目录

[一、 课程设计题目 3](#_Toc104726623)

[二、 课程设计目标及要求 3](#_Toc104726624)

[三、 需求分析 3](#_Toc104726625)

[（1）进程模块 3](#_Toc104726626)

[（2）内存模块 4](#_Toc104726627)

[（3）文件模块 5](#_Toc104726628)

[（4）设备模块 5](#_Toc104726629)

[（5）CPU设计 5](#_Toc104726630)

[（6）中断设计 5](#_Toc104726631)

[（7）UI设计 6](#_Toc104726632)

[四、 开发环境 6](#_Toc104726633)

[（1）本地开发环境 6](#_Toc104726634)

[（2）远程项目管理环境 7](#_Toc104726635)

[（3）在线协作调试环境 8](#_Toc104726636)

[五、 总体设计 9](#_Toc104726637)

[六、 详细设计 11](#_Toc104726638)

[（1）进程模块 11](#_Toc104726639)

[（2）内存模块 14](#_Toc104726640)

[（3）文件系统 16](#_Toc104726641)

[（4）设备模块 25](#_Toc104726642)

[（5）CPU设计 26](#_Toc104726643)

[（6）中断设计 29](#_Toc104726644)

[（7）UI设计 31](#_Toc104726645)

[七、 程序清单 34](#_Toc104726646)

[（1）进程模块 34](#_Toc104726647)

[（2）内存模块 37](#_Toc104726648)

[（3）文件系统 38](#_Toc104726649)

[（4）设备模块 43](#_Toc104726650)

[（5）CPU设计 44](#_Toc104726651)

[（6）中断设计 45](#_Toc104726652)

[（7）UI设计 46](#_Toc104726653)

[八、 测试报告 47](#_Toc104726654)

[1）测试环境 47](#_Toc104726655)

[2）测试的功能 48](#_Toc104726656)

[3）针对每个功能的测试情况 49](#_Toc104726657)

[九、 课程设计总结 60](#_Toc104726658)

[1) 设计过程中遇到或存在的主要问题及解决方案 60](#_Toc104726659)

[2) 改进建议 61](#_Toc104726660)

[3) 体会/收获 61](#_Toc104726661)

## 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，

## 课程设计目标及要求

目标：设计并实现一个具有操作系统基本功能的软件

**具有操作系统的基本功能：**

➢ 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。

➢ 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。

➢ 文件系统。 目录/文件的创建和删除、空间分配和回收

➢ 设备管理。 设备的申请、分配、使用、释放等

➢ UI界面。

**中断机制**

要求：给出各阶段的设计文档和实现成果

## 需求分析

### （1）进程模块

* + 为便于管理，每个进程需要有进程控制块PCB
  + 进程的基本操作（创建、撤销、阻塞、唤醒合同步控制）
  + 进程需要考虑五种进程状态在何时进行状态转移
  + 进程需要能够换入换出。
  + 需要使用调度算法对进程进行调度
  + 进程允许并发，即需要使用时间片分给不同进程
  + 由于时间片的提出，需要计时器中断
  + 进程间与文件系统需要进行互斥操作

### （2）内存模块

* 分配：当进程创建时，内存需要分配相应大小的内存空间给进程，包括内核内存分配用于存放PCB和页表信息，以及用户内存以页为单位分配给进程存放代码段和数据段内容；
* 释放：当进程结束后，需要将该进程所占用的空间释放，并对内存空间进行整理；
* 访问：当进程运行时，要对该进程的相应数据进行访问，内存需要通过逻辑地址到物理地址的映射找到所需要的数据，如果缺页需要触发缺页中断；
* 监控：需要能够实时的显示当前内存的占用情况。
* 交换：由于加入了虚拟内存，进程页可以在内存中换入换出

### （3）文件模块

* 需要确定文件系统目录的逻辑结构（树形Tree）合物理结构
* 确定文件系统文件控制块FCB
* 进程对文件进行操作时文件或目录的状态以及实现方式
* 确定文件访问方法
* 文件的权限保护
* 文件系统分配方式（连续、链接合索引分配）
* 空闲空间管理
* 物理存储空间的设计

