# 实验项目：UCore启动实验

姓名：张伟焜 学号：17343155 邮箱：[zhangwk8@mail2.sysu.edu.cn](mailto:zhangwk8@mail2.sysu.edu.cn)

院系：数据科学与计算机学院 专业：17级软件工程 指导教师：张永东

**【实验题目】**

UCore启动实验

**【实验目的】**

操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件-bootloader 来完成这些工作。为此，我们需要完成一个能够切换到x86 的保护模式并显示字符的bootloader，为启动操作系统ucore 做准备。lab1 提供了一个非常小的bootloader 和ucoreOS，整个bootloader 执行代码小于512 个字节，这样才能放到硬盘的主引导扇区中。通过分析和实现这个bootloader 和ucore OS，实验者可以了解到：

* 基于分段机制的存储管理
* 设备管理的基本概念
* PC 启动bootloader 的过程
* bootloader 的文件组成
* 编译运行 bootloader 的过程
* 调试 bootloader 的方法
* ucore OS 的启动过程
* 在汇编级了解栈的结构和处理过程
* 中断处理机制
* 通过串口/并口/CGA 输出字符的方法

**【实验要求】**

根据指导，完成练习1~6。

**【实验方案】**

实验环境：老师提供的虚拟机（Virtual box），无特殊硬件要求

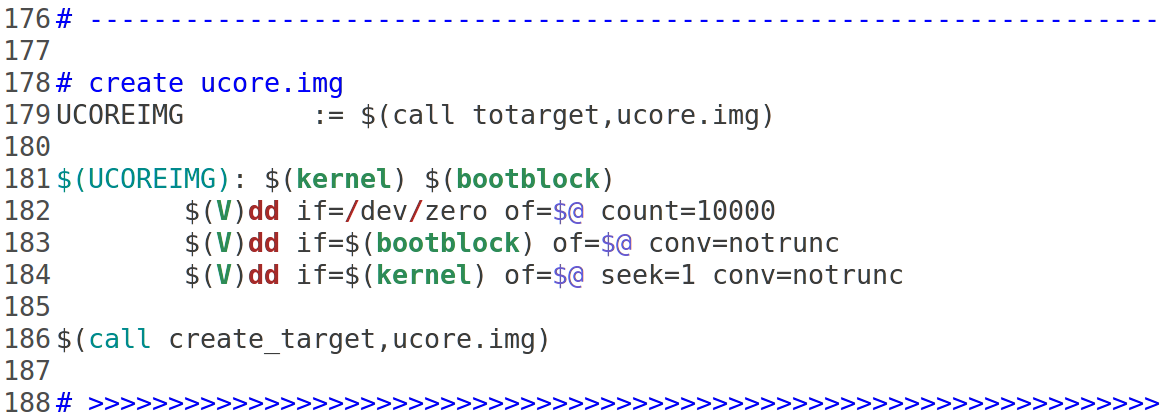
实验思路：根据实验指导，先了解理论知识，再进行实验

**【实验过程】**

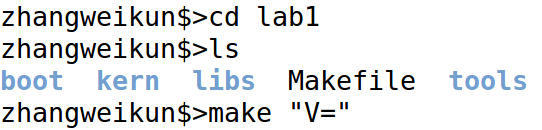
**练习1：理解通过make生成执行文件的过程。**

**1.**

查看Makefile代码：



运行Makefile：



显示结果及相关注释：

+ cc kern/init/init.c //编译init.c

gcc -c kern/init/init.c -o obj/kern/init/init.o

+ cc kern/libs/readline.c //编译readline.c

gcc -c kern/libs/readline.c -o

obj/kern/libs/readline.o

+ cc kern/libs/stdio.c //编译stdlio.c

gcc -c kern/libs/stdio.c -o obj/kern/libs/stdio.o

+ cc kern/debug/kdebug.c //编译kdebug.c

gcc -c kern/debug/kdebug.c -o obj/kern/debug/kdebug.o

+ cc kern/debug/kmonitor.c //编译komnitor.c

gcc -c kern/debug/kmonitor.c -o

obj/kern/debug/kmonitor.o

+ cc kern/debug/panic.c //编译panic.c

gcc -c kern/debug/panic.c -o obj/kern/debug/panic.o

+ cc kern/driver/clock.c //编译clock.c

gcc -c kern/driver/clock.c -o obj/kern/driver/clock.o

+ cc kern/driver/console.c //编译console.c

gcc -c kern/driver/console.c -o

obj/kern/driver/console.o

+ cc kern/driver/intr.c //编译intr.c

gcc -c kern/driver/intr.c -o obj/kern/driver/intr.o

+ cc kern/driver/picirq.c //编译prcirq.c

gcc -c kern/driver/picirq.c -o

obj/kern/driver/picirq.o

+ cc kern/trap/trap.c //编译trap.c

gcc -c kern/trap/trap.c -o obj/kern/trap/trap.o

+ cc kern/trap/trapentry.S //编译trapentry.S

gcc -c kern/trap/trapentry.S -o

obj/kern/trap/trapentry.o

+ cc kern/trap/vectors.S //编译vectors.S

gcc -c kern/trap/vectors.S -o obj/kern/trap/vectors.o

+ cc kern/mm/pmm.c //编译pmm.c

gcc -c kern/mm/pmm.c -o obj/kern/mm/pmm.o

+ cc libs/printfmt.c //编译printfmt.c

gcc -c libs/printfmt.c -o obj/libs/printfmt.o

+ cc libs/string.c //编译string.c

gcc -c libs/string.c -o obj/libs/string.o

+ ld bin/kernel //链接成kernel

ld -o bin/kernel

obj/kern/init/init.o obj/kern/libs/readline.o

obj/kern/libs/stdio.o obj/kern/debug/kdebug.o

obj/kern/debug/kmonitor.o obj/kern/debug/panic.o

obj/kern/driver/clock.o obj/kern/driver/console.o

obj/kern/driver/intr.o obj/kern/driver/picirq.o

obj/kern/trap/trap.o obj/kern/trap/trapentry.o

obj/kern/trap/vectors.o obj/kern/mm/pmm.o

obj/libs/printfmt.o obj/libs/string.o

+ cc boot/bootasm.S //编译bootasm.c

gcc -c boot/bootasm.S -o obj/boot/bootasm.o

+ cc boot/bootmain.c //编译bootmain.c

gcc -c boot/bootmain.c -o obj/boot/bootmain.o

+ cc tools/sign.c //编译sign.c

gcc -c tools/sign.c -o obj/sign/tools/sign.o

gcc -O2 obj/sign/tools/sign.o -o bin/sign

+ ld bin/bootblock //根据sign规范生成bootblock

ld -m elf\_i386 -nostdlib -N -e start -Ttext 0x7C00

obj/boot/bootasm.o obj/boot/bootmain.o

-o obj/bootblock.o

//创建大小为10000个块的ucore.img，初始化为0，每个块为512字节

dd if=/dev/zero of=bin/ucore.img count=10000

//把bootblock中的内容写到第一个块

dd if=bin/bootblock of=bin/ucore.img conv=notrunc

//从第二个块开始写kernel中的内容

dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc

我们可以得到ucore.img的生成过程：

(1)编译所有生成bin/kernel所需的文件

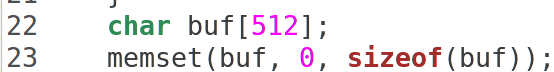
(2)链接生成bin/kernel

(3)编译bootasm.S bootmain.c sign.c

(4)根据sign规范生成obj/bootblock.o

(5)生成ucore.img（首先先创建一个大小为10000字节的块儿，然后再将bootblock拷贝过去。 生成ucore.img需要先生成kernel和bootblock）

