****

**MyOS v1.0**

**初版文档**

文 档 作 者： 李祥宇、李润杰、孟宇航、

马天成，锁佳鑫，覃韦维

学 院 ： 计算机学院

专 业 ： 计算机科学与技术

指 导 教 师 ： 李文生

**1 项目概况**

* 1. **开发目标**

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；

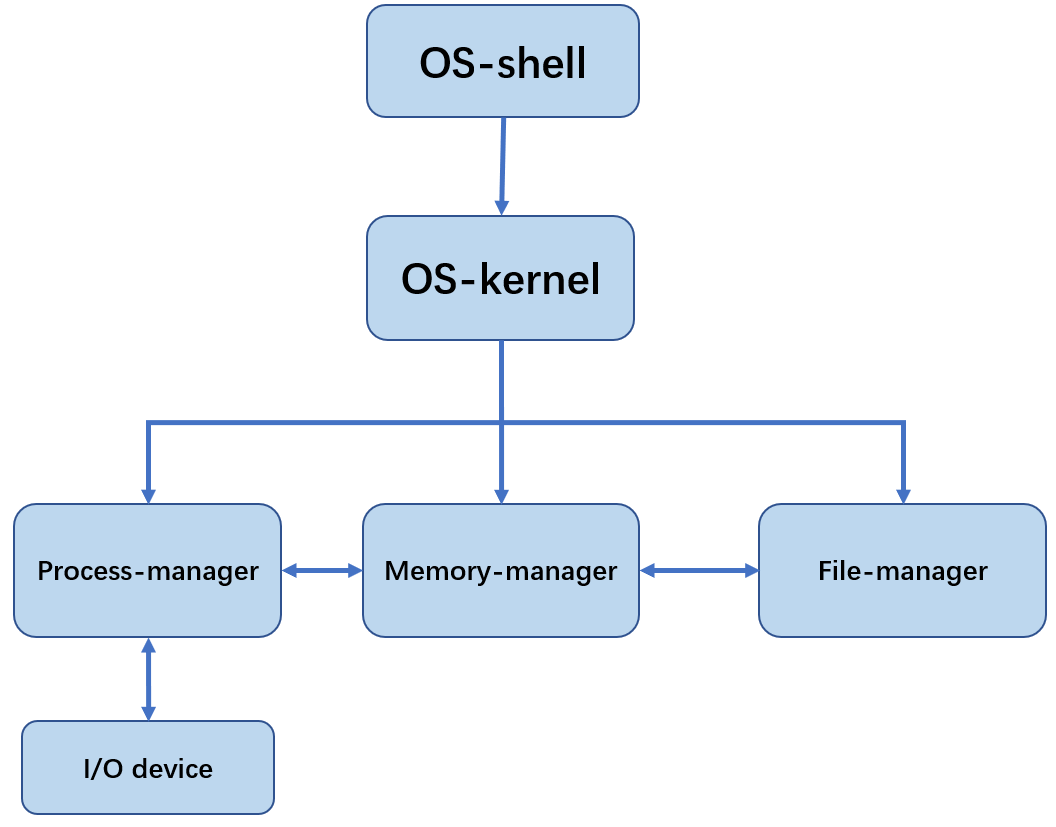
② 内存管理；

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。



**1.2 开发路线**

通过第一次会议讨论，我们选择了python3进行代码的编辑，python本身具有较好的跨平台特性，同时语言本身较为简易，能够减轻代码编辑的负担，同时使得小组成员能够更好的进行操作系统的设计与分析，以及相关算法的实现。不仅如此，通过第一次会议，我们也将项目大致分为了五个阶段，并严格执行预期计划。

第一阶段：明确项目需求，确定分工并进行初步的功能设计与分析

第二阶段：初版代码的编写及各模块的功能测试

第三阶段：进行模块的整合和不同模块间API的调试

第四阶段：进行模块功能的进一步完善，并且加入UI界面的设计

第五阶段：进行最终的功能测试，完成实验报告的编写和项目的总结

**1.3 小组分工**

|  |  |  |
| --- | --- | --- |
| **成员姓名** | **学号** | **成员分工** |
| 李祥宇 | 2020211375 | 文件管理模块 |
| 孟宇航 | 2020211377 | 文件管理模块 |
| 李润杰 | 2020211374 | 进程管理模块 |
| 马天成 | 2020211376 | 设备管理模块&部分进程 |
| 锁佳鑫 | 2020211380 | Kernel&shell模块 |
| 覃韦维 | 2020211381 | 内存管理模块 |

每位成员根据自己的分工负责自己部分的代码编写以及文档编辑，并且负责维护和更新对应代码块的功能。在完成了各个模块的编写和单元测试之后，由组长进行文档的整合，负责kernel部分的同学进行代码的集合调试，以验证各模块是否能够正常相互协作。

**2 各模块的需求分析及初步设计**

**2.1 系统基本需求分析及模块划分**

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，我们确定了初版的模块划分：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和内存资源，对处在就绪态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和内存资源的分配情况 |
| 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。 |

**2.2 模块结构与功能初步设计**

**2.2.1 Shell&Kernel**

**2.2.1.1 基本功能分析**

**Kernel需求：**

1. **系统启动和关闭管理：**Kernel需要能够管理系统的启动和关闭过程，并确保系统能够在安全的条件下启动和运行。
2. **内存管理：**Kernel需要能够有效地管理系统内存，包括内存的分配和释放，以及处理内存不足的情况。
3. **文件系统管理：**Kernel需要能够管理系统的文件系统，包括创建、删除、读取和写入文件，并确保文件系统的安全性。
4. **设备驱动程序管理：**Kernel需要能够管理系统的设备驱动程序，包括对设备的控制和数据传输。
5. **进程管理：**Kernel需要能够管理系统中的进程，包括创建、删除、调度和监控进程，并确保进程之间的安全和资源分配的公平性。

**Shell需求：**

1. **用户界面：**Shell需要提供一个友好的用户界面，让用户能够轻松地与系统进行交互。
2. **命令解析和执行：**Shell需要能够解析用户输入的命令，并执行相应的操作。这包括内置命令和外部命令。
3. **命令历史记录和自动完成：**Shell需要记录用户执行的命令历史，并支持自动完成功能，以提高用户的效率和准确性。
4. **管道和重定向：**Shell需要支持管道和重定向操作，以允许用户将多个命令连接起来并重定向命令的输入和输出。
5. **脚本编写和执行：**Shell需要支持脚本编写和执行，以允许用户将多个命令组合成一个脚本并自动执行。
6. **用户权限管理：**Shell需要支持用户权限管理，以控制用户对系统资源的访问和使用。

**2.2.1.2 模块初步设计**

**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 |
| chprmis·[文件名或目录名][权限] | | 更改文件或目录使用权限 |
| 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 -p | 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。 | |
| snapshot -e | 向用户展示设备队列的的情况，大致包括：设备名称、各设备队列情况、设备是否空闲等等。 | |
| 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 -s | 展示系统外存各磁盘块的占用状态 | | 内存模块 |
| display -m | 展示系统物理内存占用状态 | |
| 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 | |

**主要方法设计：**

**Kernel：**

**·def display\_command\_description:**

用字典类型存储所有的指令的用途和用法，可以向用户展示所有的指令（help），还可以通过字符串比较来检查用户输入的指令是否合法。

**·def monitoring：**

监视方法，用来监视所有资源的调度，方便kernel对系统进行管理，以及面向用户输出各资源的使用情况（例如用户输入snapshot -e查看设备队列的情况时，monitoring方法可以提供设备的状态）

**·def report\_error:**

用户输入不合法时报错

**·def run(self):**

利用if-else分支运行用户输入的指令

**Shell：**

**·def get\_split\_command**

接收用户输入的指令或指令集

**2.2.1.3 测试方案**

**Kernel测试：**

Kernel和shell的测试具体体现在对各个模块的分别测试和整合测试，因此kernel和shell’的测试应放于各个模块测试成功之后或已实现部分模块功能之后进行测试。

