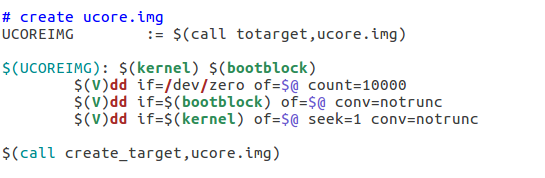
Lab1

**练习1：理解通过make生成执行文件的过程**

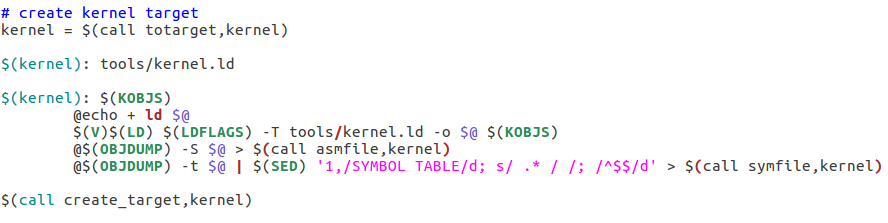
[练习1.1]

1、生成ucore.img需要kernel和bootblock

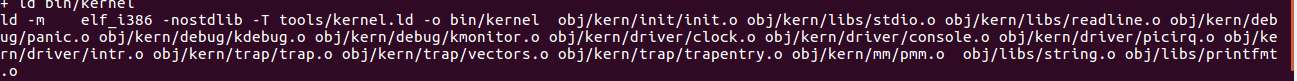


1. 生成kernel

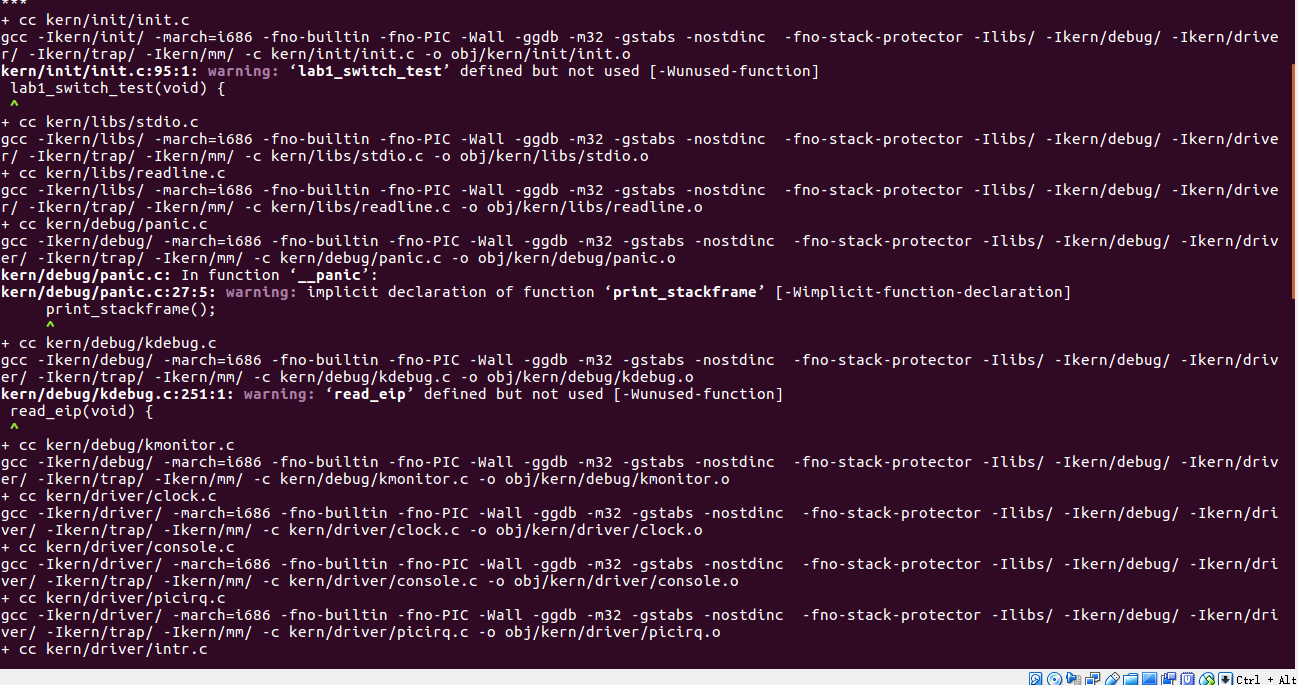
生成kernel代码如下：

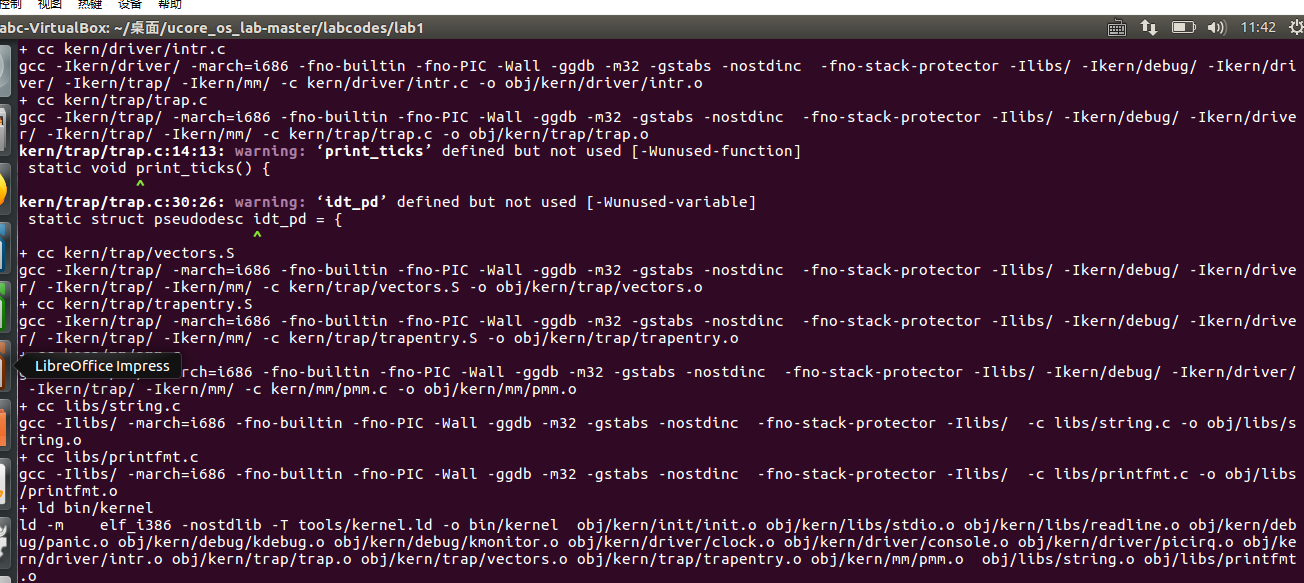


执行make “V=”得到执行的具体命令如下所示