**2.**文档中写道“sign.c是一个C语言小程序，是辅助工具，用于生成一个符合规范的硬盘主引导扇区。”因此，阅读sign.c代码：





得出，一个被系统认为是符合规范的硬盘主引导扇区的特征是：

大小为512字节；第510个字节为0x55；第511字节为0xAA

**练习2：使用qemu执行并调试lab1中的软件。**

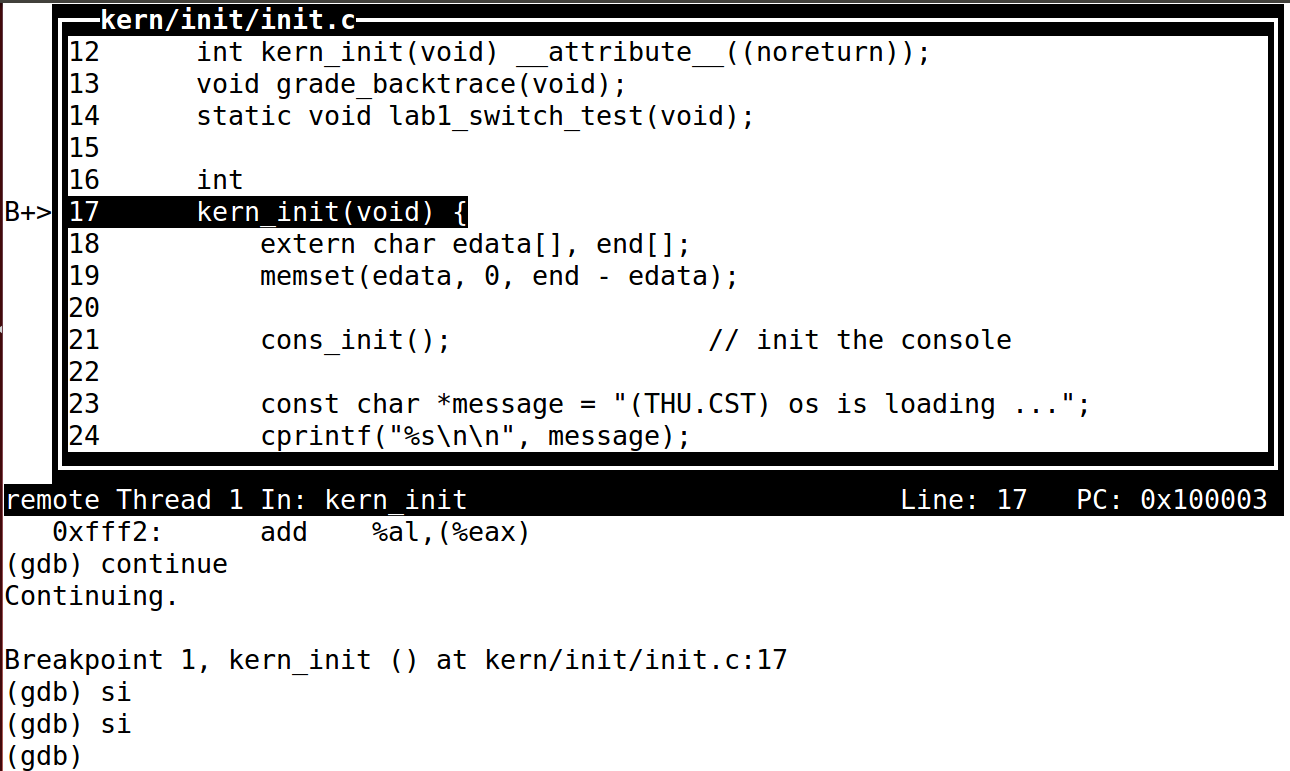
**1.**查看gdbinit文件中的指令，它在kern\_init函数设了断点，再执行了continue。我们要将continue去掉，这样make debug后就会自动停在0xffff0。



跳转至gdb界面可发现，此时停在断点0xffff0（图中红色标记）



输入continue输入si(stepi)进行单步调试：



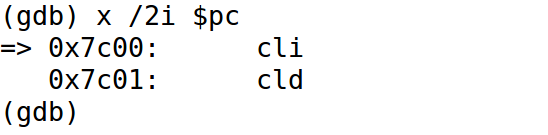
**2.**在初始化位置 0x7c00 设置实地址断点。

输入b\*0x7c00

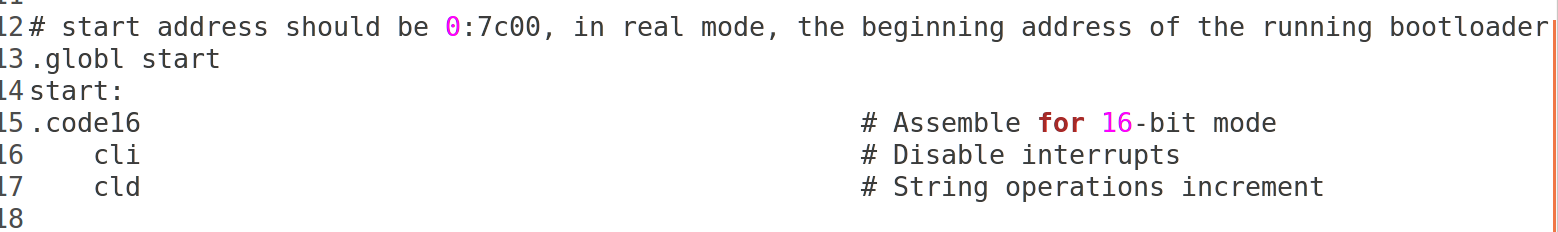
输入continue运行至断点处



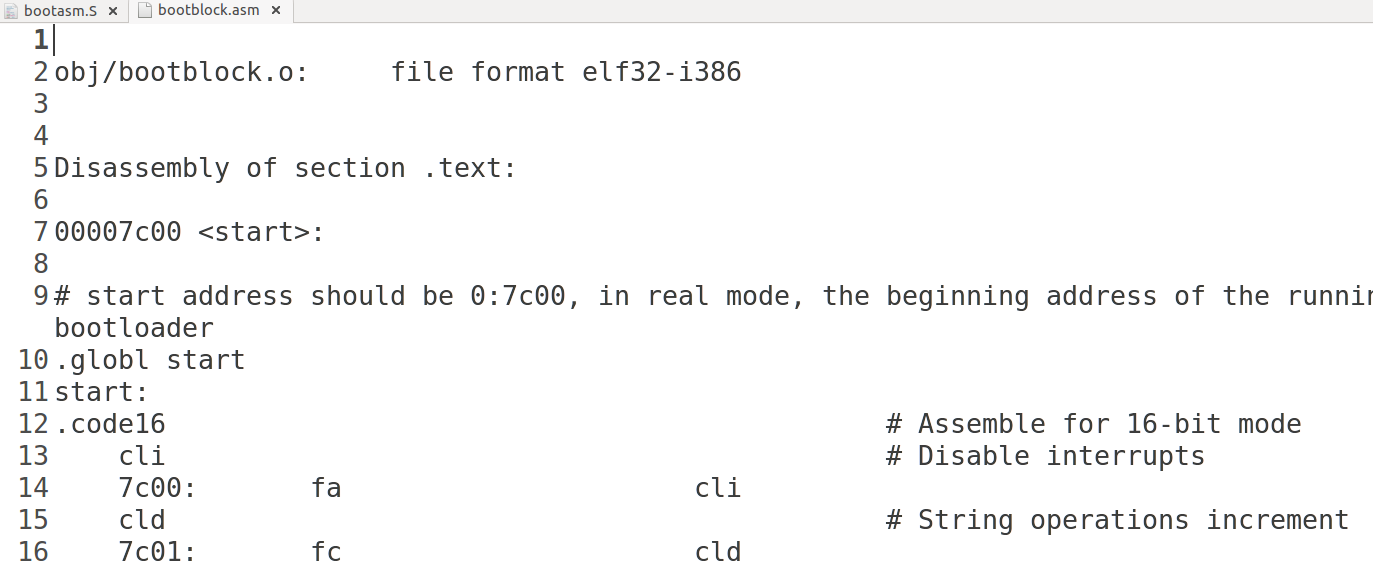
输入x /2i $pc 查看反汇编代码：



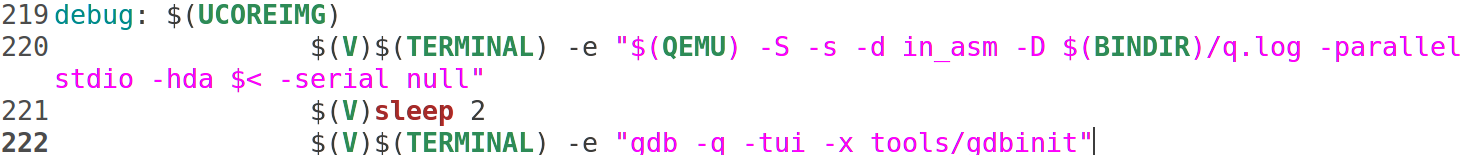
与bootasm.S中代码一致：



与bootblock.asm中代码一致:



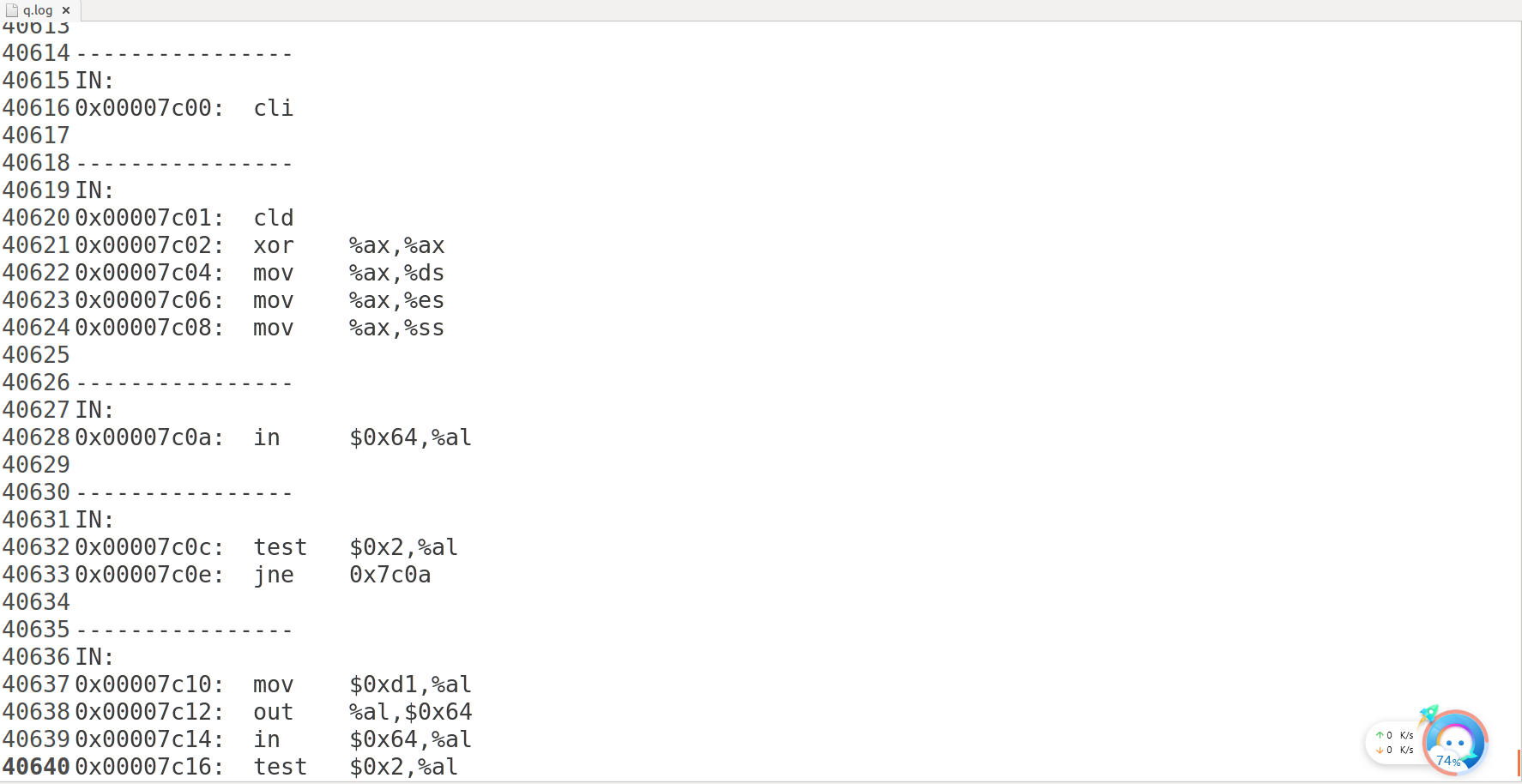
**3.**将Makefile的debug改为如下内容，再次make debug会在bin目录下生成q.log



