OS课设——TJOS项目文档

组长： 张铃沛 1953196

组员： 周心怡 1951102

成以恒 1950059

杨飞航 1952347

李梦祥 1952726

指 导 教 师：张慧娟 王冬青

学院、 专业： 软件工程

同济大学

Tongji University

目录

[OS课设——TJOS项目文档 1](#_Toc81077022)

[组长： 张铃沛 1953196 1](#_Toc81077023)

[组员： 周心怡 1951102 1](#_Toc81077024)

[成以恒 1950059 1](#_Toc81077025)

[杨飞航 1952347 1](#_Toc81077026)

[李梦祥 1952726 1](#_Toc81077027)

[指 导 教 师：张慧娟 王冬青 1](#_Toc81077028)

[学院、 专业： 软件工程 1](#_Toc81077029)

[同济大学 1](#_Toc81077030)

[Tongji University 1](#_Toc81077031)

[**1.** **项目简介** 2](#_Toc81077032)

[**2.** **TJOS操作系统工作流程** 3](#_Toc81077033)

[**3.** **编写过程** 3](#_Toc81077034)

[**3.1** **环境搭建** 3](#_Toc81077035)

[**3.2** **boot sector 引导扇区的编写** 4](#_Toc81077036)

[**3.3** **保护模式的建立** 5](#_Toc81077037)

[**3.3.1** **GDT的编写** 5](#_Toc81077038)

[**3.3.2** **从实模式到保护模式的跳转** 5](#_Toc81077039)

[**3.3.3** **特权级的设置** 5](#_Toc81077040)

[**3.3.4** **特权级的转移** 5](#_Toc81077041)

[**3.3.5** **页式存储的实现** 5](#_Toc81077042)

[**3.4** **Loader的编写和导入内存** 5](#_Toc81077043)

[**3.5** **内核雏形的形成** 5](#_Toc81077044)

[**3.6** **加入中断处理的内核形成** 6](#_Toc81077045)

[**3.7** **加入单进程的内核形成** 6](#_Toc81077046)

[**3.8** **多进程** 11](#_Toc81077047)

[**3.9** **加入输入输出系统的内核** 11](#_Toc81077048)

[**4.** **成果** 16](#_Toc81077049)

[**4.1** **用户登录及权限管理** 16](#_Toc81077050)

[**4.2** **文件系统** 18](#_Toc81077051)

[**4.3** **进程管理** 21](#_Toc81077052)

[**4.4** **应用开发** 25](#_Toc81077053)

[**4.5** **命令行功能** 38](#_Toc81077054)

1. **项目简介**

本项目主要参考《Orange’S 一个操作系统的实现》进行编写。其中，我们以Orange’S的样例代码为基础，同时参考了网上的操作系统的代码，最终完成了我们的项目。

本操作系统在Ubuntu上进行编写，通过make进行编译，生成磁盘映像并在bochs上运行。

在Orange’S代码的基础上，我们实现了用户登录与权限管理、多级文件系统、进程管理、用户级应用等功能。

1. **TJOS操作系统工作流程**

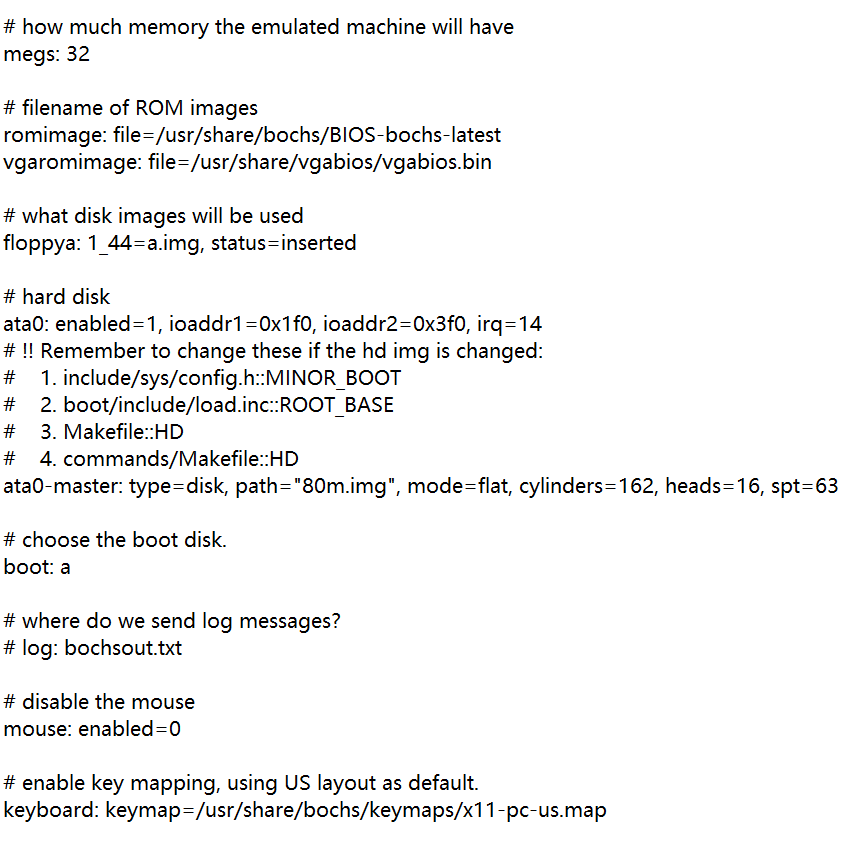
从软盘的引导扇区引导 -> 在软盘中查找‘loder.bin’ -> 跳转到’loader.bin‘中的代码并开始执行 -> 在软盘中查找操作系统内核‘kernel.bin’ -> 跳入保护模式 -> 加载‘kernel.bin‘ -> 跳转’kernel.bin‘中的代码并开始执行 -> 更新 GDT -> 初始化 IDT ->初始化 TSS -> 跳入系统主函数 -> 启动系统进程 -> 开启时钟中断 -> 开启进程调度 -> 系统开始运转

1. **编写过程**
   1. **环境搭建**

本项目在Vmware虚拟机上完成，通过Vmware安装Ubuntu linux操作系统，并在其上进行操作系统的编写。

在Terminal上直接使用命令行安装bochs，并通过bochs调试和运行TJOS操作系统。

bochs配置文件的编写（bochsrc）：



打开终端，输入命令“whereis bochs“找到bochs的全路径，根据bochs的路径更改romimage，vgaromimage以及keymap的值，并输入对内存、软盘映像、和硬盘映像的要求，完成对bochs的配置，为后续工作做好准备。

* 1. **boot sector 引导扇区的编写**

通过学习知道，当计算机开机时，会先进行POST，然后寻找启动盘。我们的操作系统是从软盘映像进行启动，所以，我们的引导扇区将会以0xAA55结束。在BIOS发现引导扇区后，会将512字节的执行码装载到内存地址0000：7c00处，并跳转到此处，将控制权完全交给这段引导代码。此后，计算机由操作系统的一部分来控制。引导扇区代码如下：

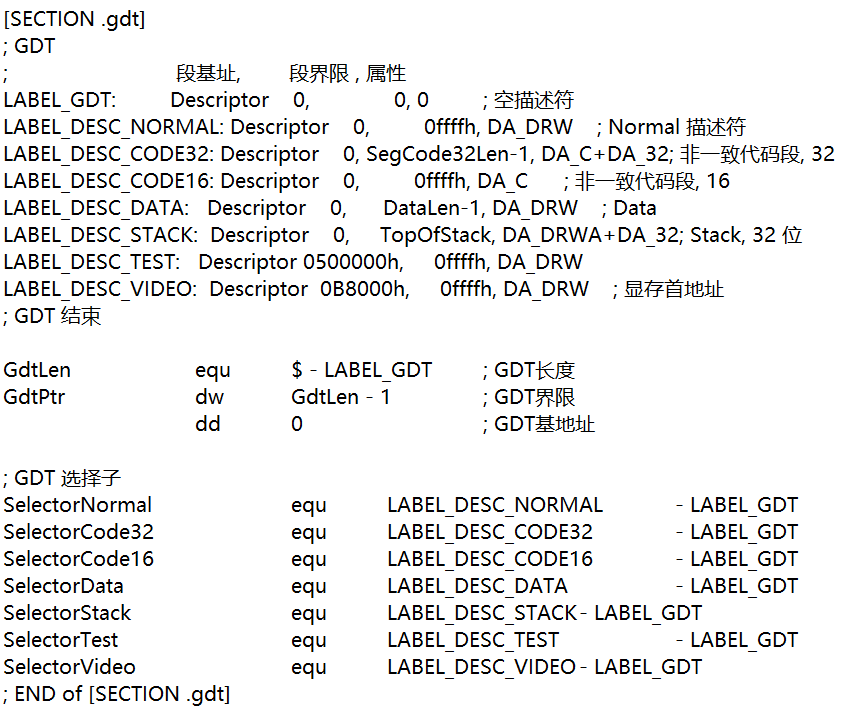


* 1. **保护模式的建立**
     1. **GDT的编写**

我们知道在IA32下，CPU有实模式和保护模式两种工作模式。在保护模式下，CPU有着巨大的寻址能力，并为强大的32位操作系统硬件保障，而我们要做的是实现由实模式进入到保护模式。

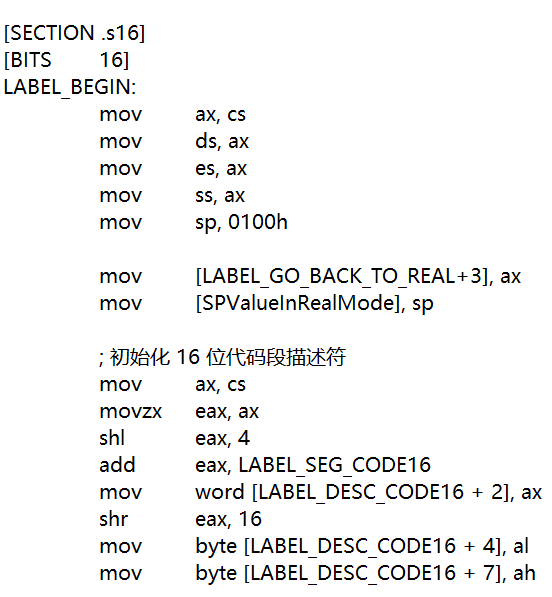
在实模式下，16位的寄存器通过“段：偏移”的方法能达到1MB的寻址能力，而对于 32位寄存器，一个寄存器就可以寻址4GB的空间。而保护模式下“段”的概念发生了根本性的变化。实模式下，段值可以看做是地址的一部分，段值为XXXXh表示以XXXX0h开始的一段内存。而保护模式下，虽然段值仍然由原来16位的cs、ds等寄存器表示，但此时它仅仅是一个索引，这个索引指向一个GDT（或者LDT）的一个描述符，描述符中详细定义了段的起始地址、界限、属性等内容。要编写保护模式，我们首先需要完成的是对GDT的编写。

GDT的作用是用来提供段式存储机制，这种机制通过段寄存器和GDT中的描述符共同提供。GDT的部分编写如下：



* + 1. **从实模式到保护模式的跳转**

先看部分代码：



在[section.s16]中，对32位代码段描述符进行了初始化，然后将GDT的物理地址填充到了GdtPtr中，再执行lgdt [GdtPtr]将其指示的数据加载到gdtr寄存器（这一步见具体文件）用于保存32位基地址和16位界限，接下来进行关中断，准备进入到保护模式。而为了访问所有的内存，我们需要把A20打开，在开机时它默认关闭。寄存器cr0的第0位是PE位，他决定CPU是处于实模式还是保护模式。此位为0时，CPU运行于实模式，为1时，CPU运行于保护模式。所以在最后一步，执行代码，将此位赋值为1，CPU跳转进入保护模式。

所以，从实模式跳入到保护模式得步骤可总结为以下4步：

1 准备GDT

2 用lgdt加载gdtr

3 打开A20

4 置cr0的PE位

5 跳转进入保护模式

但此时cs的值仍然是实模式下的值，所以还需将上述代码段中，选择子的值装入cs。

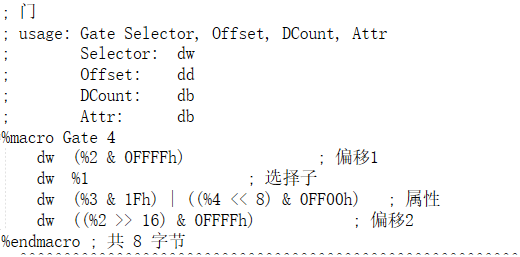
* + 1. **特权级的设置**

在IA32的分段机制中，特权级总共有4个特权级别，从高到低分别是0、1、2、3。数字越小表示的特权级越大。较为核心的代码和数据，将被放在特权级较高的层级中。处理器将用这样的机制来避免低特权级的任务在不被允许的情况下访问位于高特权级的段。处理器通过识别CPL、DPL、RPL这3种特权级进行特权级检验，如果处理器检测到一个访问请求是不合法的，将会产生常规保护错误。

