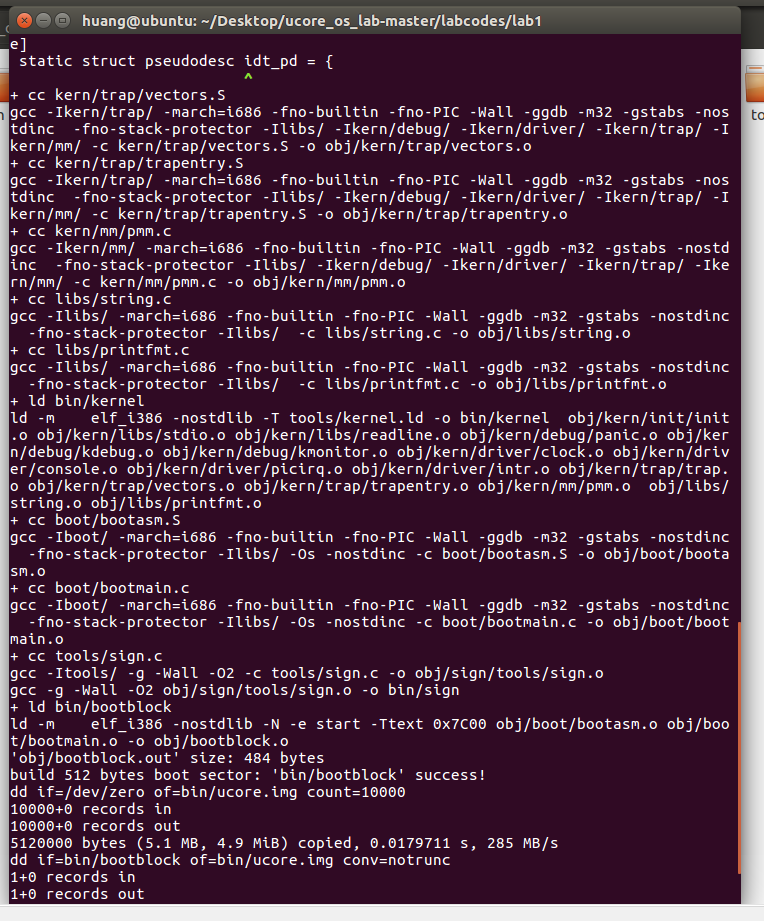
**练习1：理解通过make生成执行文件的过程。**

1. **操作系统镜像文件ucore.img是如何一步一步生成的？(需要比较详细地解释Makefile中每 一条相关命令和命令参数的含义，以及说明命令导致的结果)**