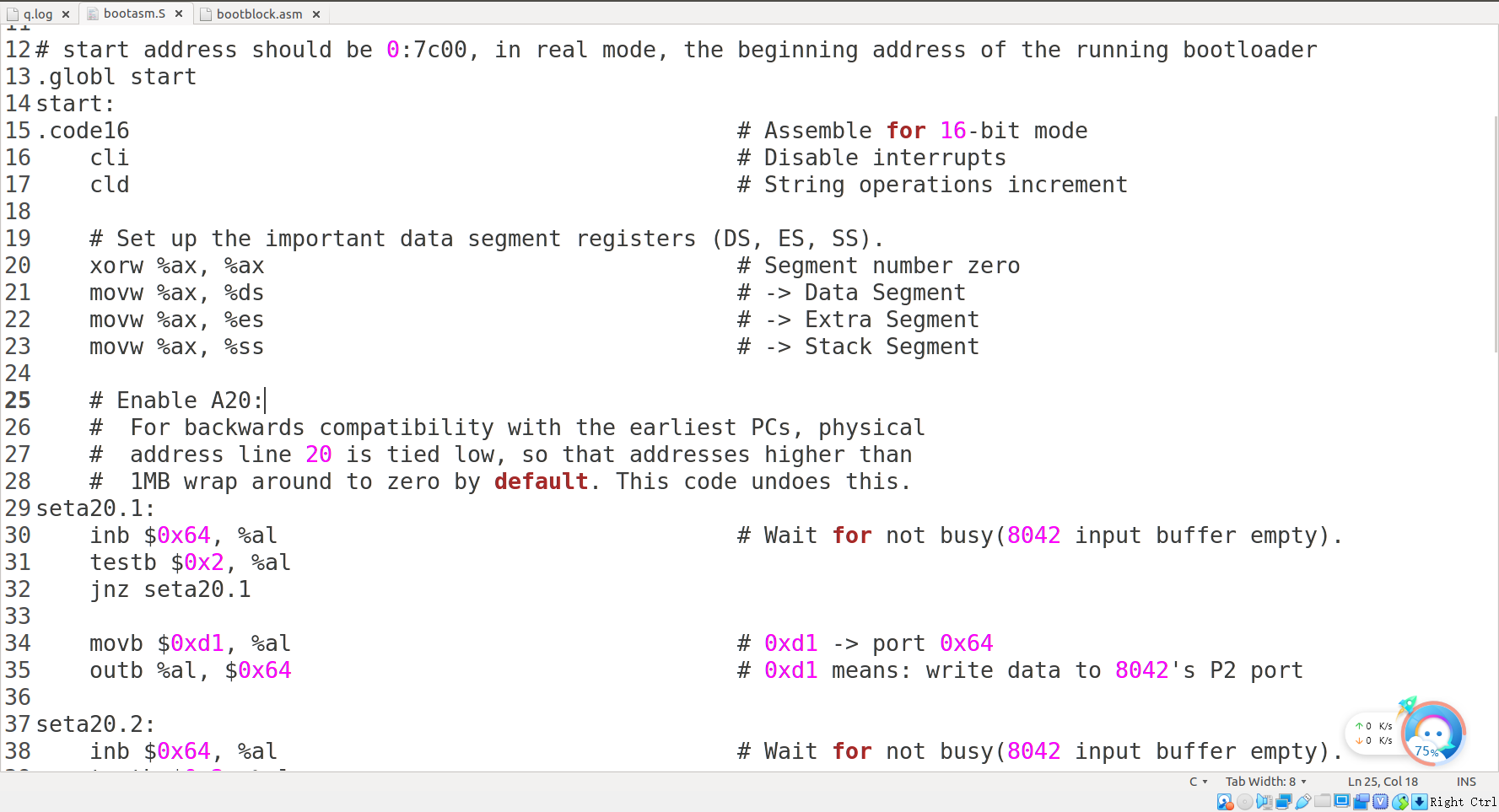


对比q.log、bootasm.S和bootblock.asm中代码，发现它们相同（q.log中断点后的代码与其余两者相同）：

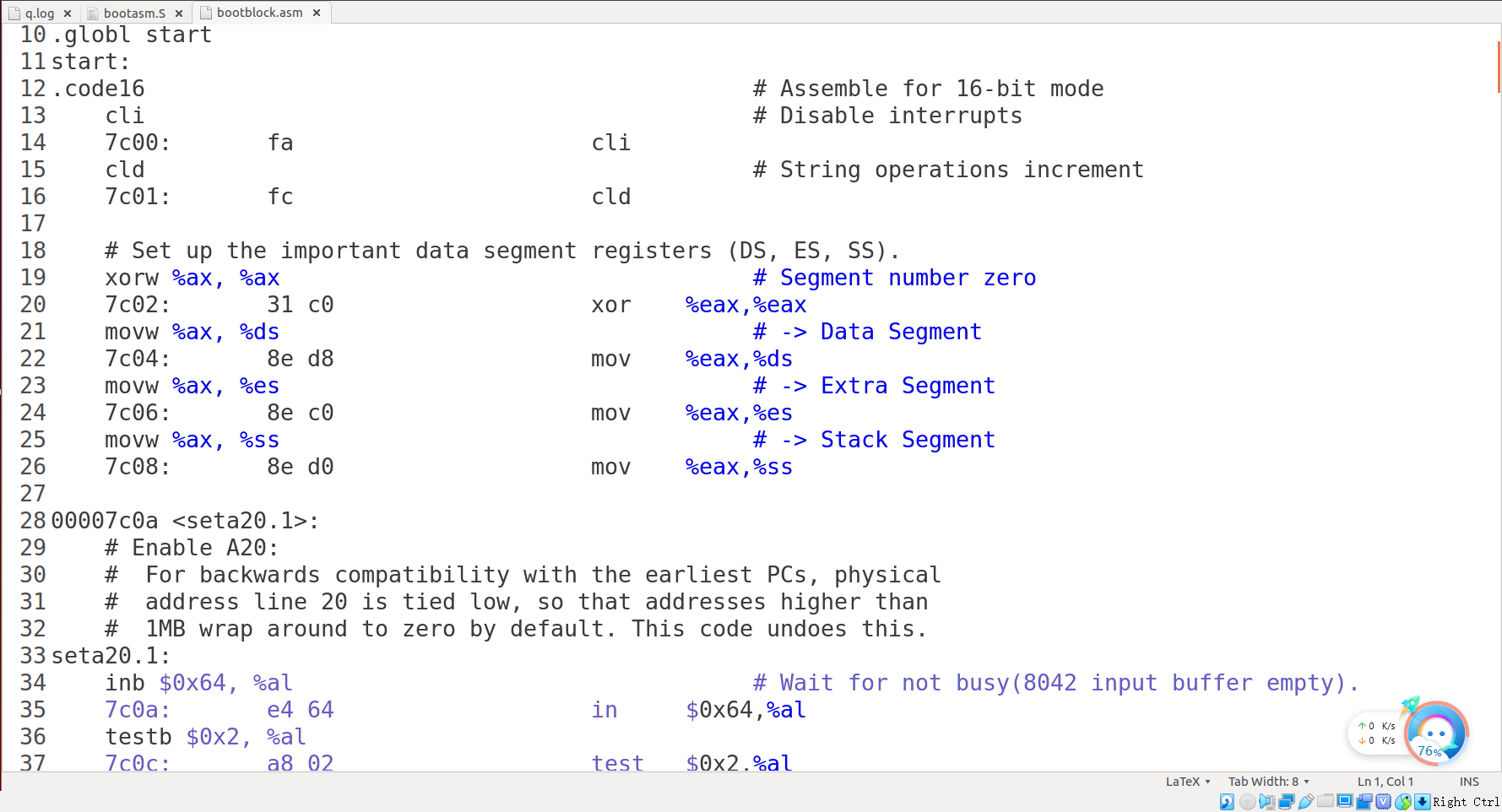
q.log:



bootasm.S：

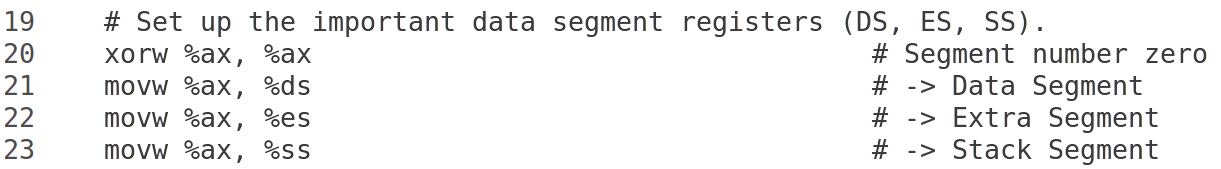


bootblock.asm：

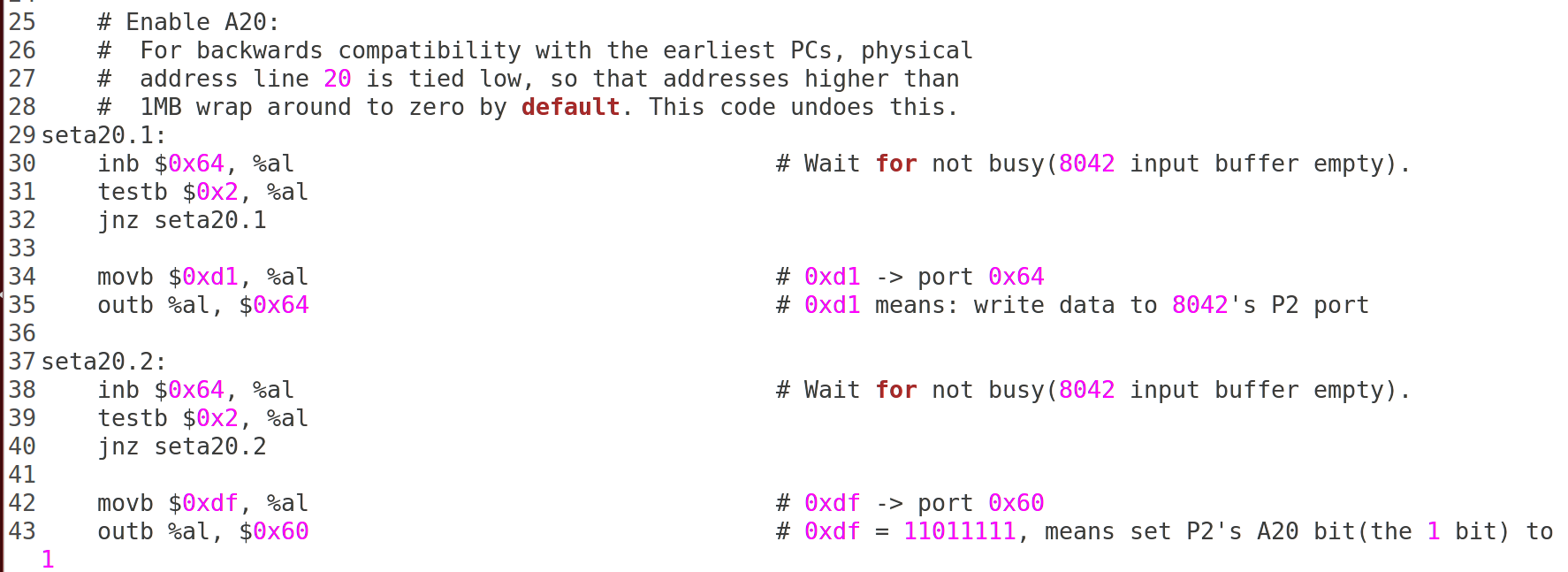


**练习3：分析bootloader 进入保护模式的过程。**

**1.**首先关闭中断，将各个段寄存器重置为0：



**2.**开启A20.（当 A20 地址线控制禁止时，处于实模式。通过修改A20 地址线可以完成从实模式到保护模式的转换，即需要通过将键盘控制器上的A20线置于高电位，使得全部32条地址线可用）



其中：

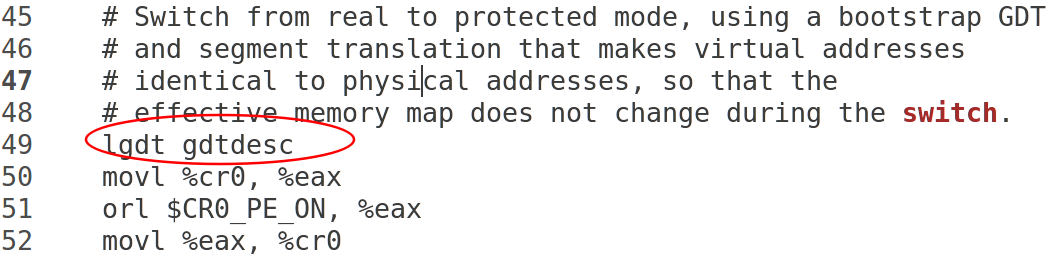
inb $0x64, %al # 读取状态寄存器,等待8042键盘控制器闲置

testb $0x2, %al # 判断输入缓存是否为空

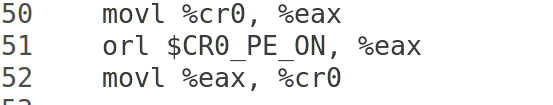
movb $0xd1, %al # 0xd1表示写输出端口命令，参数随后通过0x60端口写入

