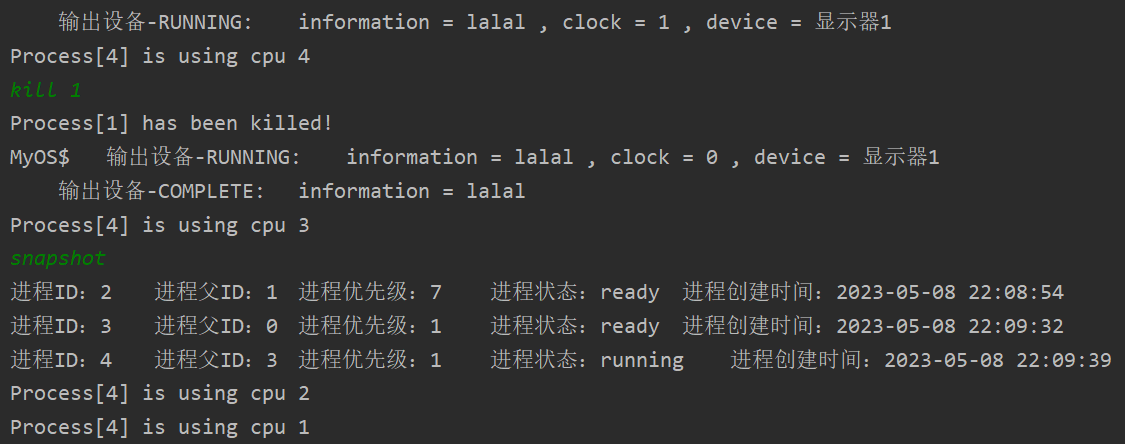


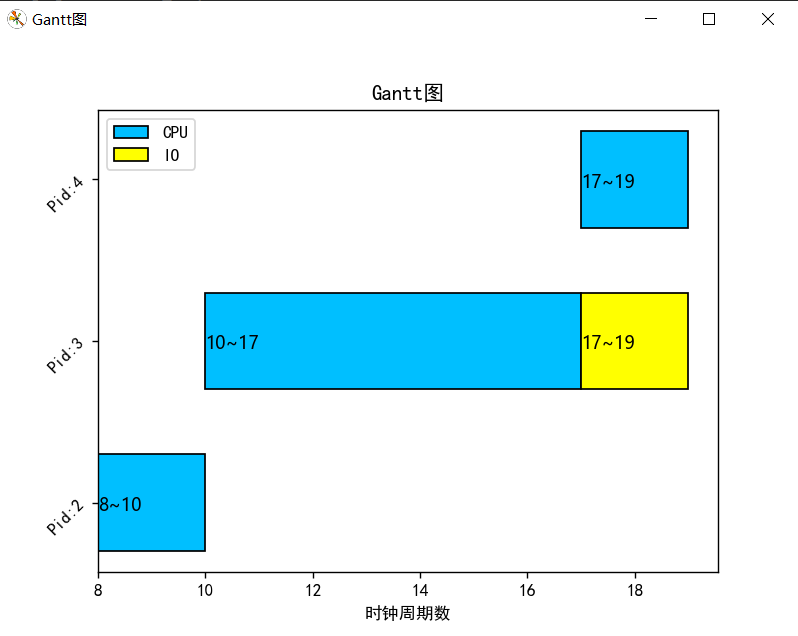
测试结果如下：

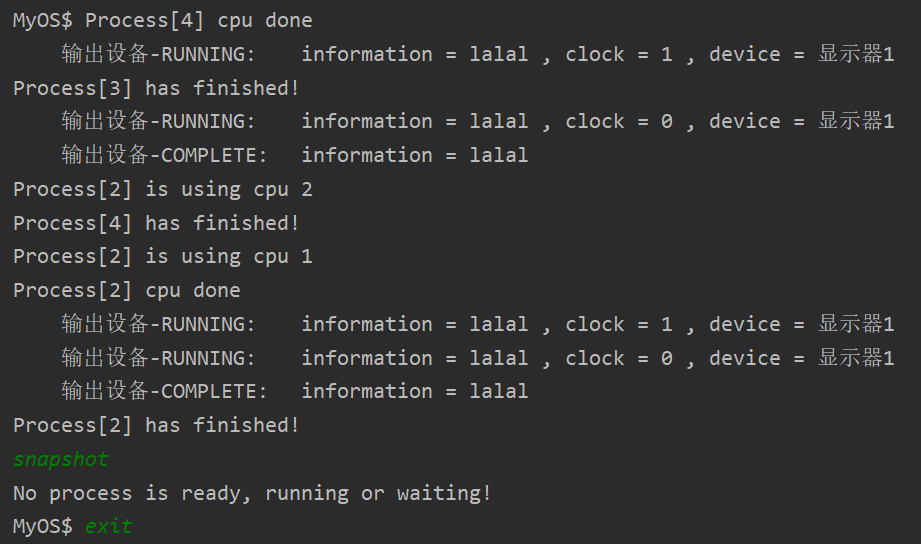












**测试情况说明：**

指令部分：exec为创建进程指令（优先级随机）。test为创建进程指令（优先级最高）。

snapshot指令为展示所有进程信息。kill为强制终止指定进程

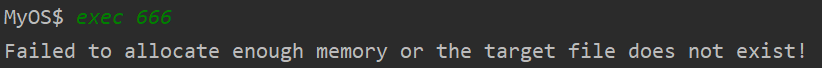
测试说明：在第一行，我们创建了一个进程，并执行了第一条fork指令，因此Process2

创建成功，4代表cpu处理还剩多少时间，由于过了一个时钟周期，因此从5变成4并不断执行，当cpu指令执行结束后会提示cpu done，随后进程1就该进行output指令，即IO指令，此时会产生IO中断，将进程1放入waiting队列中，IO设备的时钟中断与进程是同步的，因此后面会看到输出设备的提示信息与Proces2 is using cpu 4同时出现。

接下来我们执行了test指令，创建了一个优先级最高的进程，此时由于在一个时钟周期内，所以目前还是进程2在执行，这就是为什么test指令结束后的下一个输出信息是Proces2 is using cpu 3，而IO由于不需要使用CPU，因此第一个进程的IO可以正常运行。当这个时钟周期结束后创建了Process4，这就证明发生了进程的抢占并执行了优先级更高的进程3，通过snapshot指令我们也可以发现此时多了两个优先级最高的进程3和4，并且3正处于running状态。后面的Gantt图则能更进一步说明问题，由于进程1的IO已经执行结束，因此会产生IO中断，将进程1放入就绪队列中，但由于进程1还未执行完（end指令还未执行），因此Gantt图中有进程1的相关信息。接下来就是不断执行进程3的指令，cpu一直到结束并产生提示信息。

中间我们执行了kill指令，将第一个进程强制终止，此时我们可以通过snapshot指令发现进程1的信息已经没有了，并且Gantt图中也没有了进程1的相关信息，在进程3执行完毕后，由于进程4的优先级比进程2高，因此进程4开始执行，而当进程2执行结束后则意味着所有进程都执行完毕，我们再调用snapshot指令，此时控制台显示已经没有进程处于ready、running或waiting状态。至此，我们测试了调度算法的正确执行，进程创建和fork原语的正确执行，进程阻塞、唤醒、结束的正确执行，kill指令与snapshot指令的正确执行并能够通过snapshot指令来实时监控系统内的进程状态信息。