### （4）设备模块

* 设备的申请、管理以及释放
* 进程使用设备的队列

### （5）CPU设计

* CPU解释执行编译完成的二进制
* 需要接收时间中断完成进程切换
* 模拟进程对CPU的使用

### （6）中断设计

* 最重要的是中断机制的实现方式，如何模拟中断
* 设计IO中断(包括键盘中断和打印中断)和设备中断
* 设计磁盘中断
* 设计计时中断对时间片进行模拟

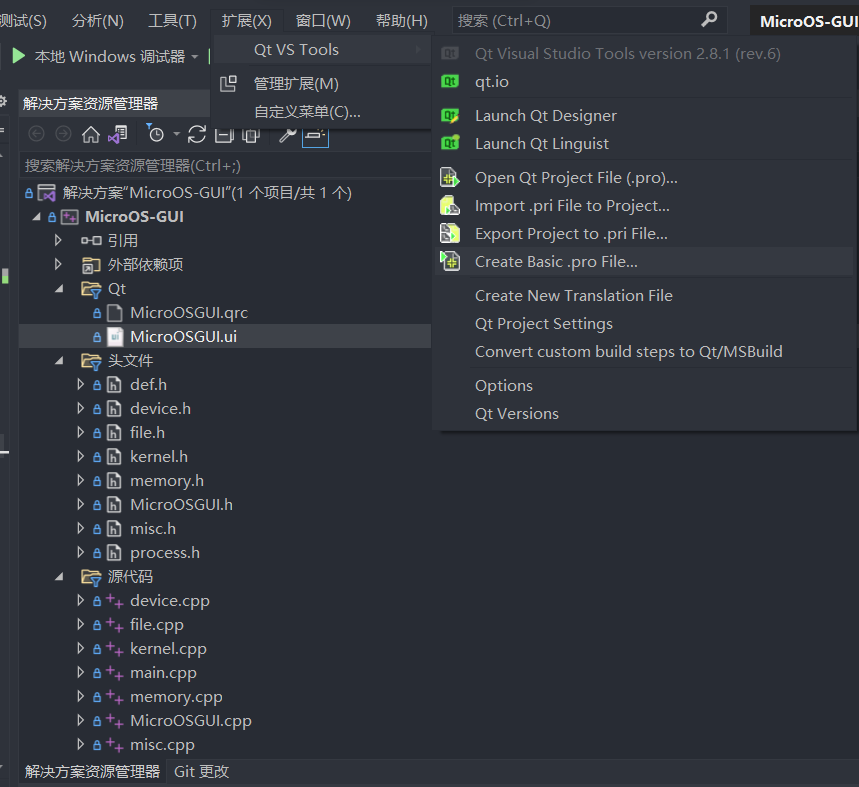
### （7）UI设计

* 设计Shell来完成最基本的命令操作
* 设计各种监控窗口进行操作系统监控

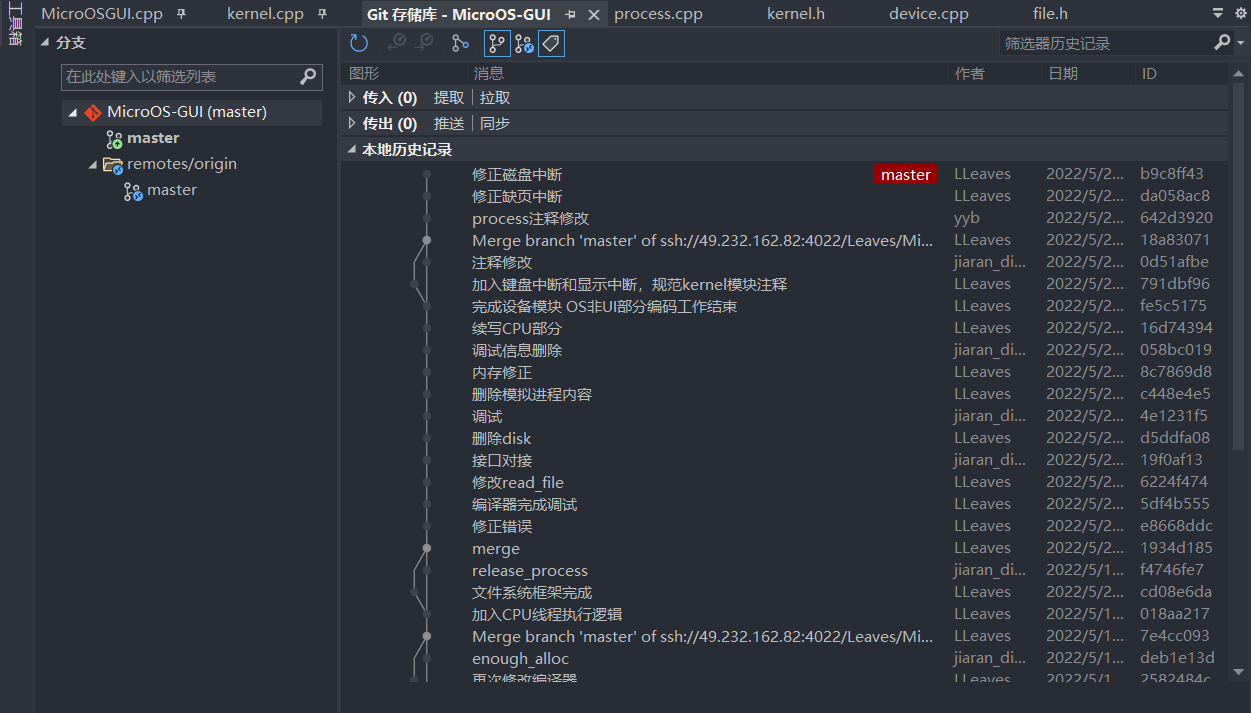
## 开发环境

### （1）本地开发环境

* 本地使用Windows作为开发环境基础
* 通过插件将 Visual Studio 2022 与Qt5.12.12联合起来开发

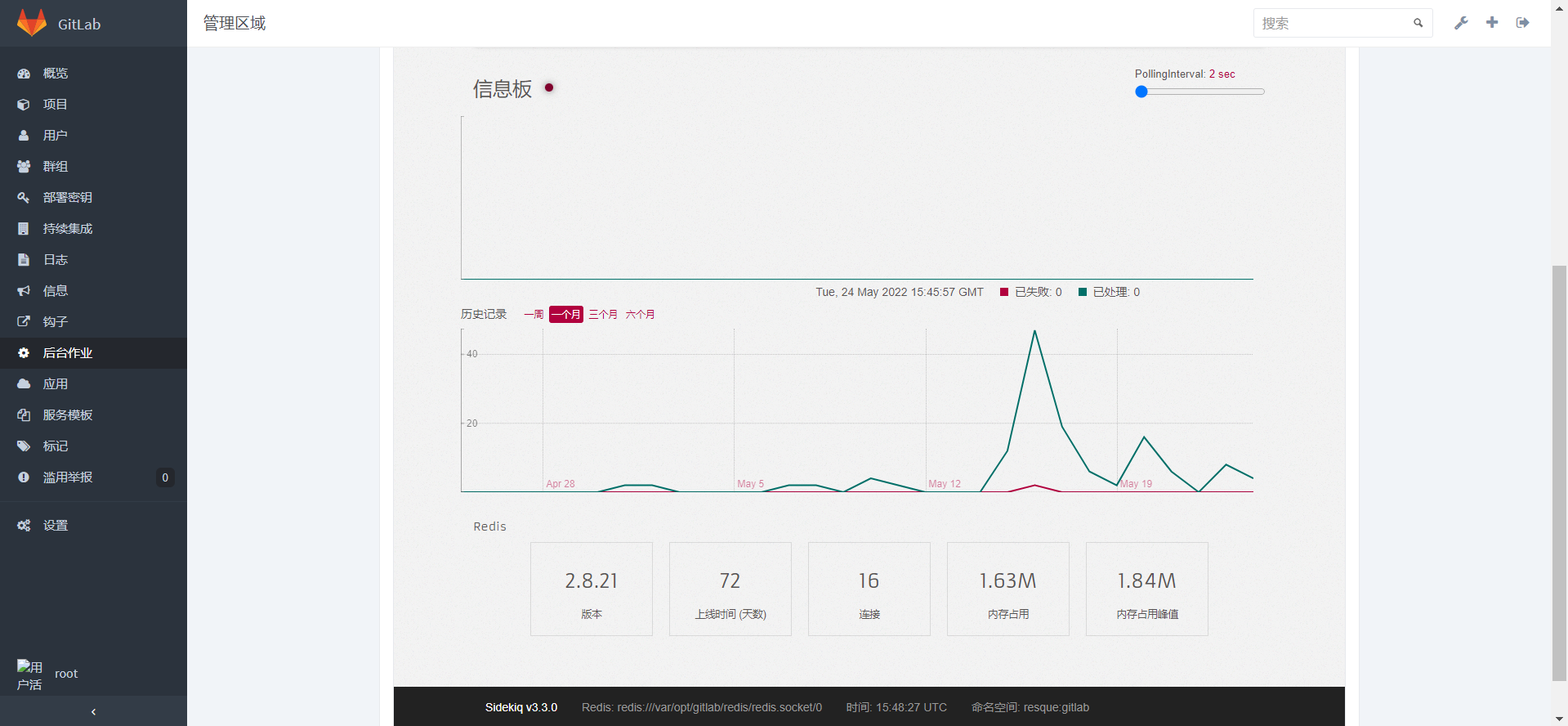


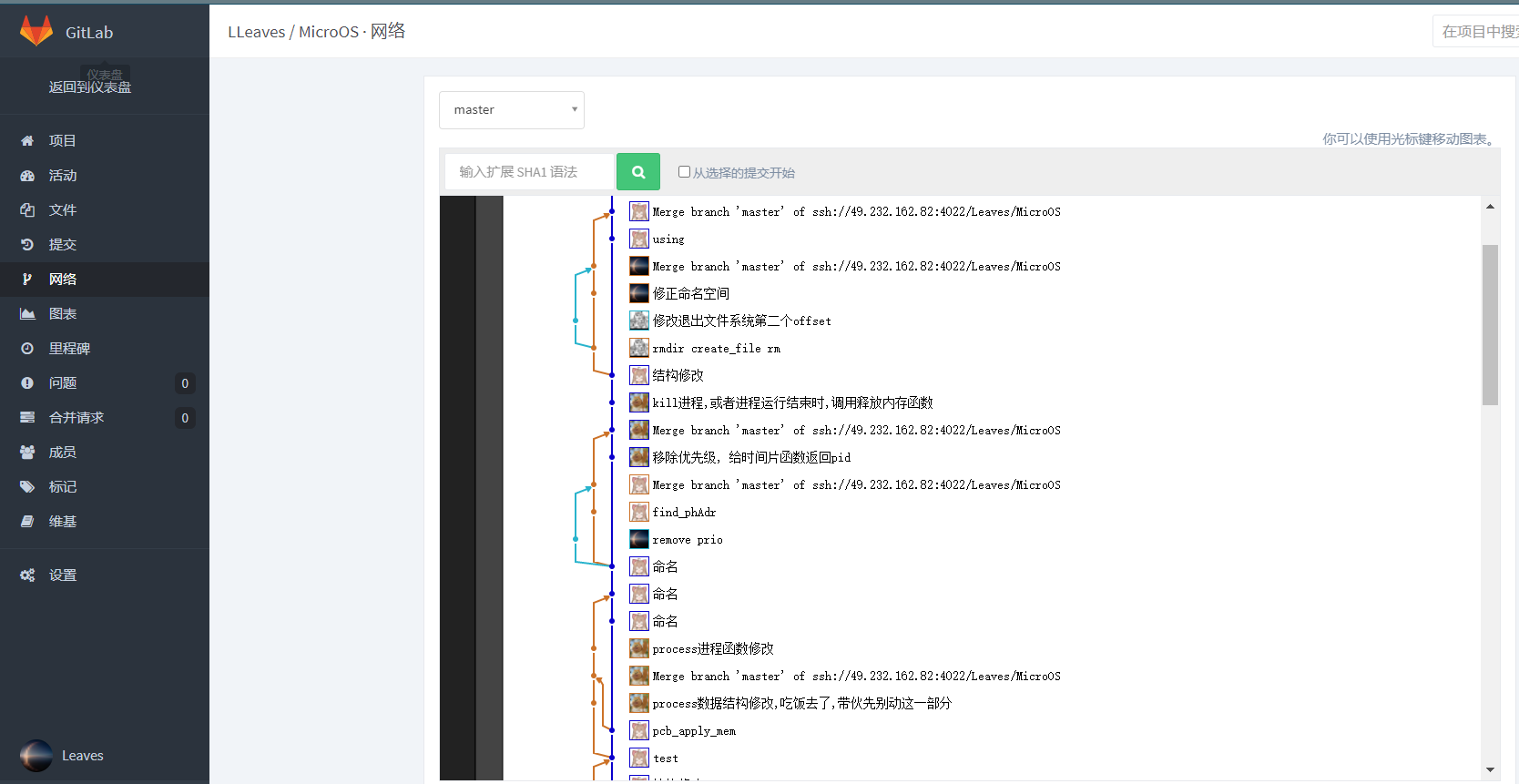
* 使用Git在Visual Studio中进行本地项目的管理



* 使用C++进行开发

### （2）远程项目管理环境

* 在远程服务器上搭建远程项目管理平台GitLab进行团队项目的整体维护和管理，极大的提升了线上协作效率



* 通过在线协作文档CodiMD进行远程文档协作

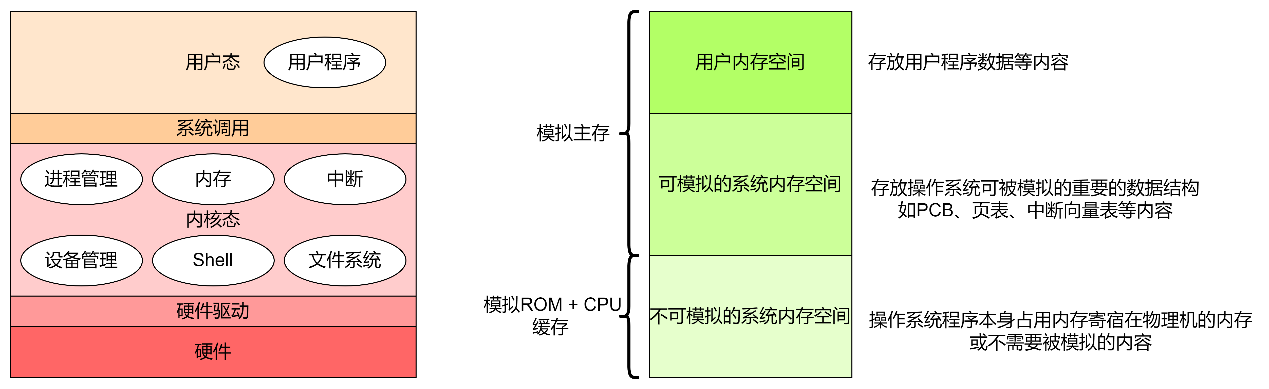


### （3）在线协作调试环境

* 通过腾讯会议配合ToDesk远程控制软件进行远程线上联调



## 总体设计



如上图所示，整个操作系统分为五部分，分别是用户态、系统调用、内核态、硬件驱动以及硬件设备。其中用户态主要包括用户程序执行部分；内核态是操作系统课设中的主要内容，包括几大主要模块：设备模块、进程模块、内存模块以及文件系统模块，还包括Shell和中断的设计；系统调用在内核态和用户态间进行联系，将部分系统接口提供给用户进程；硬件则主要设计磁盘部分和设备部分，将二进制文件模拟为磁盘并抽象设备为设备数据结构，通过软件模拟硬件驱动实现多种中断操作。

与之对应的是内存空间的设计，内存空间设计主要分为三部分：第一部分为用户的内存空间，对应用户态，该部分内存是提供给进程使用的内存空间，存放包括二进制可执行文件本身代码段和数据段的空间和进程执行额外需要的空间；第二部分是可模拟的内存空间，为提高操作系统的模拟程度，更好的掌握操作系统的内核知识，将可模拟的重要的数据结构，包括所有进程PCB、页表以及中断向量表等存放在从堆区申请的内存空间中，与整体的内存位于同一片内存空间，以此来模拟完整的主存空间；第三部分为不可模拟的内存空间，主要用于存放访问频繁户的内存空间和相较不重要的数据结构可以作为ROM部分和CPU缓存不模拟到内存空间。

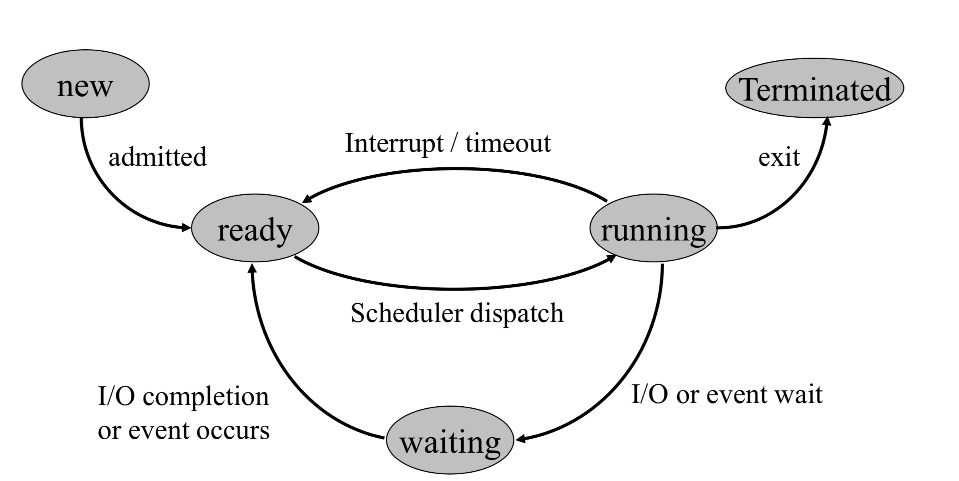
这样的设计不仅符合操作系统的基本结构设计，而且能够最大限度的通过软件模拟操作系统。

整个操作系统通过QT设计的UI与用户进行交互，Shell可以接收用户的输入，通过事件过滤器检测用户键盘按键从而完成一系列模拟shell的功能，例如用户同时按下Ctrl与S则对shell中的内容进行保存以实现写文件操作等。用户写入源代码后，通过内嵌的编译器进行编译转换成更容易被CPU识别的二进制字节码。当用户提交执行后，CPU进行RR调度分片执行提交的进程。在shell执行命令的过程中和整个进程执行过程中使用到内核态的全部模块完成操作系统的模拟，执行过程中的信息通过UI展示给用户。

## 详细设计

### （1）进程模块

#### 进程状态迁移



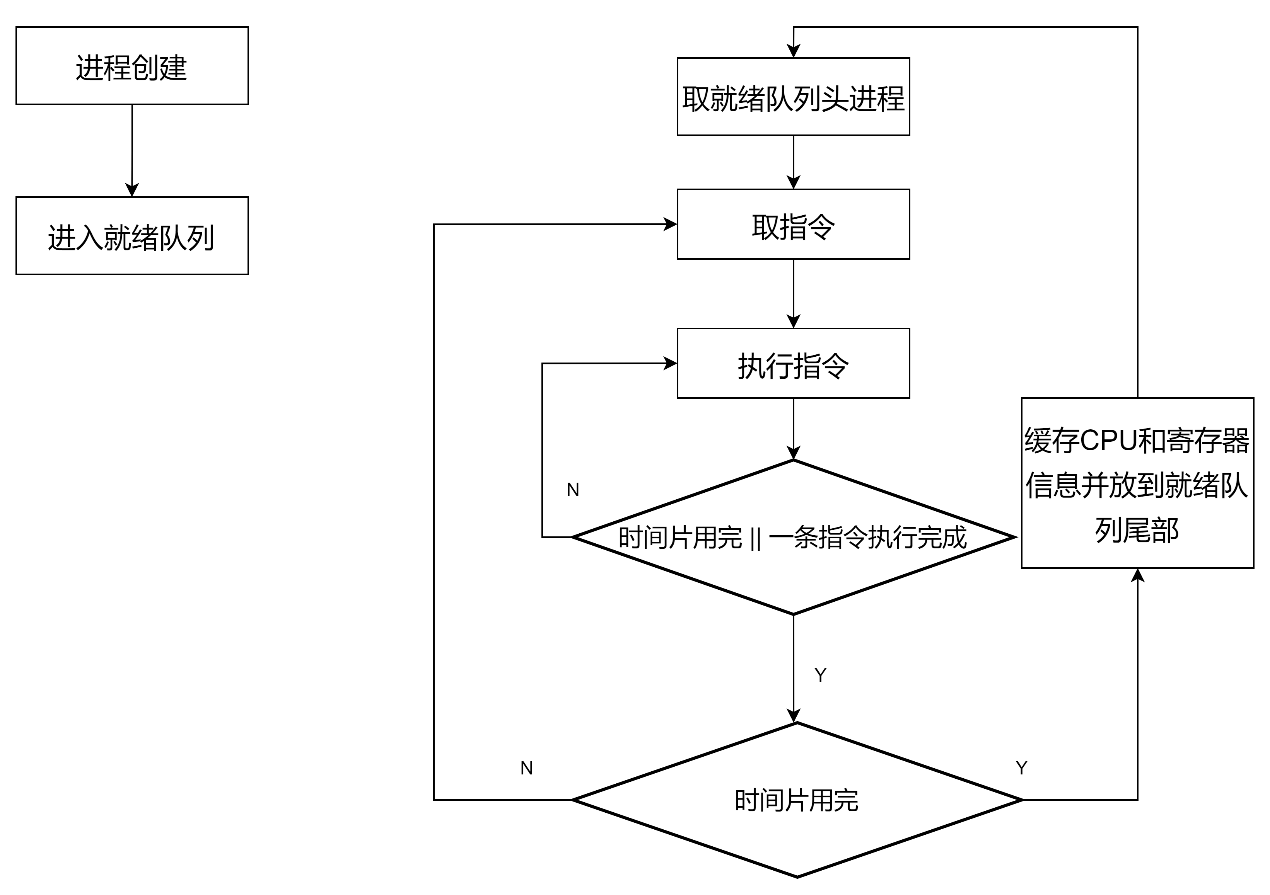
进程状态迁移使用经典五状态模型，在建立进程后，进程就从新建(new)状态转换到准备(ready)状态，等待其需要的资源。如果CPU调度到该进程，则进入运行(running)状态。在运行状态，如果发生中断或者时钟超时，则会重新回到准备(ready)状态。在运行状态，如果发生I/O或者其他事件的等待，则会从运行(running)状态中退出，进入等待(waiting)状态。在等待状态(waiting)等待其I/O完成，或者事件发生后，则从等待(waiting)状态中退出，重新进入准备(ready)状态等待CPU调度。在程序运行的过程中，如果接收到退出指令，或者程序运行完毕，则转入终止(terminated/Finish)状态。

#### 进程控制块PCB设计



进程控制块PCB由PID、进程状态、当前进程PC计数器、内存管理指针、IO状态信息指针、记账信息以及CPU缓存信息。其中内存管理信息和CPU缓存信息部分为PCB中最重要的部分，其中内存管理信息指针指向内存模块分配给当前进程的页表内存块，当进程访问内存时需要通过页表获得内存物理地址；其中CPU缓存信息是为了便于操作系统的设计和RR调度，由于每条指令都有一定的时间占用CPU资源，而又采用RR调度，所以每条指令不一定在一个时间片执行完成，加入缓存信息后能够加快进程调度的效率，从而更好的模拟执行。

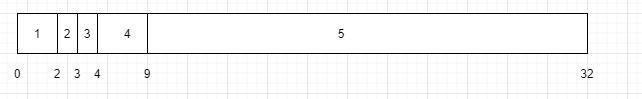
#### RR轮询调度设计



进程创建后若准备就绪则进入就绪队列，CPU位于另一个线程，一直执行一个循环取当前就绪队列的头部进程PCB准备进行执行，操作系统每次给当前进程分配200ms的时间片，在这个时间内，需要进行取指令和执行指令的过程，如果在执行过程中200ms时间片被使用完则将当前执行进程从running状态变为就绪状态，将CPU缓存信息和寄存器信息保存到PCB后加入到就绪队列尾部等待下一次分片执行（第一部分已说明状态迁移，此处流程图不进行状态迁移流程展示），如果当前时间片没被使用完则继续取指令执行下一条指令直到时间片使用完毕。

### （2）内存模块

#### 内存结构设计



内存的具体结构设计如图所示。整个内存大小为32KB（32页），其中每页的大小为1024字节。

模块1表示PCB模块，用来存放进程的PCB，占用大小为2页。

模块2表示中断向量表模块，用来存放系统的中断向量表，占用大小为1页。

模块3表示系统设备表模块，用来存放系统设备表，占用大小为1页。

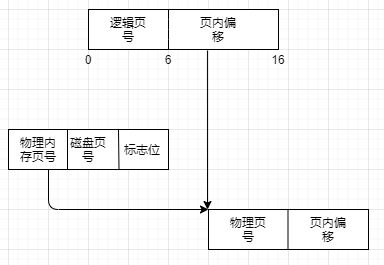
模块4表示页表模块，用来存放进程的页表，占用大小为5页。

模块5表示用户进程模块，用来存放进程代码，占用大小为23页。

#### 内存分配设计

内存分配采用页式分配的方式。对于一个进程，在创建进程时，先预加载一部分的代码到内存中。内存需要的内存空间包括指令段数据段的长度和进程指令申请的长度。进程占用内存的最大空间为三页，在加载进程时，对于需求内存大于三页的进程，只加载三页的内容到内存中，剩下的放在磁盘空间。进程加载成功后返回进程页表的起始地址，加载失败后返回0，表示内存空间不足，阻止进程的创建。