* + 1. **特权级的转移**

程序从一个代码段转移到另一个代码段之前，目标代码段的选择子会被加载到cs中。作为加载过程的一部分，处理器将会检查描述符的界限、类型、特权级等内容。如果检验成功，cs将被加载，程序控制将转移到新的代码段中，从eip指示的位置开始执行。

特权级转移可以分为两类，一类是通过jmp和call的直接转移，另一类是通过运用门描述符或者TSS进行转移。



如图，是对于调用门的宏定义。在一些情况下，可以通过调用门的使用来实现特权级的转移。

* + 1. **页式存储的实现**

页式存储，是为了使内存管理更灵活。其本质是通过一个二级页表，实现从逻辑地址到物理地址的转换。

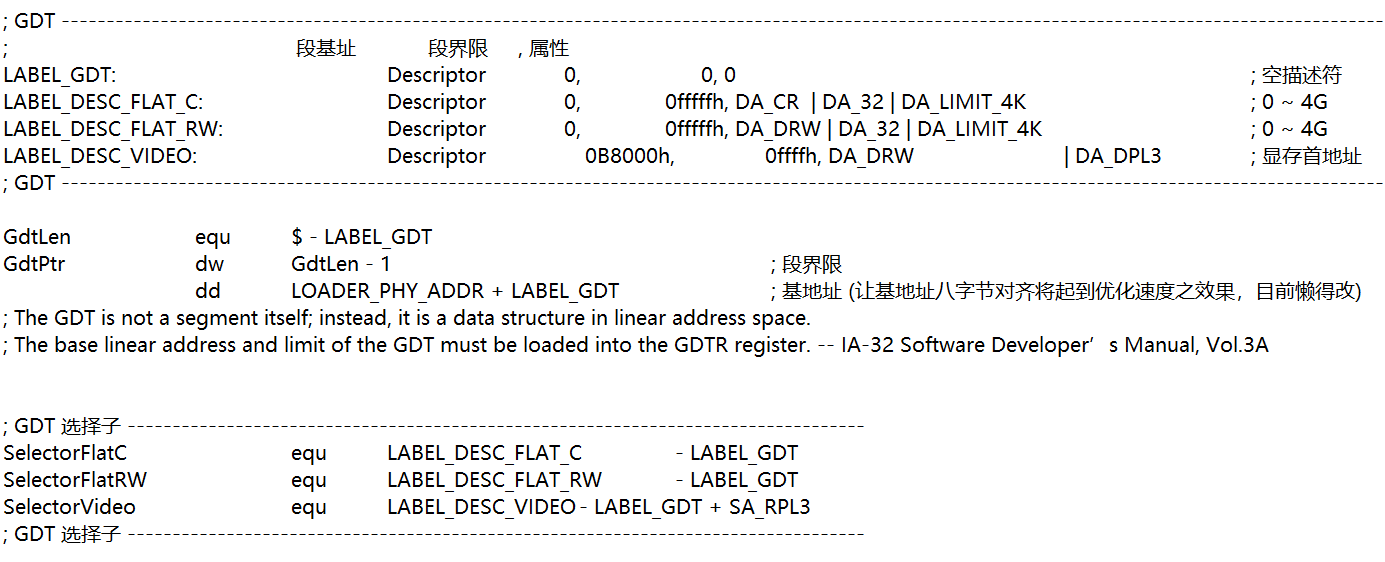
进行转换时，先是从由寄存器cr3指定的页目录中根据线性地址的高10位得到页表地址，然后在页表中根据线性地址的第12到21位得到物理页首地址，将这个首地址加上线性地址低12位便得到了物理地址。



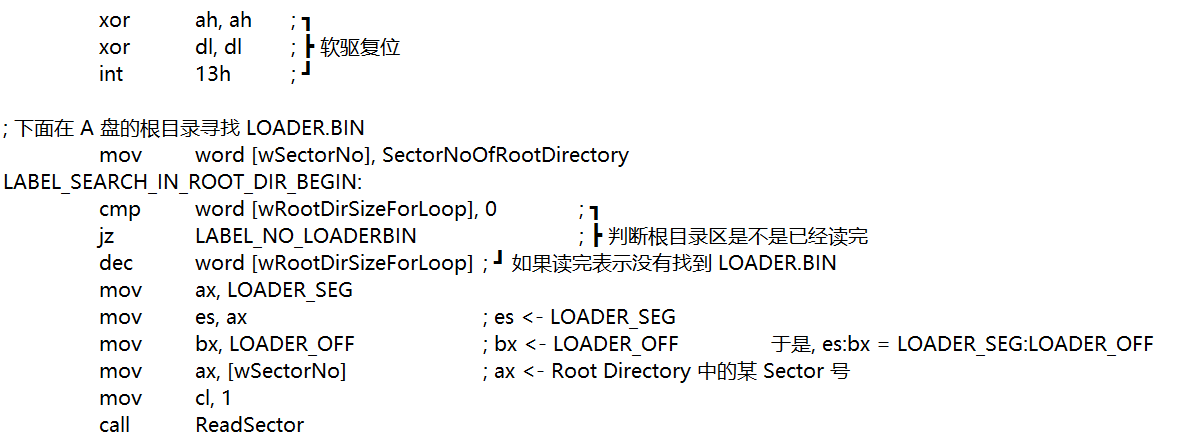
如图为启动分页机制的部分代码。PageDirBase和PageTblBase是两个宏，指定了页目录表和页表在内存中的位置。页目录表位于地址2MB处，有1024个表项，占用4KB空间，紧接着页目录表是页表，位于地址2MB+4KB处。在这里，假定最大的可能，共有1024个页表。每个页表占用4096字节，这些页表共占用4MB空间。

* 1. **Loader的编写和导入内存**

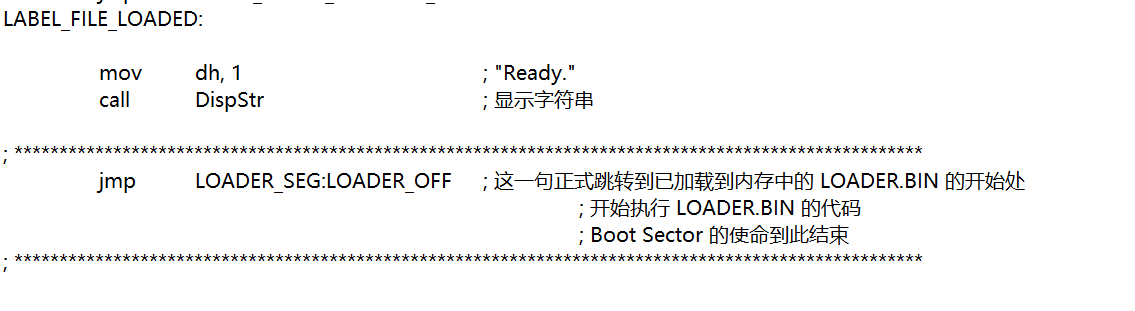
由于操作系统从开机到运行还要经过“引导→加载内核入内存→跳入保护模式→开始执行内核“等过程，但引导扇区只有512字节，因此，需要loader模块来完成这一系列工作。引导扇区负责把Loader加载入内存并交给它控制权，由loader来完成接下来的工作。



上图为loader实现的部分代码。我们先编写出loader.asm，然后修改boot.asm中的代码。



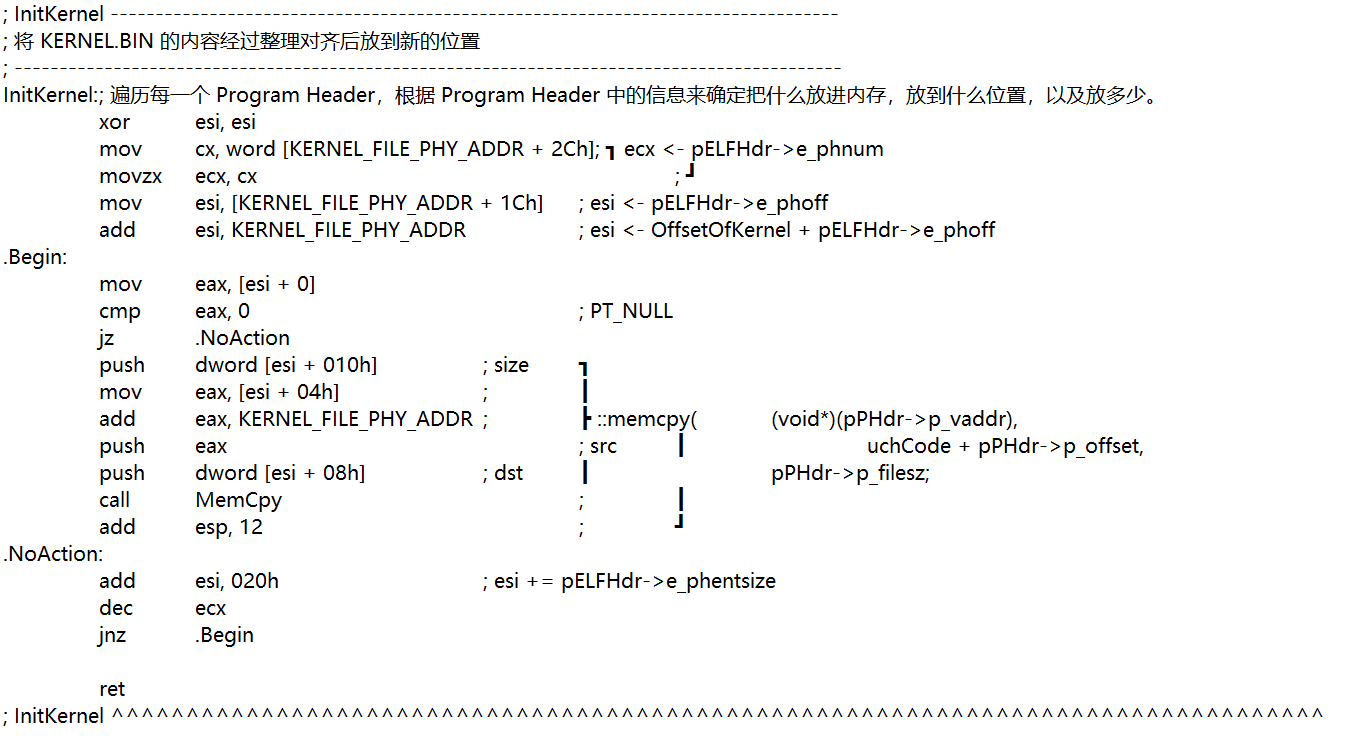
如图，在引导扇区执行后，遍历根目录区所有的扇区，将其加载入内存，并寻找loader.bin。之后我们能得到loader的起始扇区号。通过这个起始扇区号，我们可以先将该起始扇区装入内存，然后通过它找到FAT中的项，以此找到loader占用的其它扇区。



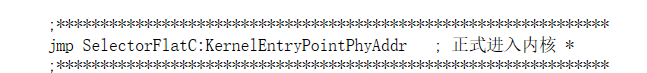
待loader装入内存后，需要将系统的控制权交给loader。调用jump进行跳转，此后开始执行loader的代码。

* 1. **内核雏形的形成**

在跳转进入loader过后，loader需要承担“加载内核到内存”以及“进入保护模式”的工作。首先，需要用Loader加载ELF，这部分与从引导扇区加载loader非常相似。接下来，我们需要调用前面编写的代码跳入保护模式。



我们编写一个简单的内核（kernel）并将kernel.bin加载进内存，使用的方法跟加载Loader.bin时相同，并通过函数将内核放置到一个合适的位置。



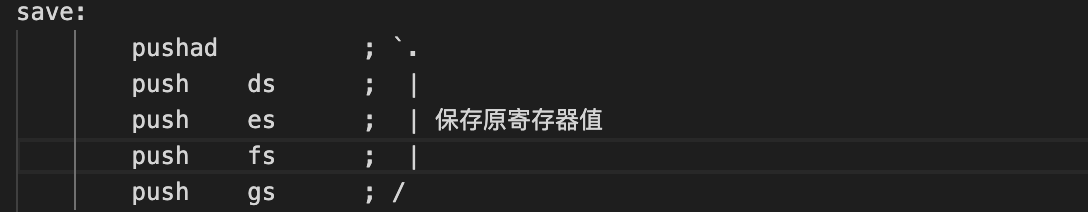
最后，调用jump命令，将系统控制权交给kernel（内核），操作系统正式开始运行。