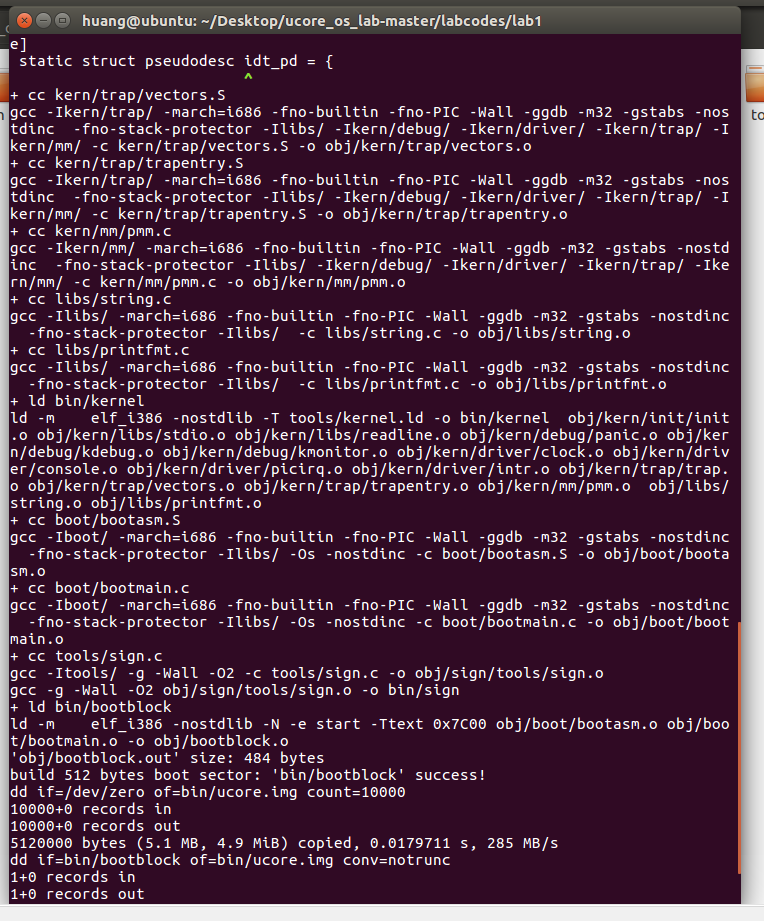
执行命令make V=

实际上是设置一个标记，使得Make它的执行过程能够展现出来

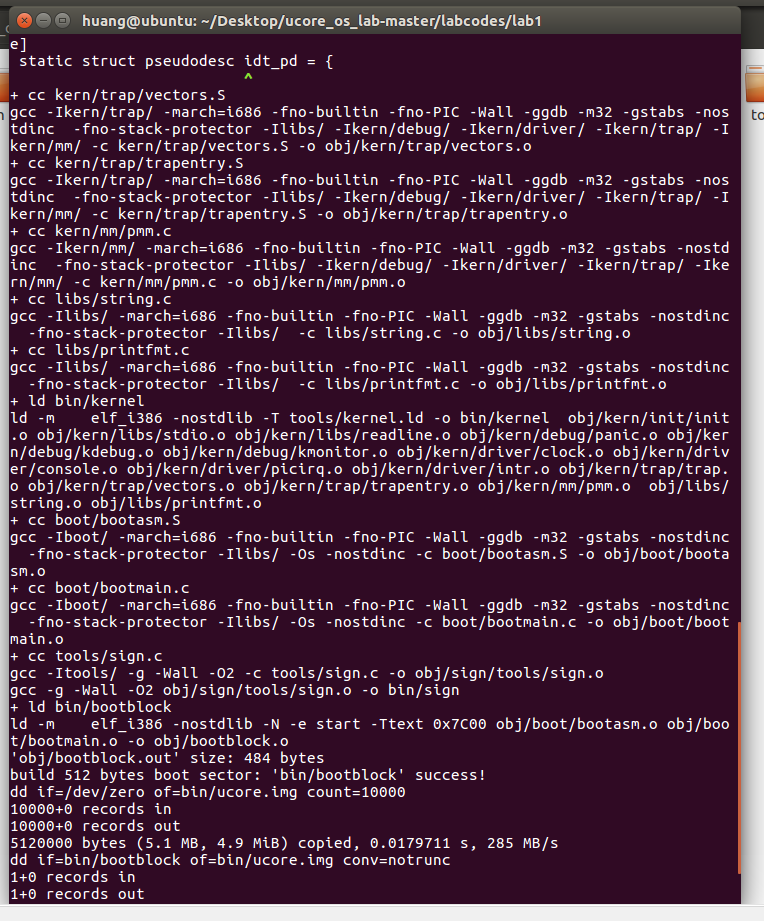
那可以看到它调了GCC，把一些C的源代码编译成了.o结尾的文件。



然后通过ld，ld会把这些目标文件转换成一个执行程序比如说在这里面会转换成这个bootblock.out（一个Bootloader执行程序）



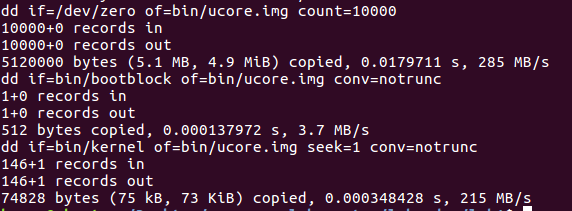
最后用dd可以把这个Bootloader放到一个虚拟的硬盘里面去，生成一个虚拟硬盘叫uCore.img count。