暂定设计思想：

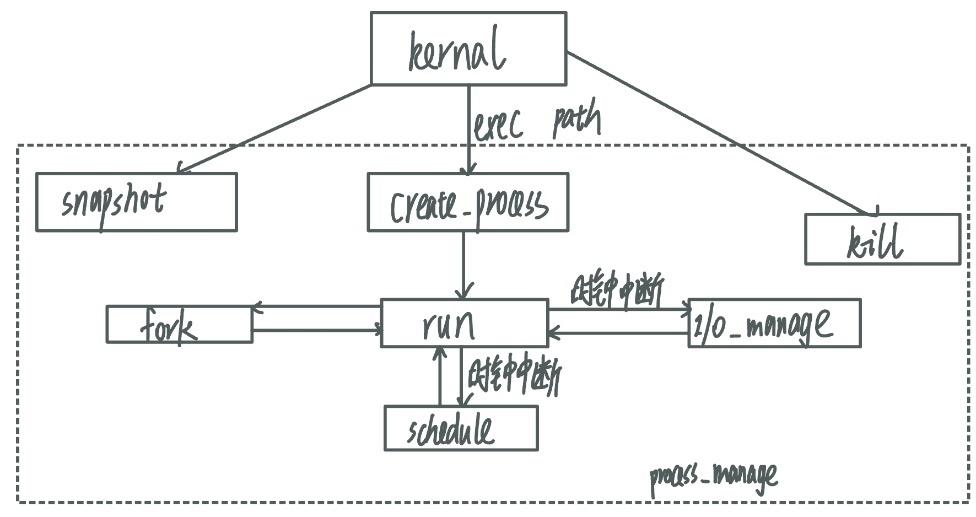
* 尝试使用命令行界面启动系统
* 确认内核支持所需的硬件和设备。
* 确认内核是否正确处理和保护系统的资源，例如进程、内存和文件系统。
* 尝试使用命令行界面关闭系统

**Shell测试：**

* 测试shell对常用命令和语法的支持，例如cd、ls、mkdir等。
* 确认shell支持管道和重定向等高级命令。

**2.2.2 Process\_manager**

**2.2.2.1 基本功能分析（主要展示模块，部分功能性函数不做展示）**



* create\_process模块：负责创建一个进程并将其放入就绪队列中，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。
* 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id为当前进程，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）。
* I/O\_manage模块：对于模拟的输入输出设备进行管理。
* run模块：整个进程管理的核心模块，主要功能是根据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应该强制终止。
* snapshot模块：根据参数输出当前设备使用情况或PCB队列中的所有进程的详细信息，相当于对整个系统当前状态进行一个快照。

**2.2.2.2 模块初步设计**

**PCB的结构设计（不包括\_\_str\_\_、\_\_lt\_\_、\_\_eq\_\_方法）**

进程PCB的主要属性有进程的父进程pid（由操作系统创建就为0，fork创建则为父进程的pid）、本进程pid、进程的优先级（数越小优先级越高）、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）、进程的创建时间、进程大小、进程的类型（是由操作系统创建还是fork创建）、进程状态。

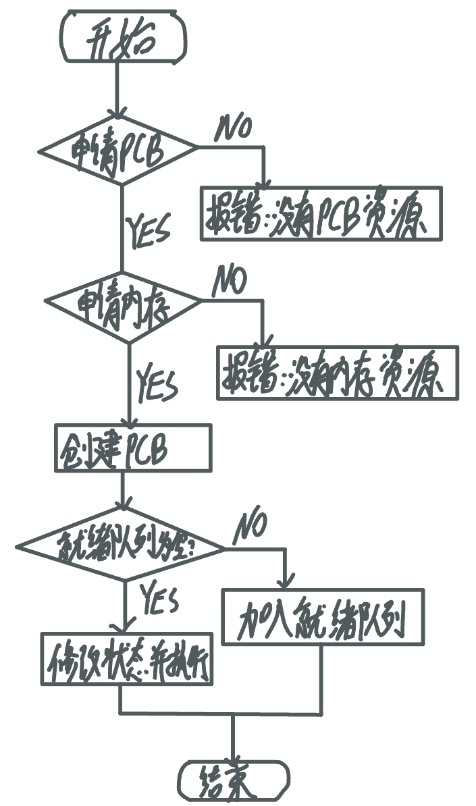
**进程管理的结构设计**

进程管理的主要属性有当前分配的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）、进程就绪队列、当前正在运行的进程cur\_process、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、标记是否有新进程加入的标志位new\_process、进程等待队列。

**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分配的，二者都涉及到申请PCB队列和内存资源的过程，这里就主要说明进程创建。

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，如果就绪队列此时为空则直接将进程状态改为running并运行，同时告知进程管理有新进程加入。



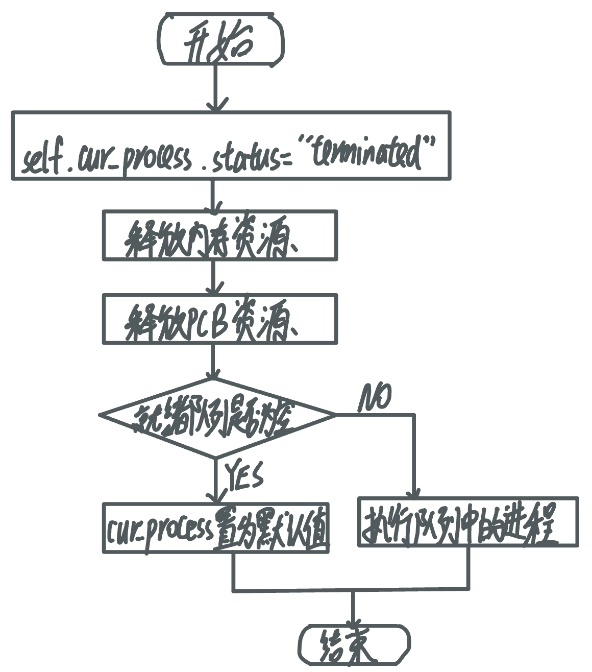
**进程的调度算法**

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，暂定算法有FCFS、抢占式优先级调度、非抢占式优先级调度和轮询算法。截止到目前为止，我们已经完成了抢占式优先级调度的编写，这部分代码同时涉及到优先级调度和抢占式调度，完成了这部分的编写，对于FCFS、非抢占式优先级调度和轮询算法的编写就更加游刃有余。

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中断事件，此时就会调用schedule函数去判断是否有抢占动作，new\_process是用来判断在这一个时钟周期内是否有新的进程加入到PCB队列中，如果有就去查看当前队列中优先级最高的进程是否发生变化，如果发生了变化就进行抢占，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。

**进程的阻塞、唤醒、结束**

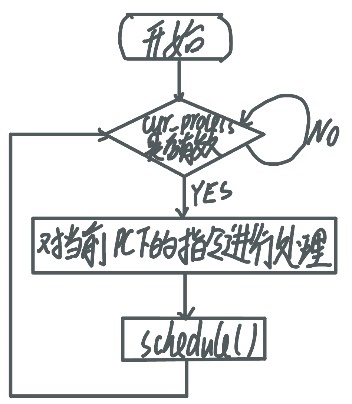
进程阻塞主要按照对应的算法规则进行实现，发生抢占导致进程阻塞时，会将进程状态改为ready，然后添加进就绪队列中，此时进程的程序计数器pc会记录当前指令进行到什么位置。进程唤醒主要通过PCB中的程序计数器pc来进行，每次进程唤醒后都会从pc处开始执行。（详见调度算法代码中发生抢占时的处理）下面是进程结束部分的流程图：



进程结束或撤销时首先先修改进程的状态，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。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。

**run模块的结构设计**

run模块实际上是一个循环的过程，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。

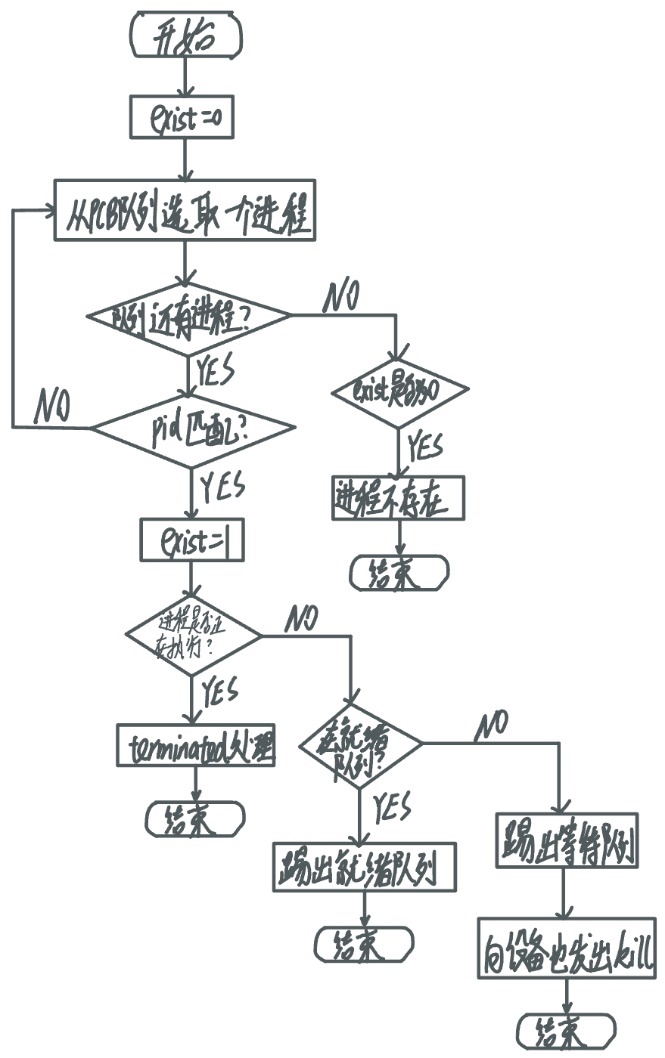


**snapshot模块的结构设计（进程部分）**

这部分实现比较简单，由于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。

**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和内存资源后需要将其从ready队列中删除；如果进程处在waiting对列中则释放掉PCB和内存资源后需要将其从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。