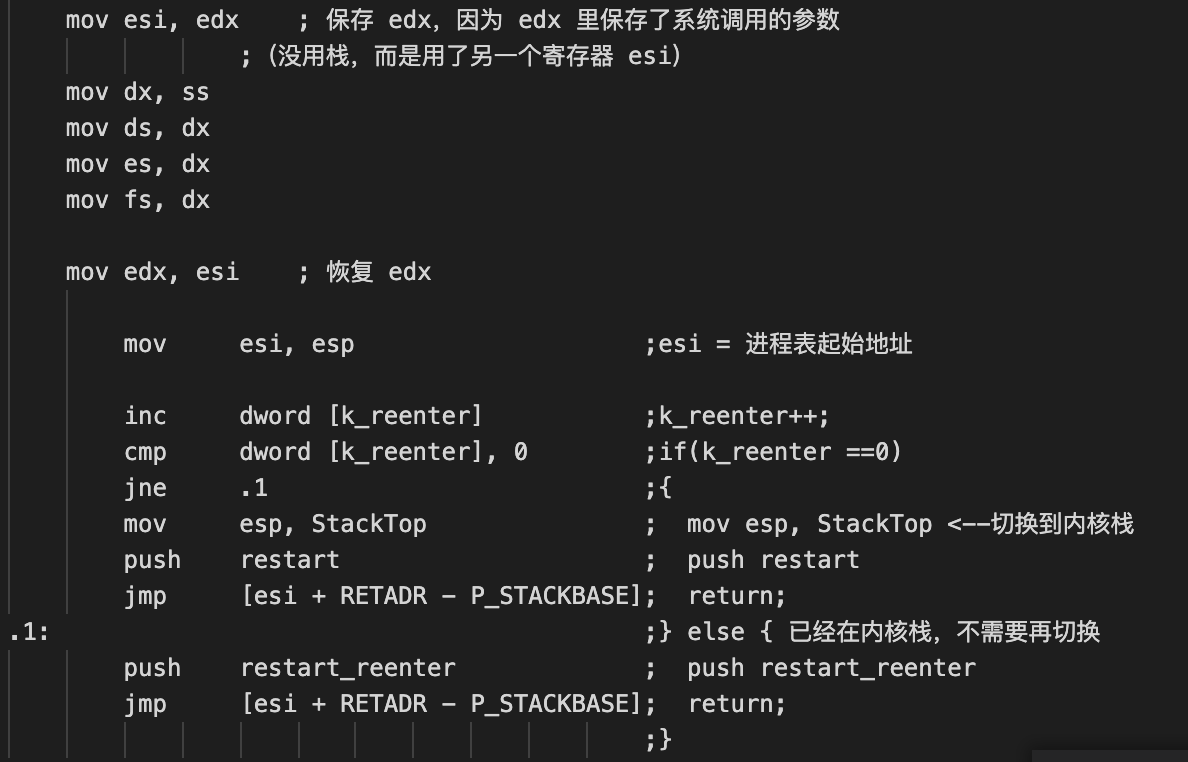
接下来，将对操作系统的内核进行扩充。

* 1. **加入中断处理的内核形成**

中断是进程实现中重要的一个环节。我们在使中断起作用的同时还要使中断进行情况可视化，参考书本，我们通过改变操作台第0行、第0列字符的方式来说明中断正在进行。

在保存原寄存器值之后， 到`mov esp, StackTop'之前，中间不能使用push或pop指令，因为当前的esp指向的是进程表内的某个位置，此时push会破坏掉进程表，导致灾难性的后果。所以时钟的中断处理实现代码如下





* + 1. **现场的保护与恢复**

前面提到，在进程的实现中，我们使用进程表来保存进程的状态，这使得程序在进行中断处理后能够更方便地恢复现场。上述代码中保存愿寄存器值即在进行现场的保护功能。

* + 1. **内核栈的使用**

上述“mov esp,StackTop”语句是为了实现钱还进程到内核栈的功能。若不切换内核栈，随着中断历程扩大，所出现的错误将难以被发现。

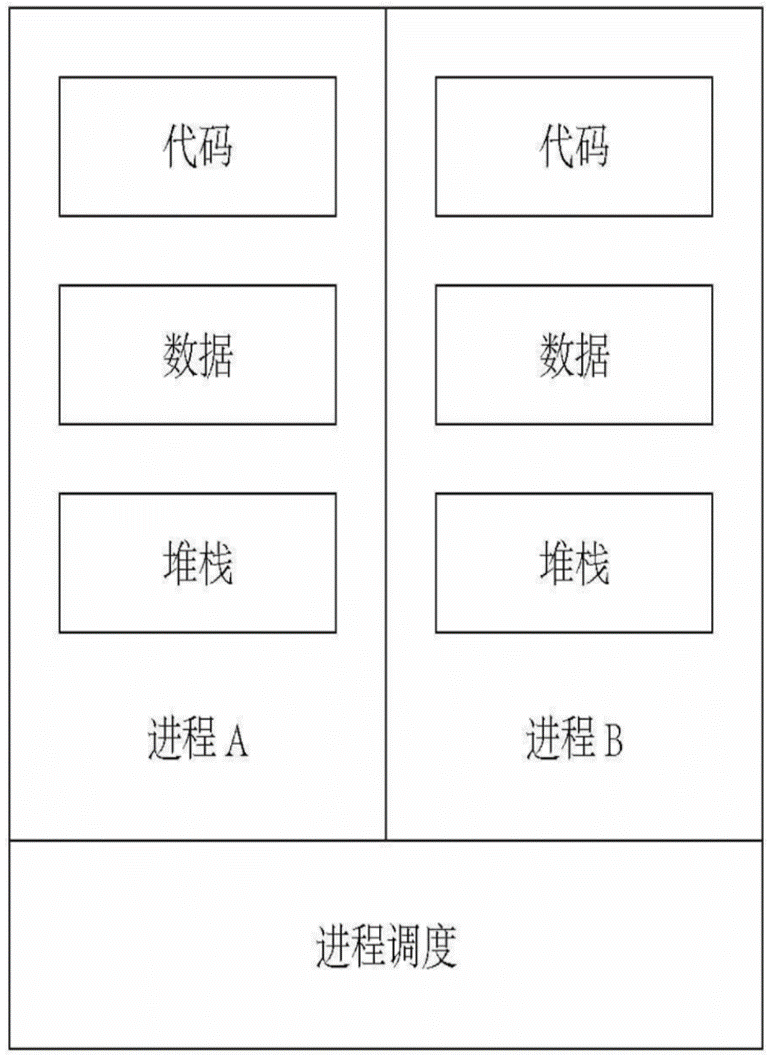
* + 1. **中断重入**

这个模块致力于解决中断嵌套问题，即在中断处理的过程中，是否允许另一个中断发生。在这之中核心点是中断处理的程序是被动的，它不管中断在什么时间发生，所以设置一个全局变量来使得中断知道自己是否处于嵌套执行状态，从而针对嵌套与否进行下一步不同的处理。

* + 1. **多进程中断实现**

实现多进程中断的核心技术点在于定位所要恢复的进程，实现这一点只需要将esp指向目的进程表即可，即给指向进程表结构的执政赋值。

* 1. **加入单进程的内核形成**
     1. **进程的理解**



在对进程的处理过程中，我们首先需要考虑单进程的情况，即两个进程之间进行简单的切换。进程切换需要经过的步骤为：

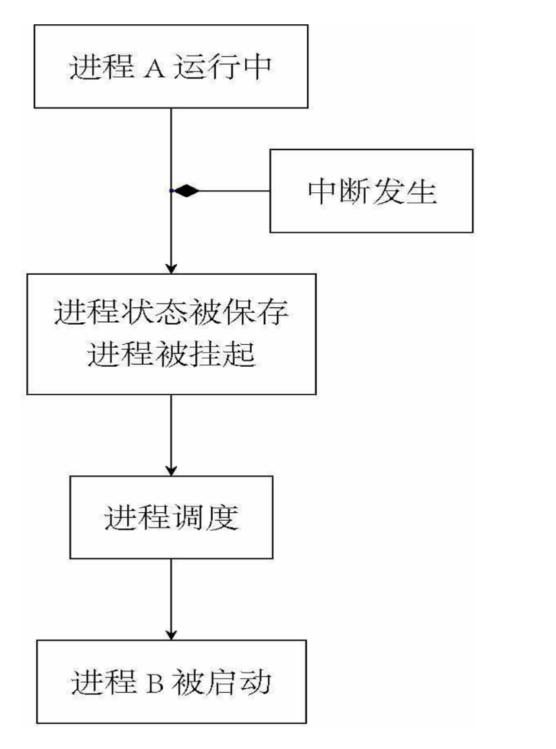
1. 进程A运行中。

2. 时钟中断发生，ring1→ring0，时钟中断处理程序启动。

3. 进程调度，下一个应运行的进程（假设为进程B）被指定。

4. 进程B被恢复，ring0→ring1。

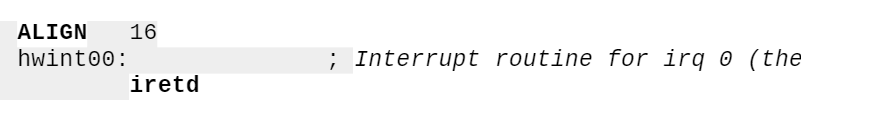
5. 进程B运行中。

****

* + 1. **实现简单的进程**

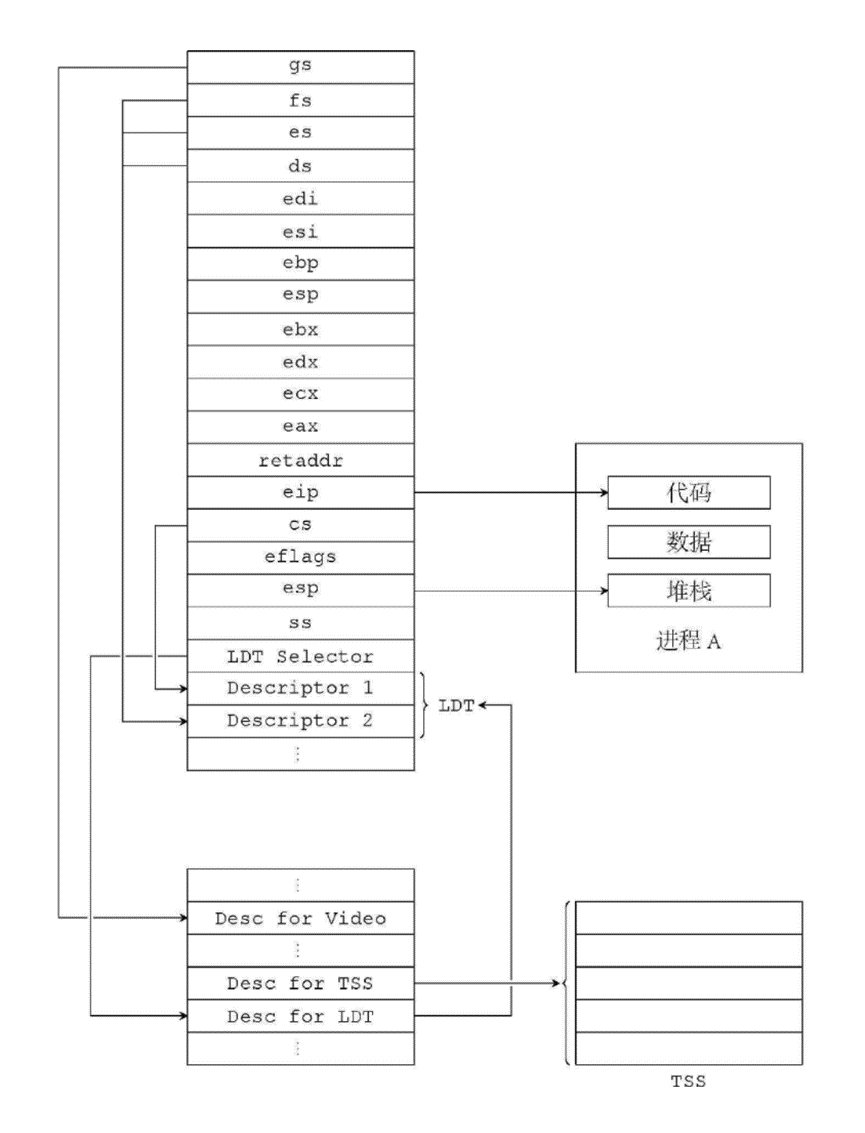
关键技术包括：保存CPU中寄存器的值 (采用pushad 指令)，执行 pop 及 iretd指令回到进程B，引入进程表描述进程 (独立于进程之外)，注意进程栈和内核栈的问题（考虑esp指向何处），以及ring0和ring1之间的特权级变换。