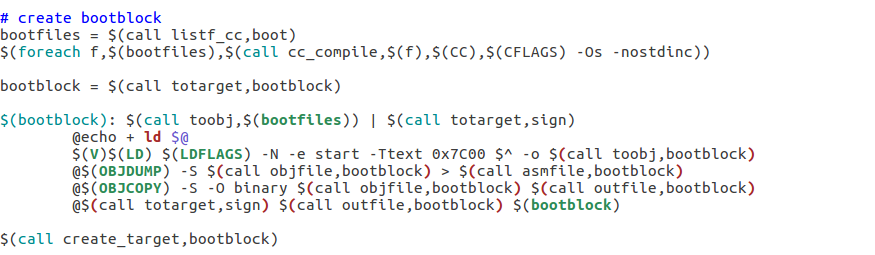
要生成kernel，需要用GCC将kern目录下所有的.c文件全部编译成.o文件，如下



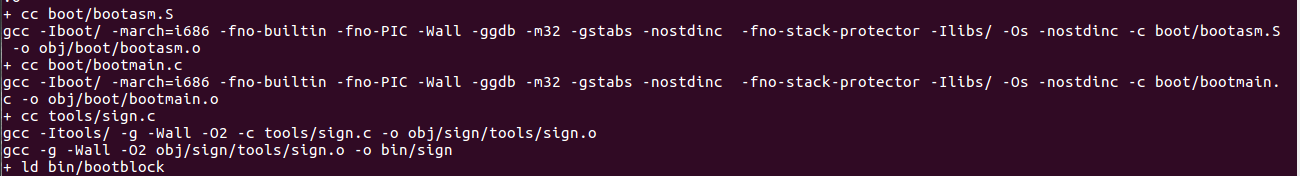


1. 生成bootblock

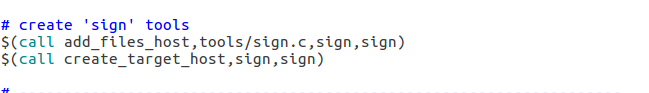
生成bootblock代码如下：



根据make “V=”的结果，要生成bootblock,首先要生成bootasm.o、bootmain.o、sign



生成sign的代码如下：



然后可以生成bootblock

捕获

[练习1.2]