**进程对于I/O设备的功能支持**

1. Output的发起（传输字节和相应控制信息）
2. Output执行结果的接受和处理（可能中断处理）
3. Input的接收和执行（传输字节和相应控制信息，以及根据系统当前状态进行处理，可能中断执行）
4. Input执行结果的发送和处理

所以对于CPU进程调度模块我们能看到的主要是4个功能：

1. 发送任务

2. kill任务

3. 根据时钟周期跑设备管理

4. 获取设备运行结果

**用户接口设计**

**指令集：**

* **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。
* **cpu time:** 模拟使用cpu，时长为time（整型）个时钟周期。
* **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。
* **write filename order:** 将order内容写进给定文件并保存，时长为1个时钟周期。
* **access address:** 访存指令，访问逻辑地址为address的内存内容，时长为1个时钟周期。
* **input/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。
* **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。

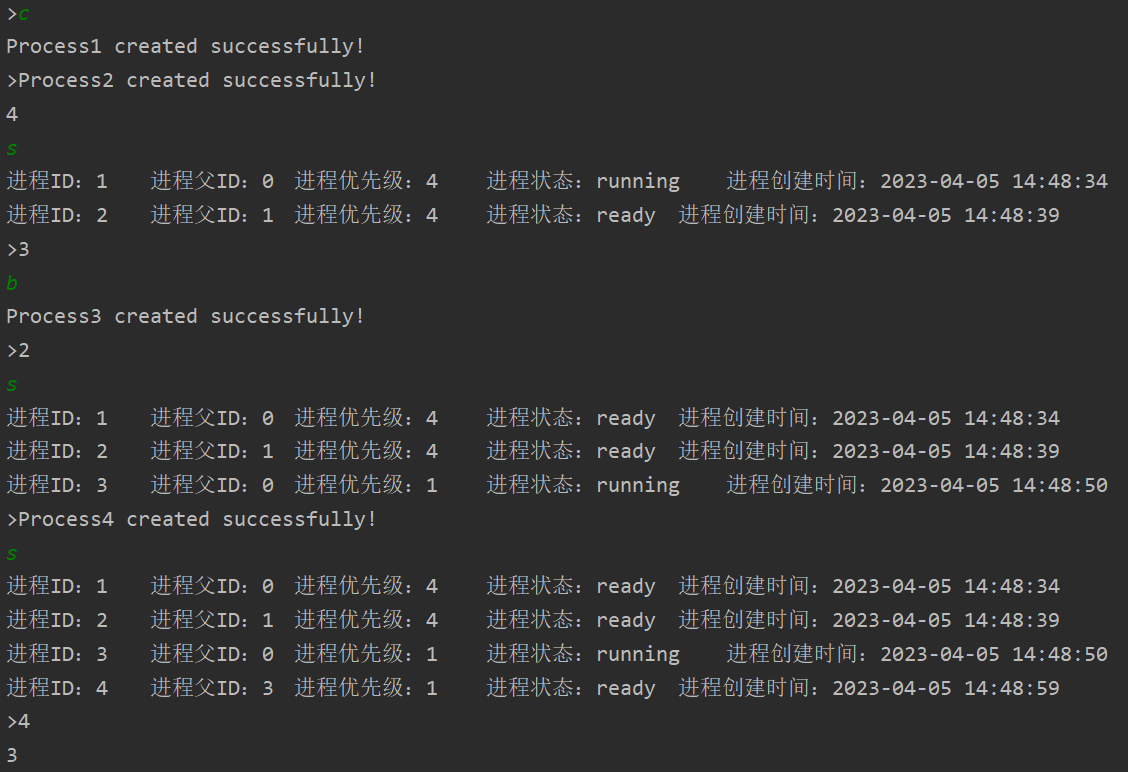
**用户指令：**

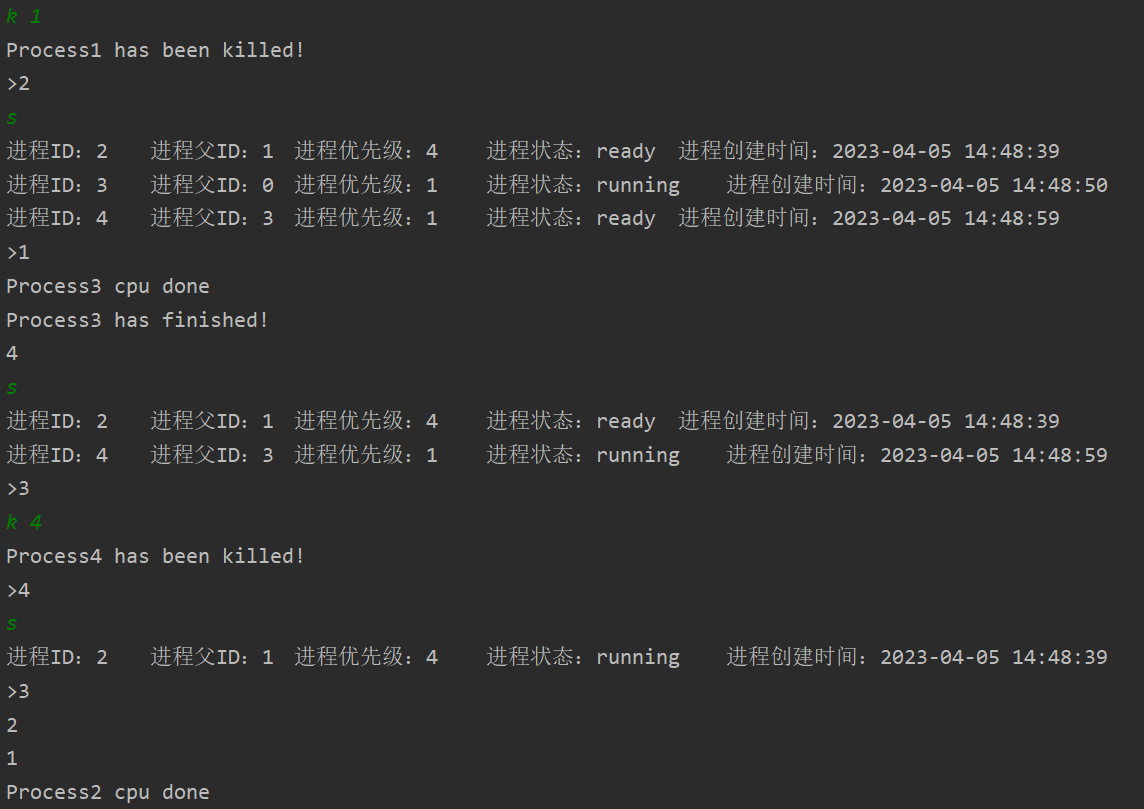
* **exec path：**创建进程并执行用户所指定的文件路径的文件内容。
* **snapshot -p：**向用户展示各并发进程的状态，大致包括：进程的pid、优先级、创建时间和进程当前时刻所处的状态。
* **snapshot -e：**向用户展示设备队列的的情况，大致包括：设备名称、各设备队列情况、设备是否空闲等等。
* **kill pid：**强行终止一个用户指定pid的未完成的进程，并将其所占用的内存和设备资源全部释放。

**2.2.2.3 测试方案**

对于进程管理的测试，我们的设想是首先需要通过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 -p指令来对任一时刻各进程的状态进行监测，同时，通过snapshot -e指令来对任意时刻cpu以及外设的占用情况进行检测，通过这两个指令相结合来测试进程是否创建成功，是否按照正确的调度算法进行进程的调度，在调度过程中是有进程的阻塞（running->waiting/ready）与唤醒过程（ready->running），对设备的使用是否按照设备队列中的顺序进行，在进程状态到达terminated时，再通过内存提供的相关命令也可以观察进程结束时是否释放了相应的资源。以此来测试整个进程管理模块。

目前进程管理的基本功能已经实现的差不多，也设计了几个单元测试函数来测试进程管理的功能，但目前的测试只是针对进程管理模块的单元测试，并不是对整个操作系统的系统测试，系统测试还需要与其它管理模块结合后才可以测试，单元测试结果如下：







**测试情况说明**

**指令部分：**c为创建进程指令（优先级随机）。b为创建进程指令（优先级最高）。

s为snapshot指令，展示所有进程信息。k为kill指令，需要进程pid作为参数。

**进程文件内容：**”fork\ncpu 5\nend”（所有的进程文件都是这个内容）

**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，因为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，这里我们通过s指令看到此时确实有两个进程，第一个进程在running，而第二个进程由于是fork出来的子进程，所以目前处于ready状态.