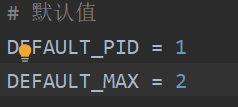
* 文件访问失败情况

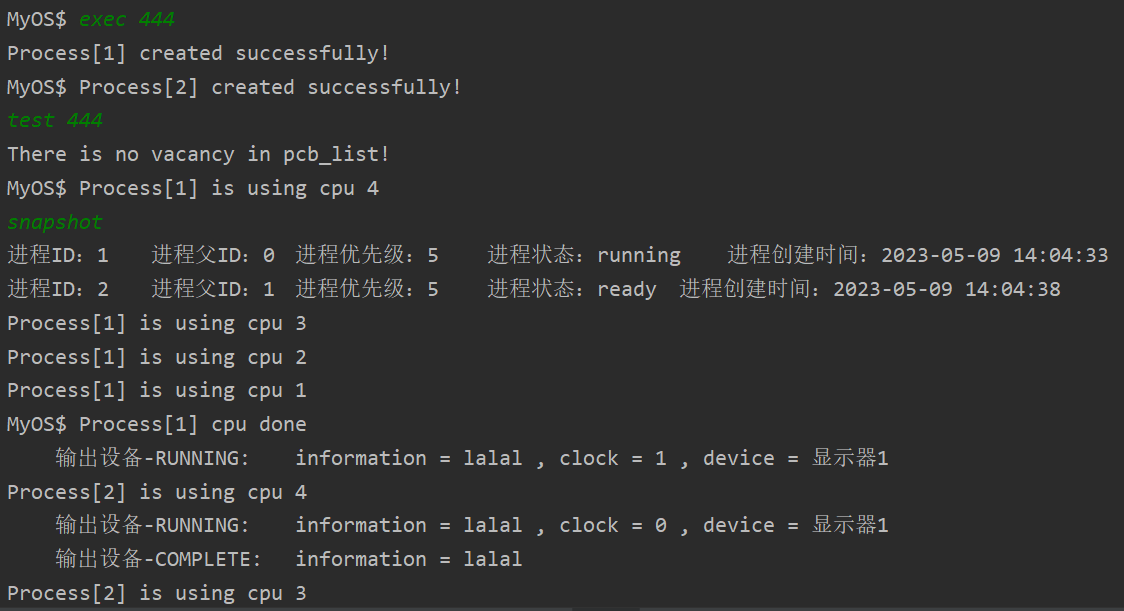


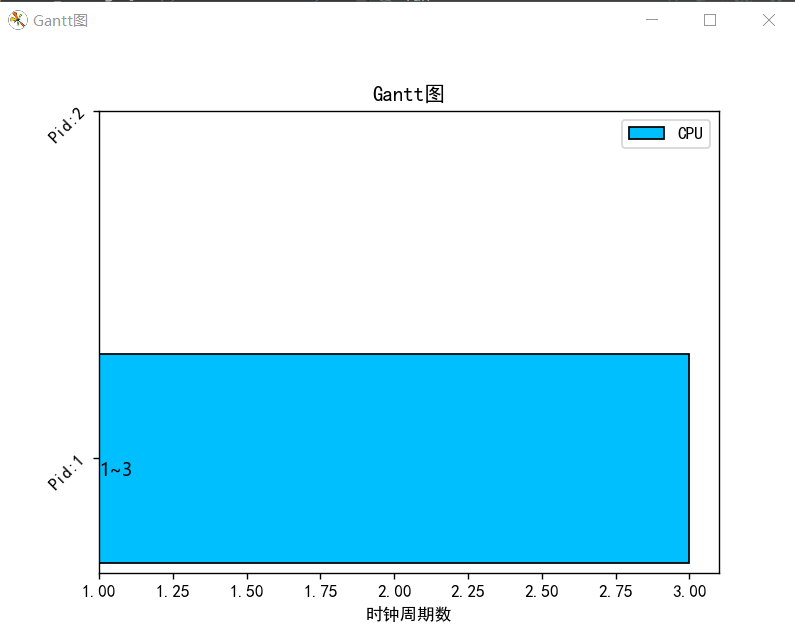
测试情况说明：

这里我们执行了一个不存在的文件，系统提示没有足够的内存空间或者目标文件不存在，测试结果正确。

PCB资源不够情况（文件内容与正常情况相同）





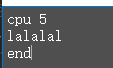


测试情况说明：

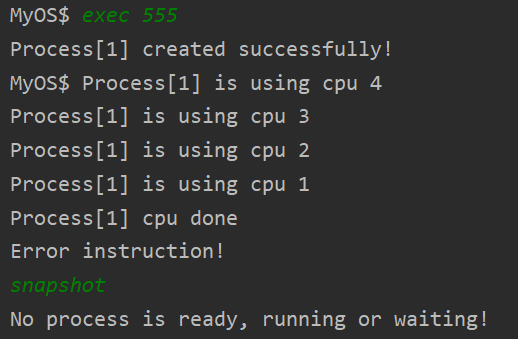
为了能够测试PCB资源不够用的情况，我先在常量设置中将PCB队列最大值从20设为2，然后我们执行444文件，调用完fork指令后此时PCB队列已经达到最大值，此时再执行444文件就可以发现系统提示PCB队列中没有空位，而且我们调用snapshot指令也可以发现进程3根本就没有创建出来，测试结果正确。

指令不存在情况

测试文件内容：（文件名：555）



测试结果如下：



测试情况说明：

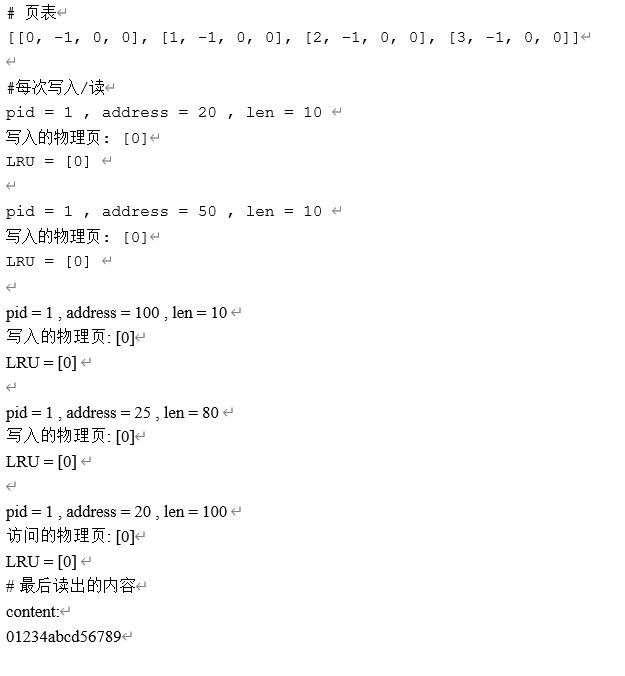
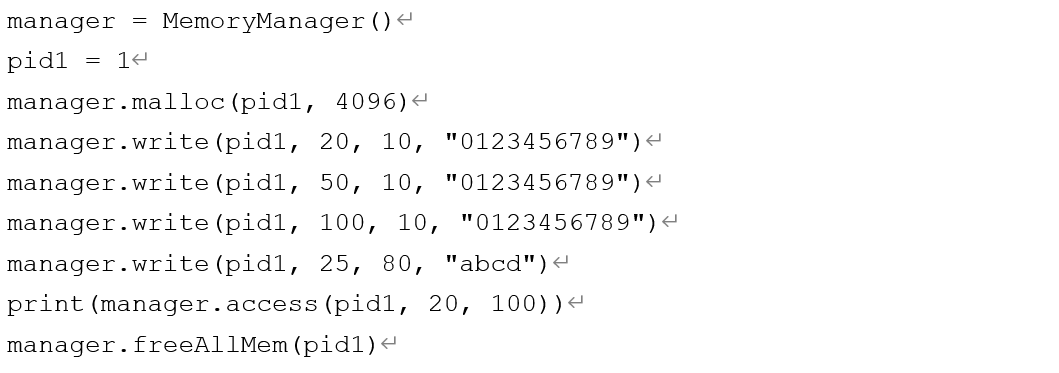
通过测试文件我们可以发现，在cpu指令和end指令中间有一条非指令lalalal，在执行555文件时，cpu指令正常执行，但检测到lalalal时，由于没有对应的指令，因此会提示指令错误并结束掉当前进程，因此我们在调用snapshot指令时会得到系统提示当前没有进程处在ready、running、waiting状态，测试结果正确。

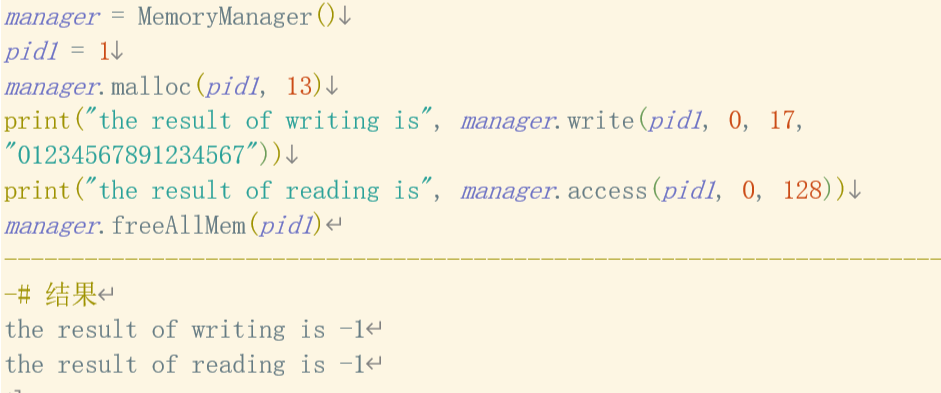
### 8.1.4 Memory\_manager本地模块测试

下图实例化了一个内存模块，模仿pid为1的进程，为其申请了4096字节大小的内存空间。并先后往[20, 29]写入“0123456789”，[50, 59]写入“0123456789”， [100, 109]写入“0123456789”，[25, 104]写入“abcd”。依次序列写入，最后一次写入时[50, 59]的数据会被全

部覆盖，[100, 109]的数据会被部分覆盖。之后读出[20, 119]的内容，最后释放进程的内存。

第二部分是一个错误的使用，只申请了13字节的内存空间，却写入了更多字节的内容，最后也访问了不存在的地址。可见返回结果都是-1，代表访问错误。

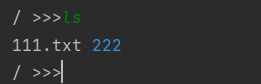




### 8.1.5 File\_manager本地模块测试

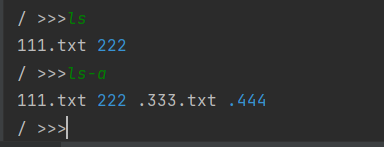
* ls函数功能测试：

无参数



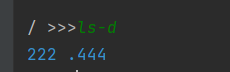
结果说明：只打印当前目录下的非隐藏文件和文件夹

-a



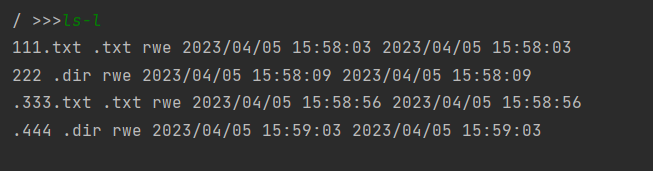
结果说明：文件名前加’.’为隐藏文件，此选项将隐藏文件也展示出来

-d



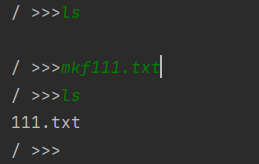
结果说明：此选项仅展示文件夹

-l



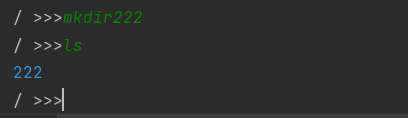
结果说明：此选选项展示文件和文件夹的具体信息

* mkf函数测试：此函数主要用来创建文件



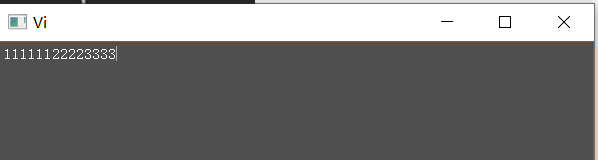
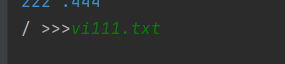
结果说明：执行指令前，当前目录为空，执行后，生成111.txt文件