时钟中断处理程序：

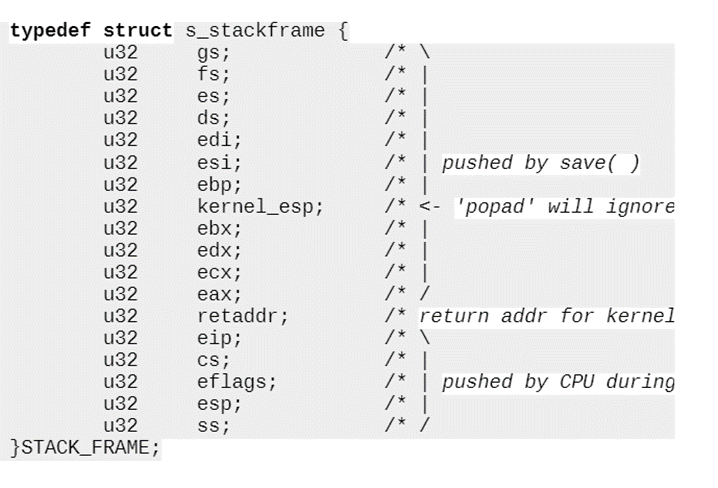


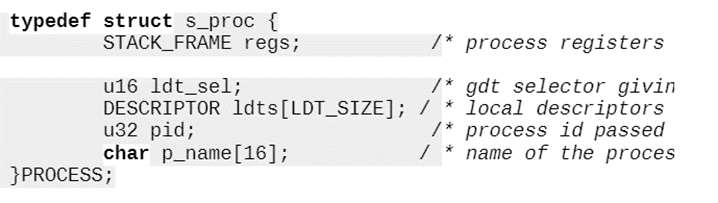
初始化进程表、进程体、GDT、TSS：

其中进程表及相关数据结构对应关系如图所示：

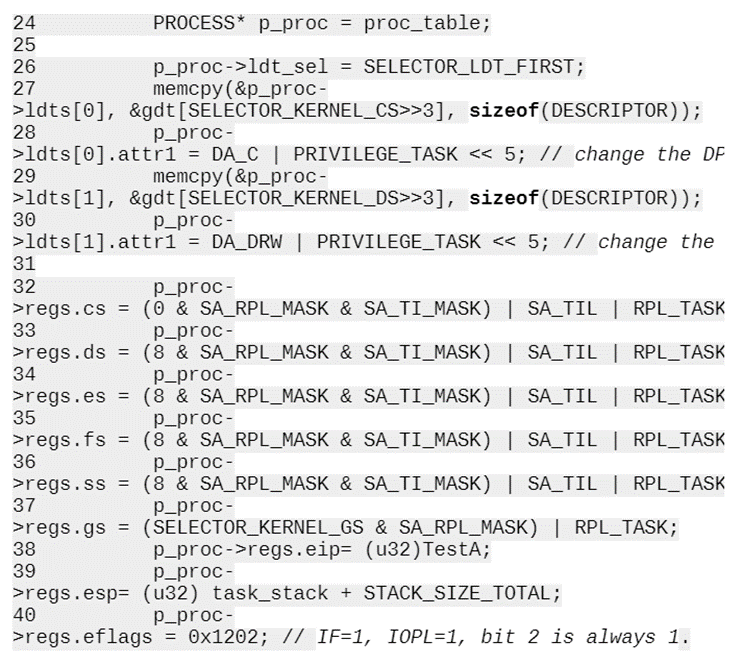


进程表结构体定义：

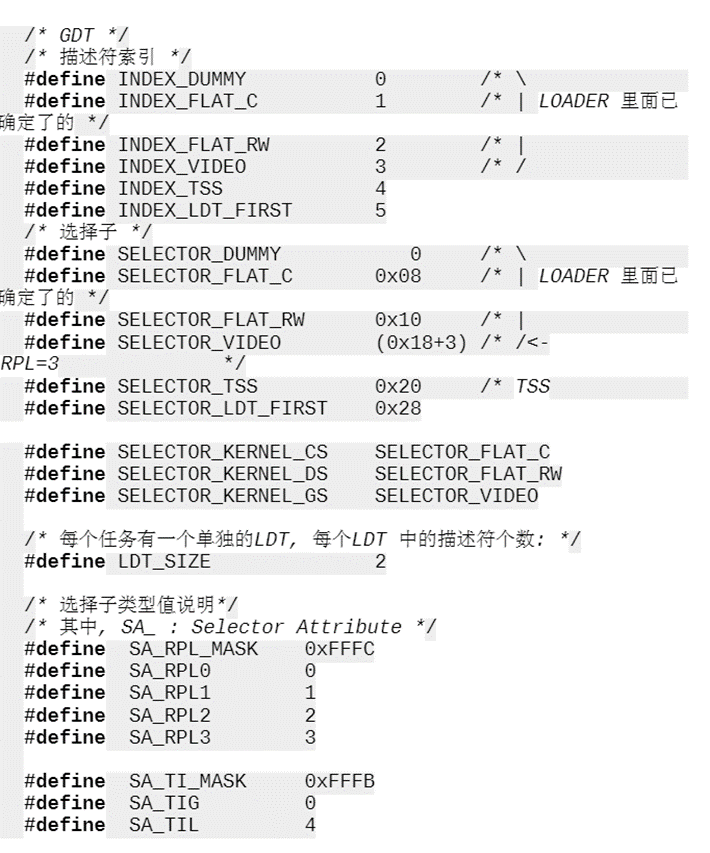




初始化进程表：

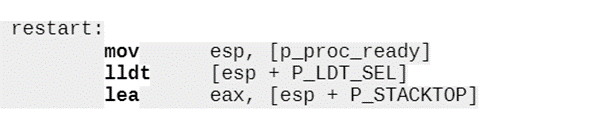


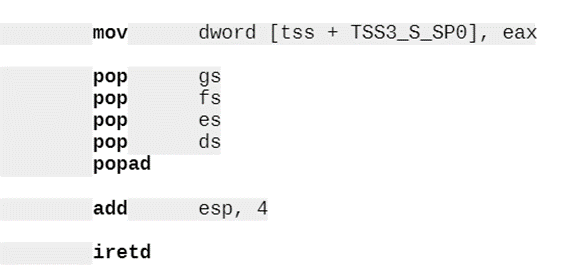
宏定义：



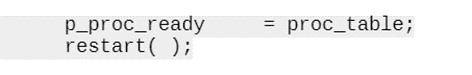
* + - 1. **iretd**

简化版restart函数：





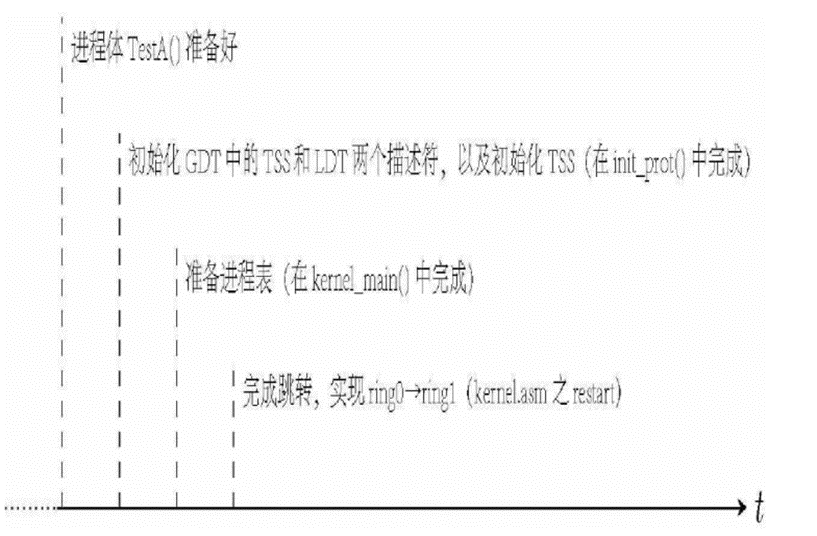
在此之后，所有的准备工作均已完成，需要在kernel.main函数中添加代码：



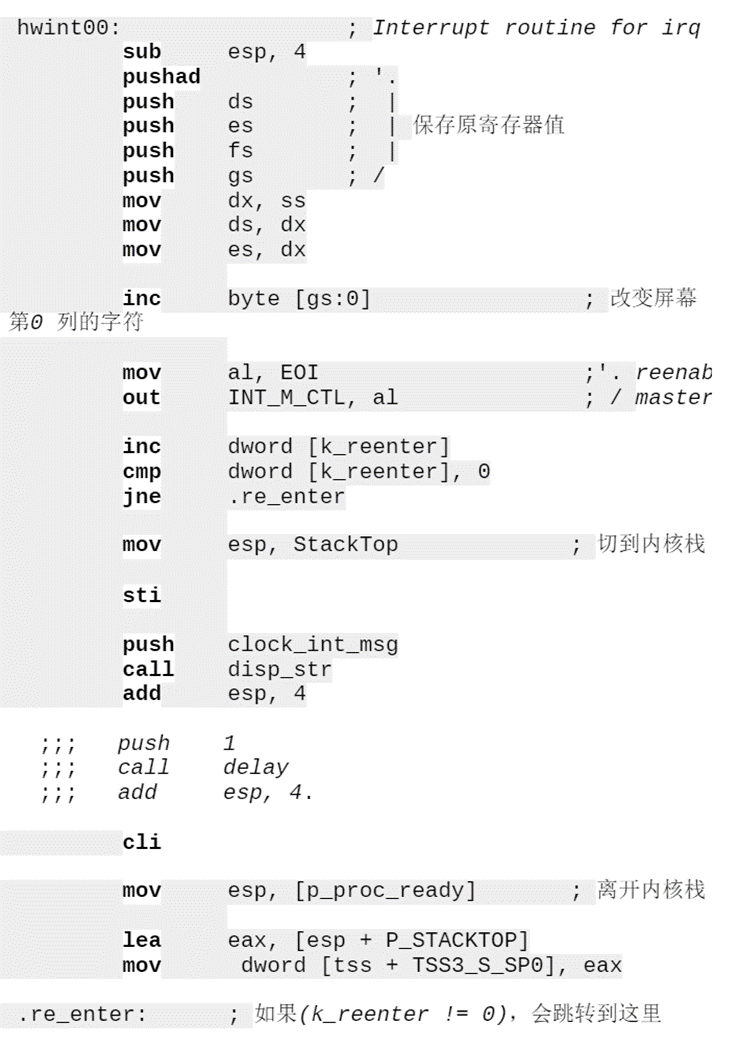
make，运行，单进程执行成功。

* + 1. **单进程启动过程**

在此图中，进程体TestA()在内核被LOADER放置到内存中之后就准备好了。



在此之后，打开时钟中断程序，使得时钟中断程序并不是只发生一次，并依次实现如下功能：现场的保护与恢复，赋值tss.esp0，内核栈，解决中断重入问题，部分代码如下图所示:

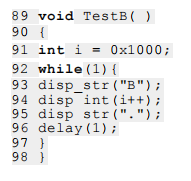


* 1. **多进程**
     1. **多进程的理解**

在单一进程的实现基础上，我们可以引入多进程。在单一进程中，进程已经拥有了两种状态：运行和睡眠。由此不难想到，我们已经具备了处理多个进程的能力，只需要让其中一个进程处在运行态，其余进程处在睡眠态就可以了。

* + 1. **添加进程体**

核心代码如下所示

****

此处i的初始值被设成了0x1000，为的就是在将来程序运行时能清晰地分辨两个进程。

* + 1. **多进程涉及的宏与变量**

考虑进程本身，由于进程具有4个要素：进程表、进程体、GDT、TSS。接下来要关注的问题是进程表如何初始化。

方法一：参考第一个进程的初始化操作

将其中的几个关键成员赋值就可以了。可以将这份代码复制一份，将其中涉及进程 A的内容统统改成与进程B相关的代码。可是显然那样有些麻烦与复杂，因为不可能每增加一个进程就复制一份代码，最好能够让代码在某种程度上实现一点自动化，让增加一个进程变得简单而迅速。

方法二：

建立一个数组，将此数组的每一项定义好一个进程的开始地址、堆栈等，在初始化的时候，只要用一个for循环依次读取每一项，然后填充到相应的进程表项中就可以了。

**3.8.4LDT**

每一个进程都会在GDT中对应一个LDT描述符。于是在for循环中，我们将每个进程表项中的成员p\_proc- >ldt\_sel赋值。可是，选择子仅仅是解决了where问题，通过它，我们能在GDT中找到相应的描述符，但描述符的具体内容是什么， 即what问题还没有解决。参照一个进程时的操作，将其扩展为循环即可。