movb $0xdf, %al # 通过0x60写入数据11011111 即将A20置1

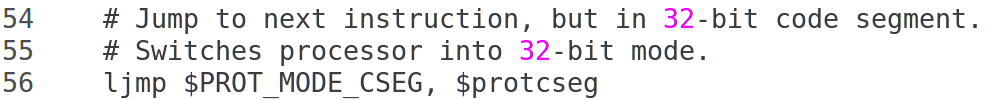
**3.**加载全局描述符表（GDT表已经存储在引导区中，直接加载即可）：



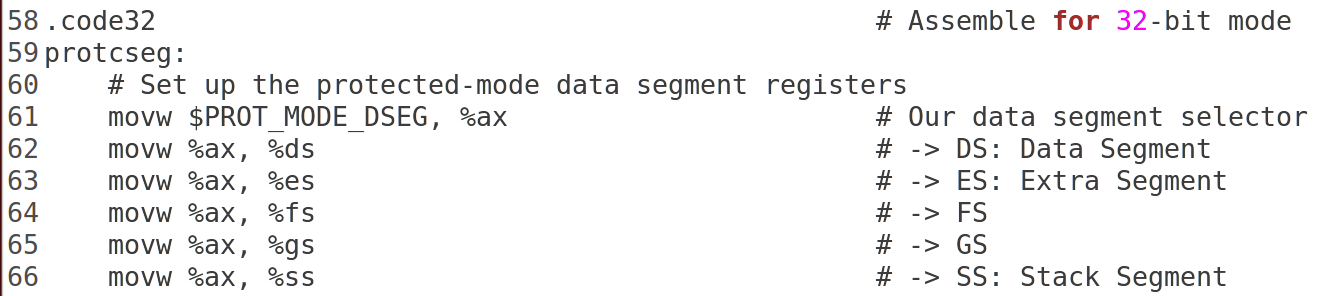
**4.**将控制寄存器CR0的第0位置为1



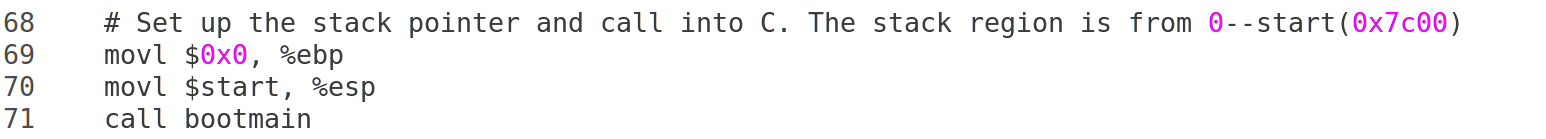
**5.**长跳转到32位代码段，重装CS和EIP



**6.**重装DS、ES等段寄存器，建立堆栈



**7.**转到保护模式完成，进入bootmain方法

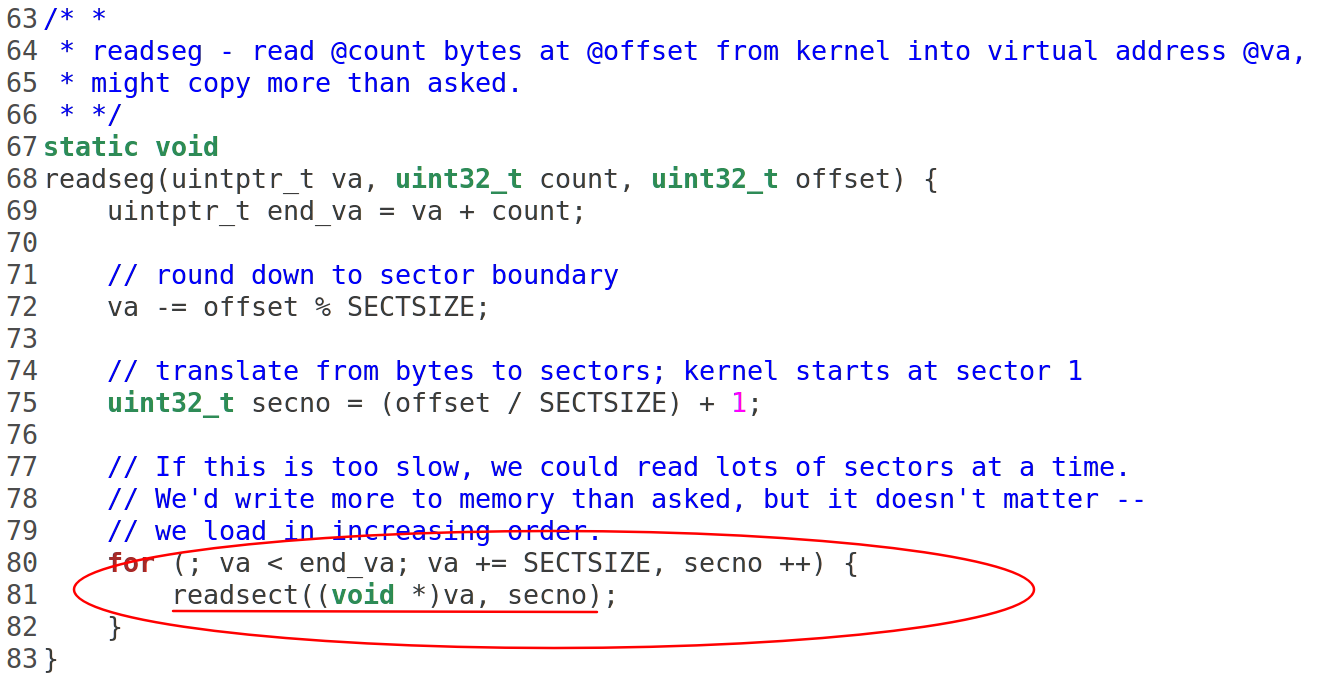


**练习4：分析bootloader 加载ELF 格式的OS 的过程。**

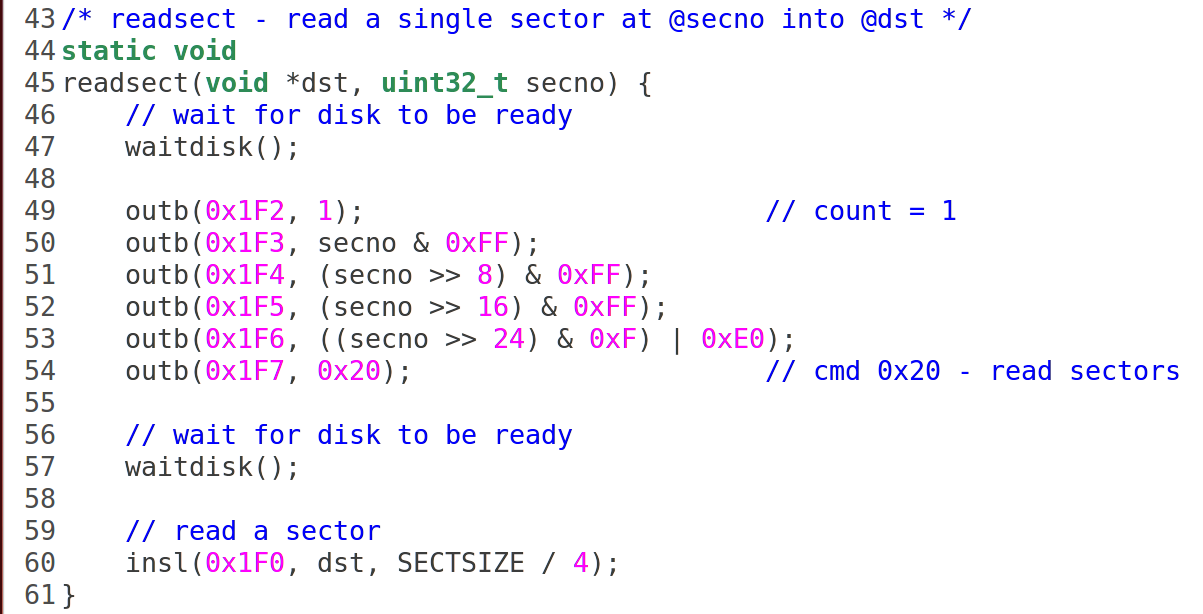
**1.**bootloader读取硬盘扇区。

由bootmain函数分析，首先是由readseg函数读取硬盘扇区，而readseg函数则循环调用了真正读取硬盘扇区的函数readsect来一次读取一个扇区。

readseg函数：



Readsect函数：



其中：

waitdisk(); // 等待硬盘就绪

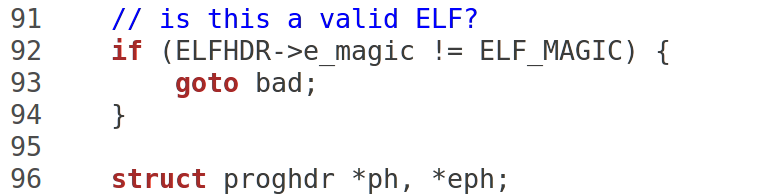
outb(…)