* mkdir函数测试



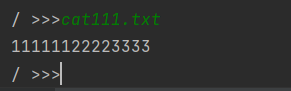
结果说明：执行指令创建文件夹222，通过ls指令发现成功创建

* vi函数功能测试



结果说明：执行指令编辑111.txt，弹出界面，界面里显示文件内容进行修改

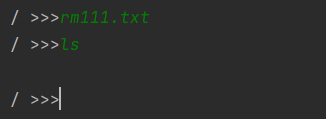
* cat函数功能测试



结果说明：查看111.txt文件内容，并输出

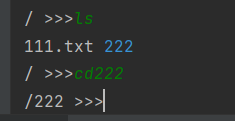
* rm函数功能测试

无参数：



结果说明：删除111.txt，通过ls指令查看，111.txt文件被删除

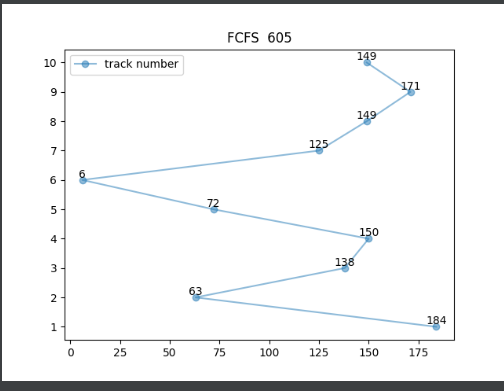
* cd函数功能测试



结果说明：指令进入222文件夹，路径显示成功进入

* imt\_disk\_seek函数功能测试





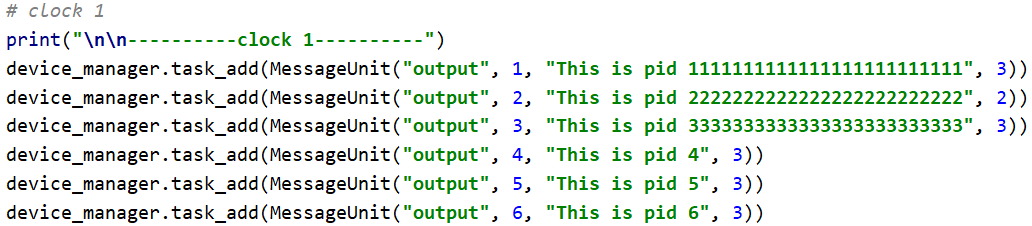
结果说明：磁道访问模式为FCFS，调度过程如图，从184磁道开始，149磁道结束，一共移动605个磁道。