**3.8.5中断处理程序**

运行后发现没有改变，检查发现每次发生时钟中断的时候，被恢复的进程还 是原来的进程，我们还没有编写任何进程切换代码。

时钟中断处理程序位于kernel.asm中，除了保存和恢复进程信息，我们只做了一件简单的事，就是在屏幕上打印一个字 符“^”。显然，进程切换的代码就应该添加在这个位置才对。

考虑一个进程由“睡眠”状态变成“运行”状态时，即是将esp指向进程表项的开始处，然后在执行lldt之后经历 一系列pop指令恢复各个寄存器的值。一切信息都包含在进程表中，所以，要想恢复不同的进程，只需要将esp指向不同的进程表就 可以了。

在离开内核栈的时候，有一个语句是为esp赋值的：

mov esp, [p\_proc\_ready] 全局变量p\_proc\_ready是指向进程表结构的指针，我们只需要在这一句执行之前把它赋予不同的值就可以了。

如此可见，进程切换是一个稍稍有点复杂的过程，因为涉及进程调度等内容。一方面，这涉及算法等一些复杂的内容，另一方

面，它应该是与硬件无关的，所以我们用C语言编写这个模块。

**3.8.6 总结**

1. 在task\_table中增加一项（global.c）。

2. 让NR\_TASKS加1（proc.h）。

3. 定义任务堆栈（proc.h）。

4. 修改STACK\_SIZE\_TOTAL（proc.h）。

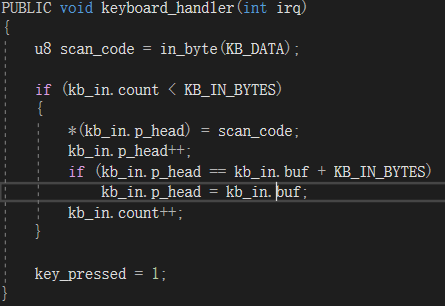
5. 添加新任务执行体的函数声明（proto.h）。

* 1. **加入输入输出系统的内核**

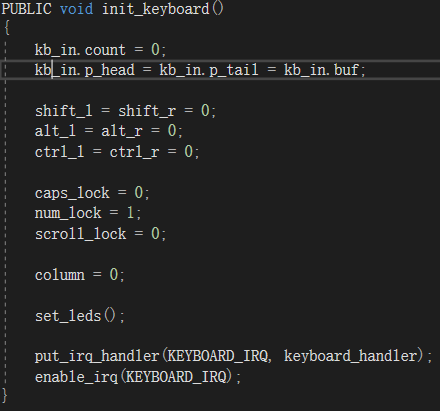
交互也是操作系统中很重要的一点，目前为止，我们的操作系统还并不能实现输入输出的交互。因此我们首先从键盘这一重要的交互方式开始。

* + 1. **键盘**

敲击键盘有两方面的含义：动作和内容。其中动作包括三类，分别是按下、保持按住的状态以及松开；内容则为键盘上的不同按键，功能键或者数字键、回车键等等。

敲击键盘所产生的编码被称作扫描码（Scan Code），它分为Make Code和Break Code两类。当一个键被按下或者保持住按下时，将会产生Make Code，当键弹起时，产生Break Code。除了Pause键之外，每一个按键都对应一个Make Code和一个Break Code。我们建立一个缓冲区，让keyboard\_handler将每次收到的扫描码放入这个缓冲区，然后建立一个新的任务专门用来解析它们并做相应处理。 

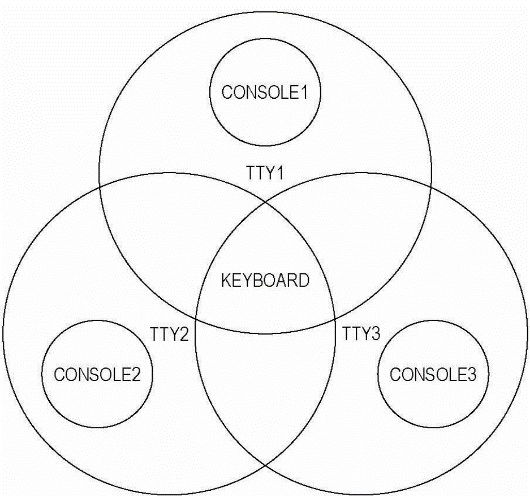
其中kb\_in需要初始化如下：



为处理键盘输入，我们需要不停调用函数keyboard\_read，函数首先判断kb\_in.count是否为0，如不为0，表明缓冲区中有扫描码，就开始读。读缓冲区开始时关闭了中断，到结束时才打开，因为kb\_in作为一个整体，对其中的成员的操作应该是一气呵成不受打扰的。读完后我们依照keymap对扫描码进行解析。

* + 1. **显示器**

随着键盘模块的逐渐完善，我们越来越需要考虑它与屏幕输出之间的关系。下面是对TTY的一种理解：



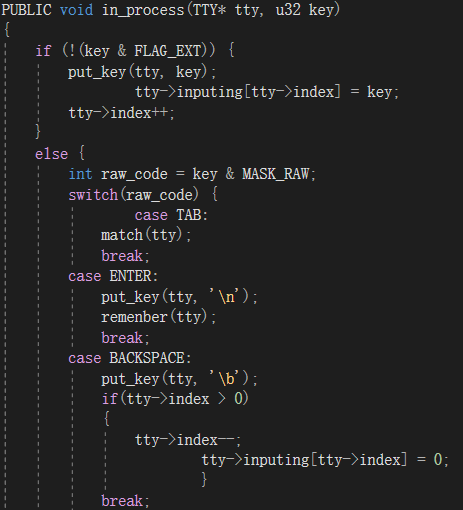
虽然不同的TTY对应的输入设备是同一个键盘，但输出却好比是在

不同的显示器上，因为不同的TTY对应的屏幕画面可能是迥然不同的。

虽然不同的TTY对应的输入设备是同一个键盘，但输出却好比是在

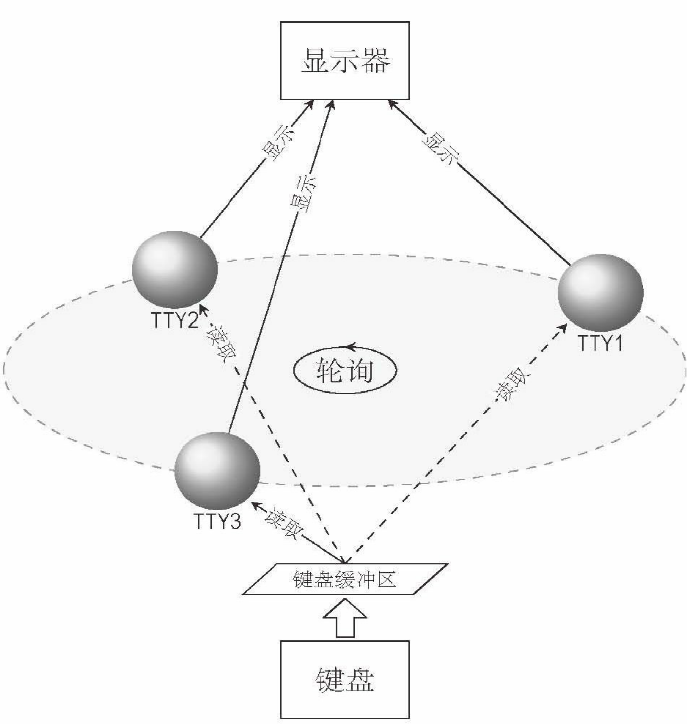
不同的显示器上，因为不同的TTY对应的屏幕画面可能是迥然不同的。

虽然不同的TTY对应的输入设备是同一个键盘，但输出却好比是在不同的显示器上，因为不同的TTY对应的屏幕画面可能是迥然不同的。因为可能显示器所显示的是显存的不同位置。而3个CONSOLE公用同一块显存，我们需要通过端口操作，使得在切换CONSOLE的瞬间，让屏幕显示显存中某个位置的内容。下为部分代码：

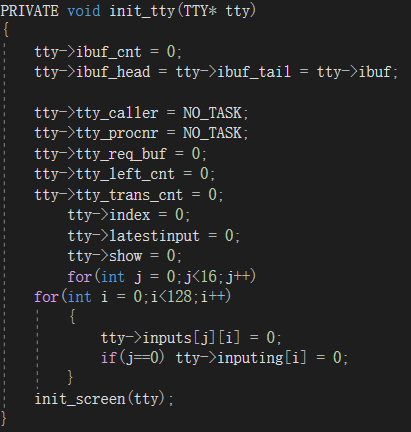


* + 1. **TTY任务**

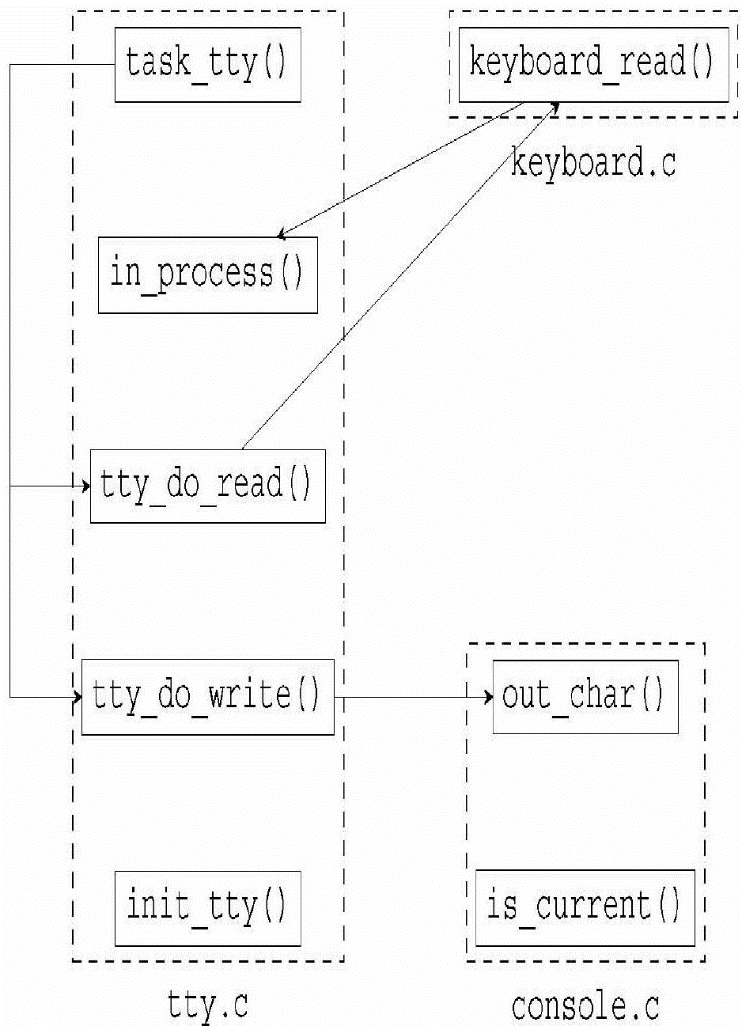
我们需要在TTY任务中执行一个循环，这个循环将轮询每一个TTY，处理它的事件，包括从键盘缓冲区读取数据、显示字符等内容。



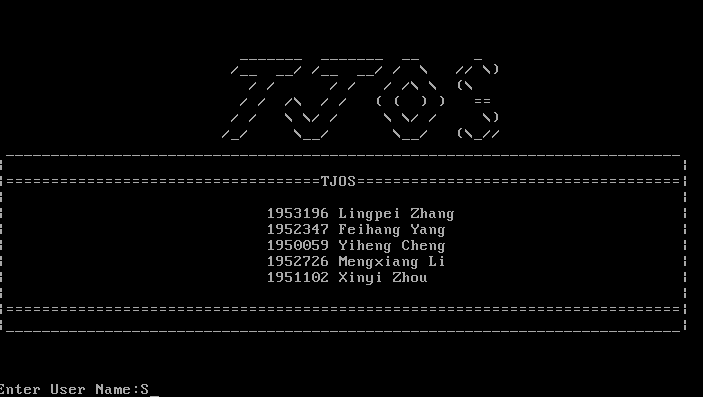
在进行了框架性的task\_tty()后，我们需要进行一些初始化操作，调用了init\_tty函数：

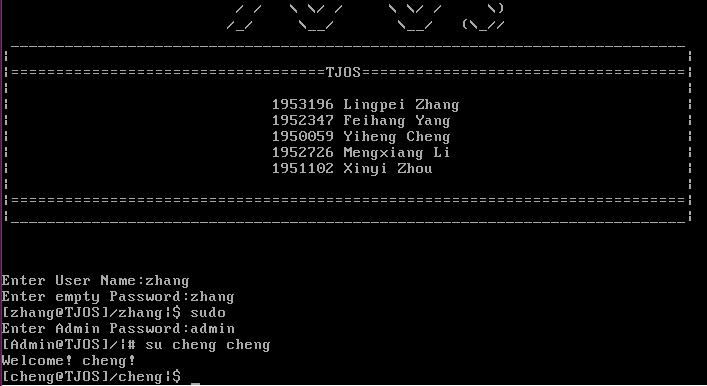


随后，我们再根据如下图所示的流程，为每个TTY任务指定一个CONSOLE，其中需要判断每个TTY任务所指定的CONSOLE是否为当前的控制台，若是则可以进行输出。