接下来我们执行了b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，所以目前还是第一个进程在执行，这就是为什么3变成了2，但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s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，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s指令发现进程1的信息已经没有了，在进程3执行完毕后再次调用s指令发现此时只有进程2和进程4，由于进程4的优先级比进程2高，因此进程4开始执行，我们在进程4执行的过程中强制终止，可以通过s指令看到进程4尽管正在执行但也终止成功此时只有进程2在执行，当进程2执行完毕后再使用s指令会提示此时已经没有进程处在running、ready、waiting状态中。至此，我们测试了抢占式优先级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（进程）指令的正确执行并能够通过s指令来实时监控系统内的进程状态信息。

**2.2.3 file\_manager**

**2.2.3.1 基本功能分析**

* 存储管理：文件系统需要能够存储大量的文件和数据，而且应该有足够的存储容量来支持将来的扩展。
* 文件管理：文件系统需要提供一种方便的方法来创建、修改、删除和查找文件。同时也需要能够支持文件的元数据，如文件名、创建日期、大小等。
* 目录管理：文件系统需要提供一种有效的方式来组织文件，以便用户可以方便地找到所需的文件。这可以通过目录（文件夹）的创建和管理来实现。
* 安全性：文件系统应该保证用户数据的安全性。这可以通过访问控制、加密和备份来实现。
* 效率：文件系统需要在存储和检索文件方面具有高效性，以便用户可以快速地获取他们需要的文件。

**2.2.3.2 模块初步设计**

**文件**

* 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
* type 表示文件类型
* size 表示文件所占磁盘空间大小，单位为 byte
* 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）
* ctime 创建时间，记录首次创建文件时间
* mtime 修改时间，记录最后一次修改文件时间
* content 为模拟文件的内容
* class File:  
   def \_\_init\_\_(self, name='', prmis='rwe'):  
   self.name = name # 文件的名字  
   self.size = 0 # 文件的大小  
   self.prmis = prmis # 文件的权限  
   self.type = '' # 文件类型  
   self.ctime = '' # 文件创建时间  
   self.mtime = '' # 文件的修改时间  
   self.content = '' # 文件内容  
   self.loc = [] # 记录文件占用块数

**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。

FCB类代码设计如下

class FCB:  
 def \_\_init\_\_(self, name, loc):  
 self.name = name # 文件的名字  
 self.loc = loc # 文件在磁盘中位置，对应的磁盘块号

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。

磁盘类代码设计如下

class Disk:  
  
 def \_\_init\_\_(self, track\_number, sec\_number, block\_size):  
 self.track\_number = track\_number # 磁道数  
 self.sec\_number = sec\_number # 扇区数  
 self.block\_number = track\_number \* sec\_number # 磁盘块的总数  
 self.block\_size = block\_size # 数据块大小  
  
 self.block\_free\_number = self.block\_number # 初始空闲块数量等于磁盘块总数  
 self.free\_map = np.zeros([track\_number, sec\_number], int)  
  
 self.blocks = [[Block(block\_size) for i in range(0, self.sec\_number)] for i in range(0, self.track\_number)]

**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。**

磁盘块类代码设计如下

class Block:  
  
 def \_\_init\_\_(self, block\_size):  
 self.block\_size = block\_size # 块内的大小  
 # 文件项  
 self.file = File()  
 self.file.empty\_file()

**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。

磁盘块类代码设计如下：

class FileManager:  
 file\_sep = os.sep # 获取系统文件分隔符  
 root\_path = os.getcwd() + file\_sep + 'OS\_files' # 根目录  
  
 def \_\_init\_\_(self, track\_number, sec\_number, block\_size):  
 self.disk = Disk(track\_number, sec\_number, block\_size)

# 字典树初始化

self.file\_dict = self.init\_file\_dict()  
  
 # 当前目录，初始化为根目录  
 self.cur\_dict = self.file\_dict['/']  
 self.cur\_path = ['/']

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

**外部接口：**

1. **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**：

-a 显示所有文件及目录 (. 开头的隐藏文件也会列出)

-d 只列出目录（不递归列出目录内的文件）。

-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

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

1. **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**

1. **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。

1. **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。

1. **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。**

1. **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**

1. **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。**

1. **chpmis(mode='',name='')**

**描述：更改文件或目录使用权限**。path为当前路径；mode为权限模式,，默认全部权限；name为文件或目录名。

**mode**相关参数：

-w 写权限

-r 读权限

-e运行权限

1. **redistd(void)**

**描述：整理已分配磁盘块，减少外部碎片。**执行函数将整理磁盘使用空间，减少外部碎片，增加可利用空间

**2.2.3.3 测试方案**

**文件系统命令方面：**对于文件系统命令的测试，主要测试思路为在命令行界面，编写测试设计命令的代码并执行，根据执行效果判断命令是否设计成功，此步骤会综合测试命令的不同参数mode。

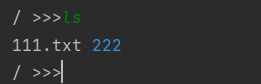
**磁盘方面：**对于磁盘分配算法的测试，可以通过观察位示图的分配情况；对于磁头寻道算法，可以通过输出磁道访问序列，来判断算法是否正确运行。

我们将按照以上测试思想，设计对应测试集。

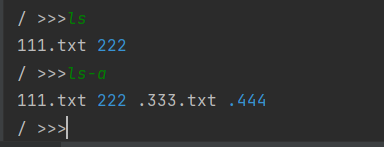
**基本功能测试**

ls函数功能测试：

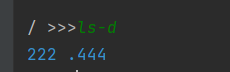
无参数：



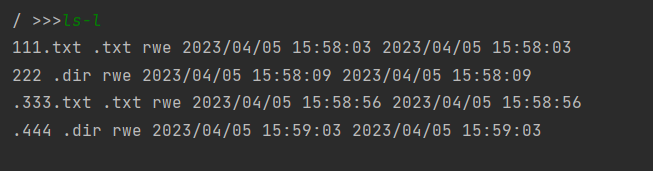
-a：



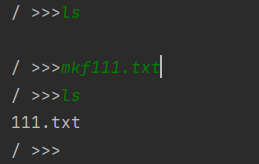
-d:



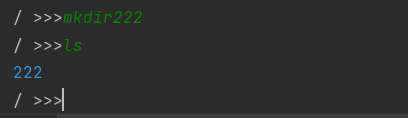
-l:



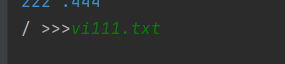
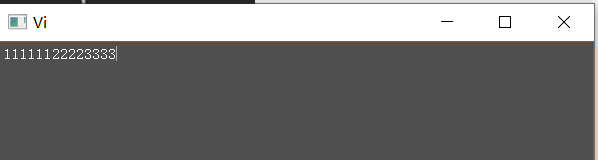
mkf函数测试：



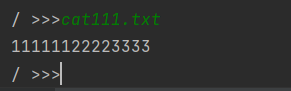
mkdir函数测试：



vi函数功能测试：

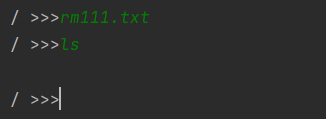
 

cat函数功能测试：

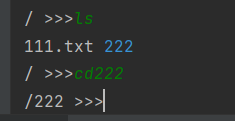


rm函数功能测试：

无参数：



cd函数功能测试：



**安全测试**

1. 测试文件和文件夹的权限设置是否生效
2. 测试文件系统的数据安全性和完整性
3. 测试文件系统的访问控制和安全性策略是否生效

**2.2.4 Memory\_manager**

**2.2.4.1 基本功能分析**

1. **内存分配：**操作系统需要为每个运行的程序或进程分配一定的内存空间，以便程序可以存储和执行它的代码和数据。内存管理器可以根据程序的需求动态地分配内存空间，同时保证不同程序之间的内存空间互相隔离，防止程序之间的内存干扰。
2. **内存回收：**当程序结束运行时，它所占用的内存空间需要被回收，以便其他程序可以使用这些空间。内存管理器可以监控程序的状态，并及时回收已经不再使用的内存空间。
3. **虚拟内存管理：**虚拟内存是一种内存管理技术，可以将物理内存和磁盘空间组合成一块连续的内存空间，使得程序可以访问比物理内存更大的内存空间。内存管理器可以使用虚拟内存管理技术，对内存空间进行动态调整和管理，以便程序可以访问更大的内存空间。