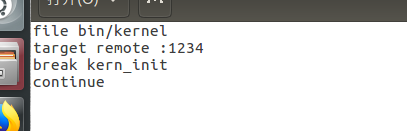
1. 磁盘主引导扇区只有512字节
2. 磁盘最后两个字节为0x55AA

**练习2：使用qemu执行并调试lab1中的软件。**

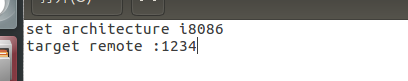
[练习2.1]

1. 默认的gdb需要进行一些额外的配置才能进行qemu的调试任务，qemu和gdb之间使用网络端口1234进行通信。

Lab1/tools/gdbinint的内容如下：



可见，这里是对内核代码进行调试，并且将断电设置在内核代码的入口地址，即kern\_init函数。为了从CPU加电后的第一条指令开始调试，修改内容如下：

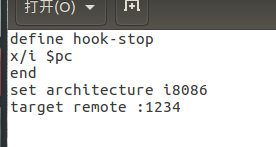


1. 通过上一个步骤gdb可以连接到qemu，此时qemu会进入停止状态，听从gdb的命令。另外，可能需要qemu在一开始便进入等待模式，所以使用make debug开始系统的运行。

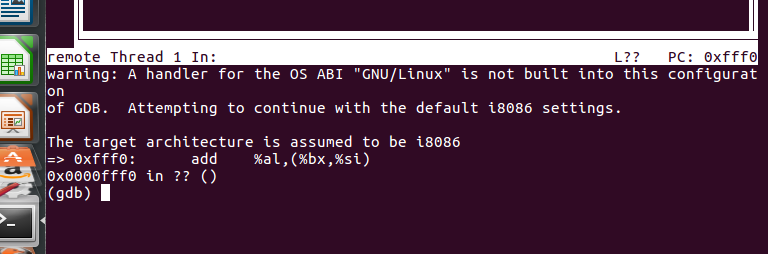
在lab1目录下执行make debug，弹出qemu和gdb窗口：



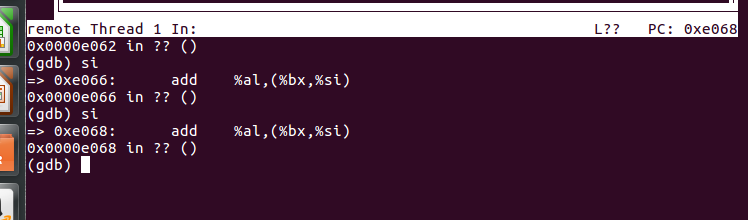
没有显示反汇编代码，再次修改Lab1/tools/gdbinint的内容如下：



再次运行，结果如下：

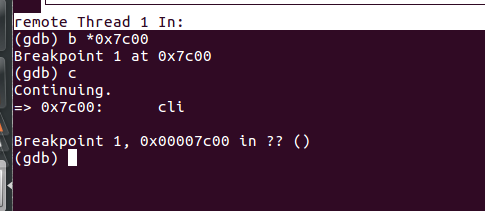


用si命令单步执行机器指令，单步跟踪BIOS：



[练习2.2]