…// 写地址0x1f2~0x1f5,0x1f7,发出读取磁盘的命令

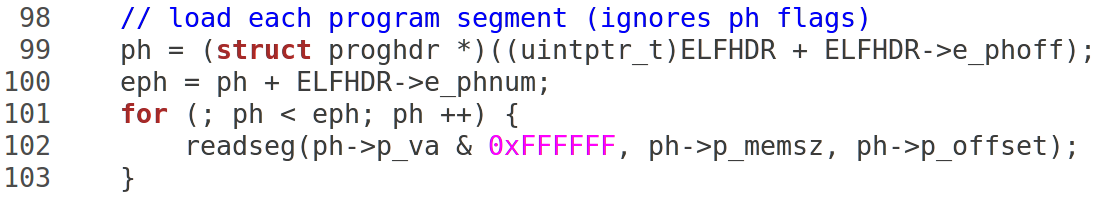
insl(0x1F0, dst, SECTSIZE / 4);//读取一个扇区

**2.**bootloader加载ELF格式的OS。

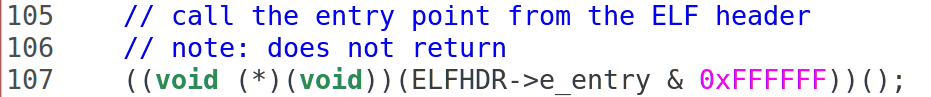
首先判断是不是ELF格式的文件。若不是，goto bad。



ELF头部有描述ELF文件应加载到内存什么位置的描述表，这里读取出来将之存入ph，并按照程序头表的描述，将ELF文件中的数据载入内存。



根据ELF头表中的入口信息，找到内核的入口并开始运行

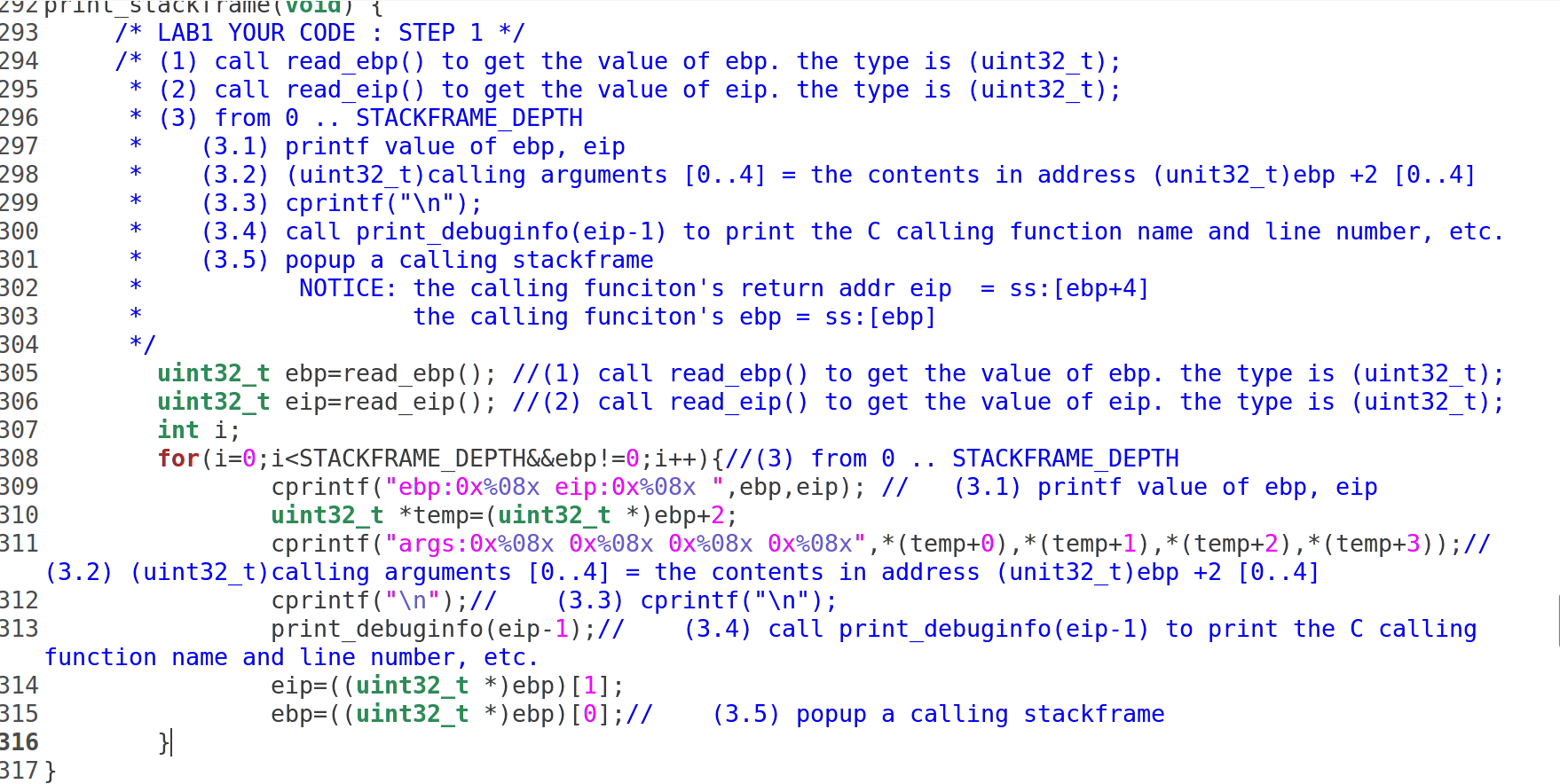


**练习5：实现****函数调用堆栈跟踪函数。**

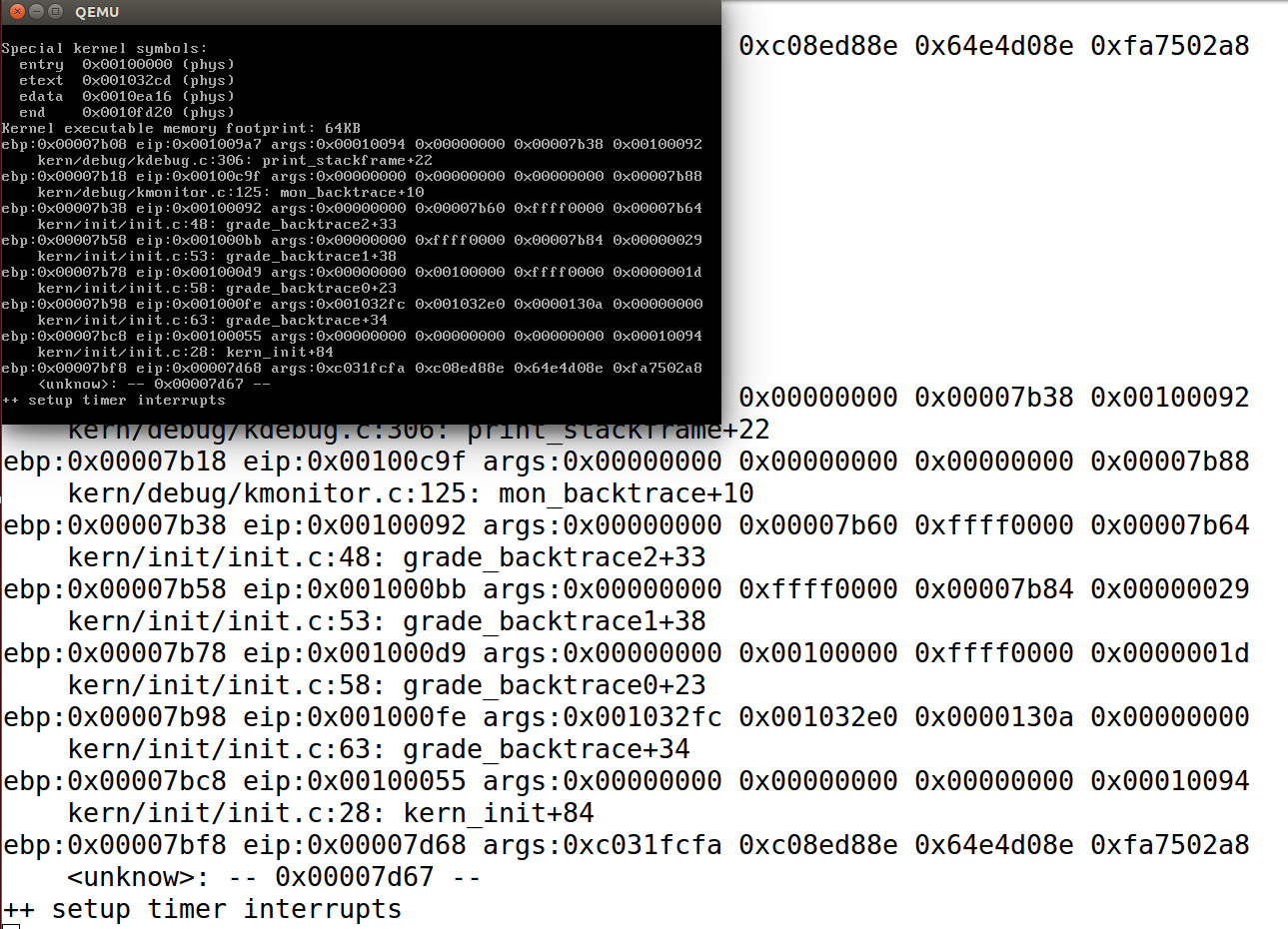
思路：

先通过read\_ebp()和read\_eip()函数来获取当前ebp寄存器以及eip 寄存器信息。然后通过ebp+12,ebp+16,ebp+20,ebp+24来输出4个参数的值，最后更新ebp：ebp=ebp[0],更新eip：eip=ebp[1]。直到ebp 对应地址的值为0（表示当前函数为bootmain）。

根据提供的注释编写代码:

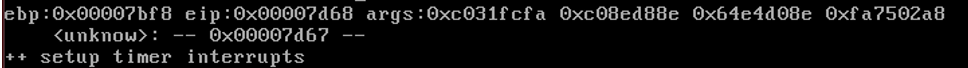


运行结果：



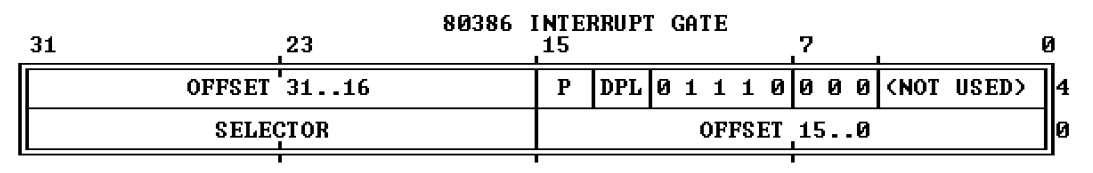
最后一行的解释：

最后一行对应的是第一个使用堆栈的函数，bootmain.c中的bootmain。（因为此时ebp对应地址的值为0） bootloader设置的堆栈从0x7c00开始，使用”call bootmain”转入bootmain函数。call指令压栈，所以bootmain中ebp为0x7bf8。