硬件模拟器呢会基于这个虚拟硬盘中的数据来执行相应的代码

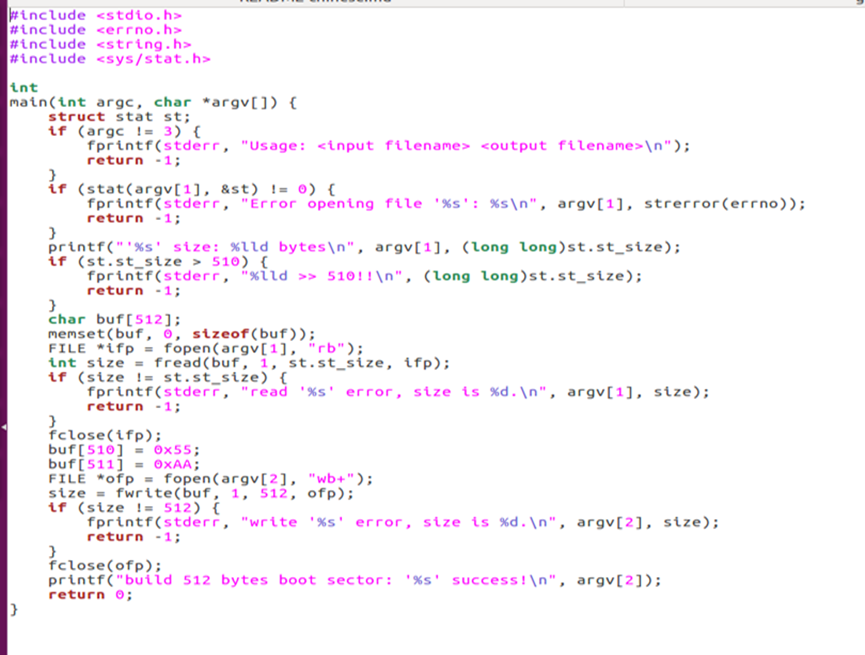
这里面其实生成了两个软件

第一个是Bootloader 第二叫kernel，kernel实际上是uCore的组成部分



1. **一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？**

硬盘主引导扇区的规范写在了sign.c文件中



一个被系统认为是符合规范的硬盘主引导扇区的特征有以下几点：

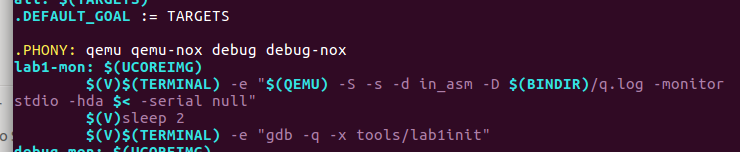
- 磁盘主引导扇区只有512字节

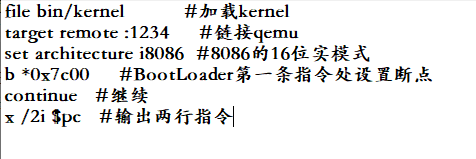
- 磁盘最后两个字节为0x55AA

- 由不超过466字节的启动代码和不超过64字节的硬盘分区表加上两个字节的结束符组成

**练习2：使用qemu执行并调试lab1中的软件。**

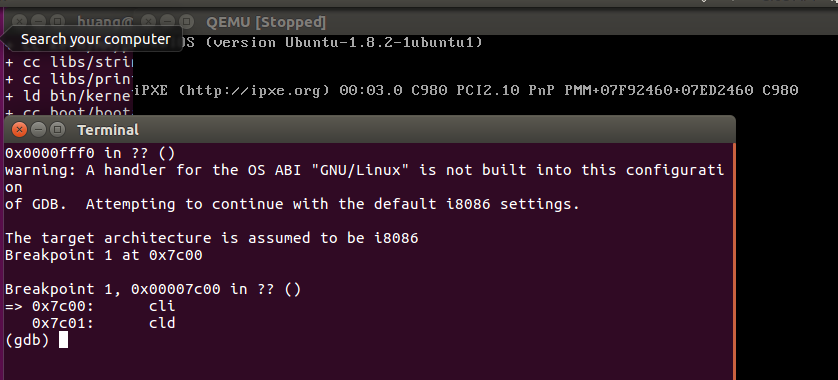
**为了熟悉使用qemu和gdb进行的调试工作，我们进行如下的小练习：**

1. **从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。**

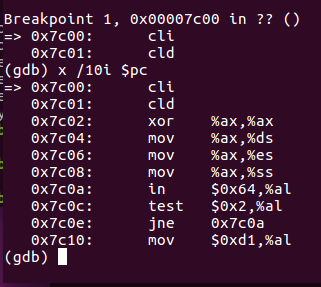
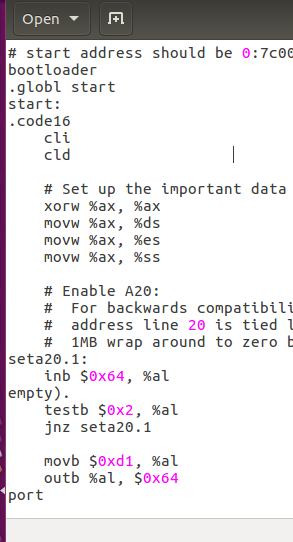