#### 虚拟内存设计



虚拟内存的转换能够根据需要访问的逻辑地址找到对应的页表项的物理地址，能够实现根据逻辑地址访问到内存中的具体内容。

在进行逻辑地址的转换时，有可能需要访问的页不在内存中，这样就会触发缺页中断，进而执行页面置换算法。

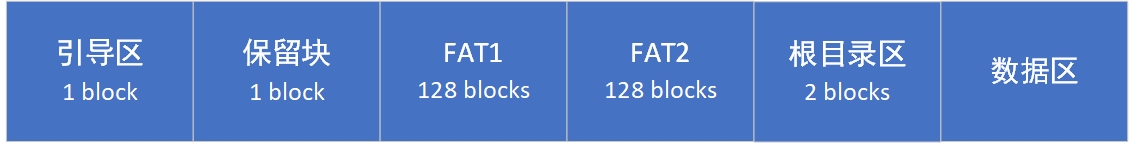
页面置换算法使用LRU算法，利用数组timeInMem来记录内存中的页在内存中存在的时间。每进行一次逻辑地址到物理地址的转换，访问到的内存页对应的timeInMem置为0，属于同一个进程的其他内存页对应的timeInMem加一。每次进行页面置换时，选择该进程的内存页中timeInMem值最大的页换出，再换入需要访问的页。换入的页对应的timeInMem置为0。

#### 进程释放设计

进程执行完时需要释放进程占用的空间。包括PCB模块清空对应进程的PCB，用户内存模块释放占用的内存块，flag，timeInMem对应的页标记全部复原为-1。并根据页表找到进程占用的虚拟内存的磁盘页号进行释放，最后清空对应的页表空间。

### （3）文件系统

#### 磁盘



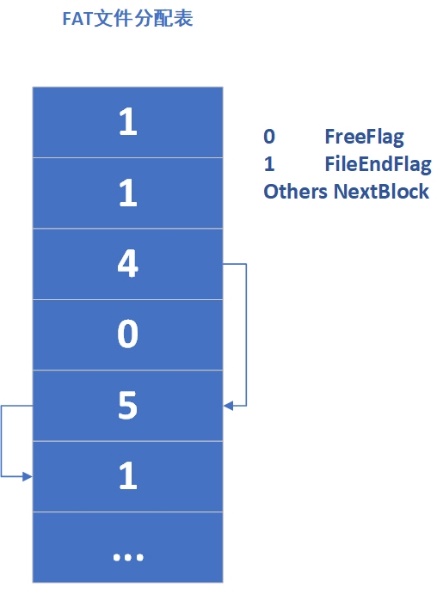
磁盘结构：类似FAT16的磁盘结构，将一个64M的2进制文件作为磁盘

文件系统的物理块大小为1KB,共计2^16块，即磁盘空间是64MB

磁盘分为引导区(1块)，保留块（1块），FAT1（占128块）,FAT2(备份，占128块)，根目录区（2块）,数据区（大小可变）。

只有文件系统有权直接操作磁盘。

#### FAT文件分配表



文件分配表中每一项为16位，记录了对应的物理块相连接的下一块的位置

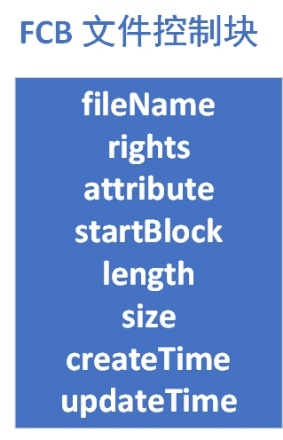
若物理块空闲，该项为0，表示可用，若物理块为文件的最后一块，该项为1，表示结束块

FAT1文件分配表大小为128k，占128个物理块

FAT2是FAT1的备份，与FAT1完全相同

FAT会在启动文件系统时读入，在退出文件系统时写回磁盘。

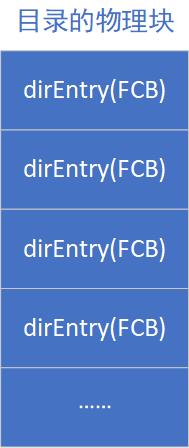
#### FCB 文件控制块



文件控制块，用以描述文件的属性，并作为目录项，每个FCB占32字节。

每个文件控制块包含：文件名、文件权限、文件属性、起始块号、文件长度、文件大小、文件创建时间、文件最后修改时间属性。

#### 目录

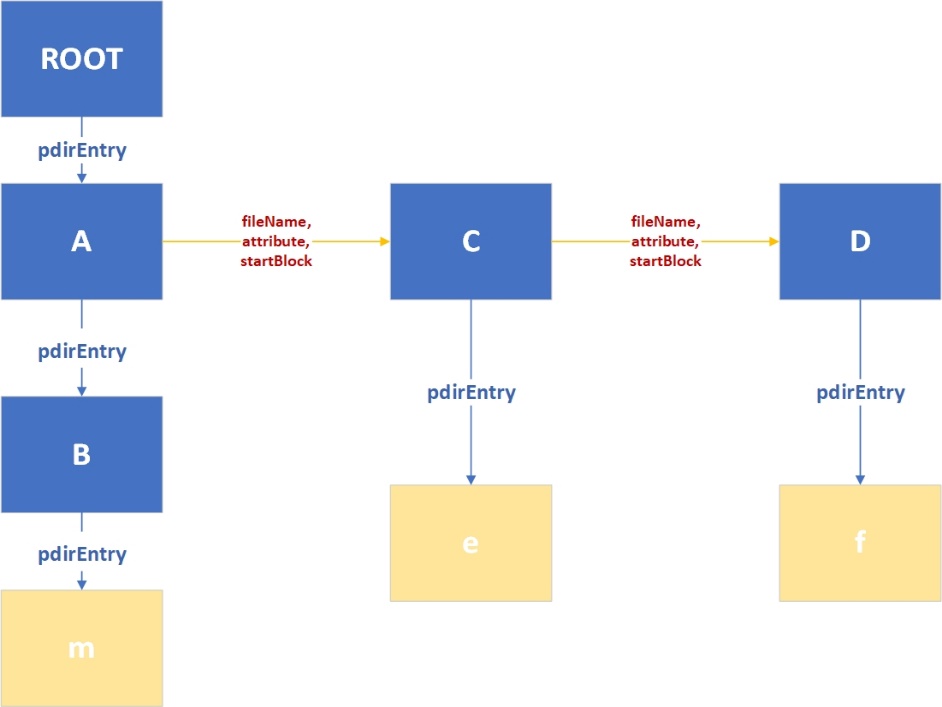


目录逻辑上是二叉树型结构，使用vector下标访问同级文件（目录）的目录项，通过同级目录项起始块号（FCB中的startBlock）来访问下一级目录内容。

根目录有固定的磁盘块位置和长度，其他目录大小固定，均占1个磁盘块，但位置不固定，根据FAT表随机分配空闲块。

目录被视为一种特别的文件，除了根目录有固定位置，其他目录均在数据分区。

文件系统会维护当前工作目录，记录当前工作目录的目录项。



访问举例：

如图，若当前工作目录为根目录，需要对D目录下的f文件进行读写。

首先根据根目录找到目录项A（通过文件名，获得下标）。

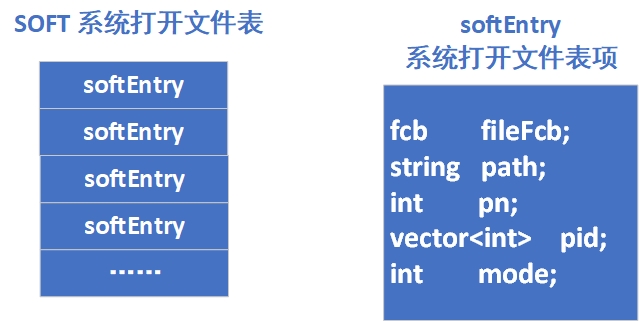
根据A的FCB ，找到存储A目录开始的物理块（fileName和atribute指示该目录项指向一个目录，startBlock指示起始块号）。

然后生成A的目录结构（即创建A的目录项vector）。

同上，接着找到C目录和D目录，最终搜索D目录，找到文件f的目录项。

根据f的FCB，获取文件长度和起始物理块号，参照FAT，即可获得完整的文件f。

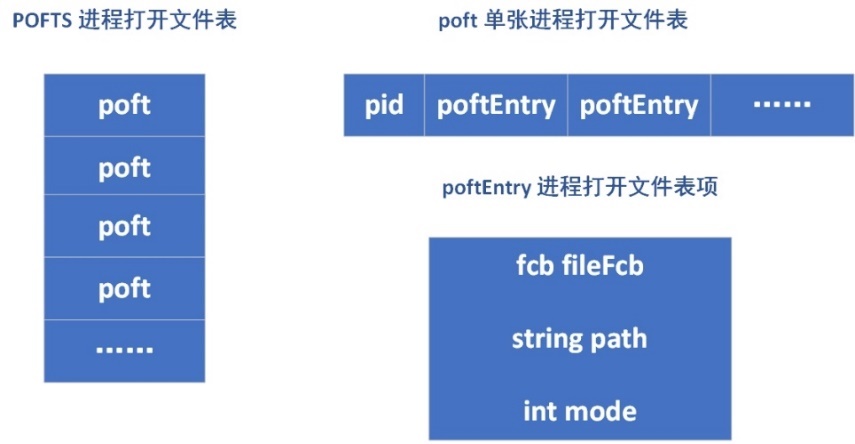
#### 系统打开文件表



系统打开文件表项以文件为主体，在进程打开和关闭文件时，对表项进行操作。

系统打开文件表项记录了文件的FCB、文件路径、被多少个进程打开、各个进程的pid、打开模式（读或者写）。

#### 进程打开文件表

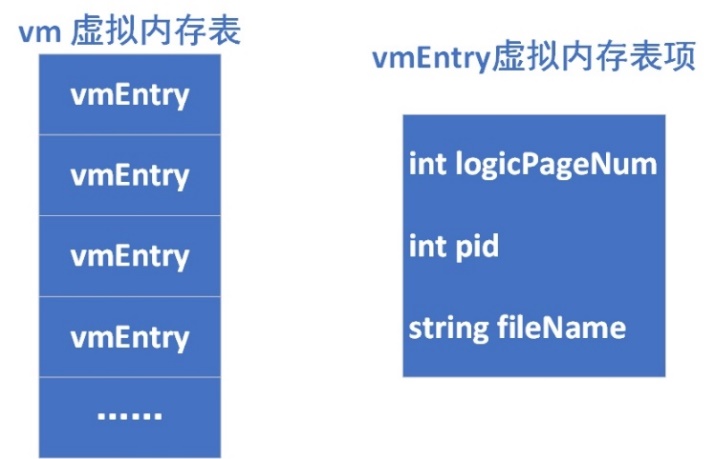


进程打开文件表(pofts)以进程为主体，在进程打开和关闭文件时，对表和表项进行操作。每个在使用文件的进程都会维护一张进程打开文件表（poft）,他们集体组成了进程打开文件表（pofts）

进程打开文件表项(poftEntry)记录了文件的FCB、文件路径、打开模式（读或者写）。

单张进程打开文件表(poft)包含一个pid和若干个表项。

#### 虚拟内存

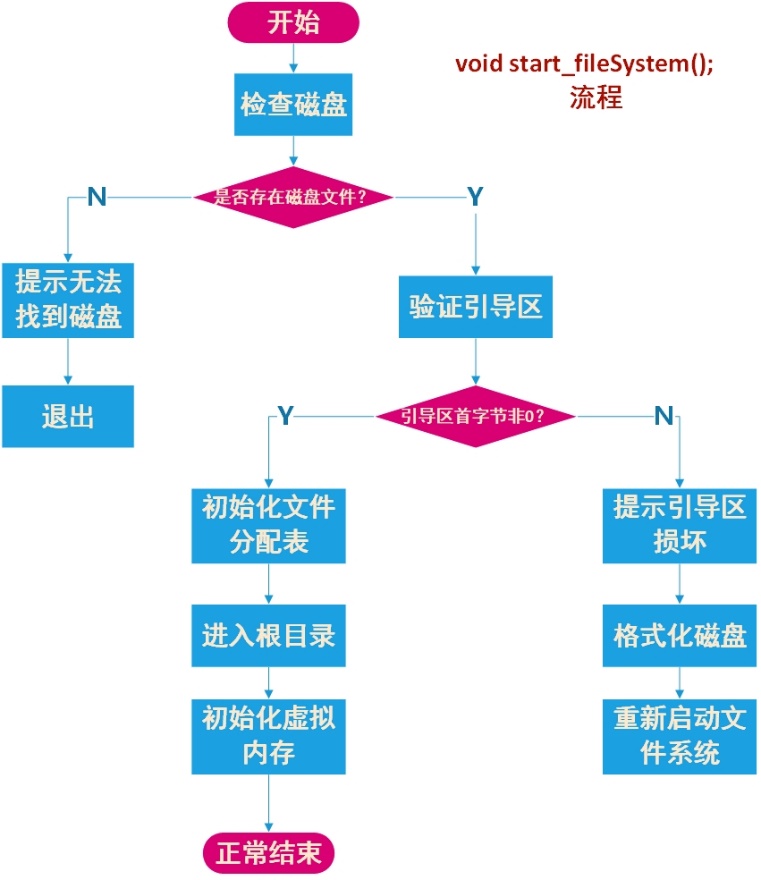


虚拟内存是一片与内存交互的区域，大小为32KB，设计上将其放在了FAT2的磁盘块上，在文件系统启动时，将FAT2的磁盘块前32块清空作为虚拟内存，在退出文件系统时，才将FAT的内容备份到FAT2的磁盘块。这样的设计，实现了FAT2磁盘块的复用，提高了磁盘的利用率，并节省了磁盘空间。