1. **成果**
   1. **用户登录及权限管理**





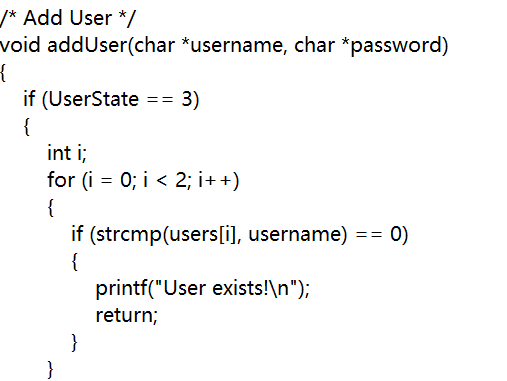
如图，虚拟机启动后，操作系统开始运行，会要求用户进行登录。

当前设置了zhang和cheng两个用户（账号和密码相同），为普通用户。在用户登录成功后，可以输入‘sudo’指令切换到管理员用户，密码为admin。

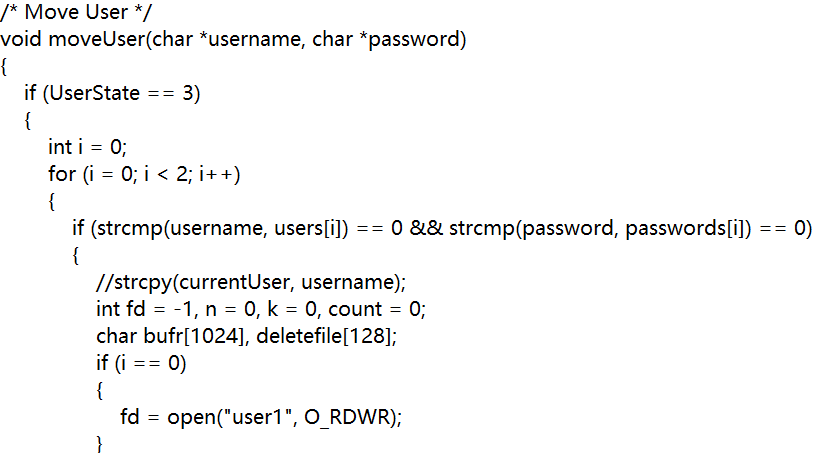
管理员拥有管理普通用户的权限，包括创建用户，删除用户，修改用户信息等。

通过输入指令‘su’可以切换用户。

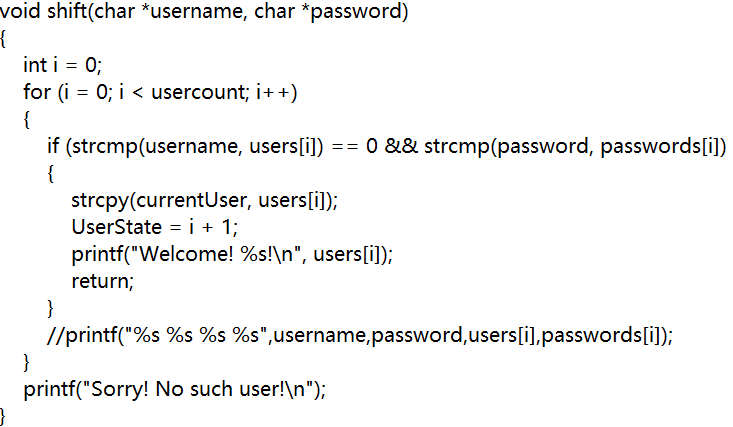
以上功能的具体实现代码如下：



管理员添加用户，调用addUser函数



管理员删除用户，调用moveUser函数



切换用户，调用shift函数

* 1. **文件系统**

在登录zhang这个用户之后，我们用ls命令可以查看当前目录下的所有文件及文件夹（这里zhang用户下有一些初始文件和文件夹）。

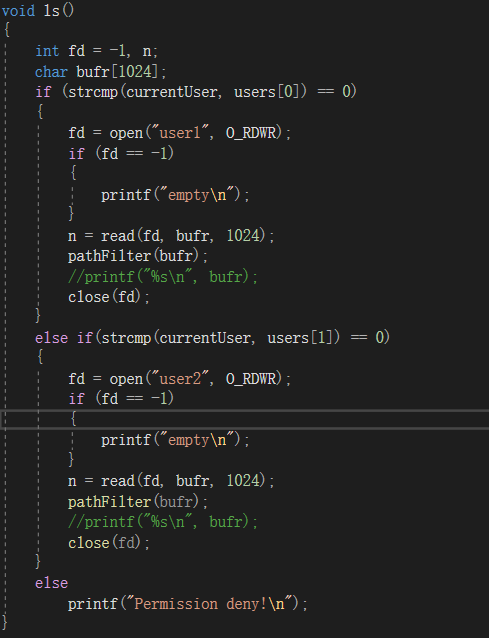


在根目录下，我们可以新建文件或文件夹。这里新建一个文件夹，并进入该文件夹。在该文件夹目录中，我们可以新建文件并编辑文件内容。

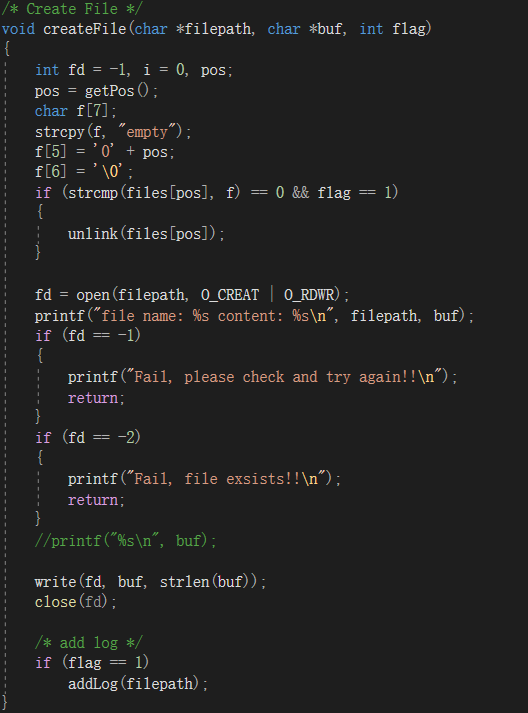


我们可以用rd命令查看文件内容，或者用wt命令对文件进行重新编辑，可以看到使用wt+命令时是在文件原本内容后添加内容，也即append的方式。此外，我们可以用del命令删除文件。

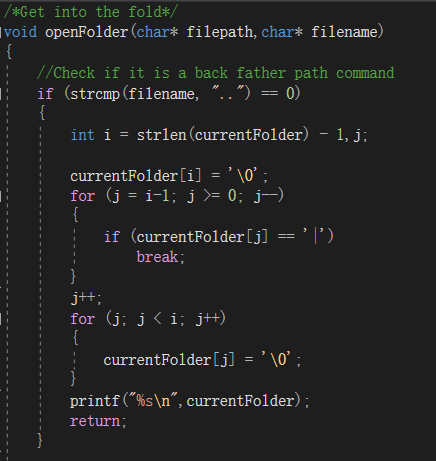
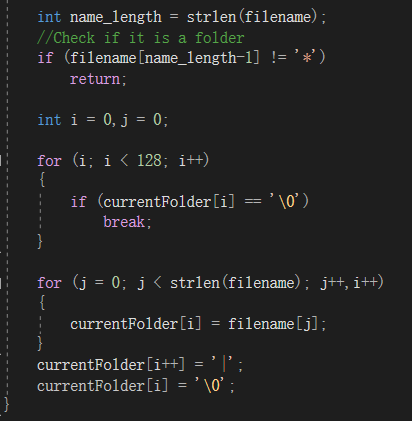
上述功能的具体实现代码如下：



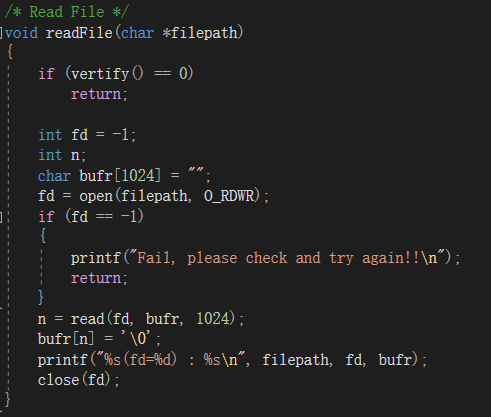
查看当前目录下所有文件及文件夹，调用ls函数



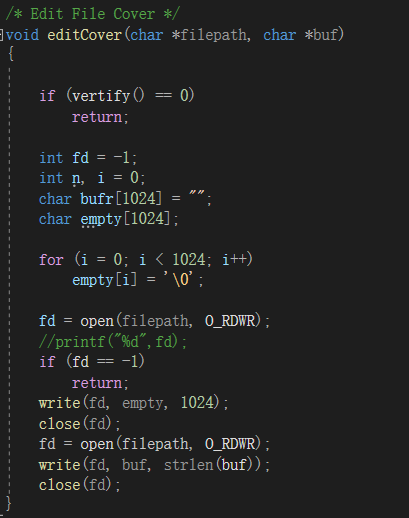
在根目录下新建文件或文件夹，调用了CreateFile函数（CreateFolder函数类似，不再贴出）

打开某一文件夹，调用openFolder函数



查看某一文件的内容，调用readFile函数



编辑文件内容（覆盖与不覆盖类似）

* 1. **进程管理**

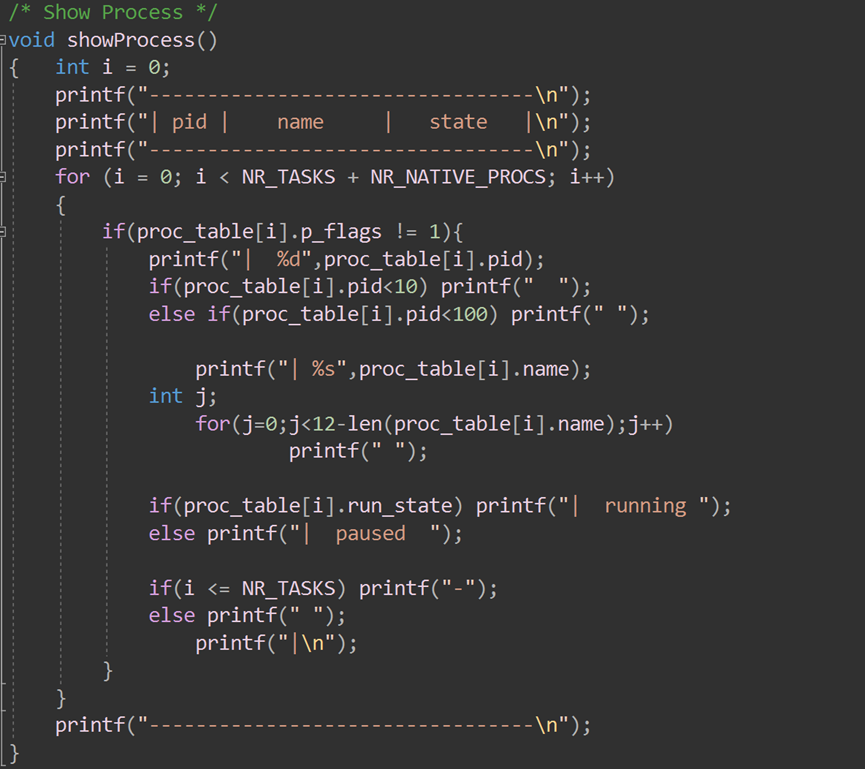
在进程管理功能中，主要包括显示正在运行的所有进程、以及对用户进程的操作。其中，对用户进程的操作包括，终止进程，暂停进程，唤醒进程这三个功能点（这里将进程分为系统进程【系统进程不可操作】和用户进程【用户进程可以操作】）。