8.1.5

### 8.1.6 Device本地模块测试

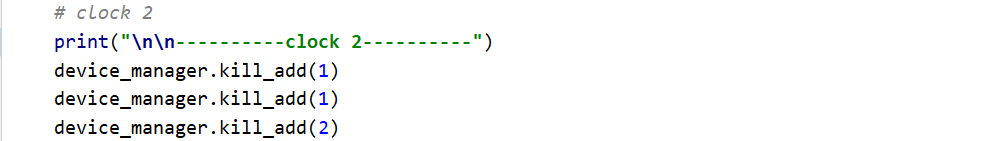
* 初步测试

第一步，在函数第一个周期进行以下指令的添加：

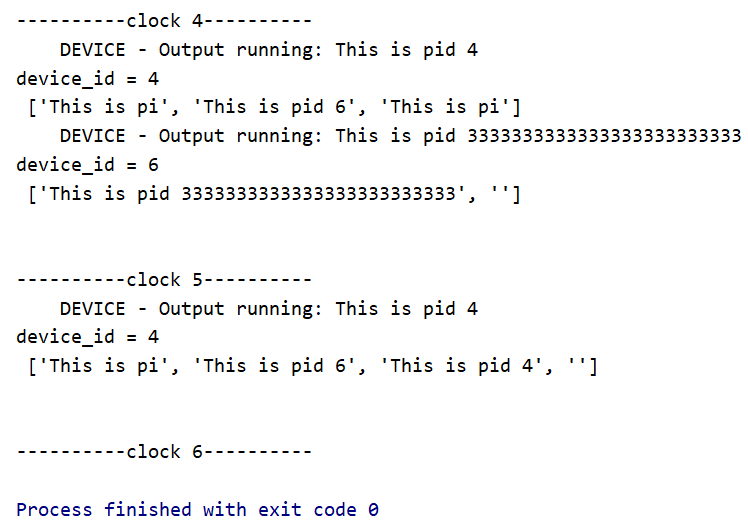


（图8-？：本地模块测试指令任务添加）

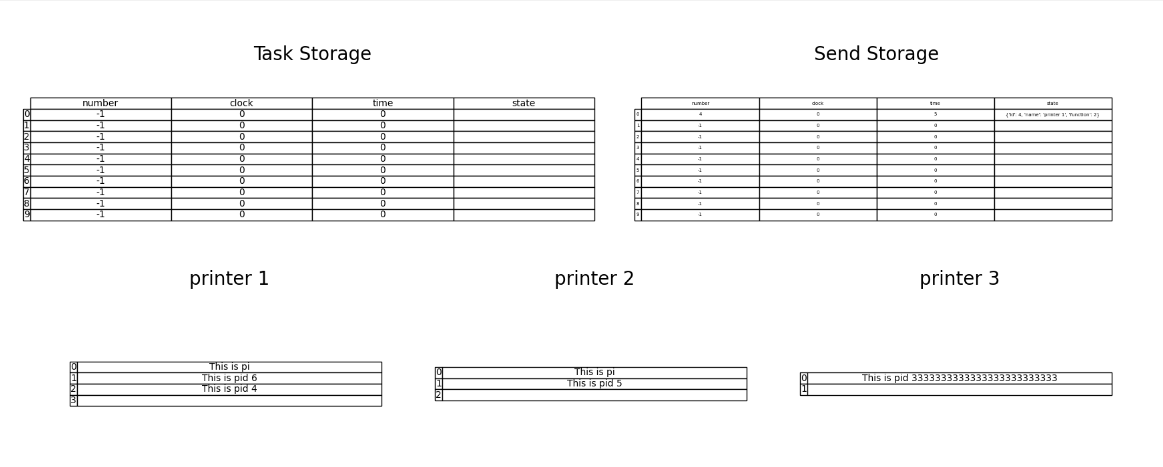
第二步，然后在第二个周期杀死12任务：



我们可以查看执行结果：



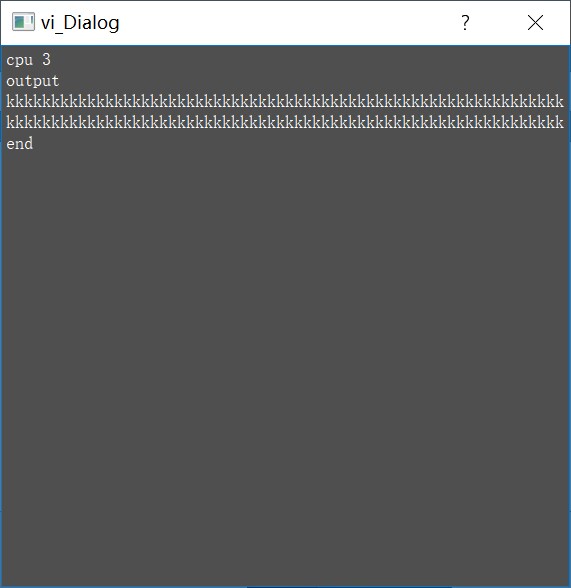
（图8-？：终端测试部分输出结果）



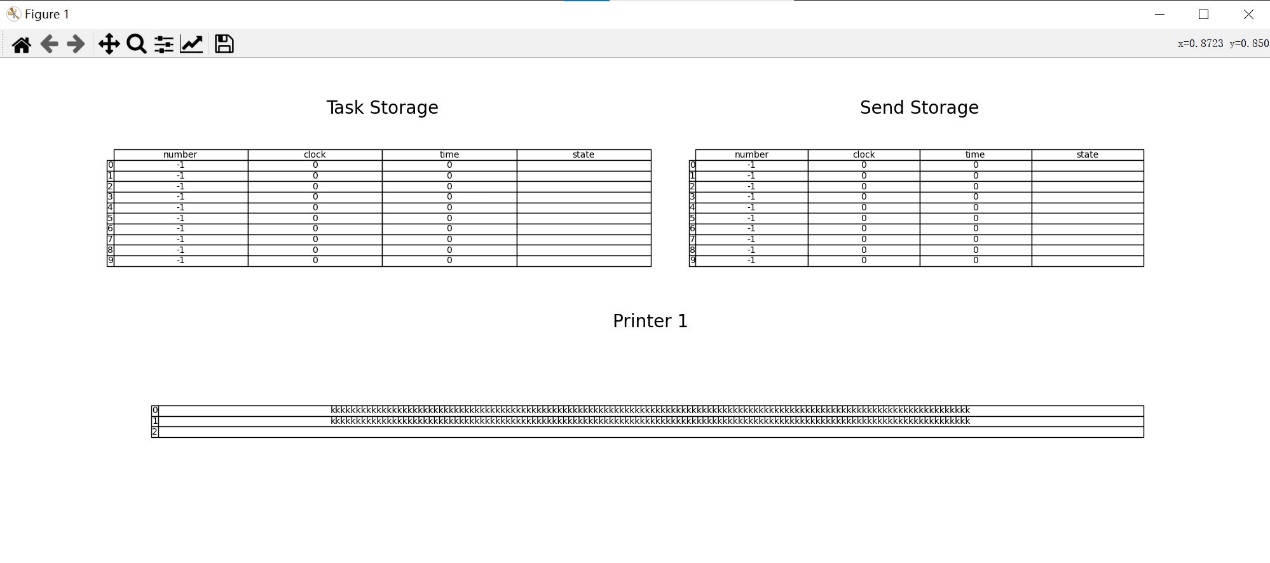
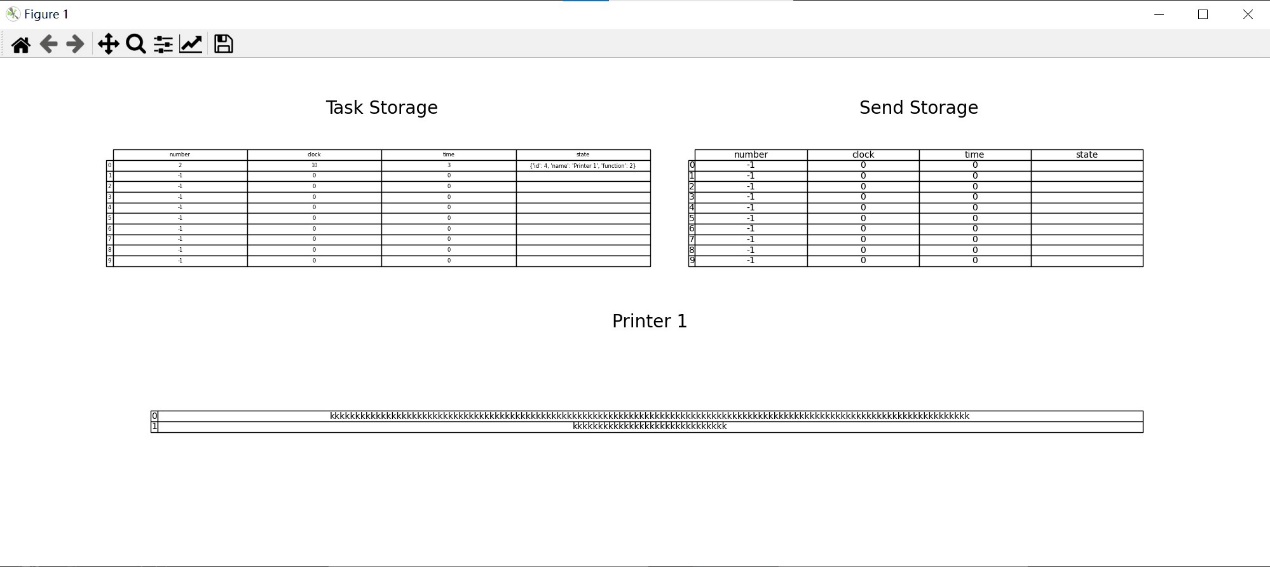
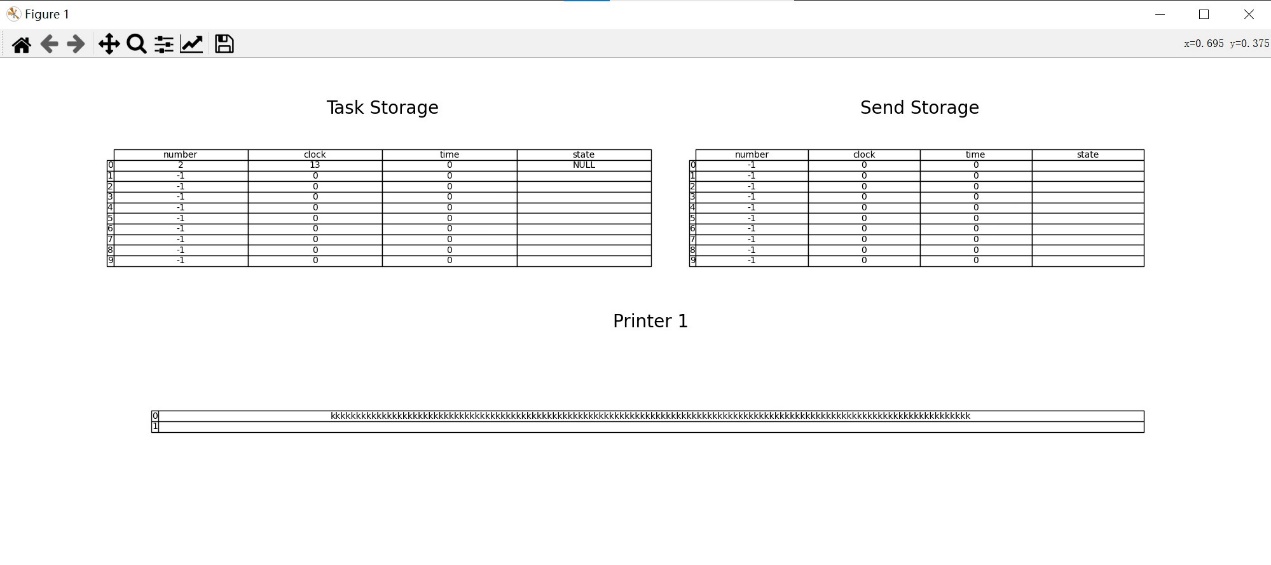
可以看到第一二个任务分配给打印机12后，输出十个字符后就被制止了，换行。而3456指令都被正确调度且显示在了正确的的位置上。

* 长指令多周期输出效果测试

第一步测试长指令多周期输出效果。我们执行程序进行指令调度，这条指令是长指令：



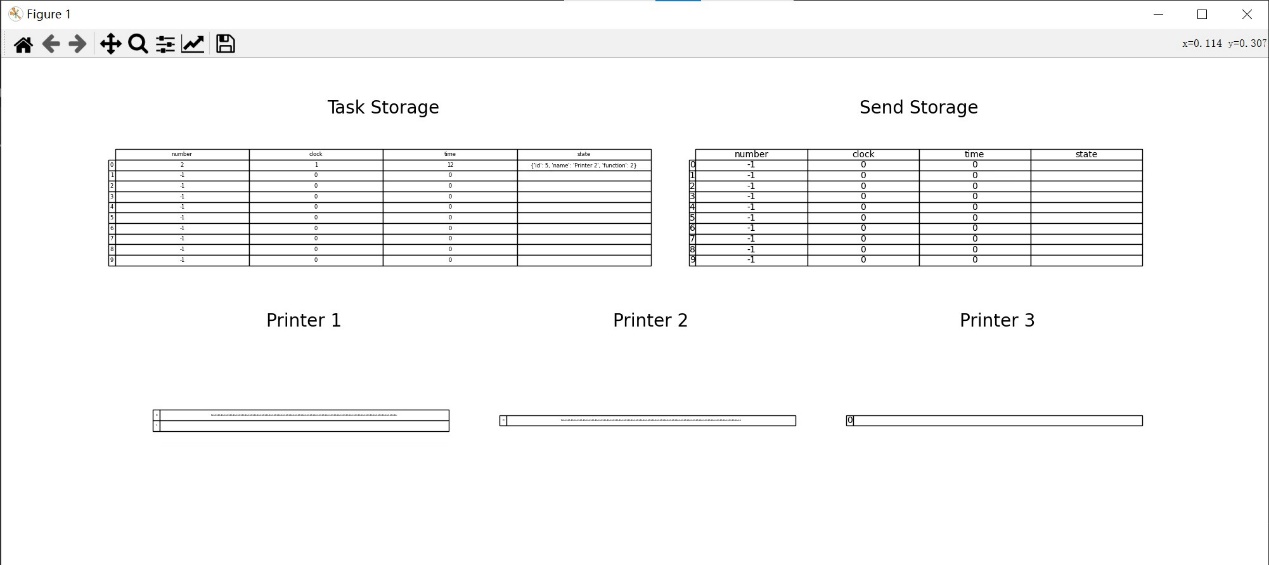
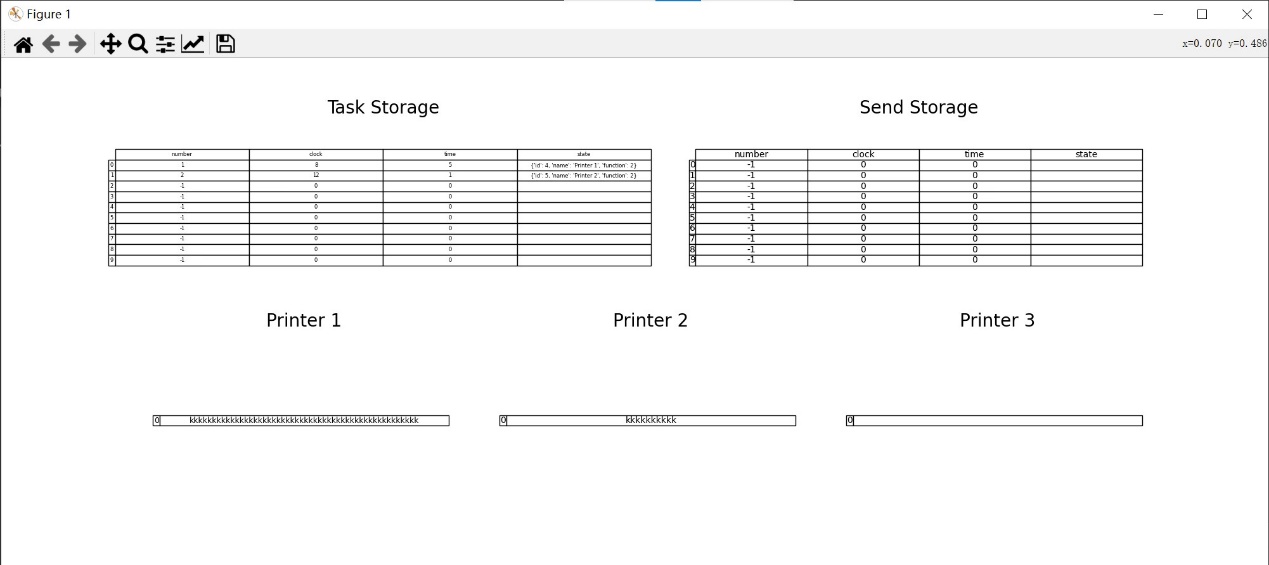
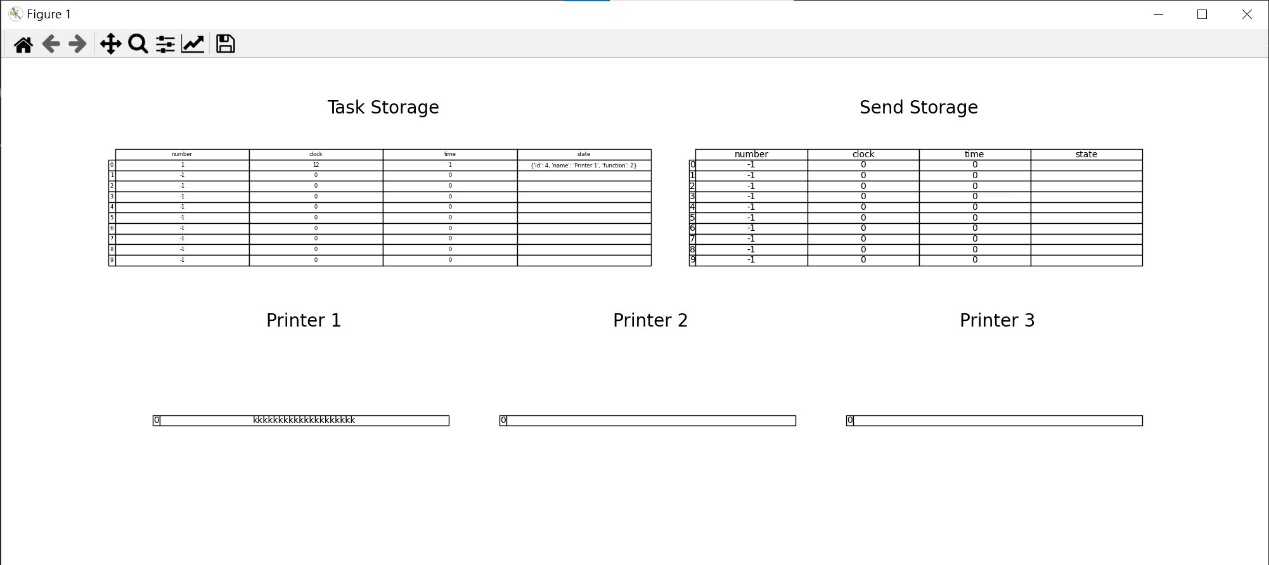
因为上一次已经执行过这个指令，所以从第二行开始看：