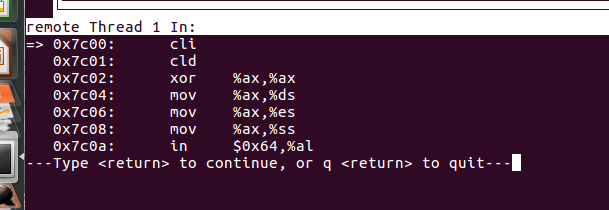
在gdb中输入b \*0x7c00设置断点

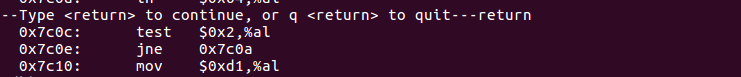


程序运行到0x7c00停止，并指向汇编指令cli

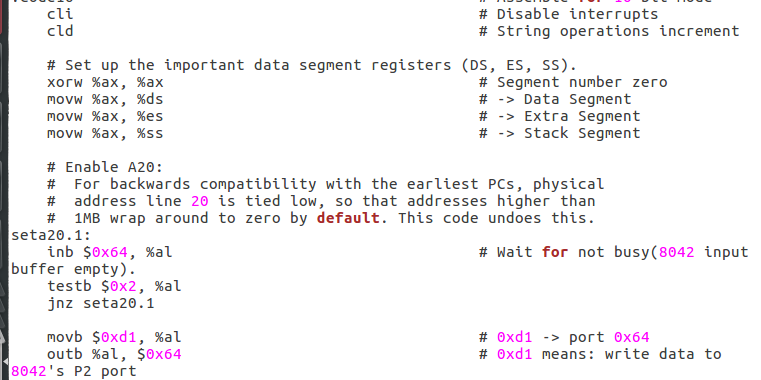
[练习2.3]

输入x /10i $pc 查看后面的指令

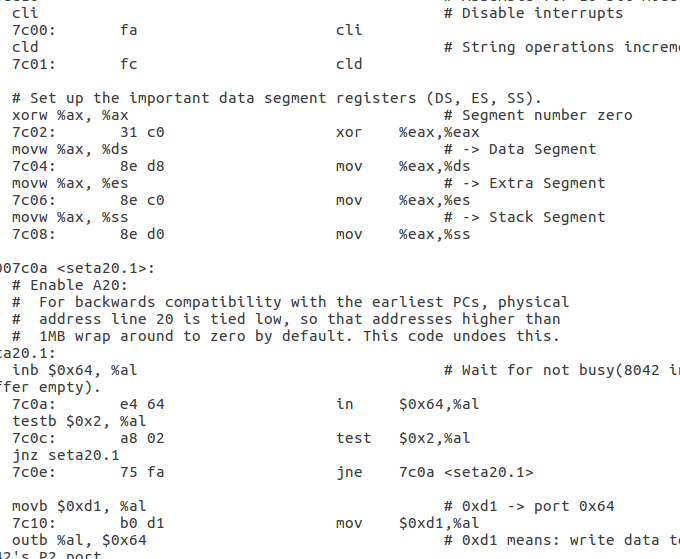




与bootasm.S比较



与bootblock.asm比较



可以看出，指令是一致的

[练习2.4]

1. 将gdbinit改回原来的命令
2. 执行make debug
3. 设置断点，输入命令b cons\_init



可以看到，遇到断点，在cons\_init函数处停止

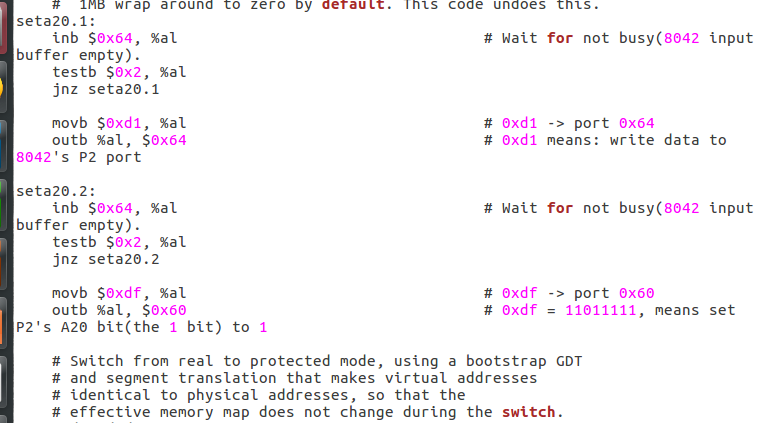
**练习3：分析bootloader进入保护模式的过程**

[练习3.1]

为何开启A20，以及如何开启A20？

为什么：在实模式下，我们可以访问超过1MB的空间，但是我们只希望访问1MB以内的内存空间。为了解决这个问题，CPU中添加了一个可控制A20地址线的模块，通过这个模块，我们在实模式下将第20bit的地址线限制为0，这样CPU就不能访问超过1MB的空间了。进入保护模式后，我们再通过这个模块解除对A20地址线的限制，这样我们就能访问超过1MB的内存空间了。