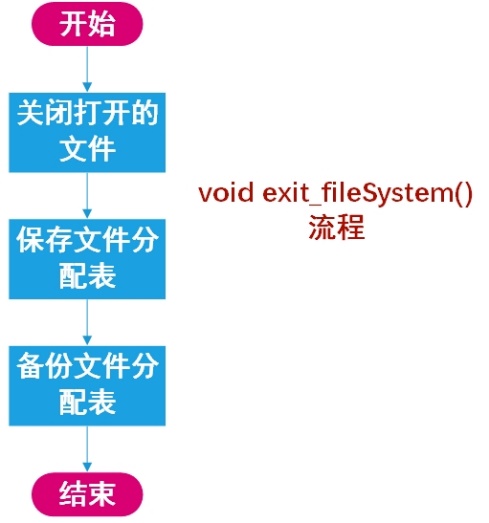
文件系统会维护一张虚拟内存表vm，用来进行页面置换。

虚拟内存表项记录了逻辑页号，进程pid和文件名，当逻辑页号为0，表明对应的虚拟内存块是空闲的。

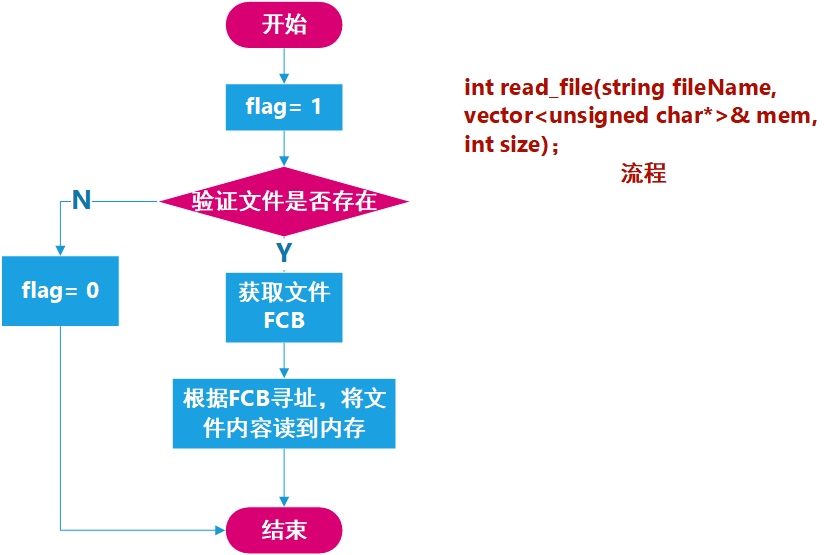
#### 启动文件系统

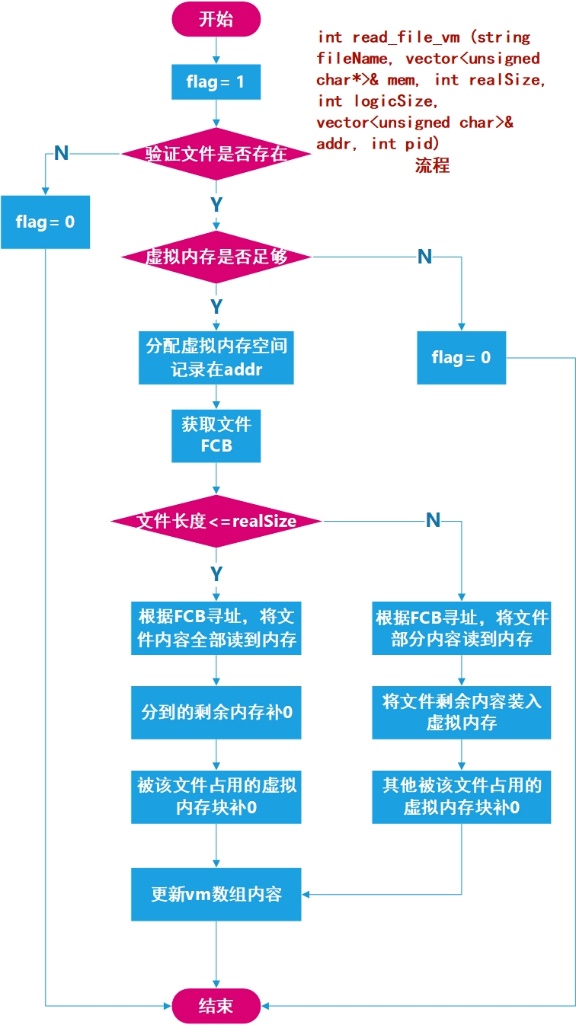


#### 退出文件系统

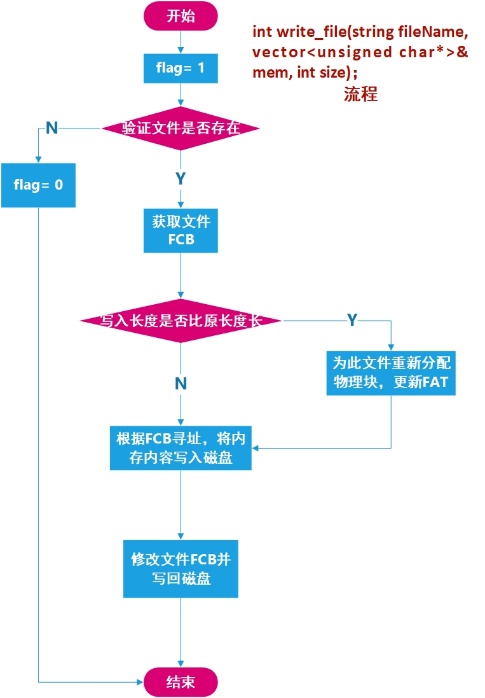


#### 读文件

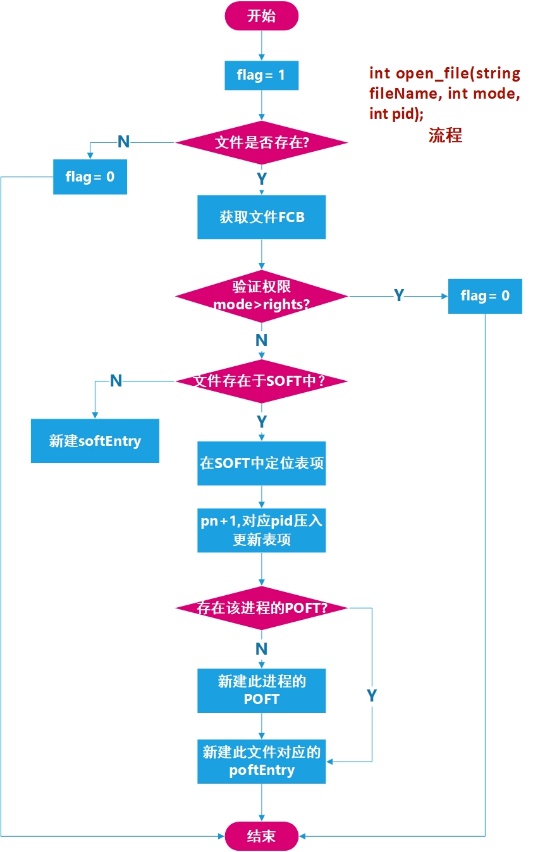




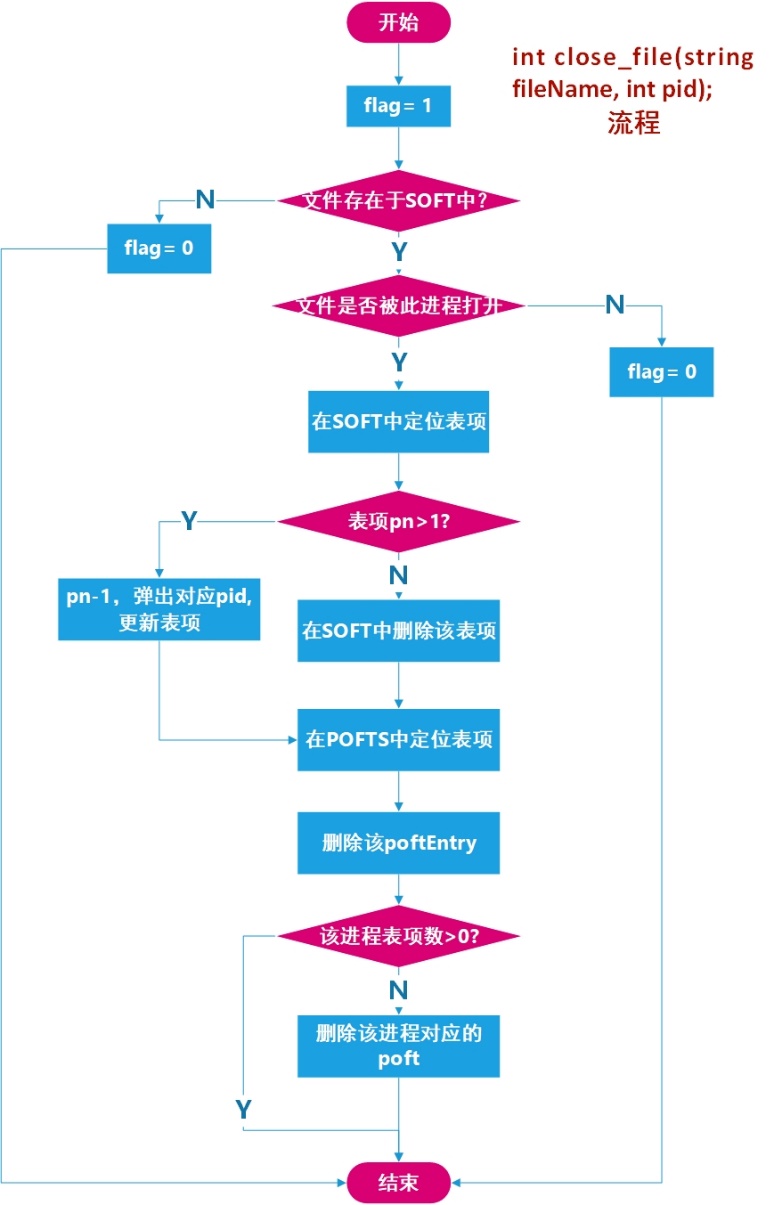
#### 写文件



#### 打开文件



#### 关闭文件



#### 页面置换

int vm\_swap(unsigned char\* mem, unsigned char pageNum);

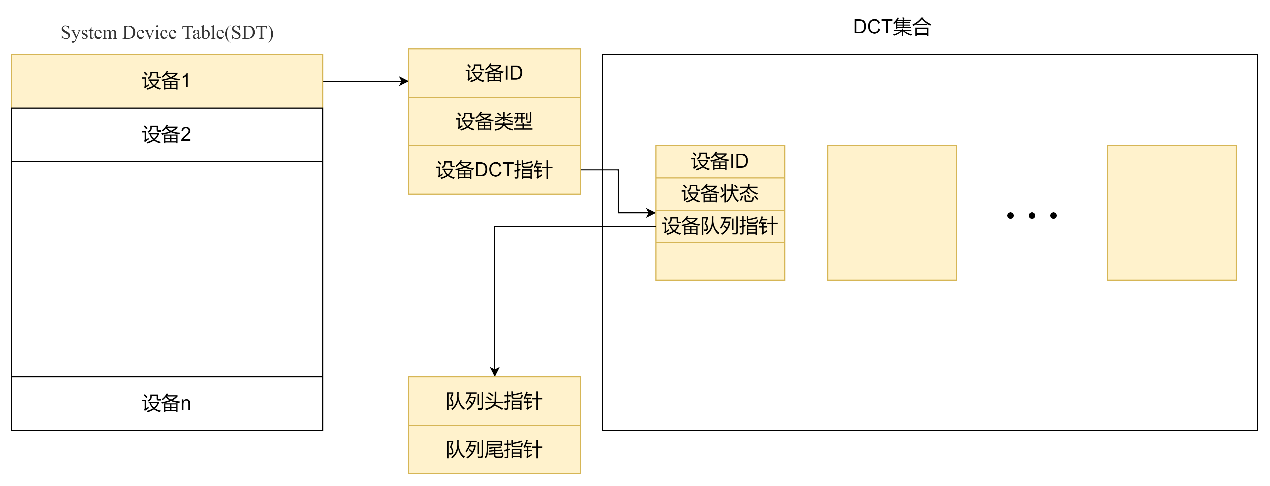
返回值：0表示失败，1表示成功

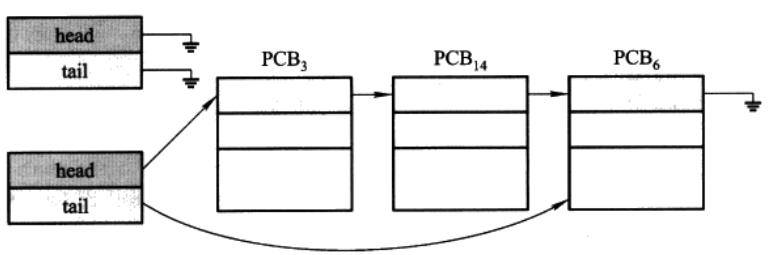
参数：mem内存地址，pageNum虚拟内存（页号）下标

将内存某一页的内容和虚拟内存某一页的内容进行交换

### （4）设备模块

#### 系统设备表SDT以及DCT设计





操作系统设备关键数据结构分有系统设备表System Device Table 和 设备控制表Device Control Table。其中系统设备表表项指向设备控制表，设备控制表有两个主要的表项，其中一个是设备状态，在模拟操作系统内用于标记该设备是否被忙占用(BUSY)，如果处于忙占用，其他进程则不能使用该设备，此时需要用到第二个主要的表项，即设备队列指针，该部分提供两个指针：队列头指针和队列尾指针，每次设备不处于busy状态时可以将设备分配给队列头的进程，如果有新的进程申请使用设备且设备忙则将进程置于设备队列尾部，等待分配设备。

#### 设备中断设计

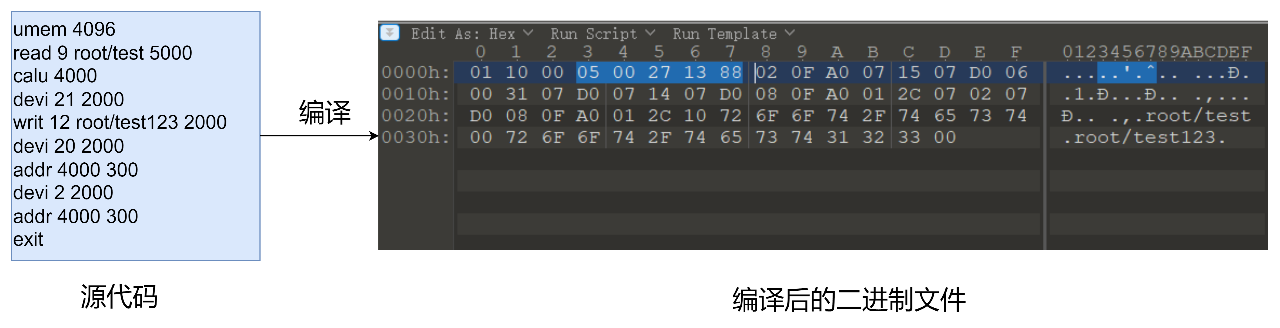
进程等待设备时进入阻塞状态，等待进程到达队列头部可以被当前进程使用时触发中断，通过线程不断轮询标志位来模拟中断，如果条件满足，当前进程可以使用设备，设备使用完成后，通过指针函数调用再次触发中断告知设备使用完成。

### （5）CPU设计

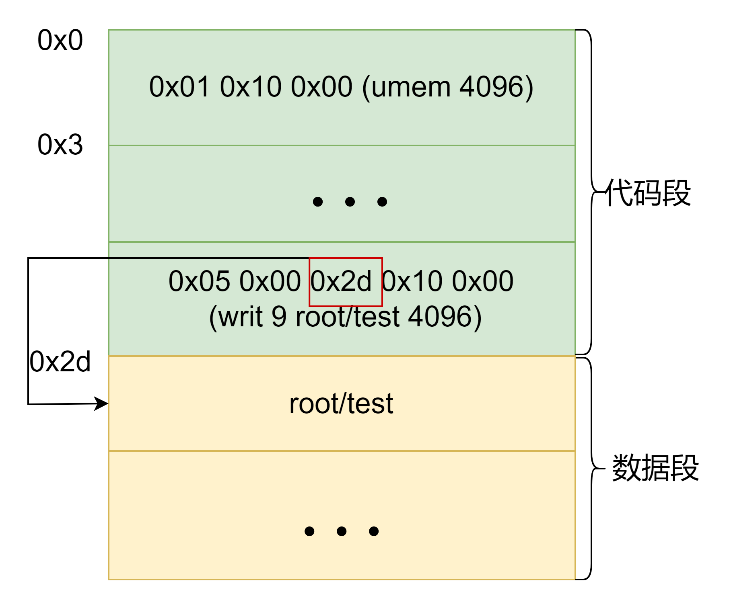
#### 编译器设计



通过操作系统的内嵌编译器将源代码编译成可被CPU识别的二进制字节码，字节码最长长度为5字节，这样的设计能够降低CPU执行取值令的时间损耗和内存损耗，更高效的模拟真实的操作系统。