**2. 在初始化位置0x7c00设置实地址断点,测试断点正常。**

运行make lab1-mon,会在0x7c00处中断



**3. 从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较。两者指令相同。**

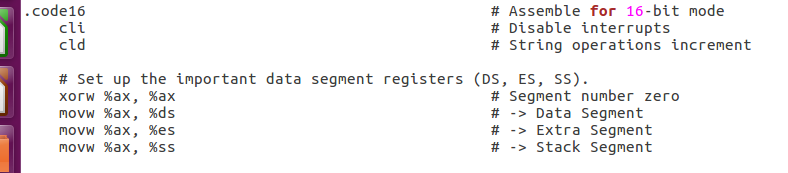
**4. 自己找一个bootloader或内核中的代码位置，设置断点并进行测试。**

**练习3：分析bootloader进入保护模式的过程。（要求在报告中 写出分析）**

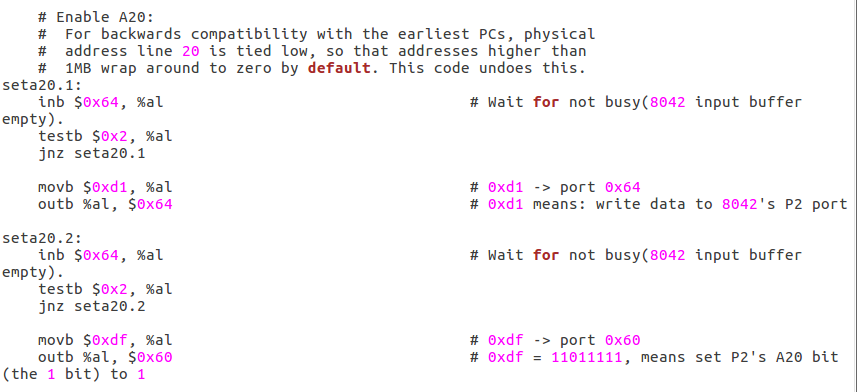
BIOS将通过读取硬盘主引导扇区到内存，并转跳到对应内存中的位置执行bootloader。请分 析bootloader是如何完成从实模式进入保护模式的。

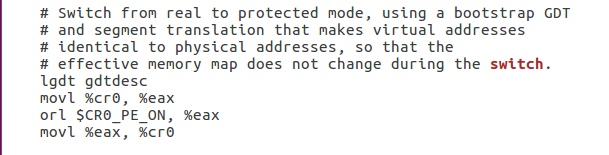
首先我们先分析一下bootloader：

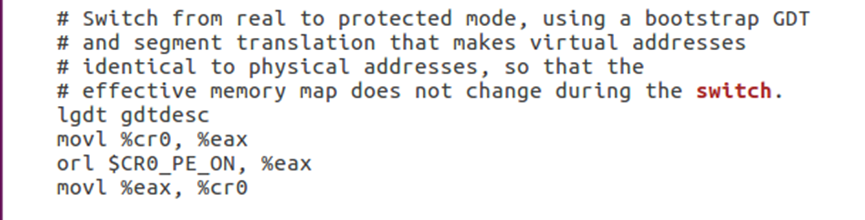
关闭中断，将各个段寄存器重置，它先将各个寄存器置0



开启A20，然后就是将A20置1。当 A20 地址线控制禁止时，程序就像在 8086 中运行，1MB 以上的地址不可访问。而在保护模式下 A20 地址线控制打开，所以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。