默认情况下，A20地址线是关闭的，因此在进入保护模式前，需要开启A20地址线。具体代码如下：



如何开启：

1、等待8042Input buffer为空

2、发送Write 8042 Output Port命令到8042 Input buffer

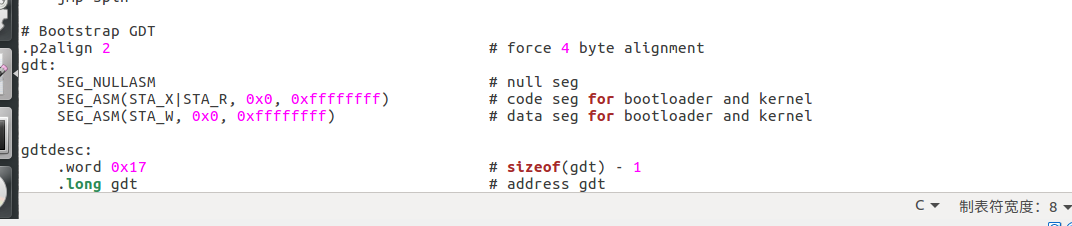
3、等待8042Input buffer为空

4、将A20置1，写入8042Input buffer

[练习3.2]

如何初始化GDT表？

具体代码如下：

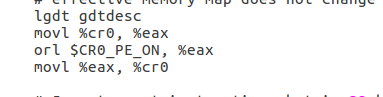


可以看出GDT表的初始化为全段，此时段偏移量为物理地址。

[练习3.3]

如何使能和进入保护模式？

开启A20，初始化GDT表后，将控制寄存器CR0的PE置为1，具体代码如下：

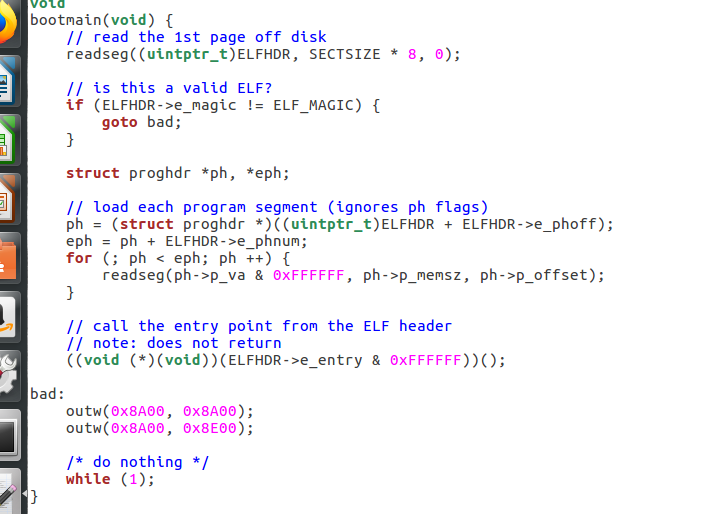


**练习4：分析bootloader加载ELF格式的OS的过程**

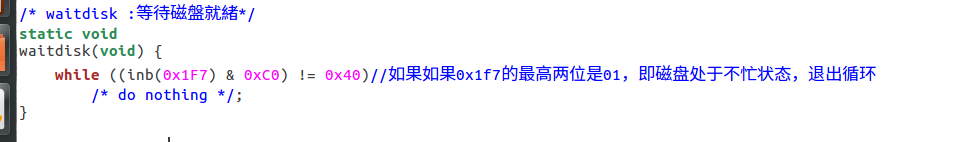
[练习4.1]