* + 1. **显示正在运行的所有进程**

显示正在运行的所有进程功能，需要用户先进行登录，在此处示例登入用户Zhang，输入指令proc，即可显示正在运行的所有进程。



其在kernel/main.c中的代码实现如下，功能对应的函数名为showProcess()。

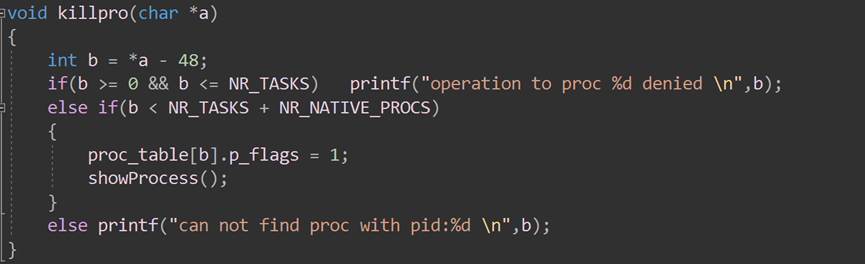


* + 1. **终止进程**

终止进程功能，输入指令kill+当前需要被终止进程的pid，即可终止进程。

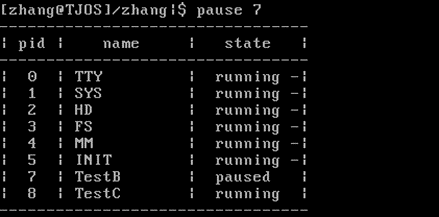


其在kernel/main.c中的代码实现如下，功能对应的函数名为killpro()。在修改完相关属性后，重新调用showProcess()函数，显示正在运行的所有进程。

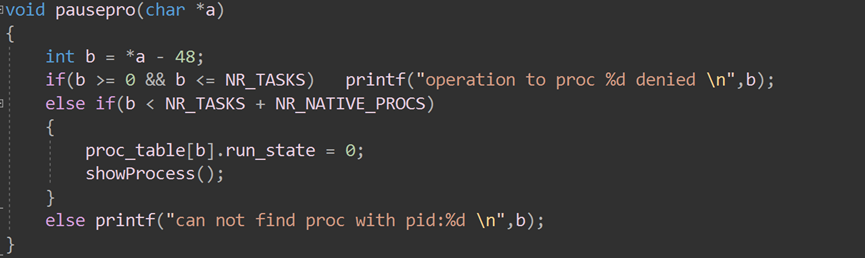


* + 1. **暂停进程**

暂停进程功能，输入指令pause+当前需要被暂停进程的pid，即可暂停进程。

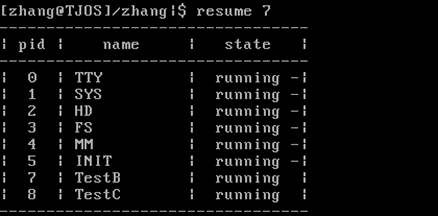


其在kernel/main.c中的代码实现如下，功能对应的函数名为pausepro()。在修改完相关属性后，重新调用showProcess()函数，显示正在运行的所有进程。

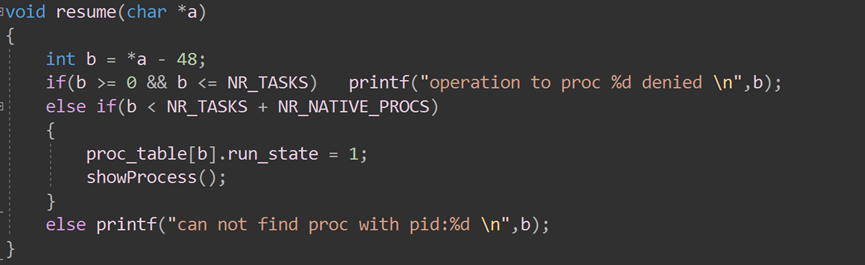


* + 1. **唤醒进程**

唤醒进程功能，输入指令pause+当前需要被唤醒进程的pid，即可唤醒进程。



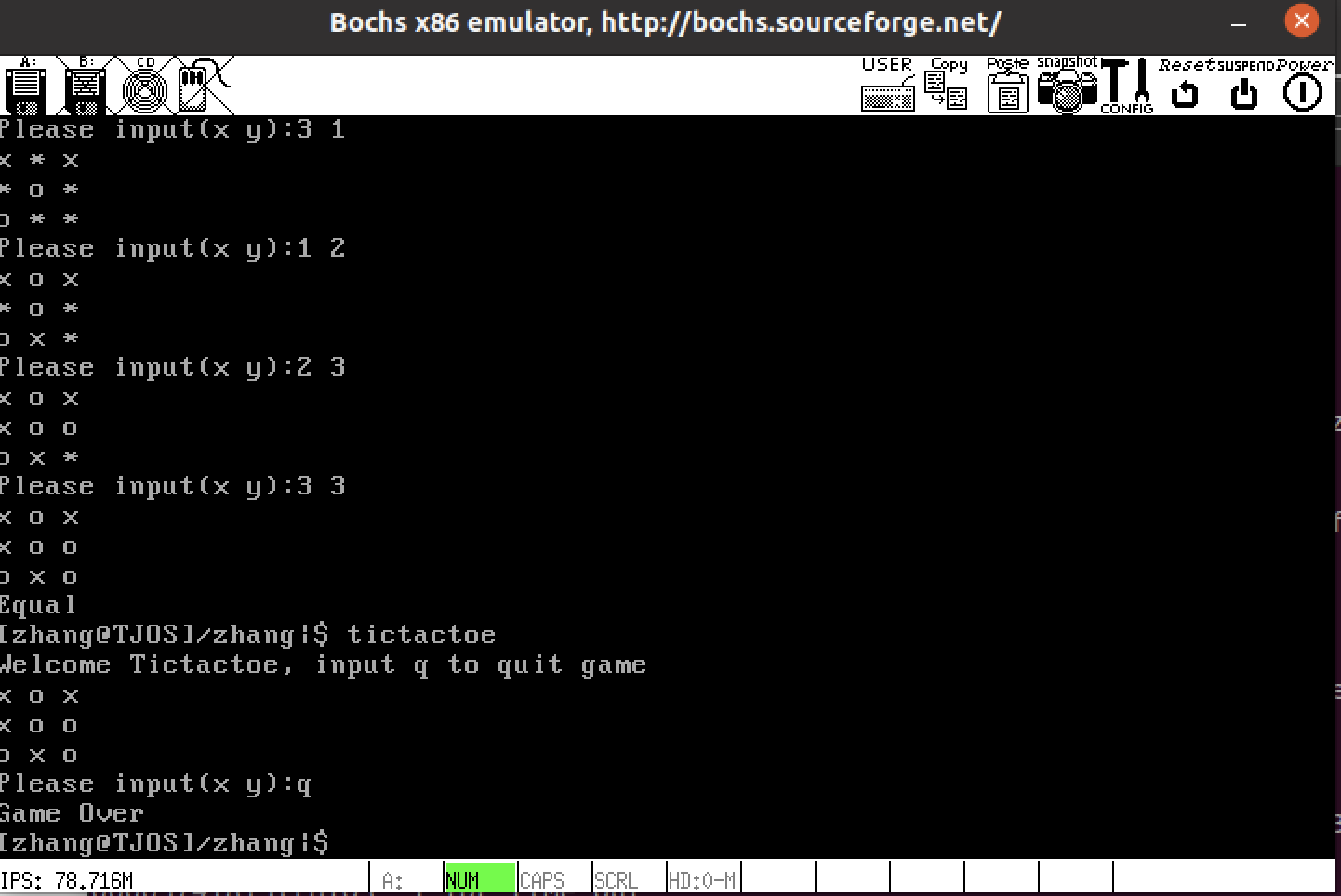
其在kernel/main.c中的代码实现如下，功能对应的函数名为resume()。在修改完相关属性后，重新调用showProcess()函数，显示正在运行的所有进程。



* 1. **应用开发**

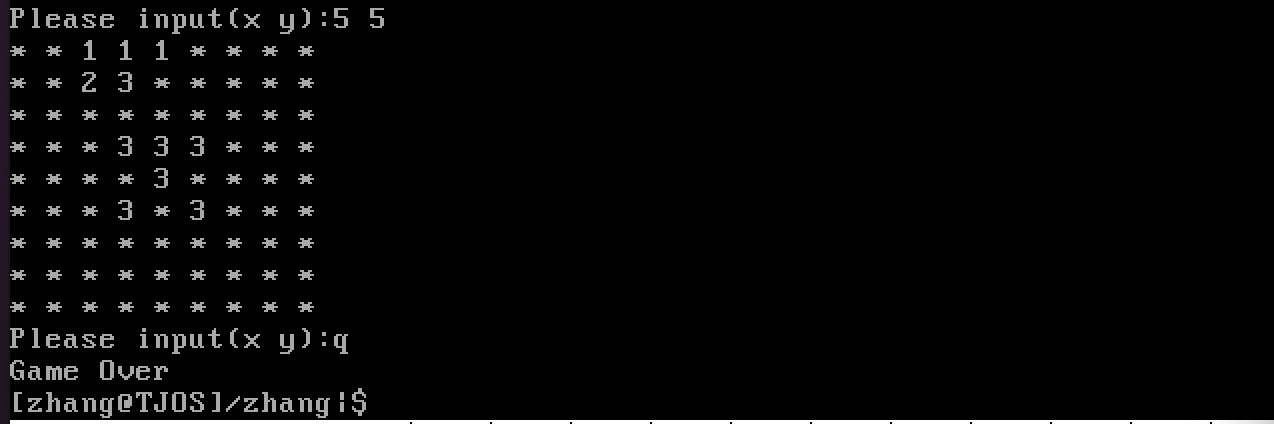
用户登入后，可对系统内置应用进行使用，本系统中有井字棋、扫雷两款应用。

使用命令行tictactoe启动游戏井字棋后，首先显示出棋盘界面，然后在命令行输入坐标命令`x y`(x,y范围为1到3,用空格隔开)落子。每一次玩家落子后，电脑AI会跟一颗子，直到有一方获胜或棋盘下满。在游戏过程中玩家可以通过输入`q`来退出游戏。



使用命令行boom启动游戏扫雷后，首先会展示出当前的游戏界面，然后用户可以输入坐标命令`x y`(x,y范围为1到9,用空格隔开)来揭开该坐标及其周围无雷的区域。若该坐标有雷，则游戏结束，若玩家最终揭开全部无雷区域，则玩家获胜。在游戏过程中玩家可以通过输入`q`来退出游戏。





上述功能的核心实现代码如下

井字棋实现代码

#include<stdio.h>

#define SIZE 3

char scane[SIZE][SIZE]={'\*','\*','\*','\*','\*','\*','\*','\*','\*'};

int grade[SIZE][SIZE]={50,10,50,10,100,10,50,10,50};

void AI\_input(int x,int y);

void display();

int check();

int main\_tic()

{

printf("Welcome Tictactoe, input q to quit game\n");

int result=0;

display();

for(int i=0;i<5;i++)

{

printf("Please input(x y):");

char input[4];

for(int i=0;i<4;i++)

{

input[i]=0;

}

char x,y;

int r=read(0,input,4);

input[r]=0;

x=input[0]-48;

y=input[2]-48;

if(x+48=='q'){

printf("Game Over\n");

return 0;

}

while(1){

if(x>0&&x<4&&y>0&&y<4&&(grade[x][y]%10==0)) break;

else {

printf("Error,input again:");

int r=read(0,input,4);

input[r]=0;

x=input[0]-48;

y=input[2]-48;

if(x+48=='q'){

printf("Game Over\n");

return 0;

}

}

}

scane[x-1][y-1]='o';

grade[x-1][y-1]=4;

if(check()==1){

printf("You win\n");

result=1;

break;

}

if(i!=4) AI\_input(x-1,y-1);

if(check()==2){

printf("You lose\n");

result=1;

break;

}

display();

}

if(result==0) printf("Equal\n");

return 0;

}

void display()

{

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

{

for(int j=0;j<SIZE;j++)

{

printf("%c ",scane[i][j]);

}

printf("\n");

}

}

void AI\_input(int x,int y)

{

int grade\_sum[8];

grade\_sum[0]=grade[0][0]+grade[0][1]+grade[0][2];

grade\_sum[1]=grade[1][0]+grade[1][1]+grade[1][2];

grade\_sum[2]=grade[2][0]+grade[2][1]+grade[2][2];

grade\_sum[3]=grade[0][0]+grade[1][0]+grade[2][0];

grade\_sum[4]=grade[0][1]+grade[1][1]+grade[2][1];

grade\_sum[5]=grade[0][2]+grade[1][2]+grade[2][2];

grade\_sum[6]=grade[0][0]+grade[1][1]+grade[2][2];

grade\_sum[7]=grade[0][2]+grade[1][1]+grade[2][0];

for(int i=0;i<8;i++){

if(grade\_sum[i]%10==8){

if(i==0){

if(grade[0][0]%10==0){

grade[0][0]=2;

scane[0][0]='x';

}

else if(grade[0][1]%10==0){

grade[0][1]=2;

scane[0][1]='x';

}

else if(grade[0][2]%10==0){

grade[0][2]=2;

scane[0][2]='x';