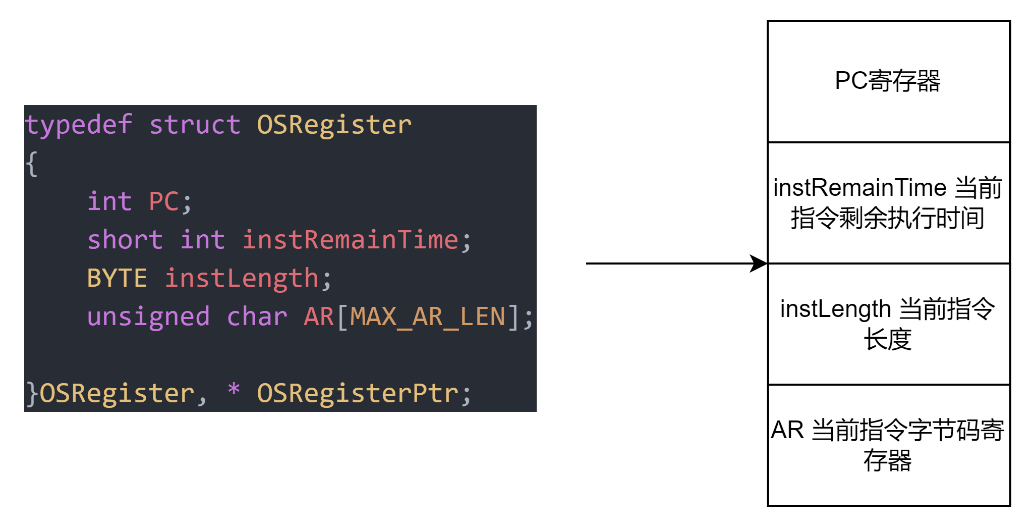


编译过程是一个翻译的过程，每条指操作对应一个字节码，指令中的立即数直接转换为二进制，例如umem 4096代表进程额外需要4096字节内存空间，umem字节码为0x01写入第一字节，4096转为hex为0x100，则将10 00 写入第二字节和第三字节。



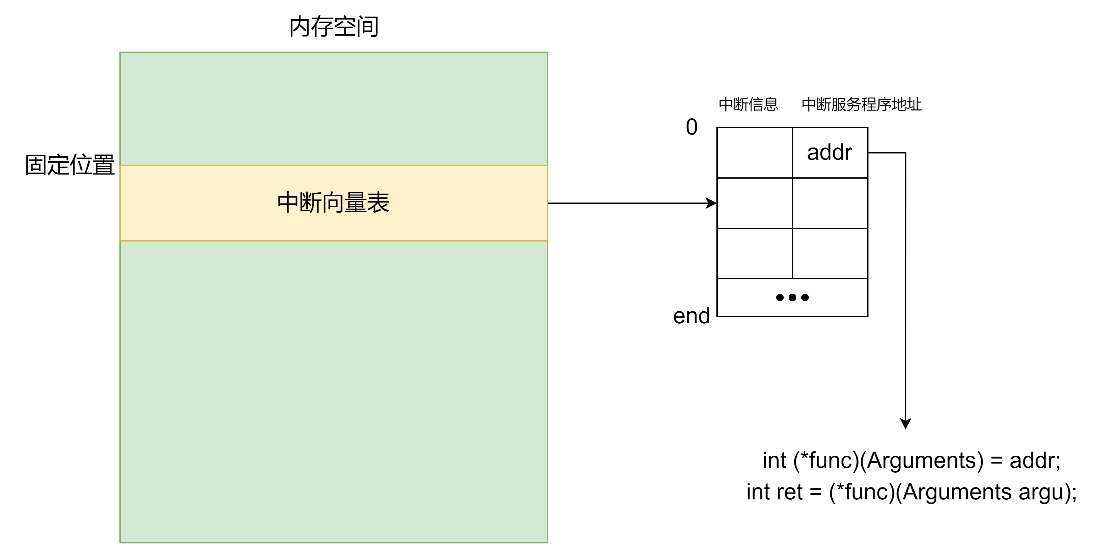
除此之外编译过程涉及到地址回填技术，因为在源码中存在文件名，而在代码执行过程中如果存在文件名，指令长度将由文件名长度决定。而使用地址回填技术，先记录数据段内容和要回填的位置，之后将数据段写到代码段后，并将地址回填到对应字节位置。如此构造好处有二：其一为更好的模拟操作系统进程代码段和数据段分离的基本设计；其二是能够更高效的利用CPU的时间片。

#### 寄存器设计



CPU执行过程已经在进程调度部分给出，此处不再赘述。在此次主要说明CPU缓存以及寄存器设计。如上图所示，寄存器组有四个寄存器，分别是PC寄存器、instRemainTime寄存器、 instLength寄存器以及AR寄存器，其中PC为当前指令虚拟内存地址；instRemainTime则记录当前指令剩余执行时间，次寄存器使得CPU执行时不需要频繁到内存中取PCB中的字段，提高CPU的执行效率；instLength则用于记录当前指令的长度，便于CPU进行指令分析；AR寄存器则借鉴经典寄存器设计的AR寄存器，用于暂存指令。

### （6）中断设计



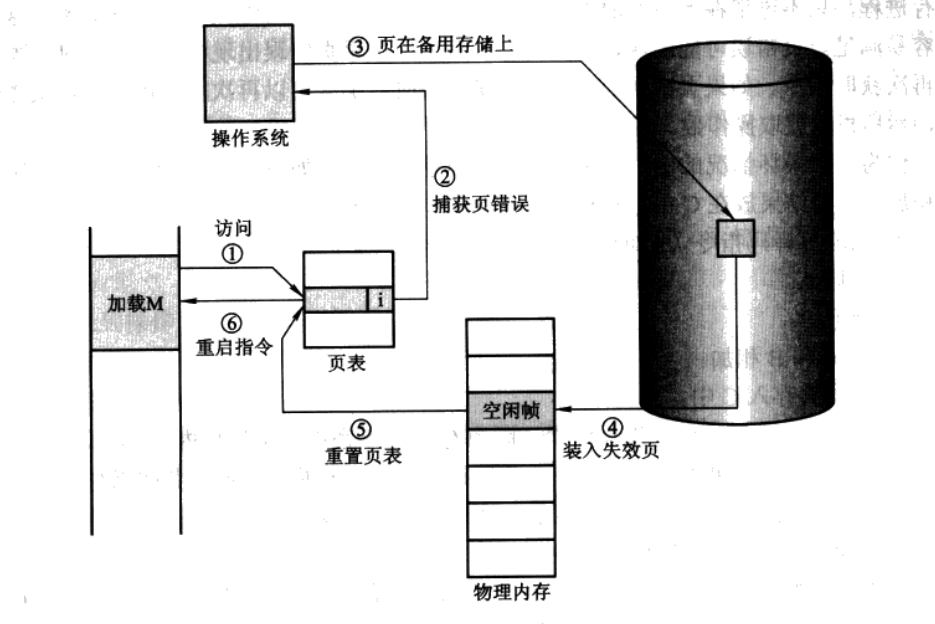
除Timer中断外的中断全部注册到中断向量表中，通过函数指针调用进行中断模拟。Timer中断则通过计时器进行模拟。

#### Timer中断设计



Timer中断通过创建计时器来模拟振晶，每200ms触发一次中断，在中断处理程序内部将全局PULSE信号置为1，代表脉冲来临，CPU通过轮询PULSE信号模拟接受脉冲，当脉冲收到时代表时间片用完，则将PULSE置为0，代表脉冲下降沿。一个Timer中断处理完成。

#### 缺页中断



由于采用虚拟内存编址的设计，在指令集中加入访问地址的指令addr ，当访问到的地址经过查询页表不在内存中的时候将会通过指针调用触发缺页中断，由内存和文件系统负责进行内存页面的换入换出完成缺页处理。

缺页中断设计与经典缺页中断设计基本一致，但是么有重启指令这一步，因为模拟操作系统设计的指令不是真实的指令，而是带有时间的指令片，因此只需要继续执行即可。

#### 磁盘中断

模拟操作系统设计磁盘中断，在进程进行文件读写的时候将进程设为阻塞态并使用多线程进行文件读写计时，以此来模拟磁盘IO操作时不占用CPU，当时间到达后，线程通过指针调用告知CPU文件读写结束，CPU将进程阻塞态设为就绪态代表完成，同时文件系统完成文件打开表等一系列修改动作。

#### 设备中断

模拟操作系统设计设备中断，在进程进行设备的时候将进程设为阻塞态并使用多线程进行设备使用申请和使用计时，以此来模拟设备操作时不占用CPU。在线程内部，进程首先会通过轮询来申请设备的使用权，直到设备可以被当前进程使用时开始计时，当计时完成后，线程通过指针调用告知设备使用结束，CPU将进程阻塞态设为就绪态代表完成，同时设备模块修改设备控制表来释放当前进程对设备的占用。

其中键盘和显示也被归入设备，分别是1号设备和0号设备。

### （7）UI设计

#### 设计思路

1）对于进程操作的展示，初步设计一个Text Browser 和一个Table Widget 。Text Browser 用文字描述来展示进程的操作；Table Widget以表格形式实时动态展示进程状态,展示的状态包含当前所有进程的进程名，pid，进程状态，PCaddress。

（2）内存操作的展示，初步设计一个Text Browser 和一个Table Widget 。Text Browser 用文字描述来展示内存变化的操作；Table Widget以表格形式动态展示内存状态,展示的状态包含所以的内存块的块号，块状态，占用该块的进程pid。

(3)进程执行过程期间，对资源的使用情况，这里展示进程的内存使用情况，设备使用情况，文件使用情况，在内存表，和文件打开表，以及设备队列展示表中显示。

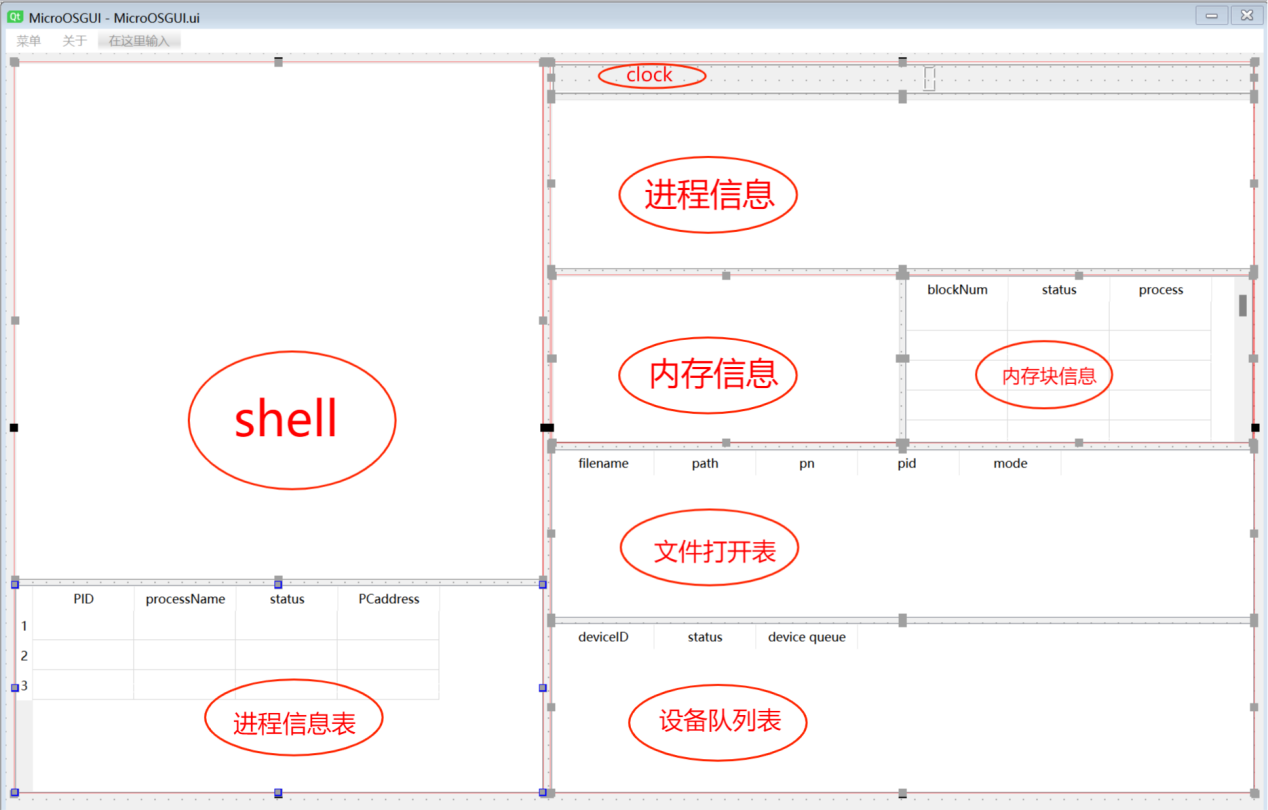
(4)进程的状态变化、在各种队列之间的迁移,见（1）.在各级队列调度时，会以文字信息描述变化过程。