加载GDT表

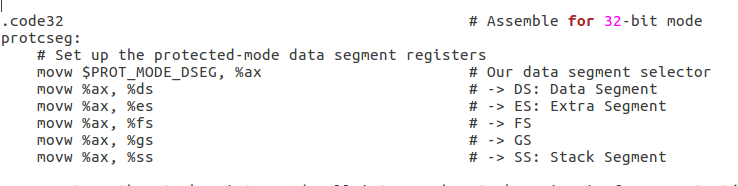
将CR0的第0位置1

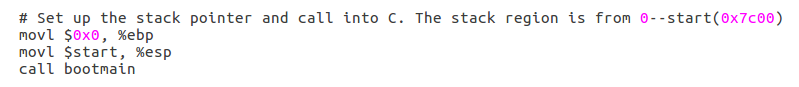


长跳转到32位代码段，重装CS和EIP



重装DS、ES等段寄存器等

转到保护模式完成，进入boot主方法



**练习4：分析bootloader加载ELF格式的OS的过程。（要求在报 告中写出分析）**

**通过阅读bootmain.c，了解bootloader如何加载ELF文件。通过分析源代码和通过qemu来运 行并调试bootloader&OS，**

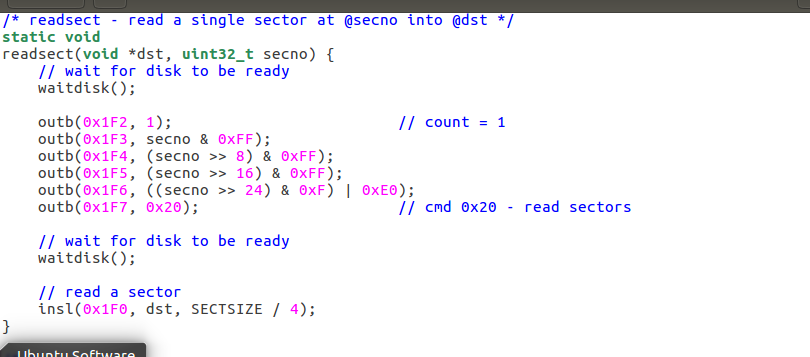
**bootloader如何读取硬盘扇区的？**

**bootloader是如何加载ELF格式的OS**

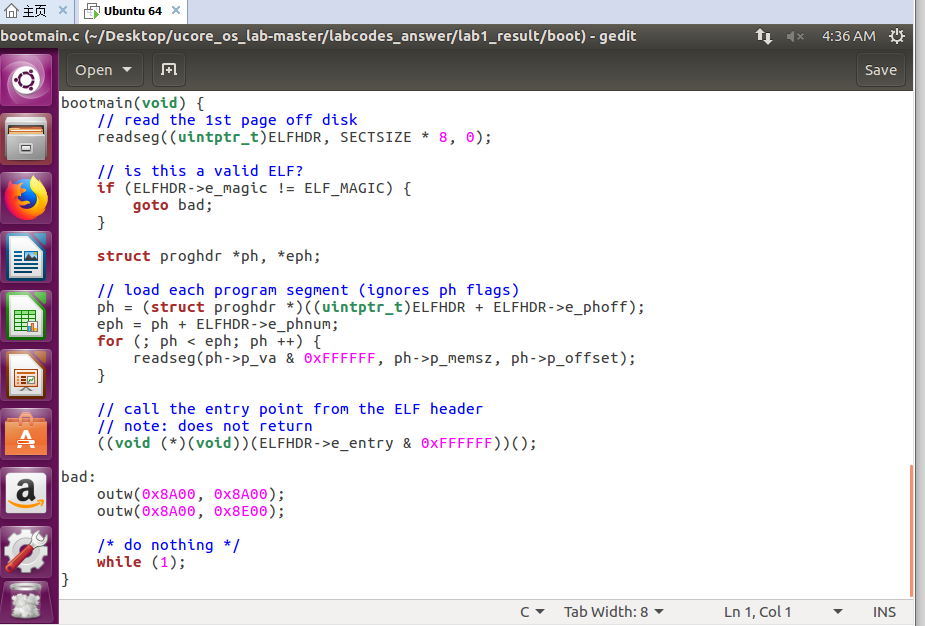
**bootloader读取硬盘扇区**

根据上述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来每次读出一个扇区 。读取扇区它用到了in b

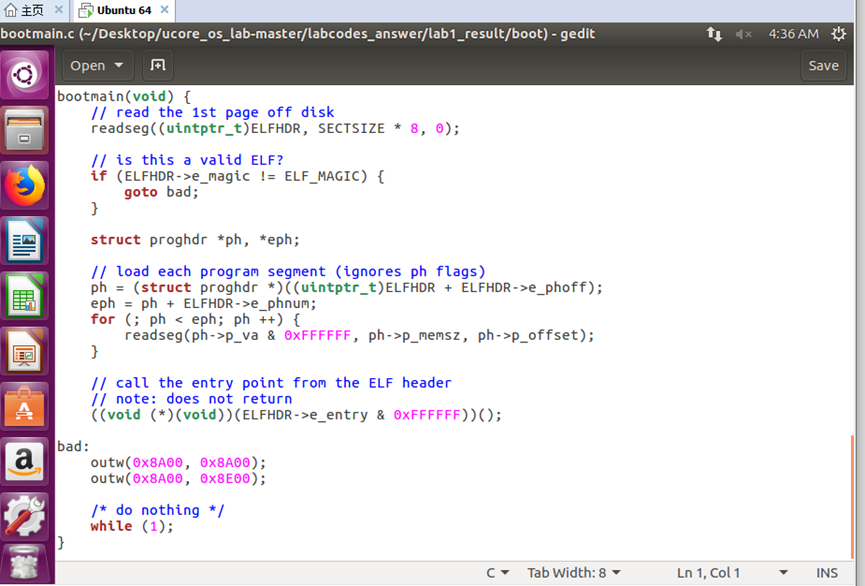
和out b这种机器指令。inb和outb的实现都是在lab0中介绍到过的内联汇编来实现的，它采取了一种IO空间的寻址方式，能够把外设的数据给读到内存中来，这也是一种X86里面的寻址方式。Bootloader把相应的扇区给读进来去进一步的分析