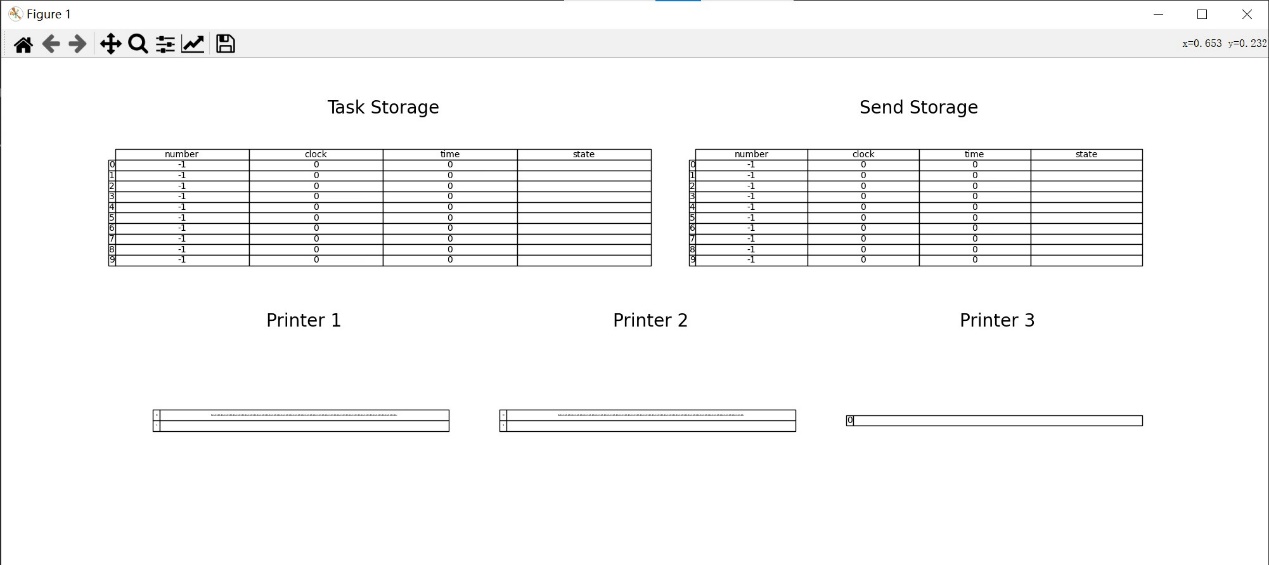


（图8-？：长指令测试结果）

* 多指令调度测试

第二步，测试多指令调度。将指令调度进行两次，并设置3个打印机外设，即三个外设调度，执行两次上一个程序。



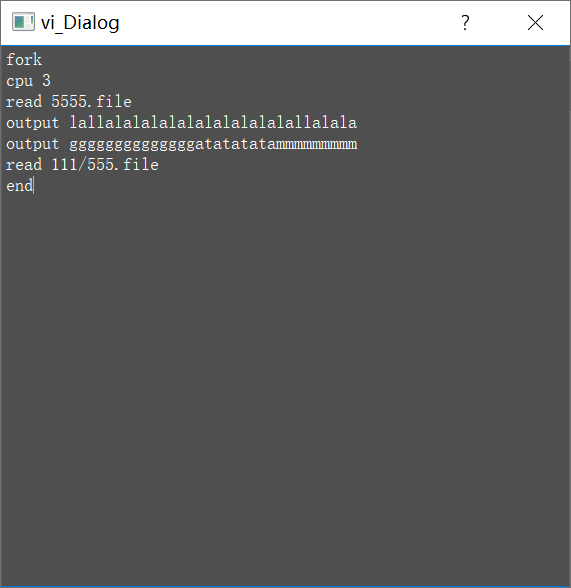


（图8-？：多指令调度测试结果）

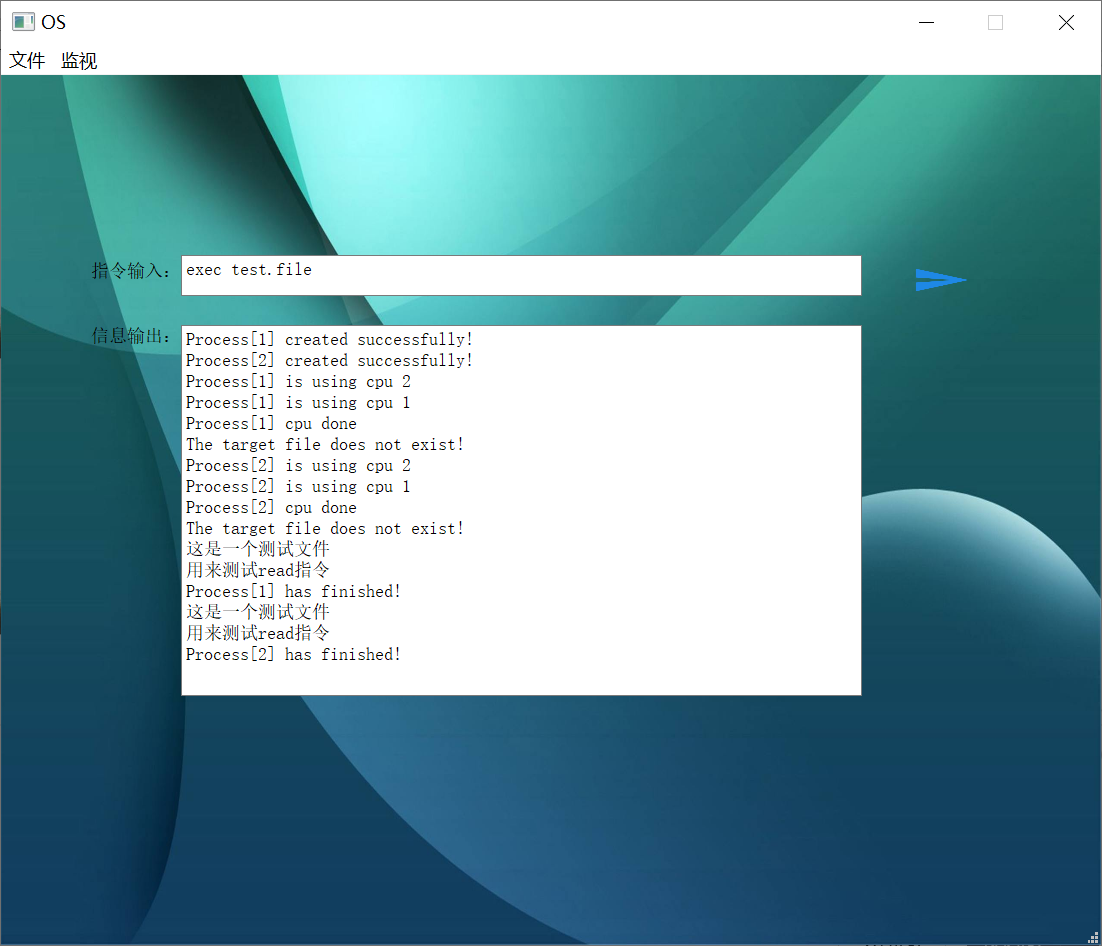
可以看到两个指令执行分别调度到Printer1和Printer2并执行完毕，在上述的两个存储结构中也能看到相应的变化（clock是剩余执行时钟数，time是该任务当前在外设的时钟数）