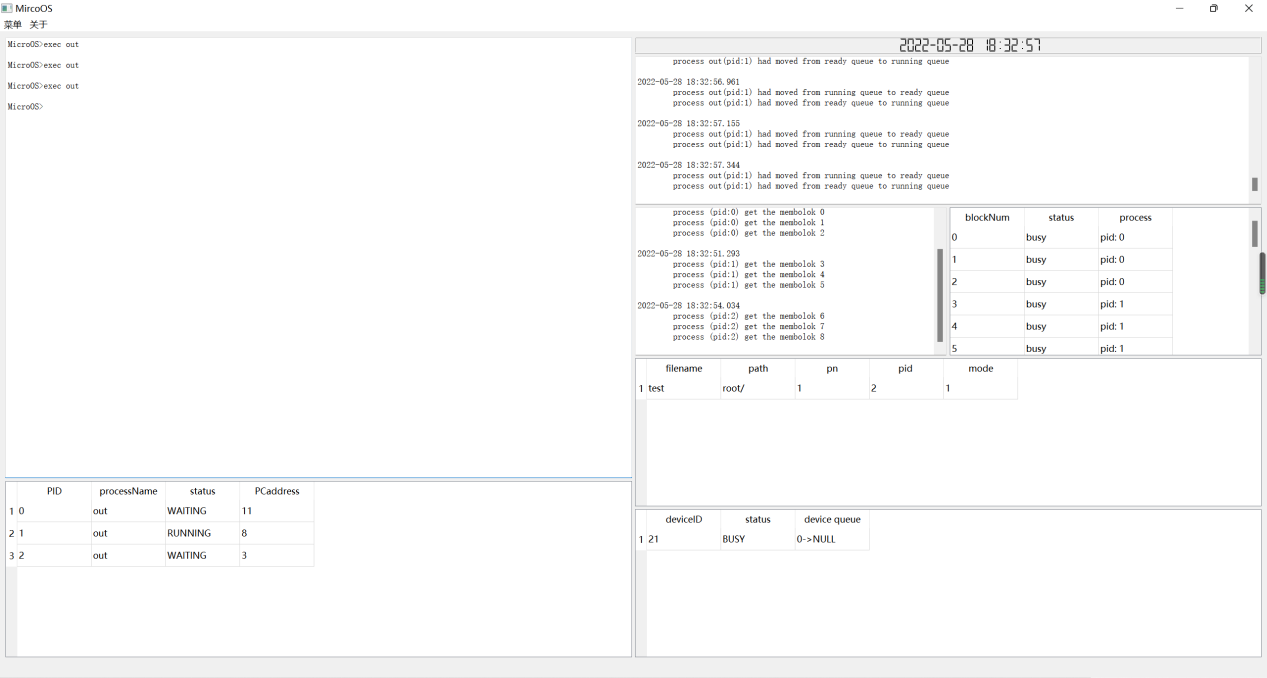
(5) 系统时钟使用lcd模块来展示

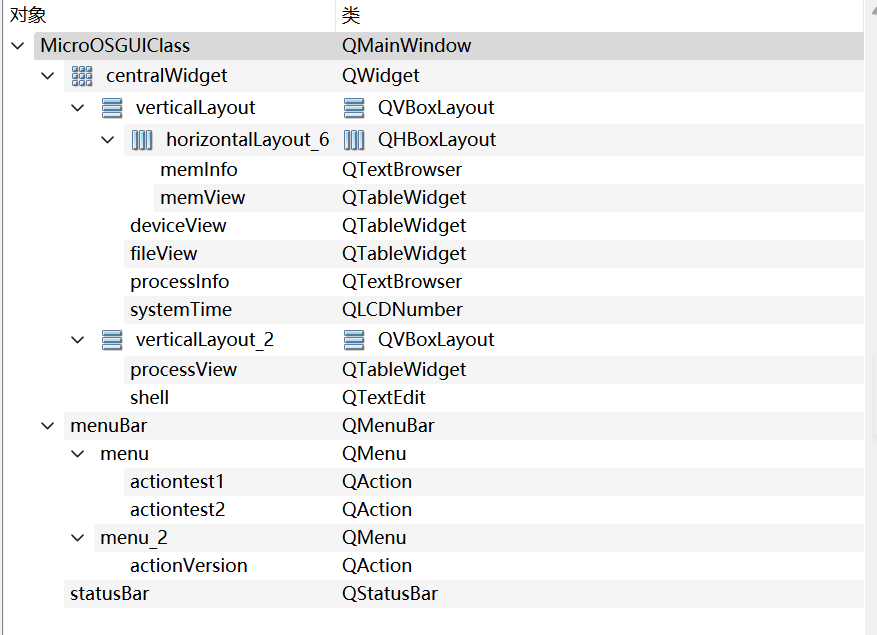
(6)设备使用情况，用Table Widget以表格形式实时动态展示设备状态和设备队列。

(7)文件打开表，用Table Widget以表格形式实时动态展示文件打开表状态和内容。

#### 布局设计







#### UI信号槽设计

## 程序清单

### （1）进程模块

#### 1.创建进程

//创建新进程

void init\_proc(string filename)

{

BYTE\* mem\_address = apply\_mem\_process(filename, latestPid + 1);

if (mem\_address!=NULL) {

pidPool[latestPid + 1] = (PCBptr)pcb\_apply\_mem(latestPid + 1);

PCBptr p = pidPool[latestPid + 1];

p->memPtr = mem\_address;

p->pid = latestPid + 1;

latestPid = p->pid;//更新当前最新pid

p->address = 0x0;

p->state = BEGIN;

p->createTime = time(0);

p->startTime = -1;

memset(p->instCache, 0, MAX\_AR\_LEN);

p->instLength = 3;

p->isInstFinished = true;

p->instRemainTime = 0;

p->isBegin = false;

start\_proc(p->pid);//进入ready队列

}

else {

cout << "Space not enough!" << endl;

}

}

#### 2.调度

int sche\_proc()

{

ready\_lock.lock();

//ready队列不空时,取队列的第一个pid

if (!ready\_queue.empty()){

int curPid = ready\_queue.front();

ready\_queue.pop\_front();

ready\_queue.push\_back(curPid);

ready\_lock.unlock();

return curPid;

}

else {

cout << "ready队列为空" << endl;

ready\_lock.unlock();

return -1;

}

}

#### 3.进程结束

// 运行结束退出

void exit\_proc(int pid)

{

if (pidPool[pid]->state == RUNNING) {

pidPool[pid]->state = FINISHED;

map<int, PCBptr>::iterator it = pidPool.find(pid);

pidPool[pid]->state = FINISHED;

release\_process(pid);//顺序不能错

pidPool.erase(it);

}

else

cout << "该进程未在运行中" << endl;

}

#### 4.进程状态转换相关函数

// 发生I/O或事件的等待

void running\_to\_waiting(int pid)

{

if (find\_in\_list(pid, running\_queue)){

running\_queue.remove(pid);

waiting\_queue.push\_back(pid);

pidPool[pid]->state = WAITING;

}

else

cout << "该进程未在运行中!" << endl;

}

// 发生I/O或事件的完成

void waiting\_to\_ready(int pid)

{

if (find\_in\_list(pid, waiting\_queue)) {

waiting\_queue.remove(pid);

ready\_lock.lock();

ready\_queue.push\_back(pid);

ready\_lock.unlock();

pidPool[pid]->state = READY;

}

else

cout << "该进程未在waiting队列!" << endl;

}

// 发生中断

void running\_to\_ready(int pid)

{

if (find\_in\_list(pid, running\_queue)) {

running\_queue.remove(pid);

ready\_lock.lock();

ready\_queue.push\_back(pid);

ready\_lock.unlock();

pidPool[pid]->state = READY;

}

else

cout << "该进程未在运行中!" << endl;

}

// 发生调度

void ready\_to\_running(int pid)

{

if (find\_in\_list(pid, ready\_queue)) {

ready\_lock.lock();

ready\_queue.remove(pid);

ready\_lock.unlock();

pidPool[pid]->state = RUNNING;

if (pidPool[pid]->startTime == -1) {

pidPool[pid]->startTime = time(0);

}

running\_queue.push\_back(pid);

}

else

cout << "该进程未在ready队列!" << endl;

}

### （2）内存模块

#### 内存数据结构

typedef struct Mem {

BYTE pcbMem[PCB\_SIZE];

BYTE ivtMem[IVT\_SIZE];

BYTE selMem[SEL\_SIZE];

BYTE pageTable[PAGE\_TABLE\_SIZE];

BYTE useMem[USE\_SIZE];

}Mem,\*MemPtr;

逻辑地址寻址

BYTE\* find\_phAdr(int pid, int vAdr)

{

PCBptr ptr;

ptr = pidPool[pid];

int vPageNum = vAdr / PAGE\_SIZE;

int offset = vAdr % PAGE\_SIZE;

BYTE\* pagetableAdr = &ptr->memPtr[3 \* vPageNum];//对应页表项的起始地址

int pPageNum;

int pageflag = pagetableAdr[2];

if (pagetableAdr[0] == -1 && pagetableAdr[2] == -1 && pagetableAdr[1] == -1)

return (BYTE\*)(-1);

else if (pageflag == 0)

return 0;

else

{

pPageNum = pagetableAdr[0];//找到对应内存块号

for (int i =0; i < USE\_SIZE / PAGE\_SIZE;i++)

{

if (flag[i] == pid)

{

if (i == pPageNum)

timeInMem[i] = 0;

else

timeInMem[i]++;

}

}

return &memPtr->useMem[pPageNum \* PAGE\_SIZE + offset];

}

}

#### 页面置换

void pagefault\_manage(int pid, int logAdr)

{

int pageNum = logAdr / PAGE\_SIZE;

BYTE\* pagetableAdr = pidPool[pid]->memPtr;

BYTE diskNum = pagetableAdr[pageNum \* 3 + 1];//需要换入的磁盘号

int maxTime = 0;

int outPage = 0;

for (int i = 0;i < USE\_SIZE / PAGE\_SIZE;i++)//根据LRU找到需要换出的内存块

{

if (flag[i] == pid && timeInMem[i] > maxTime)

{

maxTime = timeInMem[i];

outPage = i;

}

}

vm\_swap(&memPtr->useMem[outPage \* PAGE\_SIZE], diskNum);

for (int i = 0;i < 64 \* 3;i++)

{

if (pagetableAdr[i] == outPage)

{

pagetableAdr[i + 2] =0;

pagetableAdr[i + 1] = diskNum;

break;

}

}

modify\_table((BYTE)pageNum, (BYTE)outPage, 0, 1, pagetableAdr);

}

### （3）文件系统

// 打开文件

int open\_file(string fileName, int mode, int pid) // 打开文件

{

fileLock.lock();

int flag = 1;

// 支持绝对路径

// 寻找该文件的fcb并验证权限

// 在系统文件打开表中找有无该文件的表项，如果没有，新建这一项

// 如果有，验证打开模式，只有读模式才能打开（写模式不允许多个进程同时打开文件）

// 验证成功，允许打开

// 修改系统级文件打开表的相关项

// 新建进程级文件打开表项（若进程是第一次打开文件，那么新建一张该进程的表）

string rowName = tempDirName;

string path;

string name;

get\_path\_and\_name(fileName, path, name);

flag = cd(path);

if (flag > 0)

{

int location = get\_dir\_entry\_location(name);

if (location >= 0) //存在目录项

{

fcb file = tempDir[location];

if (file.attribute == COMMONFILE && file.rights >= mode) // 验证权限成功

{

// 在系统级文件打开表定位该项

int softPosition = get\_soft\_position(path, name);

if (softPosition < 0) // 定位失败，说明该文件首次被打开，生成表项追加到文件打开表

{

int pn = 1; // 只有一个进程打开该文件

vector<int> pidArr;

pidArr.push\_back(pid);

softEntry entry = { file, path, pn, pidArr, mode };

soft\_entry\_op(entry, MKSOFTENTRY);

}

else // 系统级定位成功 ,验证打开模式（只允许读），对应表项pn+1,压入pid;

{

if (mode == RO)

{

softEntry entry = soft[softPosition];

// 是否重复打开？验证pid向量

for (int i = 0; i < entry.pid.size(); i++)

{

if (entry.pid[i] == pid)

{

flag = 0; // 不允许同一个进程重复打开同一个文件

}

}

if (flag > 0)

{

entry.pn += 1;

entry.pid.push\_back(pid);

soft\_entry\_op(entry, UPDATEENTRY); // 更新操作

}

}

else // 打开模式为写模式，拒绝，不可以有多个进程写同一个文件

{

flag = 0;

}

}

}

else // 非法操作：写目录 或者是写只读文件

{

flag = 0;

}

if (flag > 0) // 对进程级打开表的操作

{

// 根据pid在进程级打开文件表中搜索是否存在该进程的表

int position = get\_poft\_position(pid);

// 生成表项

poftEntry entry = { file, path, mode };

// 如果已经存在它的表说明打开过其他文件

if (position >= 0)

{

// 创建表项

poft\_entry\_op(entry, pid, MKPOFTENTRY);

}

// 不存在这张表，则创建一张

else

{

poft table;

vector<poftEntry> entries = { entry };

table.pid = pid;

table.entries = entries;

// 创建表

poft\_op(table, MKPOFTTABLE);

}

}

}

}

cd(rowName);

fileLock.unlock();

return flag;

}

// 写文件

int write\_file(string fileName, vector<unsigned char\*>& mem, int size)

{

// 只有存在文件才可以写文件

// 需要修改该文件的目录项utime

// 可能需要申请物理块，更新fat表

// 按块将内容写入磁盘

// 可能需要需要修改fat的size和length

// 完成后重新加载一遍目录

int flag = 1;

string rowName = tempDirName;

flag = exist\_file(fileName);

int blockSize = size <= 1024 ? 1 : size / 1024;

if (flag >= 0)

{

// 只能够按页（块）操作

// 比较需要写入的大小是否超过了原先的大小（长度）

string path;

string name;

get\_path\_and\_name(fileName, path, name);

cd(path);

int location = get\_dir\_entry\_location(name);

fcb fileFcb = tempDir[location];

unsigned short fileLength = fileFcb.length;

unsigned short startBlock = fileFcb.startBlock;

vector<unsigned short> fileAddr;

time\_t lt;

lt = time(NULL);

unsigned int utime = (unsigned int)lt;

fileFcb.updateTime = utime;

flag = get\_fats(startBlock, fileAddr, fileLength); // 文件原来的物理块

vector<unsigned short> fileBlocks;

if (flag >= 0)

{

if (fileLength < blockSize) // 写入大小超过了原大小，重新分配物理块

{

// 申请新的物理块

int newBlockSize = blockSize - fileLength;

vector<unsigned short> blocks;

get\_blocks(newBlockSize, blocks);

// 将申请到的物理块和文件原物理块相连,并更新文件分配表

// 生成文件的所有块号 记在 fileBlock中。

vector<unsigned short> nextBlocks;

for (int i = 0; i < fileLength; i++)

{

fileBlocks.push\_back(fileAddr[i]);

if (i < fileLength - 1)

{

nextBlocks.push\_back(fileAddr[i + 1]);

}

}

for (int i = 0; i < blocks.size(); i++)

{

fileBlocks.push\_back(blocks[i]);

nextBlocks.push\_back(blocks[i]); // next的数量比file少一，这里保持一致即可

}

// next 补上结束标记

nextBlocks.push\_back(FENDFLAG);

// 更新文件分配表

update\_fat(fileBlocks, nextBlocks);

fileFcb.size = size;

fileFcb.length = blockSize;

}

else

{

fileFcb.size = size;

fileBlocks = fileAddr;

}

// 申请好物理块之后，或者本身长度就够了

// 逐块的写文件

for (int i = 0; i < fileBlocks.size(); i++)

{

// 定位

int offset = fileBlocks[i] \* BLOCKSIZE;

disk.open(DISKNAME, ios::binary | ios::in | ios::out);

disk.seekp(offset, disk.beg);

disk.write((char\*)mem[i], BLOCKSIZE \* sizeof(unsigned char));

disk.close();

}

// 更新目录项

cd(path);

qDebug() << update\_dir\_entry("", fileFcb);

cd(path); // 刷新当前路径

}

}

cd(rowName);

return flag;

}

### （4）设备模块

int apply\_device(int pid, unsigned char deviceID) {

int index = get\_index\_by\_deviceID(deviceID);

DCTItem\* tmpDCTPtr;

PCBPtrQueue\* tmpPCBPtr;

if (index == -1) {

qDebug() << "UNKNOW DEVICE : " << deviceID << endl;

return 0;