bootloader加载 ELF格式的 OS

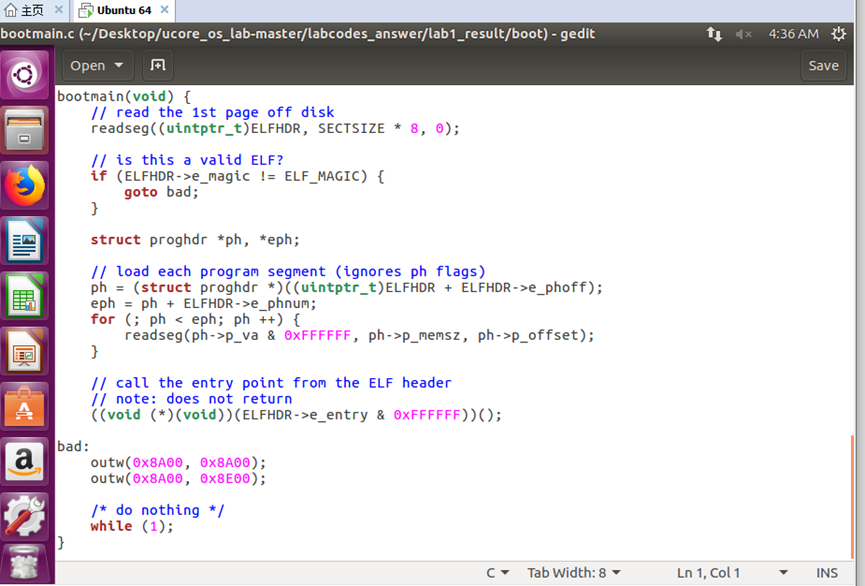
读取完磁盘之后，开始加载ELF格式的文件。



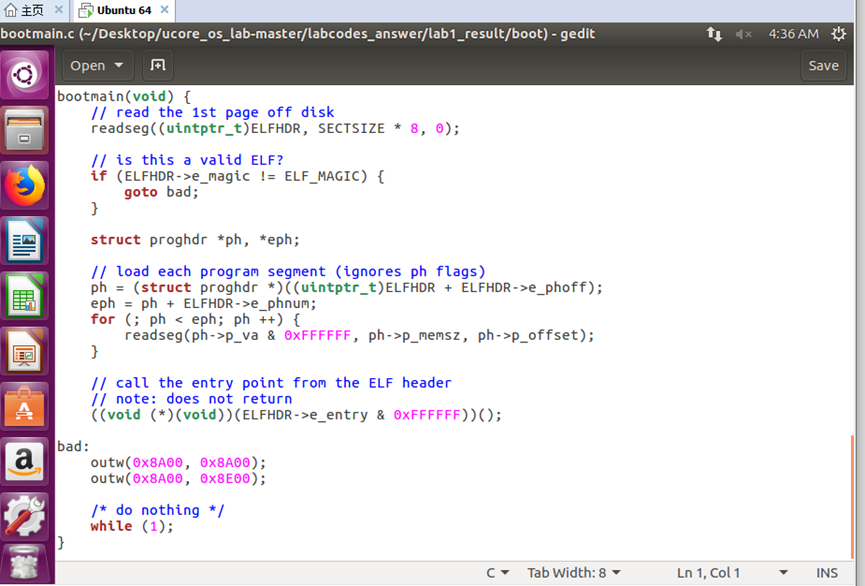
首先判断是不是ELF



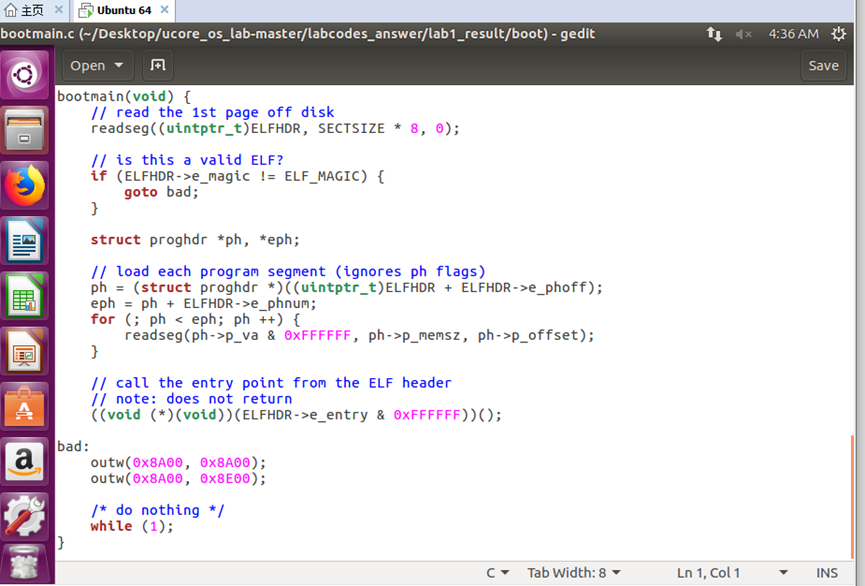
ELF头部有描述ELF文件应加载到内存什么位置的描述表，这里读取出来将之存入ph