## 8.2 总体测试

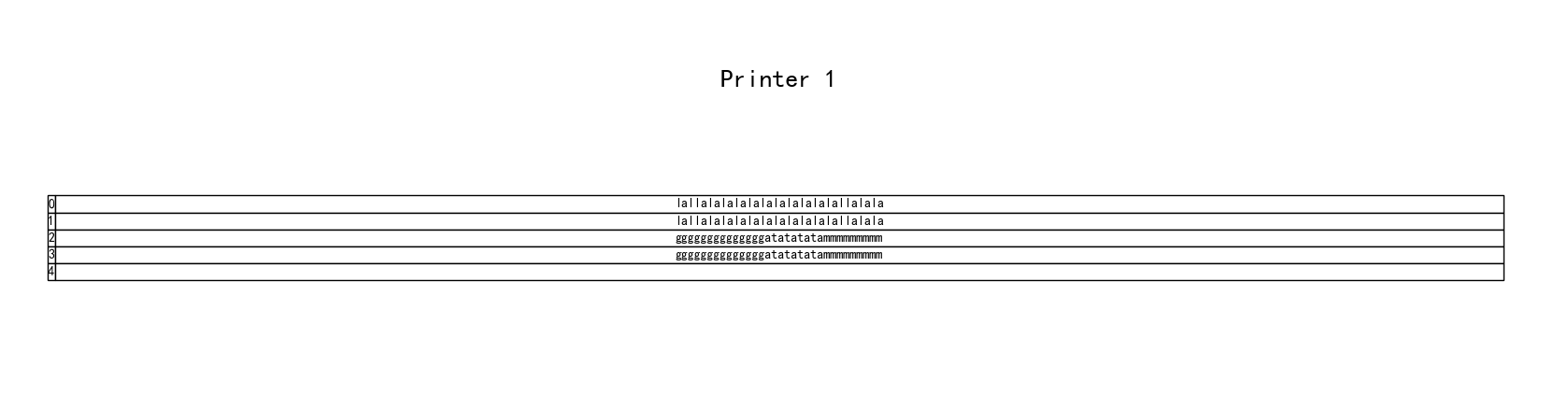
整体测试程序如下图所示：



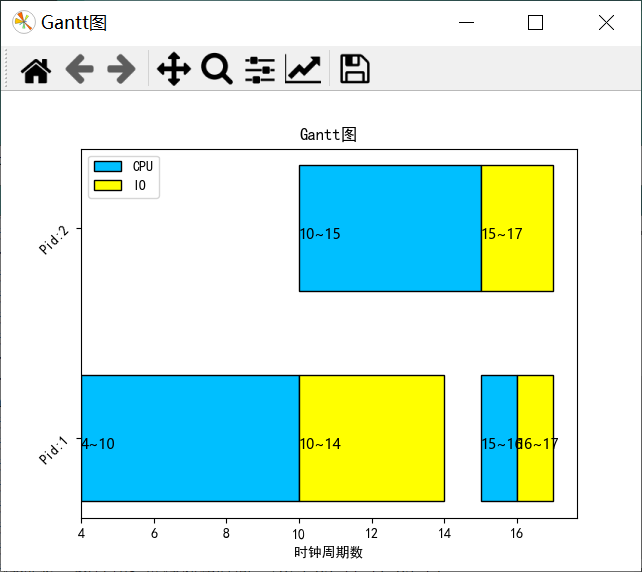
该执行文件的流程为fork指令进行子进程的创建，cpu 3表示该进程使用3个时钟周期的cpu，read表示将该文件的内容进行读出，output表示使用I/O设备进行输出，end表示进程执行结束。



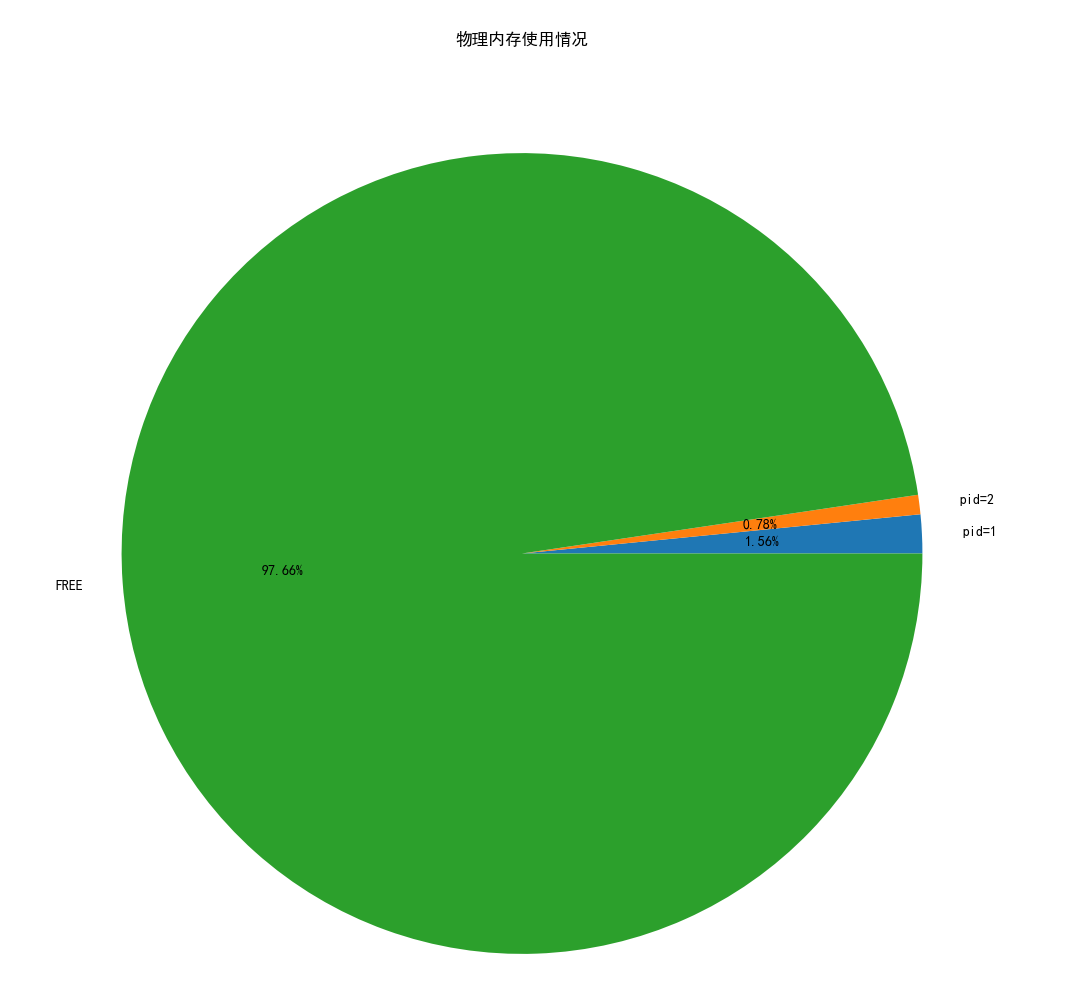
运行测试文件可以看到如上输出结果，首先是process[1]创建成功，然后程序运行第一条fork指令，创建第一个子进程，可以看到输出信息为第二个进程创建成功，之后第二个进程进入ready队列进行等待，此时进程1执行相关指令，首先是使用cpu时间，之后是读取目标文件，可以看到输出信息为目标文件不存在，因为在我们的文件系统中并没有定义相应名字的文件，因此输出文件不存在的调试信息，之后可以看到进程发生调度，这是由于进程一开始执行I/O操作，此时进程一由原先的running状态变为waiting状态进入waiting队列中进行等待，等待I/O操作的完成，此时CPU空闲，调入进程二进行执行，可以看到这里的进程二开始执行CPU时间，在执行到输出指令时同样进入waiting队列进行等待，然后是I/O之后的读取文件指令可以看到首先是进程一执行完毕，因为进程一首先进入到I/O中操作，因此其执行时间应该是最快的，之后是进程二进入I/O中进行操作，最终完整的进程过程就执行完毕。



该图片展示的是I/O设备最终的输出结果，可以看到按照要求成功输出指令的结果。



这里是进程在执行过程中所有进程的Gantt图，可以明显的看到在第十个时钟周期，进程一执行完CPU时间之后进入到I/O时间，此时进行CPU调度，执行进程二，之后二进入I/O时间进入进程一进行执行。



该图片代表的是进程占用的物理内存的比例。

# 9．课程设计总结

## 9.1 设计过程中遇到或存在的主要问题及解决方案

* 进程模块

进程管理部分遇到的第一个主要问题是抢占类算法的实现，抢占类算法不同于非抢占类算法一样有一个固定的执行顺序（对于FCFS来说就是进入就绪队列的顺序，对于优先级非抢占来说就是优先级的顺序），因此，对于非抢占类算法只需要在进程结束时根据顺序将下一个进程调入进CPU执行即可，而抢占类算法（尤其是优先级抢占）在任何时间面对用户的输入都有可能会发生抢占，这是一个不可控的过程。