}

else {

tmpDCTPtr = SDTPtr->SDTitem[index].DCTItemPtr;

tmpPCBPtr = (PCBPtrQueue\*)malloc(sizeof(PCBPtrQueue));

tmpDCTPtr->waitingQueueEnd->next = tmpPCBPtr;

if (tmpPCBPtr != NULL) {

tmpPCBPtr->pcbPtr = pidPool[pid];

tmpPCBPtr->next = NULL;

tmpDCTPtr->waitingQueueEnd = tmpPCBPtr;

qDebug() << "pid: " << pid << "APPLY" << endl;

return 1;

}

else {

return 0;

}

}

}

### （5）CPU设计

while (1) {

// 取一个准备执行的进程(在Ready状态)

currentPid = sche\_proc();

if (currentPid != -1) {

current = pidPool[currentPid];

// 取到后变为Running状态

if( current != NULL) ready\_to\_running(current->pid);

}

else {

continue;

}

if (current == NULL) continue;

// 取指

if (!load\_inst(current)) {

exit\_proc(current->pid);

continue;

}

else {

pulse = 0;

}

while (!pulse) {

intervel = 0;

start = clock();

// 判断时间片结束和指令是否结束执行

while (!pulse && intervel < osRegister.instRemainTime) {

// 执行

if (!current->isBegin) {

inst\_enter(current);

}

current->isBegin = true;

end = clock();

// 计算CPU耗时

intervel = end - start;

}

// 指令结束完成

if (intervel >= osRegister.instRemainTime) {

inst\_leave(current);

current->isBegin = false;

current->isInstFinished = true;

current->instLength = osRegister.instLength;

if (current->state != RUNNING) break;

if (!load\_inst(current)) {

exit\_proc(current->pid);

break;

}

intervel = 0;

}

}

// 如果进程是RUNNING状态将其放置到ready队列

if( current->state == RUNNING ) running\_to\_ready(current->pid);

// 修改PCB进行缓存

cache\_pcb(current, intervel);

// 脉冲下降沿

pulse = 0;

}

### （6）中断设计

void init\_ivtable() {

int i = 0;

ivTable = (IVTableptr)memPtr->ivtMem;

ivTable->size = 3;

// 异步IO中断 传输完成后告知CPU

ivTable->ivtItem[0].id = 0x0;

ivTable->ivtItem[0].accept = true;

ivTable->ivtItem[0].type = IO\_INTERRUPT;

memcpy(ivTable->ivtItem[0].info, "IO", 2);

ivTable->ivtItem[0].function = (unsigned char \*)DISK\_interrupt;

// 1 2 中断留空

// 设备中断 设备使用完成通过中断通知CPU

ivTable->ivtItem[3].id = 0x3;

ivTable->ivtItem[3].accept = true;

ivTable->ivtItem[3].type = IO\_INTERRUPT;

memcpy(ivTable->ivtItem[3].info, "DEVICE", 6);

ivTable->ivtItem[3].function = (unsigned char\*)DEVICE\_interrupt;

// 缺页中断 访问到不在内存中地址时需要触发缺页中断调入页面

ivTable->ivtItem[4].id = 0x4;

ivTable->ivtItem[4].accept = true;

ivTable->ivtItem[4].type = EXCEPTION;

memcpy(ivTable->ivtItem[4].info, "Page", 4);

ivTable->ivtItem[4].function = (unsigned char\*)pagefault\_manage;

}

### （7）UI设计

要将各个模块产生的信息实时输出至ui，这里设计使用qt特有的信号槽机制来实现。

使用timer发送计时信号，将文本信息输出与动态表格输出和timer绑定起来，构成信号与槽，控制ui模块定时进行输出。

//新建一个QTimer对象

timer = new QTimer();

//设置定时器每个多少毫秒发送一个timeout()信号

timer->setInterval(1);

//启动定时器

timer->start();

文本信息的输出：将各模块需要输出的信息存在各模块对应缓存区中，ui模块会定时读取缓存区，若缓冲区非空，将会将区内内容输出至对应ui控件，再清空缓冲区。

动态表格的输出：表格中的数据，会随着数据源的变化而实时变化。Ui模块定时清除表格，同时读取数据源，生产新表格，实现数据的动态展示。

//信号和槽

connect(timer, SIGNAL(timeout()), this, SLOT(onTimerOut()));

connect(timer, SIGNAL(timeout()), this, SLOT(refreshMemTable()));

connect(timer, SIGNAL(timeout()), this, SLOT(refreshProTable()));

connect(timer, SIGNAL(timeout()), this, SLOT(refreshFileTable()));

connect(timer, SIGNAL(timeout()), this, SLOT(refreshDeviceTable()));

connect(timer, SIGNAL(timeout()), this, SLOT(readMemBuffer()));

connect(timer, SIGNAL(timeout()), this, SLOT(readProcBuffer()));

## 测试报告

### 1）测试环境

Windows

### 2）测试的功能

#### 进程的创建、调度、控制

#### 进程执行期间对资源的使用情况

#### 进程状态变化，在各种队列之间的迁移

#### 进程申请内存总大小小于3帧时内存情况

#### 进程申请内存总大小大于3帧，且访问大于3帧部分的内容（触发缺页中断）

#### 验证LRU算法

#### 内存空间不足时进程创建情况

#### 设备队列测试

#### 创建及删除文件，文件夹

#### 读文件，写文件

#### 文件打开表测试

#### 多程序运行时文件打开表的管理：

### 3）针对每个功能的测试情况

#### 进程的创建、调度、控制

umem 4096

calu 4000

devi 21 2000

read 9 root/test 5000

devi 20 2000

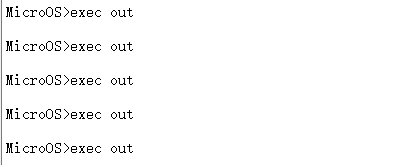
addr 4000 300

writ 12 root/test123 2000

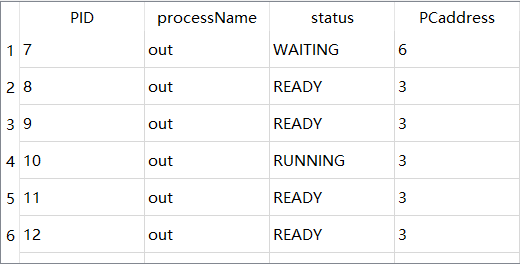
devi 2 2000

addr 4000 300exit

创建进程。



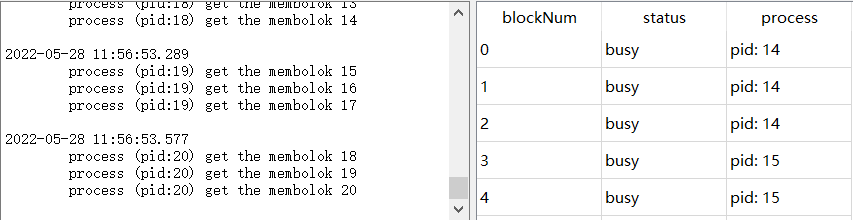
进程调度及控制。



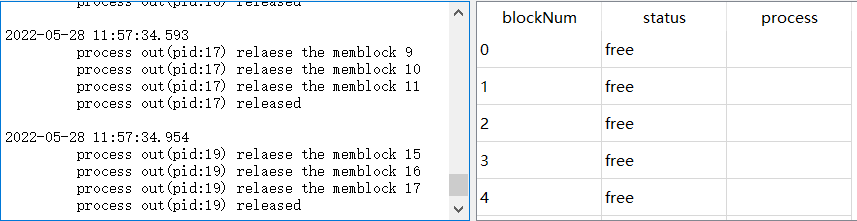
采用轮询调度，当进程位于就绪队列首位时进入RUNNING。发生中断由RUNNING进入READY，发生I/O或事件等待由RUNNING进入WAITING，发生I/O或事件的等待由WAITING 进入READY。

#### 进程执行期间对资源的使用情况

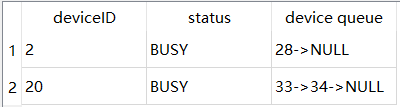
进程执行时使用内存块信息:当进程创建成功之后,内存模块会为该进程的PCB分配内存,相关信息由PCB中的内存指针存储。



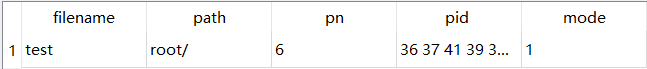
进程结束后释放,当进程运行结束之后,由内存模块释放与该进程相关的内存信息,内存块的状态置为free.



设备使用情况,设备被使用后状态置为BUSY,同时由device queue 显示其使用队列的顺序。

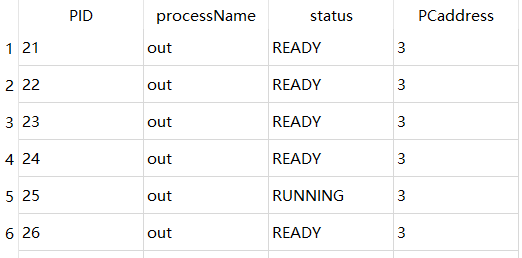


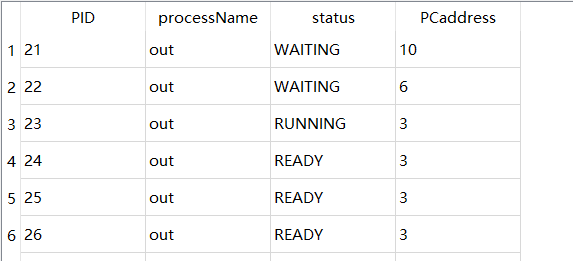
文件使用情况,当进程读写文件时,显示文件路径及使用队列信息.



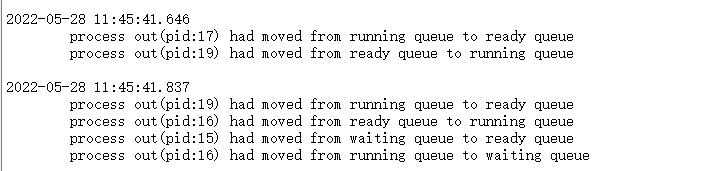
#### 进程状态变化，在各种队列之间的迁移

状态变化。



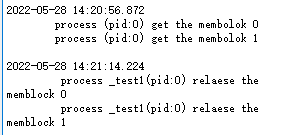


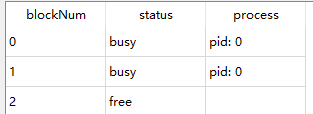
队列之间迁移。



#### 进程申请内存总大小小于3帧时内存情况

umem 1024  
read 9 root/test 5000  
calu 4000  
devi 21 2000  
writ 12 root/test123 2000  
devi 20 2000  
addr 4000 300  
devi 2 2000  
exit

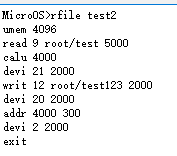


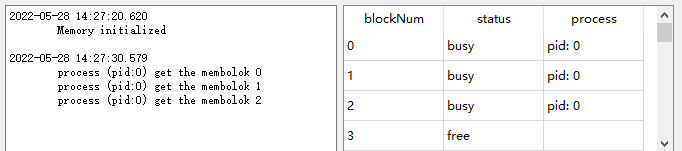


进程总共申请了2页内存，在进程结束后释放。内存占用展示模块表明该进程执行途中占用了内存帧0和1。

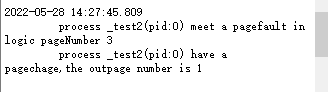
#### 进程申请内存总大小大于3帧，且访问大于3帧部分的内容（触发缺页中断）

umem 4096  
read 9 root/test 5000  
calu 4000  
devi 21 2000  
writ 12 root/test123 2000  
devi 20 2000  
addr 4000 300  
devi 2 2000  
exit





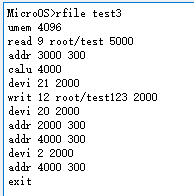
可以看到该进程在执行的过程中占用了帧0、1和2。



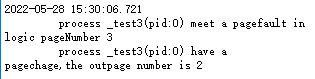
由于访问了不在内存中的内容，所以发生了一次缺页中断，并进行了一次页面置换，根据LRU算法，置换出去的帧号为1。

#### 验证LRU算法

umem 4096  
read 9 root/test 5000  
addr 3000 300  
calu 4000  
devi 21 2000  
writ 12 root/test123 2000  
devi 20 2000  
addr 2000 300  
addr 4000 300  
devi 2 2000  
addr 4000 300  
exit



可以看到程序指令先访问帧3，再访问帧2。

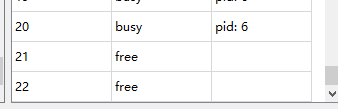


这次选择换出的帧号为2，和测试2中的帧号1不同，且进行置换后再次访问该逻辑页没有发生缺页，符合LRU算法。

#### 内存空间不足时进程创建情况

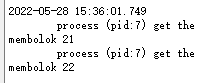
连续执行测试f)中的进程8次

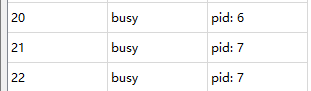




可以看到最后一次执行没有成功由于内存不足。

最后内存还剩下2帧的大小，尝试一下执行测试d)中的进程。





可以看到执行成功，刚好将最后剩下的两帧分配给了进程7.