**2.2.4.2 模块初步设计**

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。具体设计方案如下。

|  |
| --- |
| **class** **MemoryManager:**  **def** \_\_init\_\_**(**self**,** pageSize**,** phyPageNum**,** virPageNum**):**  self**.**pageSize **=** pageSize  self**.**phyMem **=** Memory**(**pageSize**=**pageSize**,** pageNum**=**phyPageNum**)**  self**.**virMem **=** Memory**(**pageSize**=**pageSize**,** pageNum**=**virPageNum**)**  # key = pid, value = ProcessMemory  self**.**processTable **=** **{}** |

MemoryManager管理所有物理内存和虚拟内存，processTable为字典，以进程ID为键，每一个单独的processMemory为值。

|  |
| --- |
| **def** mallocMem**(**self**,** pid**,** size**):**  **def** accessMem**(**self**,** pid**,** address**):**  table **=** self**.**processTable**.**get**(**pid**)**  **if** table **is** **not** **None:**  changes **=** table**.**access**(**address**)**  **def** freeMem**(**self**,** pid**,** size**):**  **def** freeAllMem**(**self**,** pid**):** |

MemoryManager对外提供四个个函数，对进程的释放、占用和访问内存的请求做出响应。对内帮助process Memory进行虚存和实存的交换。

|  |
| --- |
| **class** **Memory:**  **def** \_\_init\_\_**(**self**,** pageSize**=**1024**,** pageNum**=**4096**):**  self**.**pageSize **=** pageSize  self**.**pageNum **=** pageNum  # record used memory, kb  self**.**usedMem **=** 0  self**.**maxMem **=** pageSize **\*** pageNum  # [how many use:<= pageSize]  self**.**pageTable **=** **[**0 **for** i **in** **range(**self**.**pageNum**)]**  **def** malloc**(**self**,** pid**,** size**):**  num **=** size **//** self**.**pageSize **+** 1  offset **=** size **%** self**.**pageSize  ret **=** **[]**  **sum** **=** 0  **for** i **in** **range(**self**.**pageNum**):**  **if** self**.**pageTable**[**i**][**0**]** **==** 0**:**  ret**.**append**(**i**)**  **sum** **+=** 1  **if** **sum** **==** num**:**  **for** j **in** **range(len(**ret**)):**  self**.**pageTable**[**ret**[**j**]][**1**]** **=** pid  **if** j **==** **len(**ret**)** **-** 1**:**  self**.**pageTable**[**ret**[**j**]][**0**]** **=** offset  **else:**  self**.**pageTable**[**ret**[**j**]][**0**]** **=** self**.**pageSize  **return** ret  **return** **-**1  **def** free**(**self**,** pid**,** size**):**  **for** i **in** self**.**pageTable**:**  **if** i**[**1**]** **==** pid**:**  i**[**1**]** **=** **-**1  i**[**0**]** **=** 0 |

Memory模拟内存，每一页为pageTable数组里的一项，在数组中的位置即是页号，元素值代表该页被占用的kb数，接收MemoryManager释放或请求内存的请求，所需的页号以数组方式返回。

ProcessMemory类为管理单独的进程，pageTable为页表，进行进程中页号与实存和虚存的映射，pageQueue为LRU队列，其长度为该进程分配的物理页数量，nowVir属性表示目前该进程已经申请的虚页数。

|  |
| --- |
| **class** **ProcessMemory:**  **def** \_\_init\_\_**(**self**,** allowedPhyPage**=**16**,** allowedVirPage**=**64**,** pageSize**=**1024**):**  # list, [virPageNum, phyPageNum, how many use:<= pageSize, vaildation]  self**.**pageTable **=** **[[-**1**,** **-**1**,** 0**,** **-**1**]** **for** i **in** **range(**allowedVirPage**)]**  # list, use LRU to complete page replacement, [proPageNum]  self**.**pageQueue **=** **[]**  # to control the size of queue  self**.**maxQueue **=** allowedPhyPage  self**.**maxVirPage **=** allowedVirPage  self**.**nowQueue **=** 0  self**.**nowVir **=** **-**1  self**.**pageSize **=** pageSize |

向上提供modify函数使得MemoryManager能修改其页表。内部维护LRU队列，及时通知MemoryManager进行页面的换入换出。

|  |
| --- |
| **def** pageReplace**(**self**,** num**):**  **for** i **in** self**.**pageQueue**:**  **if** i **==** virPageNum**:**  self**.**pageQueue**.**remove**(**i**)**  self**.**pageQueue**.**append**(**i**)**  **return** **[-**1**,** self**.**pageTable**[**num**][**1**]]**  **if** self**.**nowQueue **<** self**.**maxQueue**:**  self**.**pageQueue**.**append**(**virPageNum**)**  self**.**nowQueue **+=** 1  **return** **[**virPageNum**,** **-**1**]**  **else:**  self**.**pageQueue**.**append**(**virPageNum**)**  **return** **[**virPageNum**,** self**.**pageQueue**.**pop**()]**  **def** access**(**self**,** address**):**  proPageNum **=** address **//** self**.**pageSize  offset **=** address **%** self**.**pageSize  # overstep the boundary  **if** proPageNum **>** self**.**nowVir**:**  **return** **-**1  **if** self**.**pageTable**[**proPageNum**][**2**]** **-** 1 **<** offset**:**  **return** **-**1  self**.**pageReplace**(**self**.**pageTable**[**proPageNum**][**1**])**  **def** modify**(**self**):** |

**2.2.4.3 测试方案**

1. **内存分配测试：**编写测试用例，通过MemoryManager的alloc方法申请内存空间，并检查内存是否分配成功。测试用例应包括正常情况下的内存分配和异常情况下的内存分配，如内存不足等。
2. **内存回收测试：**编写测试用例，通过MemoryManager的free方法释放已经申请的内存空间，并检查内存是否释放成功。测试用例应包括正常情况下的内存回收和异常情况下的内存回收，如重复释放已经释放的内存空间等。

**2.2.5 Device\_manager**

**2.2.5.1 基本功能分析**

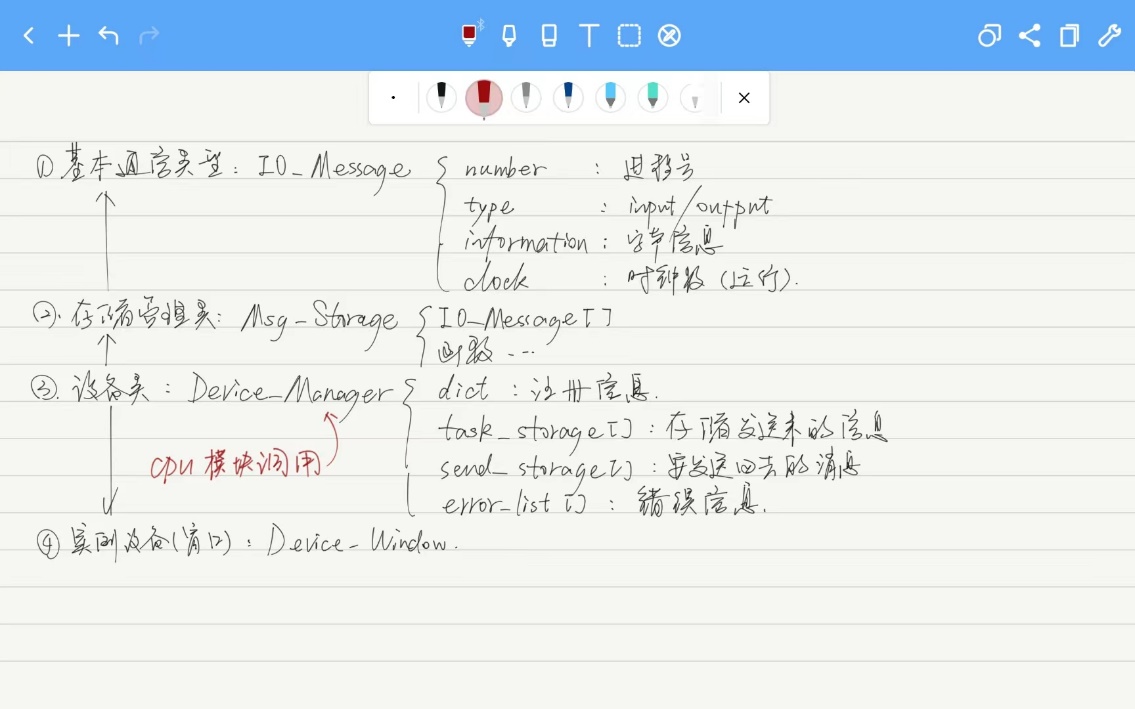
**操作系统CPU相关的需求：**

1. Output的发起
2. Output执行结果的接受和处理
3. Input的接收和执行
4. Input执行结果的发送和处理

**设备管理器**

1. Output指令的接收
2. Output指令的分配（给特定实例外部设备或者）
3. Output指令的执行结果发送（未找到实例外部设备执行或者执行完成或者执行错误）
4. Input指令的生成（根据实例外部设备传输的信息）
5. Input指令的传输
6. Input指令执行结果接收和处理