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 （需要编程）**

**我们需要在lab1中完成kdebug.c中函数print\_stackframe的实现，可以通过函数 print\_stackframe来跟踪函数调用堆栈中记录的返回地址。**

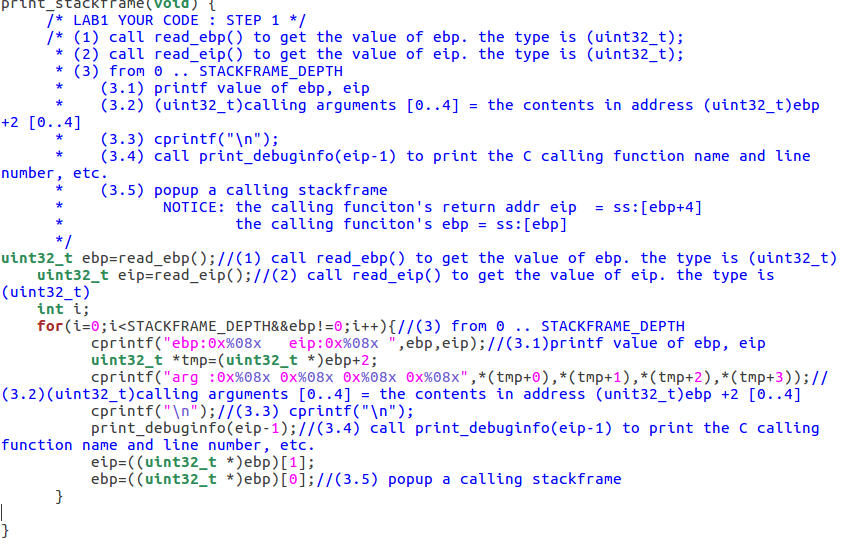
**请完成实验，看看输出是否与上述显示大致一致，并解释最后一行各个数值的含义。**