#### 设备队列测试

umem 4096

read 9 root/test 5000

calu 4000

devi 21 2000

writ 12 root/test123 2000

devi 20 2000

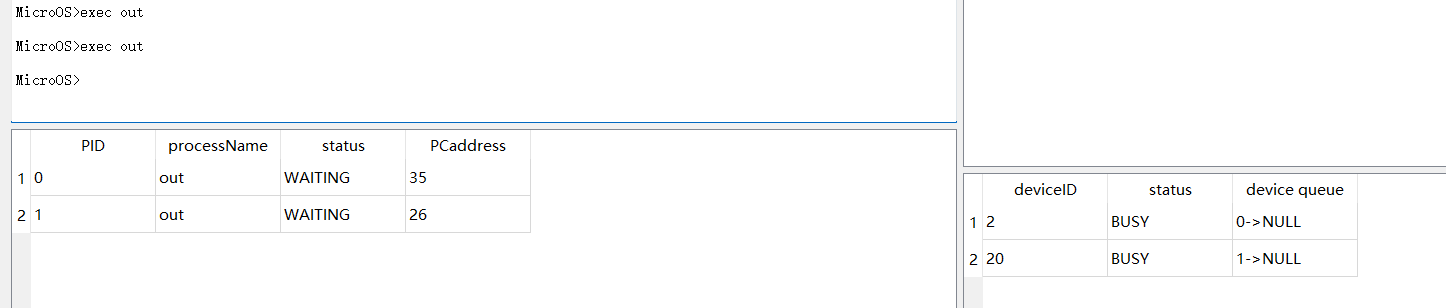
addr 4000 300

devi 2 2000

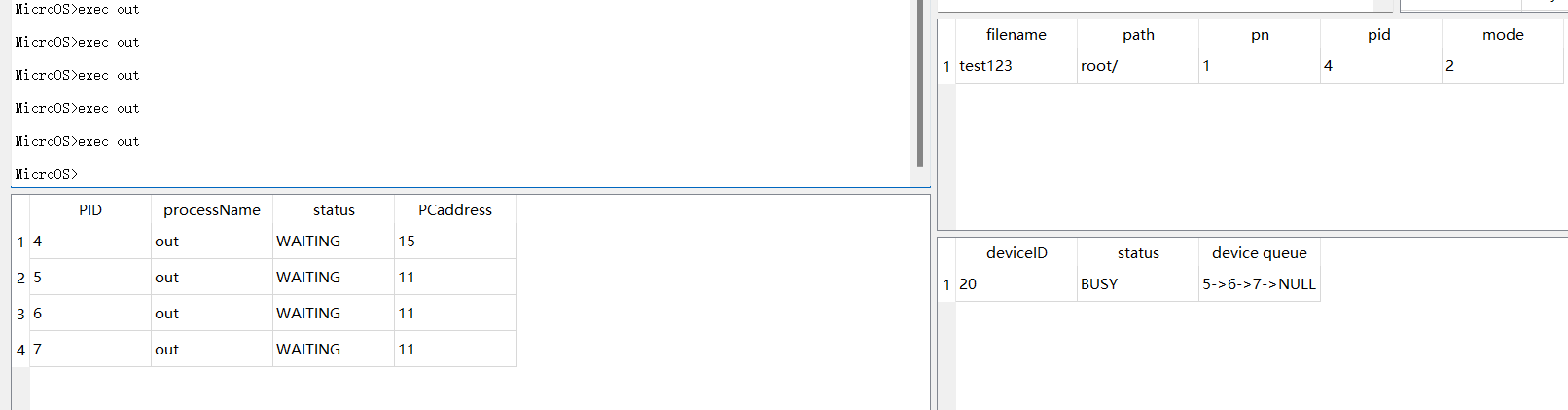
addr 4000 300

exit

如图所示，当执行两个out进程时，两个进程分别在实验不同的设备，并且device queue展示了使用情况，与预期相符。

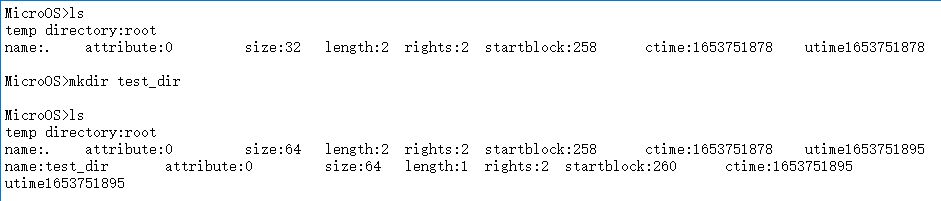


若同时执行多个进程，且同时申请有个设备，如图所示 5 6 7三个进程PC值都为11，即都阻塞在使用20号设备，设备队列则显示申请顺序为5 ->6-> 7，与预期相符。

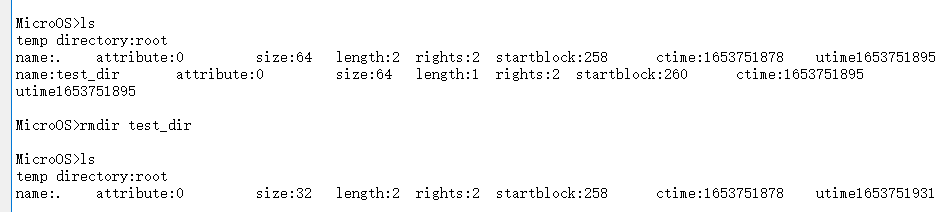


#### 创建及删除文件，文件夹

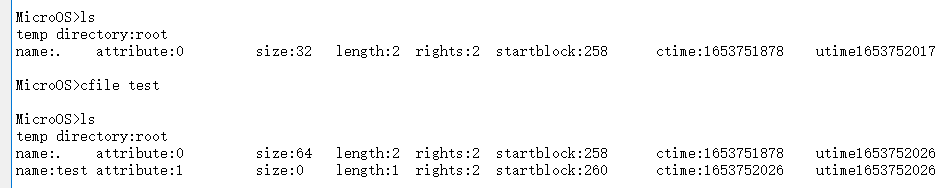
在任一目录下，在输入框输入mkdir dir\_name 即可新建文件夹



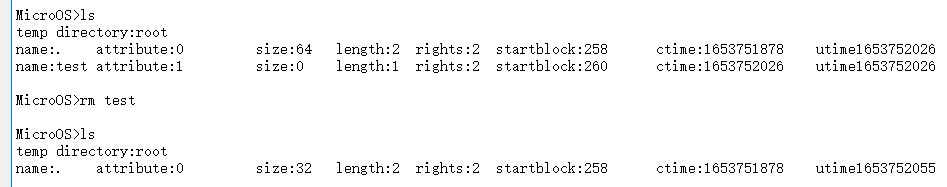
在任一目录下，在输入框输入rmdir dir\_name 即可删除文件夹



在任一目录下，在输入框输入cfile file\_name 即可新建文件

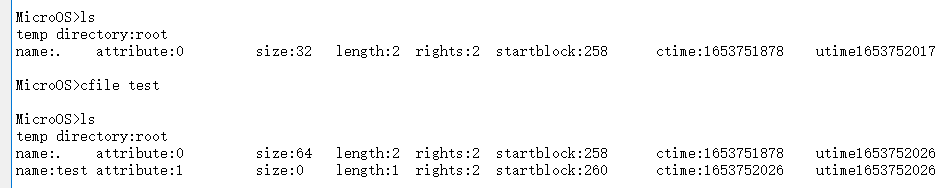


在任一目录下，在输入框输入rmdir file\_name 即可删除文件

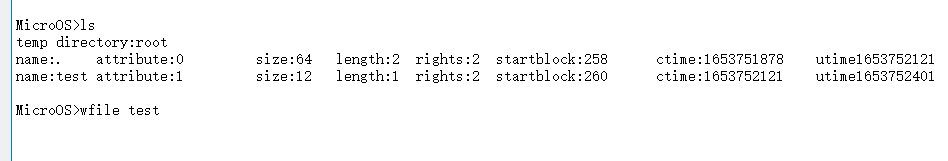


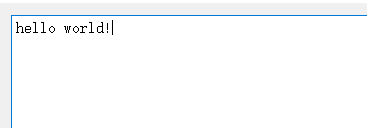
#### 读文件，写文件

写文件：首先创建文件“test”，在输入框输入cfile test 即可新建文件下图演示了创建“test”文件的过程

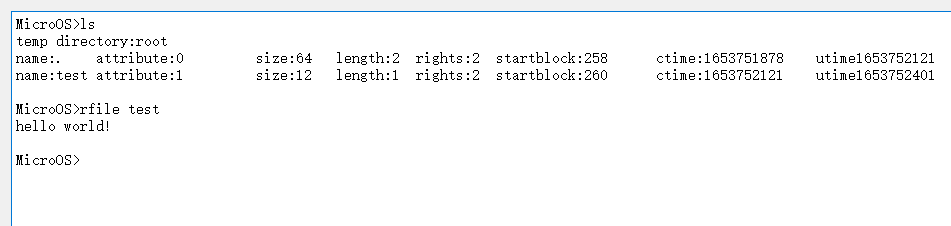


写入“hello world！” ， 在输入框输入wfile file\_name 即可写文件下图演示了写“test”文件的过程



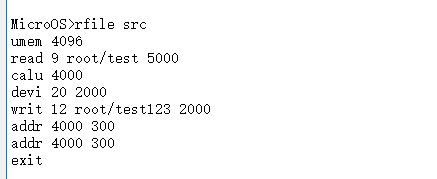


写完之后用“Ctrl”+“S”保存并退出书写框，在输入框输入rfile file\_name 即可读文件，下图演示了读取“test”文件的过程

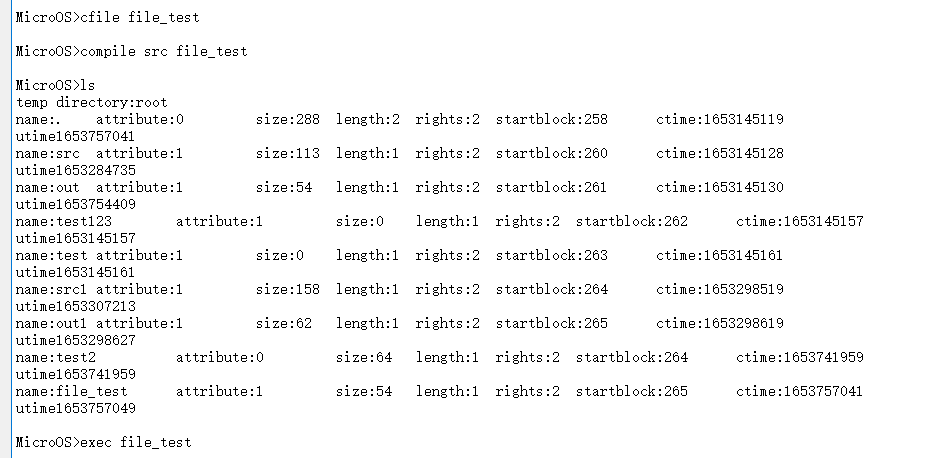


#### 文件打开表测试

首先编写程序“src”，其中有read和write 文件的指令：

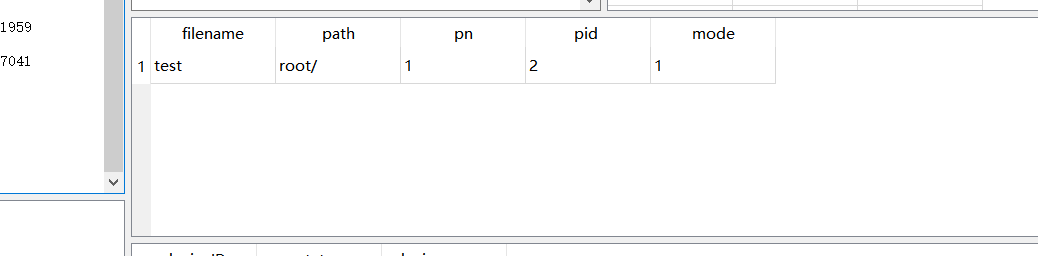


然后编译src，创建文件“file\_test”，编译输出可执行程序“file\_test”，并运行“file\_test”

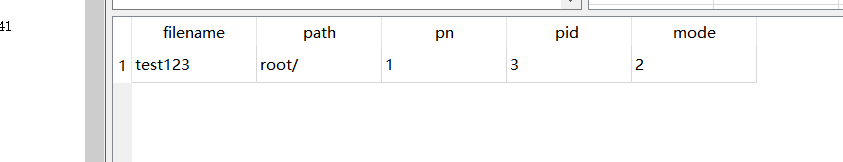


在右侧的文件打开表中有：

可执行程序中“read”指令所打开的文件：

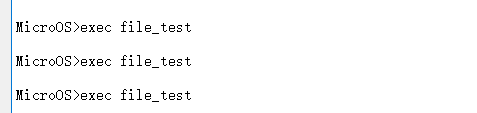


可执行程序中“write”指令所打开的文件：

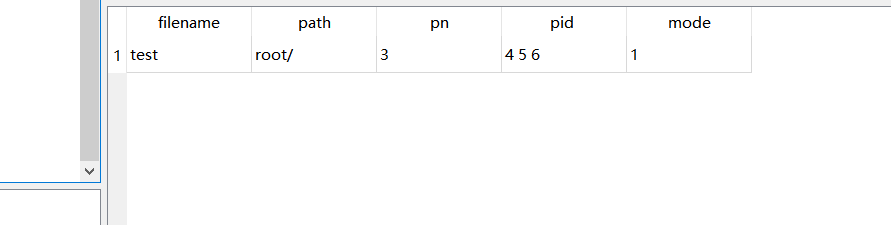


#### 多程序运行时文件打开表的管理：

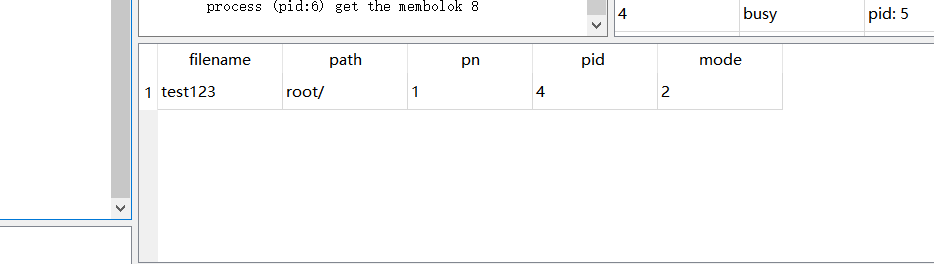
首先同时运行3次“file\_test”程序：



当多个程序同时存在读同一文件时，文件打开表会分别使用pid进行标记



当多个程序同时存在写同一文件时，只有第一个打开文件的pid，后续文件打开失败继续执行后续指令，不存在pid的输出



## 课程设计总结

### 1) 设计过程中遇到或存在的主要问题及解决方案

#### 在执行进程的过程中,ready\_queue等队列中的数值会溢出:忽略了对于临界资源的判断.shell和UI为多线程,会同时访问操作队列等临界资源.因此需要加上互斥锁.

#### 不知道其他模块的函数如何调用:开启腾讯会议进行联调

#### 在概要设计时，文件系统中的各种“表”如目录、系统文件打开表、进程打开文件表等等都暂定使用链表来实现。当时是考虑到普通的数组删除操作会非常麻烦，才选择使用链表。但是在详细设计阶段，发现这样的做法会极大增加工作量，即便是遍历链表这样最基础的操作，代码量也会很大，而且还有追加、删除等操作。这样反而和选择链表的初衷背道而驰了。解决办法也很简单，选择一种追加、删除、更新操作都很简单的数据结构就可以了。C++刚好提供了这样的数据结构：向量。它其实可以作为一种特殊的不定长的数组，并且库函数中就囊括了这些“表”需要的操作。这样就极大地减少了编写代码阶段的工作量。

#### qt布局设计和实际展示的不一致，控件位置不理想。解决办法：将每一个控件都至少处于一个layout中，并设置相对位置来约束控件位置。

#### 其他模块数据无法输出至控件。解决办法：设置缓存区，ui模块实时读取缓存区内容并输出至控件。

#### 输出对应模块信息时，逻辑上不同时产生前后两条信息被当成一条信息输出，导致输出粘连。解决办法：实际上，两条信息产生间隔小于1ms，ui模块最快没1ms读取一次缓存区，所以无法给给这样的每条信息加上时间信息，所以给粘连的信息分行输出，给予的时间信息。

#### 在测试的过程中发现，给进程分配帧时会出现会分配给进程两倍的帧大小的情况，导致会有很多的空间无法被利用。经过测试后发现，在分配进程时会将分配进程的函数放到if-else if的判断条件中，导致内存分配的函数会执行两次。在用一个变量存储内存分配的函数的返回值后再进行if-else if判断就解决了这个问题。

### 2) 改进建议

操作系统设计还可以从更精细的角度和方向去模拟，例如使用精度更高的指令集，使用更加设计精巧的中断处理。其次还是要更贴近于真实的操作系统，这样才能收获更多。