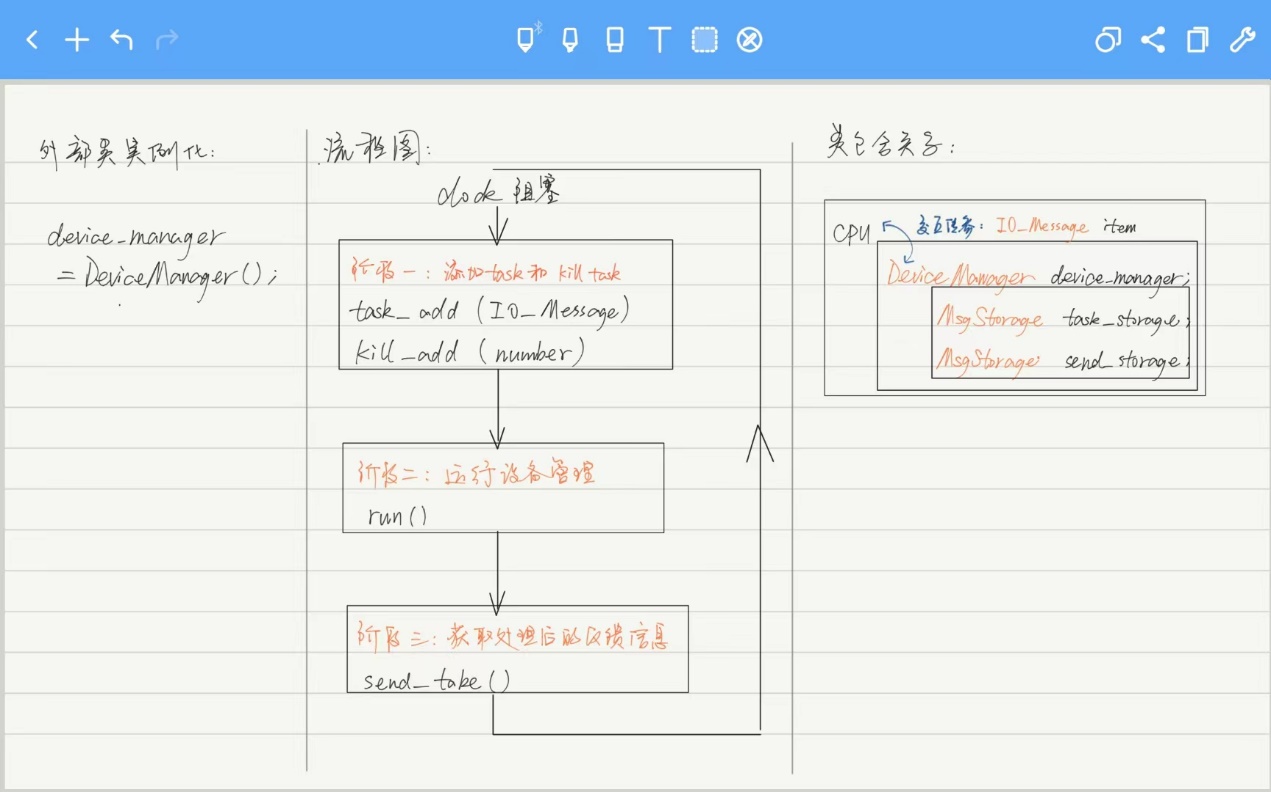
**2.2.5.2 模块初步设计**



因为很多数据类型封装是很基本的，所以我们这里只展示设备管理器的类中的对外接口：

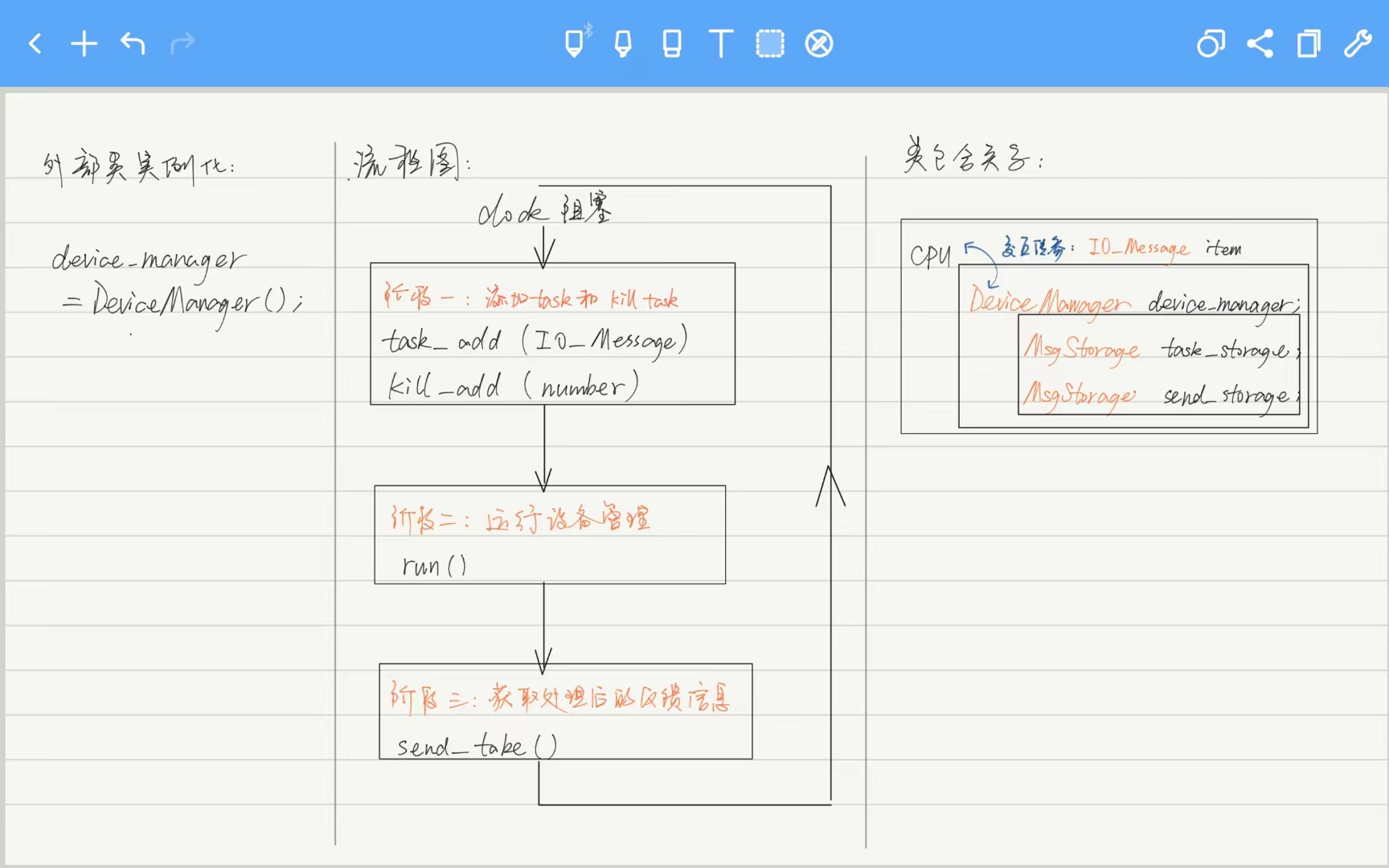
1. 外部类实例化：

为了方便其他模块调用，我们将整个设备管理器封装成package，并且只提供一个实例化接口，实现了高度集成，也方便其他模块调用。



2. 实例化后类包含关系设计

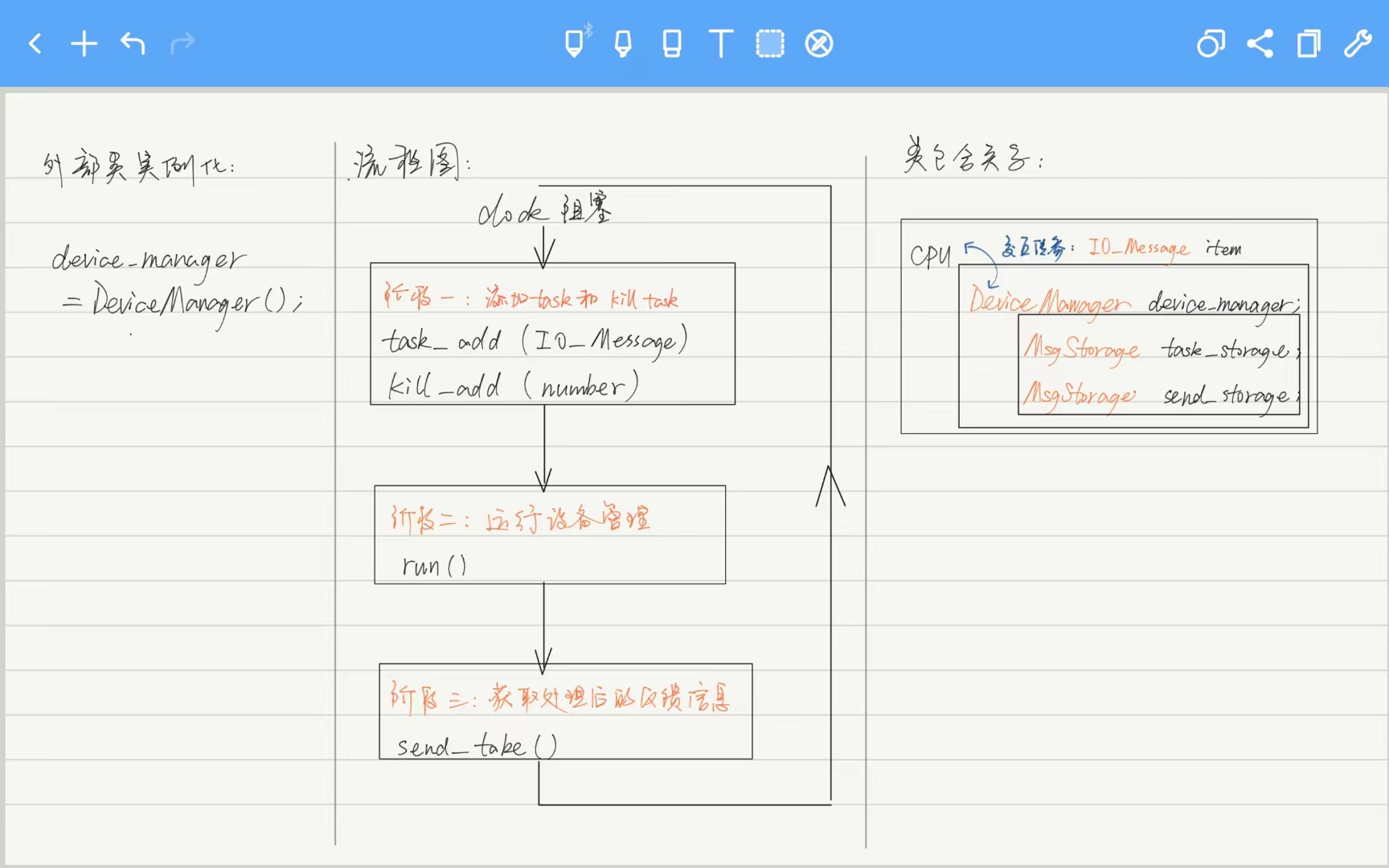
其他模块（主要针对CPU模块）实例化后，会生成一个DeviceManager的对象。在此对象里，也会生成自己定义的类：MsgStorage，两个分别对应发送来的人物列表和需要反馈的信息列表。在其中我们更是设计了一个通用交互类IO\_Message，规整化整个传参和反馈数据。MsgStorage实质上就是管理这种信息的一个数组列表。