函数堆栈的原理

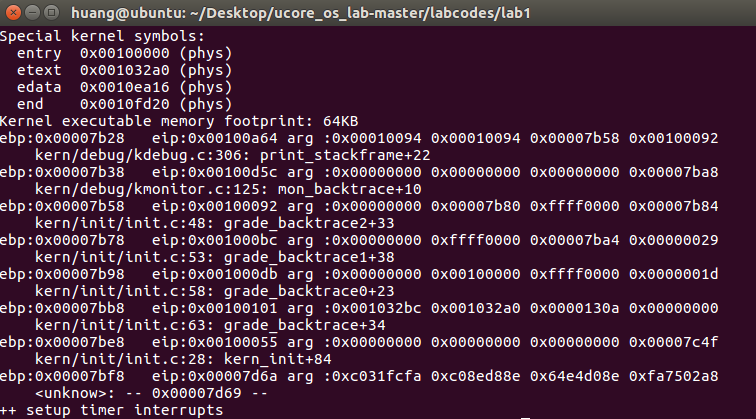
理解函数堆栈最重要的两点是：栈的结构，以及EBP寄存器的作用。

一个函数调用动作可分解为零到多个 PUSH指令（用于参数入栈）和一个 CALL 指令。CALL 指令内部暗含了一个将返回地址压栈的动作，由硬件完成。

print\_stackframe函数的实现



实验结果



最后一行是ebp:0x00007bf8 eip:0x00007d6a args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8，共有ebp，eip和args三类参数，下面分别给出解释。

ebp:0x0007bf8：

此时ebp的值是kern\_init函数的栈顶地址

eip:0x00007d6e：

eip的值是kern\_init函数的返回地址，也就是bootmain函数调用kern\_init对应的指令的下一条指令的地址。这与obj/bootblock.asm相符合。

args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8

一般来说，args存放的4个dword是对应4个输入参数的值。但这里比较特殊，由于bootmain函数调用kern\_init并没传递任何输入参数，并且栈顶的位置恰好在boot loader第一条指令存放的地址的上面，而args恰好是kern\_int的ebp寄存器指向的栈顶往上第2~5个单元，因此args存放的就是boot loader指令的前16个字节。

**练习6：完善中断初始化和处理 （需要编程）**

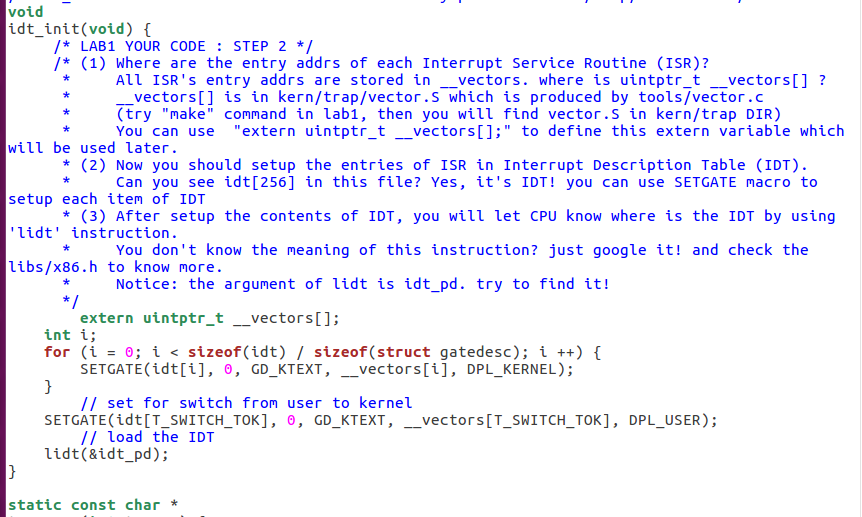
**请完成编码工作和回答如下问题：**

**1. 中断描述符表（也可简称为保护模式下的中断向量表）中一个表项占多少字节？其中哪 几位代表中断处理代码的入口？**

一个表项占八个字节。入口地址为：gd\_off\_31\_16 << 16 + gd\_off\_15\_0 ;



**2. 请编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt\_init。在idt\_init函数中， 依次对所有中断入口进行初始化。使用mmu.h中的SETGATE宏，填充idt数组内容。每个 中断的入口由tools/vectors.c生成，使用trap.c中声明的vectors数组即可。**



代码首先引入中断处理函数的入口地址\_\_vectors[],这个变量在vector.s里面生成，然后初始化idt中断描述符表，最后根据提示用lidt函数告知cpu IDT表的位置。setgate这个函数的作用是设置正确的interrupt/trap gate 描述符。注意需要对T\_SWITCH\_TOK的发生时机是在用户空间的，所以对应的dpl需要修改为DPL\_USER。lidt将idt的首地址和size装进idtr寄存器。

**3. 请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写trap函数中 处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print\_ticks子程序，向 屏幕上打印一行文字”100 ticks”。**