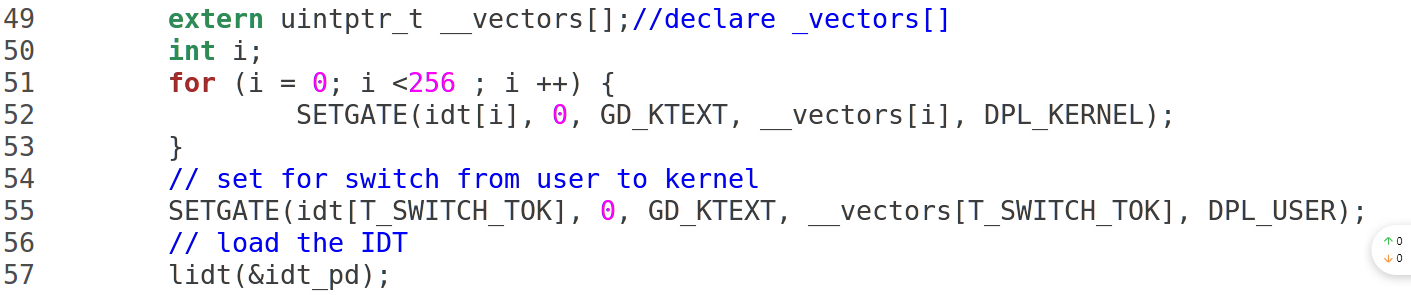
**练习6：完善中断初始化和处理。**

**1.**中断向量表中一个表项占8字节。其中0~15位和48~63位分别为offset的低16位和高16位。16~31位为段选择子。通过段选择子获得段基址，加上段内偏移量即可得到中断处理代码入口。

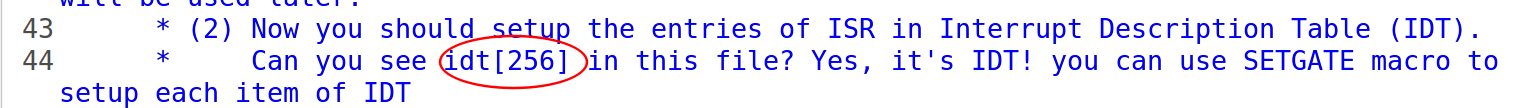


**2.**首先，声明\_\_vertors[],其中存放着中断服务程序的入口地址。这个数组生成于vertor.S中。

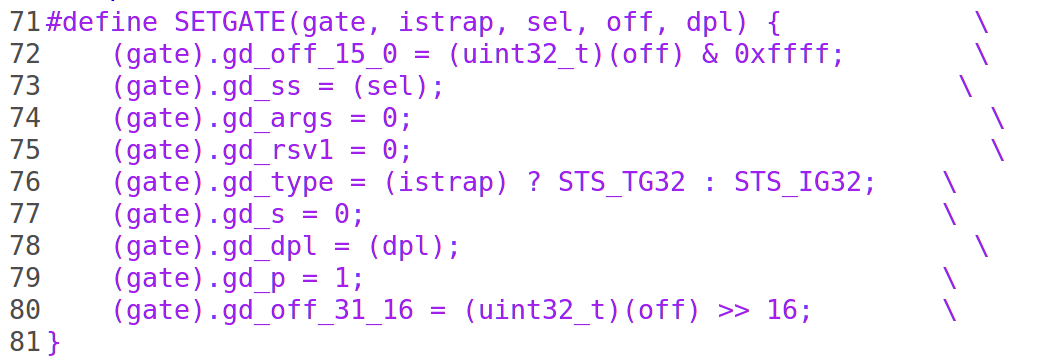
接着，填充中断描述符表IDT。然后，加载中断描述符表。



其中i<256,是参考代码中给的注释：



另外，SETGATE的定义在mmu.h中：



gate：对应的idt[]数组内容，处理函数的入口地址

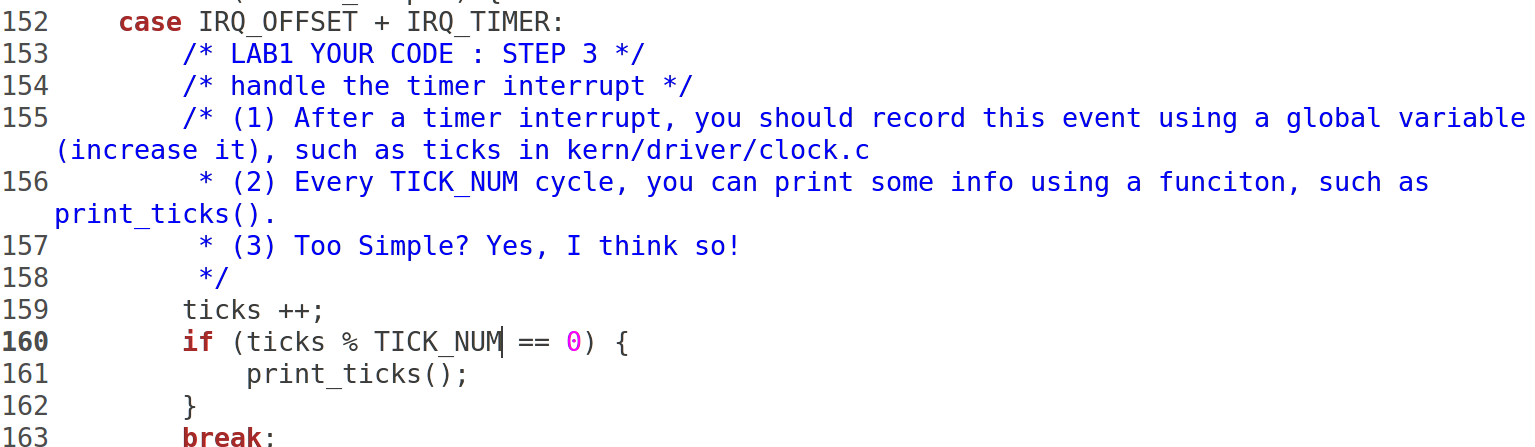
istrap：系统段设置为1，中断门设置为0

sel：段选择子

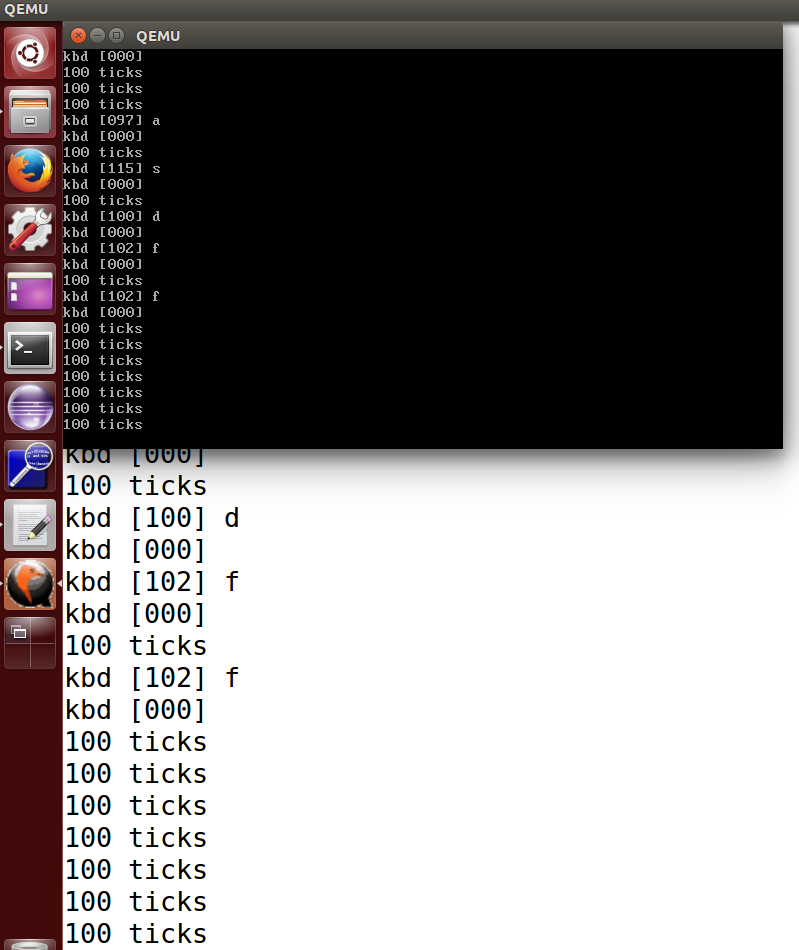
off：\_\_vectors[]数组内容

dpl：设置特权级。这里中断都设置为内核级，即第0级

**3.** trap函数调用子函数trap\_dispatch，补全trap\_dispatch函数的判断部分：



结果：每100次时钟中断显示“100 ticks”,且按下的键也会在屏幕上显示。



**【实验总结】**

初次拿到实验文档，感觉压力很大，认为实验很难，但当自己真正静心阅读实验指导时发现其实难度没有自己想象的那么大。甚至有很多问题的答案直接就写在实验指导中，如中断向量表就直接在指导手册中画出来了。但不可否认的是，实验的内容还是挺丰富的，涉及的方面也很多，包括到makefile，gdb调试等基础操作，也涉及到BIOS，扇区引导，中断机制，函数调用堆栈等内容。

实验遇到的一个困难是，开始的时候陷在makefile里无法自拔，花费了很长时间。自己过于纠结每个makefile的语句，忽略了对整个编译先后顺序的整体把握。同时，由于对gdb调试不熟悉，花了一些时间去学习，但真正做实验的时候发现需要用的指令并不是很多。

另外，在实验过程中学习了一些新操作，比如makefile的make V=帮助我们理解编译过程；在gdb中用x /2i %pc 进行反汇编查看汇编代码。

总之，通过这次试验，我对PC启动bootloader以及ucore操作系统的启动过程有了深入的认识，掌握了中断处理机制和函数调用堆栈跟踪函数。

**【参考文献】**

*《操作系统实验指导(清华大学)陈渝、向勇编著》*