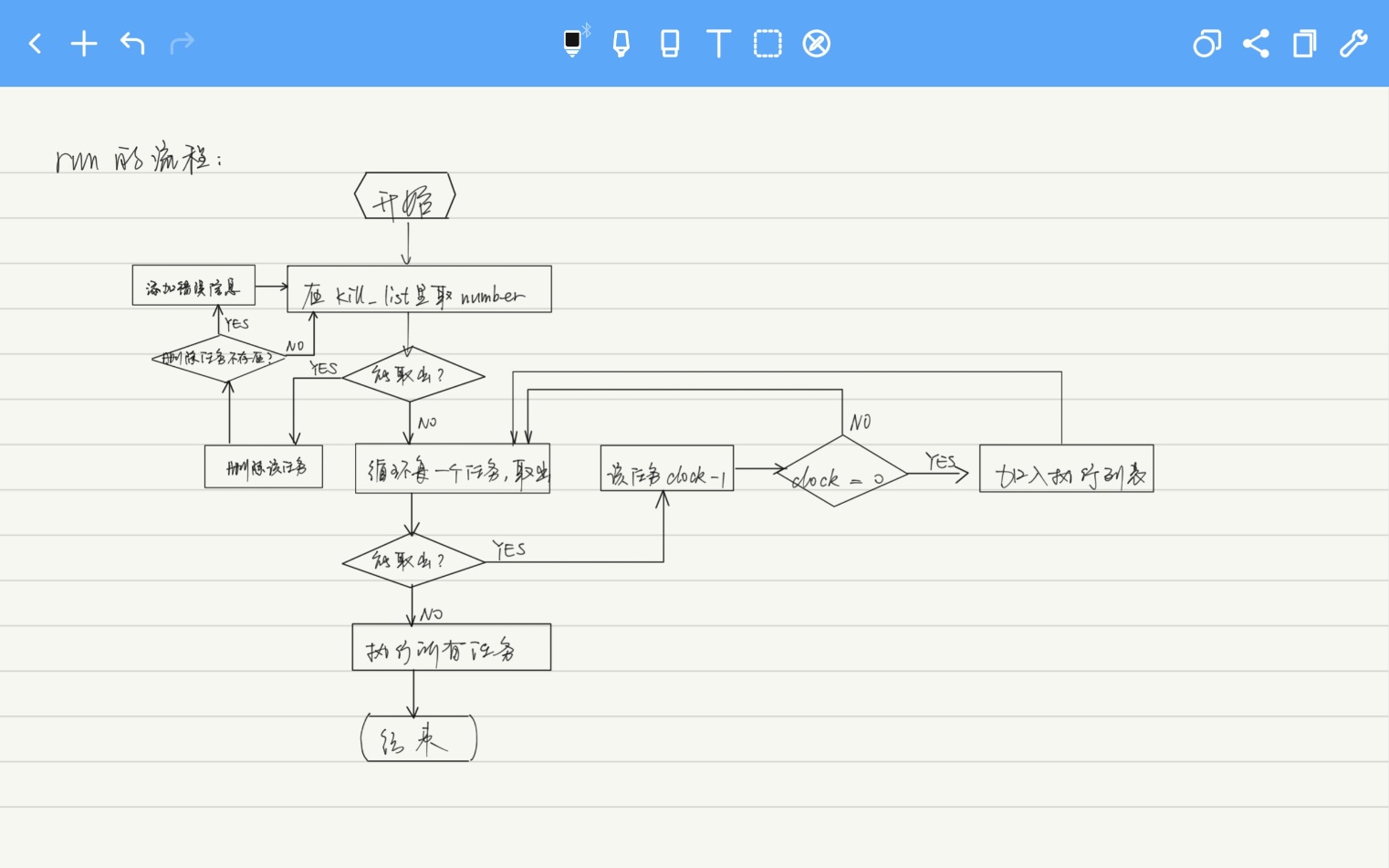
3. 对外接口调用流程图：

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）



4. run函数详细设计：



整个流程主要分几个阶段：

* 根据kill列表删除任务
* 循环每一个存在的任务，clock-1，到0则添加到执行列表
* 执行里表列的任务执行

我们会处理一些必要逻辑，以保证正常运行：

* 不可以多次删除同一个number号对应的任务
* 不可以调用不存在的设备

以上两个都会触发错误信息

5. 后续改进：

* 首先是运行设计：我们需要设计正确的设备任务队列调度算法，尽量保证不饥饿。
* 其次是实例化设备设计：争取设计出更好的模拟设备。

|  |
| --- |
| **class** DeviceManager:  *# 初始化函数* **def** \_\_init\_\_(self):  *# input，output存储数据（运行成功的进程会一一对应）* self.task\_storage = MsgStorage(TASK\_SIZE)  self.send\_storage = MsgStorage(SEND\_SIZE)  *# kill队列，反馈队列和错误信息队列（后两者作为反馈信息给进程）* self.kill\_list = []  self.send\_list = []  self.error\_list = []  *# 获取注册信息* device\_dict = json.loads(open(**'data\_devices.json'**, **'r'**, encoding=**'UTF-8'**).read())  self.input\_device\_dict = device\_dict[**"InputDevice"**]  self.output\_device\_dict = device\_dict[**"OutputDevice"**] |
| *# 放入单个任务* **def** task\_putInSingle(self, item: MessageUnit):  **if** item.type != **"input" and** item.type != **"output"**:  self.error\_list.append((id, str(item.id)+**".task.type is illegal: "**+item.type))  **return False  return** self.task\_storage.add(item) |
| *# 加入删除队列* **def** kill\_add(self, id):  *# 该删除已经有重复，必须增添错误信息* **if** id **in** self.kill\_list:  self.error\_list.append((id, **"Kill "**+str(id)+**" repeated"**))  **return False** self.kill\_list.append(id)  **return True** |
| *# 取出所有反馈信息*  **def** send\_takeAwayAll(self):  *# 取出所有的send\_storage送到反馈队列，并清空这些消息* **for** index **in** range(self.send\_storage.size):  self.send\_list.append(self.send\_storage.msg\_byIndex(index))  self.send\_storage.clear()  **return** (self.send\_list, self.error\_list) |