解决方案：经过小组讨论和实际操作系统的实现过程，我们决定通过时钟中断和标志位的方式来进行解决，我们参考了中断方式中的轮询方式来处理，既然我们没法预知何时会发生抢占，那么就让操作系统每个时钟周期都进行“轮询”，去询问是否需要发生抢占动作，即在这个时钟周期内，是否有新进程加入，新加入的进程会不会导致抢占（新进程的优先级大小），通过这种“轮询”的方式就可以解决抢占类算法的问题。

进程管理部分遇到的第二个主要问题是如何使用户在执行进程文件的同时也还能够进行其他指令的处理，如果只有单一线程的话，用户执行进程文件后，主线程就会跑进程文件的内容，进程文件没执行完也就无法调用kill、snapshot等其他指令，用户会一直等待直到进程文件执行结束。

解决方案：经过小组讨论，我们决定开一个线程专门用于执行进程文件，在没有调用exec的情况下，这个线程就会一直循环阻塞，当有新进程创建时告知该线程有新进程加入，此时就可以在后台执行进程文件，而用户也不会阻塞在这个执行进程的线程里，用户可以正常调用snapshot、kill等其它指令进行操作。

* 内存模块

在一开始我的理解有偏差，认为我的内存模拟无需真正往里存数据，只需做LRU表的换页换出，当进程 访问时给出是否访问成功，是否换页即可。 在初次的验收时，意识到真正往内存里存入数据会更符合要求。在此之下，不得不推倒了之前的设计， 重新开始考虑整个架构。这浪费了我很多的时间，我意识到在之后的学习工作中，如果有可能，应该尽 量的明确需求。 在解决这个问题之后，还有的问题是如何有效的与组员进行沟通，这很考验逻辑表达能力，我觉得我做 的也不是很好。 剩下的就是具体实现过程中对一些边界条件的考虑不周，比如存入数据为空字符串，需要读入的文件不 存在，存入数据过长等。其中的一些bug在最后整合阶段才被发现，这也是我考虑不周的原因。

* 文件模块

磁道算法实现问题，起初我们决定在通过一个批处理指令访问多个文件的方式访问多个磁盘块从而能访问多个磁道实现磁头跳转，观察磁头运动轨迹，但是因为我们的文件创建是采用索引分配，所以文件较为集中，而且受限于操作和时间，我们并不能用满磁盘空间，大部分的磁盘空间我们无法使用，磁头能触及的最远磁道就比较小，由此观测轨迹效果并不好。因此我们决定采用模拟磁道寻路的方式实现，设立随机序列，根据此序列再由指定磁道算法重新排序，去移动磁头，记录轨迹。

关于索引分配实现相关，文件大小改变时，需要计算需要的块数，并将内容分配各个块，而且当需要访问一个文件时，文件的内容因为是分配到各个块的，所以必须要聚集各个块的内容。因此我们设计了聚集文件函数和更新文件内容函数。聚集文件函数负责根据文件存储的块，将文件内容合并起来。更新文件内容函数负责重新计算更新后的内容长度，根据长度判断已有的块是否需要增添、减少或是不变。

与内存模块联系部分，主要是vi部分的内容，涉及到文件的传递问题。在用户使用vi指令后，文件系统需要将内容传递给内存，在用户修改文件后，文件系统需要将修改后的内容传递给内存。此时一个函数需要先后返回两次内容。因此我们决定采用pyqt信号量，通过发送信号，来传送两次值。

* 设备模块

我们在设计的时候出现了很多问题。前期主要精力花在了和CPU对接的问题上。CPU需要什么，以及CPU该如何调度，设备管理该如何识别任务，以及根据唯一的标识来进行任务调度。所以我们首先设计了IO通信单元MessageUnit，数据域包含了所有需要的功能和调度都需要的信息。这样我们直接存储发来的任务，然后进行调度，生成返回信息就可以了。

此外就是我们的数据管理。因为我们采用了固定大小的存储空间，所以封装增删改查以及属性值修改是很有必要的一件事。所以我们设计了MsgStorgae这个类，进行数据和功能的封装。设备管理器只需要初始化两个存储类，就可以管理所有的存储信息了。

最后是我们如何进行设备调度。我们首先要记录很多接收到外界的信息。比如加入的任务，删除的任务序列。所以我们在时钟驱动的时候，首先进行任务的删除，再进行任务的调度。任务调度的设计可以参考我们上面的流程图。我们很明确的将任务调入放在设备执行前，任务调出放在设备执行后。

* shell&kernel模块

我主要负责设计的模块是shell和kernel，其中的kernel模块需要调用其他所有组员的接口，与大家的交互很多，所以在这个过程中会出现许多问题，例如：在调用大家的接口时，我没有考虑到组员代码中也会调用其他组员的代码，导致刚开始的时候我只导入的每个组员代码中的类，于是我改为导入组员代码文件解决这个问题；与组员的交流中忽视了数据结构问题，导致我存储指令应用了列表，而组员需要我传的数据的数据结构是字符串，于是我修改了一下输入指令的数据结构。

## 9.2 改进建议

* 进程模块

最开始我们打算用命令行模拟操作系统，并没有考虑到最后会用Qt实现可视化，因此进程管理这部分跟设备管理和内存管理耦合度是比较高的，而用Qt实现的话就可以引入Qt特有的信号和槽函数的机制，使得在进程管理部分需要调用到其它模块的时候只需要发射一个信号量，其它模块捕捉到信号量后将结果传给kernal，kernal再传给进程管理部分，这样模块间的耦合度就能够大大降低，模块间的通信也仅通过kernal进行调度，这才是理想的过程，但用Qt实现可视化时整体的代码已经基本成型且进程管理部分调用其它模块的部分也比较多，短时间内也不好完全实现，可视化只是作为单纯的前端页面，因此，进程管理部分的改进建议就是可以通过Qt的信号与槽机制来降低进程管理与其它模块间的耦合度。

* 内存模块

真正的把虚拟内存放进磁盘块里，解决读入的文件可能会相互覆盖的问题。用更智能的方式决定文件存 储在内存中的位置。

* 文件模块

首先对于文件系统和内存的交互，理论上应该先在内存出现文件内容，文件内容修改后传回文件系统，我们的实验中是文件系统在修改文件后自动保存再传给内存，内存的回写并没有什么意义。因此应该修改此处逻辑。

其次对于文件的权限，我们只是简单实现了这个属性，并且默认是拥有全部权限，在实际使用中，这个权限并没有什么作用，因此更好的程序应该体现这个属性的作用。

我们程序的整个ui界面只能说是完成功能的实现，不能说是美观，因此程序的美观是一个很大的进步空间

* 设备模块

在答辩时受到老师提示，我们并没有在理论上避免读写冲突，因为我们在设计上就基本避免了这个问题。但是这个应该需要有冲突控制。针对每一个访问文件都设置一个读的锁。这样在任何冲突的时候只要读这个锁被锁住那就不能访问。

此外，我们对于I的写法并没有很好，对于用户输入的体验还是比较差，基本等于没有多线程多开功能，我们也没有进行展示。我们应该开一个线程专门处理这些用户输入信息。

* shell&kernel模块

我认为我的kernel部分可以改进的部分在于，我可以将任务调度、访问内存和其他硬件、读写文件等最底层操作放到内核来完成，由内核来完成最底层的操作，这样可以降低其他模块间的耦合性，同时使其更接近真实的操作系统内核，更好地沟通硬件和软件。

## 9.3 体会/收获

* 李祥宇

在本次操作系统课程设计当中，我担任的是组长一职，主要负责项目的整体统筹规划，

文件部分的逻辑实现，UI的设计以及最终的前后端结合调试工作。而在这个过程中，除去自己负责的文件部分的代码编写之外，也对整个操作系统的运作有了深刻的理解，对进程，文件，内存，设备，kernel，shell六个模块之间的交互和调度有了新的体会。在进行文件部分的逻辑实现时，我们实现了文件系统的基本功能，其中包括文件的创建，删除，文件夹的生成，删除，文件的读取，文件的编辑，目录的建立，磁盘的模拟等功能，并且对于磁道寻道算法进行了模拟实现，并且通过折线图进行寻道展示。当然除了这些之外，我们还进行了文件系统的UI实现，在这个过程中实现了对一个文件资源管理器的实现，能够将目录以界面的形式进行展示，并且能够将文件的属性和文件夹的基本属性进行展示。

在操作系统中，文件系统虽然相对独立，但与内存模块的交互却非常紧密。文件的读取和写入过程都离不开与内存模块的交互。因此，在实现这一过程中，需要与内存模块的同学进行频繁的沟通和合作，以商讨和定义两个模块之间信息传输的方式以及接口函数的设计等关键事项。只有当我们真正思考如何实现操作系统不同模块之间的交互时，才能真正体会到理论知识学习与实际应用之间的差异。在这个过程中，需要考虑的细节远超出我们的想象，因此必须细致入微地考虑每一个细节。这包括确保数据在内存和文件系统之间的正确传输、处理各种异常情况以及保证系统的高效性和稳定性。为此，我们需要深入研究和探讨内存管理、缓存策略、磁盘IO等相关概念和技术，以确保文件系统与内存模块之间的无缝衔接和协作。通过持续的交流、讨论和实践，我们能够建立起稳固而高效的文件系统与内存模块之间的良好协作关系，为用户提供可靠的文件读写服务。