}

}

else if(i==1){

if(grade[1][0]%10==0){

grade[1][0]=1;

scane[1][0]='x';

}

else if(grade[1][1]%10==0){

grade[1][1]=1;

scane[1][1]='x';

}

else if(grade[1][2]%10==0){

grade[1][2]=1;

scane[1][2]='x';

}

}

else if(i==2){

if(grade[2][0]%10==0){

grade[2][0]=1;

scane[2][0]='x';

}

else if(grade[2][1]%10==0){

grade[2][1]=1;

scane[2][1]='x';

}

else if(grade[2][2]%10==0){

grade[2][2]=1;

scane[2][2]='x';

}

}

else if(i==3){

if(grade[0][0]%10==0){

grade[0][0]=1;

scane[0][0]='x';

}

else if(grade[1][0]%10==0){

grade[1][0]=1;

scane[1][0]='x';

}

else if(grade[2][0]%10==0){

grade[2][0]=1;

scane[2][0]='x';

}

}

else if(i==4){

if(grade[0][1]%10==0){

grade[0][1]=1;

scane[0][1]='x';

}

else if(grade[1][1]%10==0){

grade[1][1]=1;

scane[1][1]='x';

}

else if(grade[2][1]%10==0){

grade[2][1]=1;

scane[2][1]='x';

}

}

else if(i==5){

if(grade[0][2]%10==0){

grade[0][2]=1;

scane[0][2]='x';

}

else if(grade[1][2]%10==0){

grade[1][2]=1;

scane[1][2]='x';

}

else if(grade[2][2]%10==0){

grade[2][2]=1;

scane[2][2]='x';

}

}

else if(i==6){

if(grade[0][0]%10==0){

grade[0][0]=1;

scane[0][0]='x';

}

else if(grade[1][1]%10==0){

grade[1][1]=1;

scane[1][1]='x';

}

else if(grade[2][2]%10==0){

grade[2][2]=1;

scane[2][2]='x';

}

}

else if(i==7){

if(grade[0][2]%10==0){

grade[0][2]=1;

scane[0][2]='x';

}

else if(grade[1][1]%10==0){

grade[1][1]=1;

scane[1][1]='x';

}

else if(grade[2][0]%10==0){

grade[2][0]=1;

scane[2][0]='x';

}

}

return;

}

}

int max=0;

int max\_x;

int max\_y;

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

{

for(int j=0;j<SIZE;j++)

{

if(grade[i][j]>max){

max=grade[i][j];

max\_x=i;

max\_y=j;

}

}

}

grade[max\_x][max\_y]=1;

scane[max\_x][max\_y]='x';

}

int check()

{

int grade\_sum[8];

grade\_sum[0]=grade[0][0]+grade[0][1]+grade[0][2];

grade\_sum[1]=grade[1][0]+grade[1][1]+grade[1][2];

grade\_sum[2]=grade[2][0]+grade[2][1]+grade[2][2];

grade\_sum[3]=grade[0][0]+grade[1][0]+grade[2][0];

grade\_sum[4]=grade[0][1]+grade[1][1]+grade[2][1];

grade\_sum[5]=grade[0][2]+grade[1][2]+grade[2][2];

grade\_sum[6]=grade[0][0]+grade[1][1]+grade[2][2];

grade\_sum[7]=grade[0][2]+grade[1][1]+grade[2][0];

for(int i=0;i<8;i++)

{

if(grade\_sum[i]==12) return 1;

else if(grade\_sum[i]==3) return 2;

else return 0;

}

}

扫雷实现代码

char map\_hidden[9][9]={

'\*','\*','\*','\*','\*','\*','\*','\*','\*',

'\*','\*','\*','\*','\*','\*','\*','\*','\*',

'\*','\*','\*','\*','\*','\*','\*','\*','\*',

'\*','\*','\*','\*','\*','\*','\*','\*','\*',

'\*','\*','\*','\*','\*','\*','\*','\*','\*',

'\*','\*','\*','\*','\*','\*','\*','\*','\*',

'\*','\*','\*','\*','\*','\*','\*','\*','\*',

'\*','\*','\*','\*','\*','\*','\*','\*','\*',

'\*','\*','\*','\*','\*','\*','\*','\*','\*',

};

void show\_map();

int win();

int mainboom()

{

int i,j;

for(i=0;i<9;i++){

for(j=0;j<9;j++){

map\_hidden[i][j]='\*';

}

}

char map[9][9]={

'X','2','1','1','1','1','1','X','1',

'2','X','2','3','X','2','2','2','2',

'1','2','X','3','X','3','2','X','1',

'1','2','3','3','3','3','X','3','2',

'1','X','2','X','3','X','3','3','X',

'1','1','3','3','X','3','X','2','1',

'1','2','2','X','3','3','3','3','2',

'X','3','X','3','3','X','3','X','X',

'2','X','3','X','2','1','3','X','3',

};

printf("Welcome to minesweep!\n");

show\_map();

while(1){

printf("Please input(x y):");

char input[4];

for(int i=0;i<4;i++)

{

input[i]=0;

}

char x,y;

//scanf("%d,%d",&x,&y);

int r=read(0,input,4);

input[r]=0;

x=input[0]-48;

y=input[2]-48;

if(x+48=='q'){

printf("Game Over\n");

return 0;

}

while(1){

if(x>0&&x<=9&&y>0&&y<=9&&map\_hidden[x-1][y-1]=='\*') break;

else{

printf("Error,please input again:");

int r=read(0,input,4);

input[r]=0;

x=input[0]-48;

y=input[2]-48;

if(x+48=='q'){

printf("Game Over\n");

return 0;

}

}

}

int m=x-1;

int n=y-1;

if(map[m][n]=='X'){

printf("Game Over!\n");

return 0;

}

int i,j;

for(i=m-1;i<x+1;i++){

if(i>=0&&i<=8){

for(j=n-1;j<y+1;j++){

if(j>=0&&j<=8){

if(map[i][j]!='X')map\_hidden[i][j]=map[i][j];

}

}

}

}

show\_map();

if(win()){

printf("You Win!\n");

break;

}

}

return 0;

}

void show\_map()

{

int i,j;

for(i=0;i<9;i++){

for(j=0;j<9;j++){

printf("%c ",map\_hidden[i][j]);

}

printf("\n");

}

}

int win(){

int sum=0;

int i,j;

for(i=0;i<9;i++){

for(j=0;j<9;j++){

if(map\_hidden[i][j]=='\*') sum++;

}

}

if(sum==23) return 1;

else return 0;

}

* 1. **命令行功能**

本操作系统的命令行参考ubuntu，进入控制台后控制台自动初始化，请求用户登入，登入后用户可按照以下格式执行功能



其中

\* 上键[↑]：切换上一条执行过的指令。

\* Tab键：“操作”或“文件名”智能补全，允许用户输入当前目录下的部分文件名，自动匹配到唯一文件并补全命令行。

如：

ki+[tab] => kill





vim f+[tab] => vim file





实现以上功能的代码如下

基础指令的定义

void shabby\_shell(const char \*tty\_name)

{

int fd\_stdin = open(tty\_name, O\_RDWR);

assert(fd\_stdin == 0);

int fd\_stdout = open(tty\_name, O\_RDWR);

assert(fd\_stdout == 1);

char rdbuf[128]; //读取的命令

char cmd[128]; //指令

char arg1[128]; //参数1

char arg2[128]; //参数2

char buf[1024];

initFs();

while (1)

{

if (usercount == 0)

{

printf("Enter Admin Password:");

char buf[128];

int r = read(0, buf, 128);

buf[r] = 0;

if (strcmp(buf, "admin") == 0)

{

strcpy(currentUser, "/");

UserState = 3;

break;

}

else

printf("Password Error!\n");

}

else

{

//printf("%d",usercount);

int isGet = 0;

printf("Enter User Name:");

char buf[128];

int r = read(0, buf, 128);

buf[r] = 0;

int i;

for (i = 0; i < usercount; i++)

{

if (strcmp(buf, users[i]) == 0 && strcmp(buf, "empty") != 0)

{

printf("Enter %s Password:");

char buf[128];

int r = read(0, buf, 128);

buf[r] = 0;

if (strcmp(buf, passwords[i]) == 0)

{

strcpy(currentUser, users[i]);

UserState = i + 1;

isGet = 1;

break;

}

}

}

if (isGet)

break;

else

printf("Password Error Or User Not Exist!\n");

}

}

初始化字符数组：

while (1)

{

clearArr(rdbuf, 128);

clearArr(cmd, 128);

clearArr(arg1, 128);

clearArr(arg2, 128);

clearArr(buf, 1024);

if(UserState == 3)

printf("[Admin@TJOS]%s%s# ",currentUser,currentFolder);

else

printf("[%s@TJOS]/%s%s$ ",users[UserState-1],currentUser,currentFolder);

//write(1, "$ ", 2);

int r = read(0, rdbuf, 70);

rdbuf[r] = 0;

int argc = 0;

char \*argv[PROC\_ORIGIN\_STACK];

char \*p = rdbuf;

char \*s;

int word = 0;

char ch;

do

{

ch = \*p;

if (\*p != ' ' && \*p != 0 && !word)

{

s = p;

word = 1;

}

if ((\*p == ' ' || \*p == 0) && word)

{

word = 0;

argv[argc++] = s;

\*p = 0;

}

p++;

} while (ch);

argv[argc] = 0;

int fd = open(argv[0], O\_RDWR);

if (fd == -1)

{

if (rdbuf[0])

{

int i = 0, j = 0;

/\* 命令输入 \*/

while (rdbuf[i] != ' ' && rdbuf[i] != 0)

{

cmd[i] = rdbuf[i];

i++;

}

i++;

/\* 参数1赋值 \*/

while (rdbuf[i] != ' ' && rdbuf[i] != 0)

{

arg1[j] = rdbuf[i];

i++;

j++;

}

i++;

j = 0;

/\* 参数2 \*/

while (rdbuf[i] != ' ' && rdbuf[i] != 0)

{

arg2[j] = rdbuf[i];

i++;

j++;

}

对命令的处理：

if (strcmp(cmd, "help") == 0)

{

showhelp();

}

else if (strcmp(cmd, "clear") == 0)

{

clear();

welcome();

}

else if (strcmp(cmd, "sudo") == 0)

{

printf("Enter Admin Password:");

char buf[128];

int r = read(0, buf, 128);

buf[r] = 0;

if (strcmp(buf, "admin") == 0)

{

strcpy(currentUser, "/");

UserState = 3;

}

else

printf("Password Error!\n");

}

else if (strcmp(cmd, "adduser") == 0)

{

addUser(arg1, arg2);

}

else if (strcmp(cmd, "deluser") == 0)

{

moveUser(arg1, arg2);

}

else if (strcmp(cmd, "su") == 0)

{

shift(arg1, arg2);

}

else if (strcmp(cmd, "mkfile") == 0)

{

createFilepath(arg1);

createFile(filepath, arg2, 1);

clearArr(filepath, 128);

}

else if(strcmp(cmd, "mkdir") == 0)

{

createFilepath(strcat(arg1,"\*"));

createFolder(filepath, 1);

clearArr(filepath, 128);

}

else if (strcmp(cmd, "cd") == 0)

{

createFilepath(arg1);

openFolder(filepath,arg1);

}

else if (strcmp(cmd, "rd") == 0)

{

createFilepath(arg1);

readFile(filepath);

clearArr(filepath, 128);

}

/\* edit a file appand \*/

else if (strcmp(cmd, "wt+") == 0)

{

createFilepath(arg1);

editAppand(filepath, arg2);

clearArr(filepath, 128);

}

/\* edit a file cover \*/

else if (strcmp(cmd, "wt") == 0)

{

createFilepath(arg1);

editCover(filepath, arg2);

clearArr(filepath, 128);

}

/\* delete a file \*/

else if (strcmp(cmd, "del") == 0)

{

createFilepath(arg1);

deleteFile(filepath);

clearArr(filepath, 128);

}

/\* ls \*/

else if (strcmp(cmd, "ls") == 0)

{

ls();

}

else if (strcmp(cmd, "proc") == 0)

{

showProcess();

}

else if (strcmp(cmd, "kill") == 0)

{

killpro(arg1);

}

else if (strcmp(cmd, "pause") == 0)

{

pausepro(arg1);

}

else if (strcmp(cmd, "resume") == 0)

{

resume(arg1);

}

else if (strcmp(cmd, "tictactoe") == 0)

{

main\_tic();

}

else if(strcmp(cmd, "boom") == 0)

{

mainboom();

}

else

{

continue;

}

}

}

else

{

close(fd);

int pid = fork();

if (pid != 0)

{ /\* parent \*/

int s;

wait(&s);

}

else

{ /\* child \*/

execv(argv[0], argv);

}

}

}

close(1);

close(0);

}