| *# 每个时钟驱动一次运行函数（kill的优先级最高）* **def** run(self):  *# 1 - kill任务，遍历input和output存储，删除对应的任务，并直接生成返回信息* **for** id **in** self.kill\_list:  **for** item **in** self.task\_storage.list:  **if** item.id == id:  self.task\_storage.remove(id)  print(**"Manager - killed: "**, id)  self.send\_storage.add(MessageUnit(item.type, item.id, **"Has been killed"**, item.clock))  **break** self.kill\_list.clear() *# kill任务结束，清空*   *# 2 - 在这个时钟周期中，我们减小任务时钟数。若时钟数减小到0，添加到Input和Output任务队列* Input\_list = []  Output\_list = []  **for** index **in** range(self.task\_storage.size):  *# 减小这个任务的时钟周期数* self.task\_storage.minusClock\_byIndex(index)  *# 时钟数为0，需要在这个时钟内执行* item = self.task\_storage.msg\_byIndex(index)  **if** int(item.clock) == 0:  **if** item.type == **"input"**:  Input\_list.append(item)  **elif** item.type == **"output"**:  Output\_list.append(item)  self.task\_storage.remove(item.id)  index = index-1   *# 3 - 处理input任务队列* **for** item **in** Input\_list:  print(**"RUNNING - Input:"**, item.information)  self.send\_storage.add(item)   *# 4 - 处理output任务队列* **for** item **in** Output\_list:  print(**"RUNNING - Output:"**, item.information)  self.send\_storage.add(item) |

**指令设计**

input information（输入信息，信息为修饰）

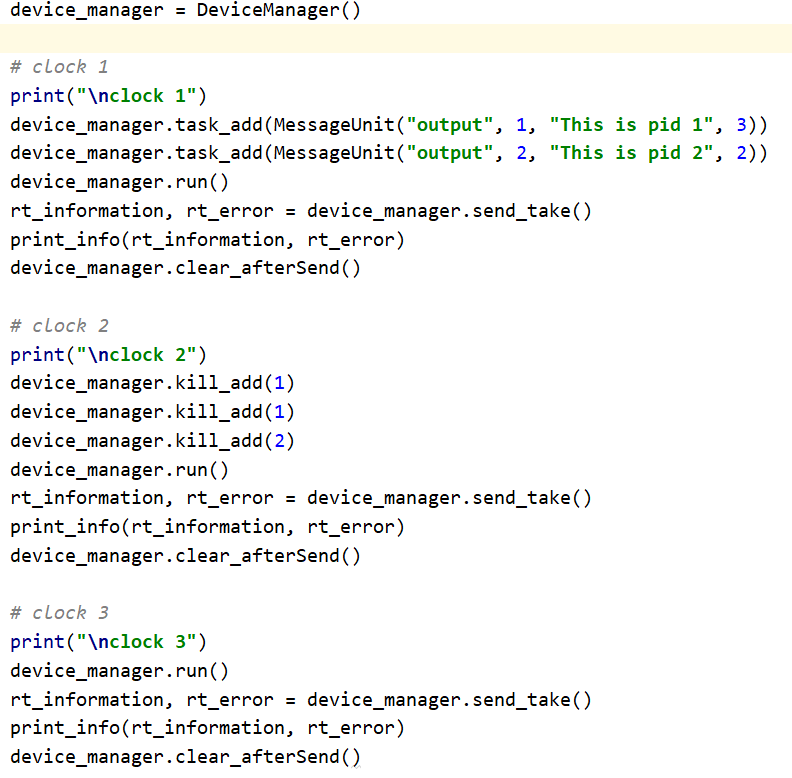
output information（输出信息，信息为内容）

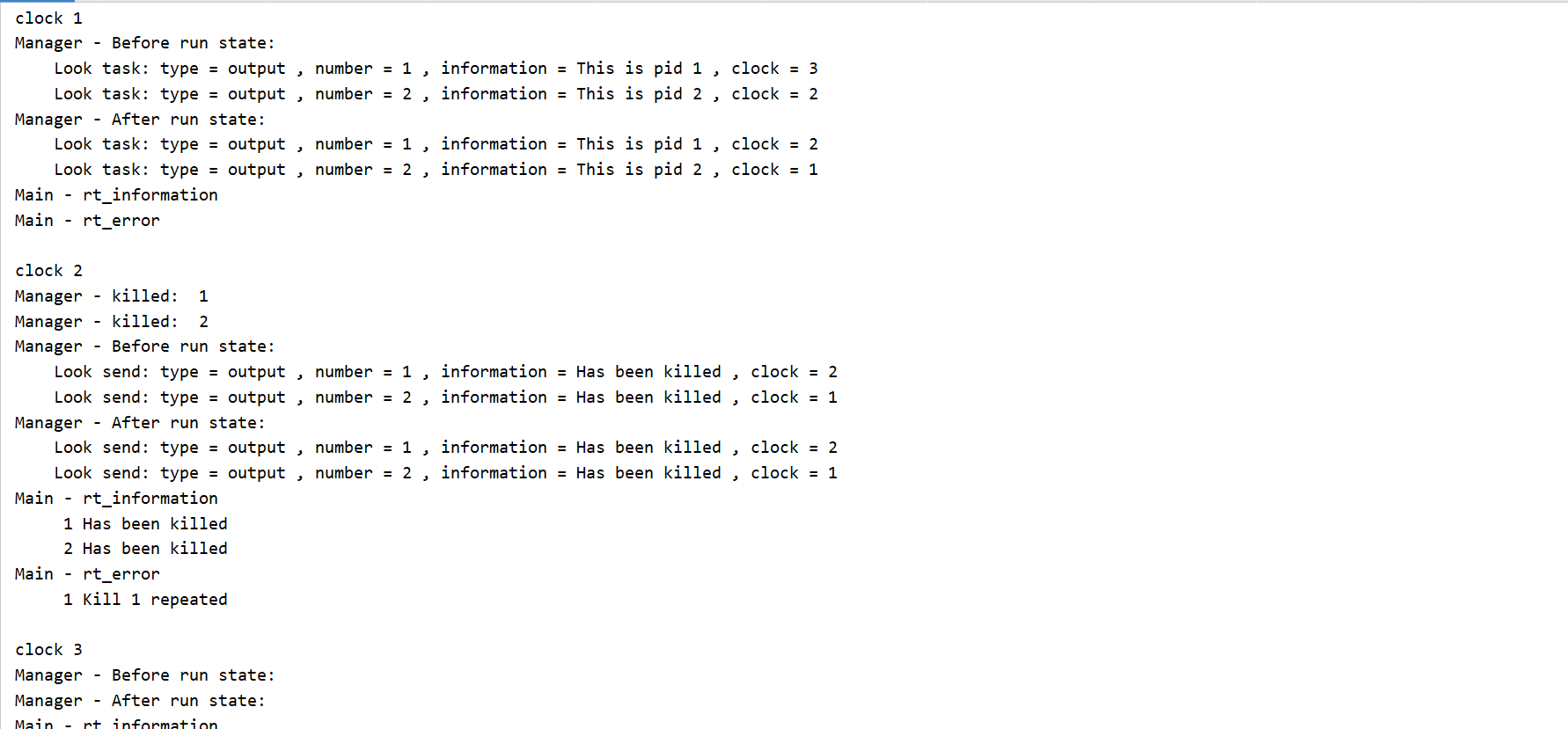
* + - 1. **测试方案**

1. **正确性测试：测试输入任务是否能正确调度。**测试一些可能转义的的字符串以及一些特殊性质的数据流，查看输入输出是否正确。
2. **功能性测试：**在多个任务下是否能正确调度，以及随时查看状态看是否正确响应和正确存储。
3. **高压测试：**通过脚本类型进行压力测试，查看响应速度是否合格。着重强调设备自身循环响应速度的测试。（因为设备可能涉及到阻塞中断之类，所以有必要进行高压测试）。

测试结果目前只进行了12，3还没有进行测试。

我们添加了三个任务，然后在其中kill，以及执行了三个时钟：

执行结果如下：



可以看到kill以及错误信息都能被很好的展示，并且在manager里，我们能很清楚的看到manager里存储的状态。我们计划这个将在debug形式里可以展示

1. **后期规划**

4月初（中期前）：初版代码的编写及各模块的功能测试

4月中（20日之前）：进行模块的整合和不同模块间API的调试

4月底：进行模块功能的进一步完善，并且加入UI界面的设计

5月初：进行最终的功能测试，完成实验报告的编写和项目的总结