当然，要成功完成这样一个项目，不仅需要掌握相关知识，还需要采用高效且完善的软件生命周期模型来进行项目开发管理。从最初的软件需求分析和软件设计，到后续的软件实现和代码测试，每个步骤都需要有相应的文档支持，以便于我们进行后续的开发和修改工作。通过这些文档，我们能够清晰地了解每个阶段的目标和需求，并有效地跟踪项目的进展。在软件生命周期的不同阶段，我们需要制定相应的文档，如需求说明书、设计文档和测试报告等。这些文档起到了重要的管理和沟通工具的作用，帮助我们明确软件开发的目标，协调开发团队的工作，并在项目进行过程中进行必要的调整和修改。文档的编写和管理能够确保项目的可控性和可维护性，并为后续的开发工作提供清晰的指导。此外，针对每个阶段，我们还需要进行代码优化工作。一旦测试完成，我们就可以针对代码进行优化，以提高程序的运行效率和响应速度。通过对算法、数据结构和编程技术的优化，我们可以减少资源消耗和提高系统性能，从而更好地满足用户的需求。

在本次项目中，我们使用了Python作为编程语言，并借助PyQt库实现了可视化部分。这个过程不仅提升了我的Python编码能力，还使我更加熟练地掌握了PyQt的使用。在这个过程中，我深入理解了Python的多线程编程和不同线程之间的信息通信机制，对PyQt中的信号和槽函数有了更深入的理解，使我能够更加灵活地利用这个工具。除了实验本身，作为项目的组长，我发现这个过程充满了挑战和磨砺。这个任务不仅考验了我对计算机专业知识，尤其是操作系统课程知识的掌握程度和综合应用能力，还对团队的协作能力、合理分工以及时间管理（高效推进项目）提出了较高要求。总体而言，本次实验给我带来了丰富的收获。它巩固了我在上学期所学操作系统相关知识，同时锻炼了我的文档编写和团队合作能力。此外，我对Python的使用也有了深入的理解。这次实验所获得的收获对未来的软件开发将极具帮助，为我奠定了坚实的基础。

* 李润杰

通过本次操作系统课程设计使我对整个操作系统的实现原理以及进程、文件、内存、设备、kernal、shell等操作系统的主要功能有了更深的理解和认识，我主要是负责进程管理部分的实现，彻底理解了进程的创建、执行、调度、阻塞、唤醒和结束这些过程，通过编写代码的形式实现了进程从创建到结束的整个进程生命周期，并能够记录进程对CPU和IO的使用情况，以Gantt图的形式进行展示。在编写代码的过程中，也加深了我对于四种进程调度算法的理解，将优先级抢占算法、优先级非抢占算法、先进先出算法、时间片轮转算法都成功实现，模拟了四种算法的调用过程。

虽然我负责的是进程管理部分，但我同时也要向内存申请进程空间、需要告知内存在进程结束时释放相应的内存资源，也要处理IO中断和IO结束产生的中断，因此，在实现的过程中也掌握了操作系统的整体实现过程，操作系统课程设计不像学习操作系统知识，在进行课程设计时需要对具体的细节进行研究和深挖，要和其它模块进行交流和接口函数的设计，需要我对整个操作系统的实现都要有一定的理解，而不是只关注进程部分的实现细节，因此本次课程设计不仅巩固了上学期所学到的操作系统的相关知识，也锻炼了我观察问题、解决问题的能力。

本次课程设计我也严格按照软件生命周期进行开发，每一阶段都有相应的文档和实现成果，完整的体验了软件开发的过程，锻炼了我们的工程化思维。同时，完成整个操作系统的模拟也需要我们进行小组合作，每个人负责一部分操作系统功能的实现，进行本地运行测试，然后再将所有部分进行整合连接，进行整体测试，完成了将复杂问题自顶向下，逐步细化，再将子问题解决方案合并为整体问题解决方案的过程，中途遇到问题或困难时也能及时与其它成员讨论，共同商量出完美的解决方案，锻炼了我的小组分工协作能力、团队合作意识以及沟通能力。

本次课程设计我们用的python语言开发，提高了我的python代码编写能力，系统的设计思路和算法也非普通的编程实验可以比拟的，锻炼了我的算法思维和解决问题的思路。总体来说，本次课程实验不仅巩固了上学期操作系统的所学知识，初步模拟了操作系统的实现流程；还提高了我的团队合作意识、沟通能力、文档撰写能力、算法实现思路和代码编写能力，能够严格按照软件工程的生命周期模型进行软件的开发和文档的撰写，对我未来进行软件开发和操作系统设计都有重大的帮助，使我受益匪浅。

* 孟宇航

通过本次对操作系统的设计和实现，我对整个操作系统的具体实现原理和各个部分具体设计实现有了更深的理解和掌握。在我们的本次实验中大致可以分为kernel、shell、文件系统、进程系统、内存管理、外部设备共六个部分，其中由我和李祥宇同学负责文件系统部分。我们共同实现了文件系统部分，在实现过程中，我深刻理解了一个文件是如何创建，存储，查找，访问，展示，删除的。通过编写代码，我们实现了一个文件相关的具体功能，在这个过程中，我对文件的物理结构和逻辑结构有了更深的认识和理解。因为文件系统和磁盘秘密相关，因此磁盘模块也是我们共同实现。我们将磁盘划分为两部分，成功用两个txt文件实现并存储了整个文件系统。对于磁盘寻道算法，我在前面遇到的问题也说过，我们采用模拟的形式实现。这样画出的磁道寻路图效果比较明显。

在本次实验中，我们文件系统不仅要负责自己的部分，我们还要与内存管理模块交流。他需要告知文件系统要访问的文件，文件系统要返回文件具体内容给内存系统，同时，内存修改了文件要写回磁盘，文件系统同样需要能够接收来自内存的文件。这是一个相互交互的过程。于这个过程中，我对操作系统整体实现又有了进一步了解，明白了具体实现的具体细节部分，这与书本中一句话简单描述不同，真正的交互是复杂的，是充满错误和困难的，要设计具体的接口函数，要与小组成员不断交流反馈，直到能够正常交互，不出错。在整个交流的过程中，我对操作系统的整体运行和操作有了一定认识和理解，复习巩固了以前学过的操作系统知识，对各个模块认识认知更深。同时增加了小组协作能力，并且文件系统是两人共同开发，对一些实现难免会有冲突，随着工作的推进和交流，我对协同开发具体方式有了实际的掌握。

在进行本次实验的同时，我们也在进行软件工程科目的学习，因此我们的软件按照软件工程的标准，每一个阶段都有相应的设计文档和实现成果，各个阶段逐步积累完成程序开发。整个开发过程体现了软件完整的生命周期。本次实验实现中，最重要的就是我们团队协作能力。小组成员从不熟悉编码习惯到流畅交流。各个成员完成自己模块，完成各个模块的单独测试，后面共同进行整合，进行测试，共同商量问题，进行代码优化，进行注释编写。这极大的锻炼了我们的团队协作能力，加强了我们的合作意识，体现了团队沟通的重要性。

本次实验我们采用python语言进行开发，并使用pyqt5进行可视化界面设计。通过本次实验，我对python语言有了进一步的熟悉和掌握。同时对pyqt5的使用也更为熟练，同时明白一个好的界面是多么难以实现。本次实验涉及的内容颇多，整个系统的涉及思路和算法对我来说都是不小的挑战，通过本次实验，锻炼了我的算法思维和设计思路及解决问题的能力，同时复习巩固了操作系统相关知识和软件工程相关知识。总的来说收获颇丰，希望在以后的学习实验中能有更多体会。

* 马天成

在编写操作系统课程设计外部设备模块的过程中，我获得了很多宝贵的经验和体会。以下是一些我所学到的重要方面：

操作系统接口的设计：在编写外部设备模块时，必须考虑如何与操作系统进行通信。这里的通讯对象主要是CPU模块。在之中我们商量了很多，主要是一些交互的细节和数据域的定义。只有定义好这些数据，设计操作系统接口，以便应用程序可以使用设备的功能。我学习了如何设计这些接口，并确保它们能够与操作系统进行良好的交互。

在编写操作系统课程设计外部设备模块这个过程中，我学会了许多技术方面的知识，比如首先复习了操作系统的基本知识，如物理内存虚拟内存，调度算法，文件系统以及CPU工作周期。之后，在编写的过程中，我也犯下了很多错误，比如一开始设计的模型就有问题，导致后续的调度算法会有一定的饥饿可能性；比如一开始的数据结构设计的数据域不完全，导致很多信息在使用时无法强关联，使得需要重新修改维护一个新的关联数据结构，花费了非常多的时间返工。

但这次实验我学会了很多，比如用Python写代码，我在其中熟悉了很多基本操作，也向团队介绍了package的制作过程。我在与CPU模块组员交互和设计的过程中，也体会到了初步设计的重要性，否则会使得工作十分不顺利。

* 覃韦唯

更加熟悉了操作系统，以及操作系统中的内存管理系统，在具体编码的过程中回归知识，搞清楚了之前 自己理解不清，或者是理解错误的地方，比如之前对访存的过程不清楚，对LRU表的原理和作用理解也 比较浅，在本次项目后，对他们都有了更深刻的理解。 深刻体会了团队协作的整个流程。在项目具体实施过程中，应该要有取舍，争取实现目标是符合小组水 平的，不能过高也不能过低。

* 锁佳鑫

在做课设的过程中我养成了记录的习惯,这样在写实验报告时能够比较完整的回忆起中间遇到的各种问题，与组员交流时也更加方便，表达的也更清晰，同时记录组员交流的重点内容，在遇到相应情况时能更快捷地解决。

我发现与组员的沟通交流很重要，例如一些细小的问题也要跟组员明确，这样会使得整个工程文件合并的时候出现的错误更少。

这次课程设计使我对linux内核的理解更加深刻，我作为将组员的部分整合到一起的角色，更加体会到封装和接口的重要性和简便性。

通过这次课程设计我明白了遇到了不确定的问题就要勇于地尝试，内核中有很多用户的指令，其中的每一个指令都有与其他模块交互的部分，这其中会出现很多微小的错误，需要细心、耐心地调试。