读一个扇区的流程如下：

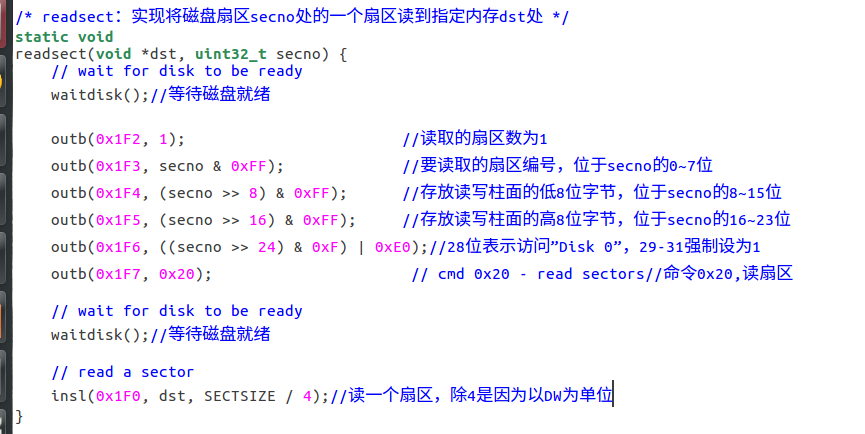
1. 等待磁盘准备好
2. 发出读取扇区的命令
3. 等待磁盘准备好
4. 把磁盘扇区数据读到指定内存



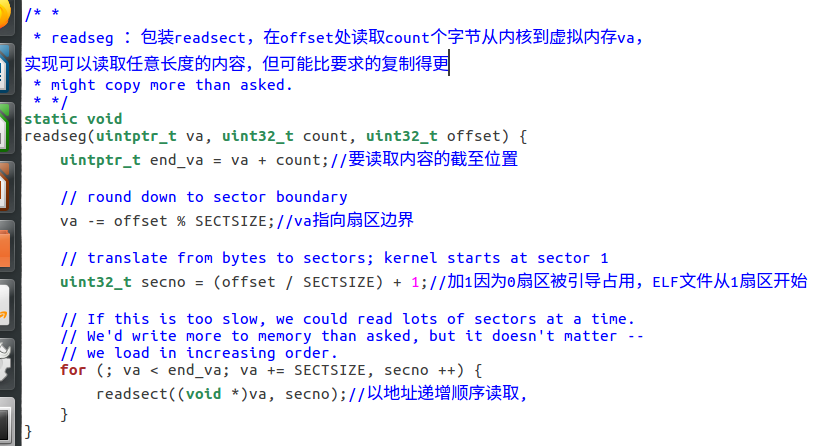
根据bootmain函数可以看出，waitdisk函数用来实现等待磁盘就绪的功能，具体代码如下：（具体分析见注释）



readsect函数实现将磁盘扇区secon处的一个扇区读到指定内存dst处。具体代码如下：（具体分析见注释）

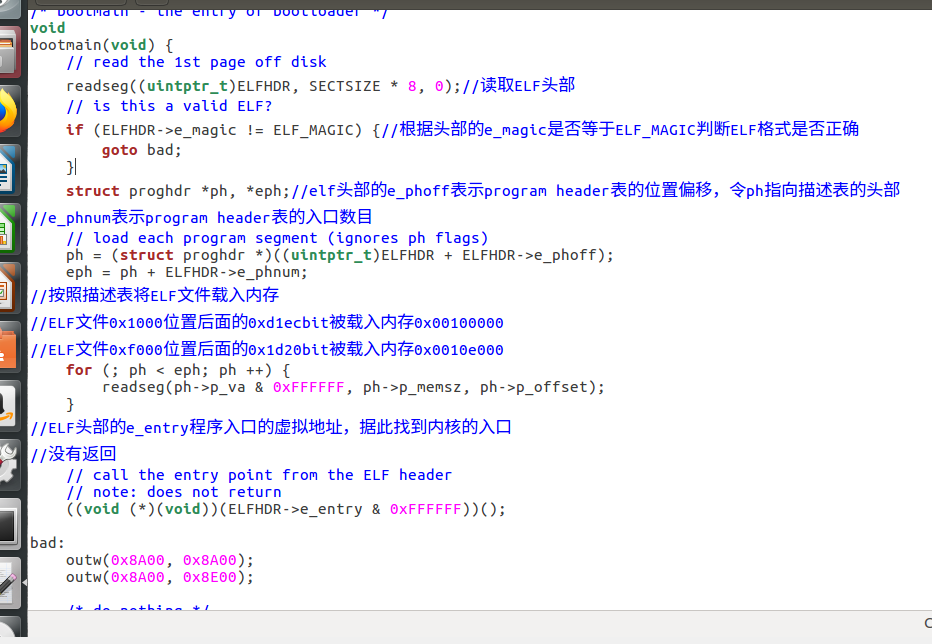


readseg函数包装readsect，在offset处读取count个字节从内核到虚拟内存。具体代码如下：（具体分析见注释）



[练习4.2]

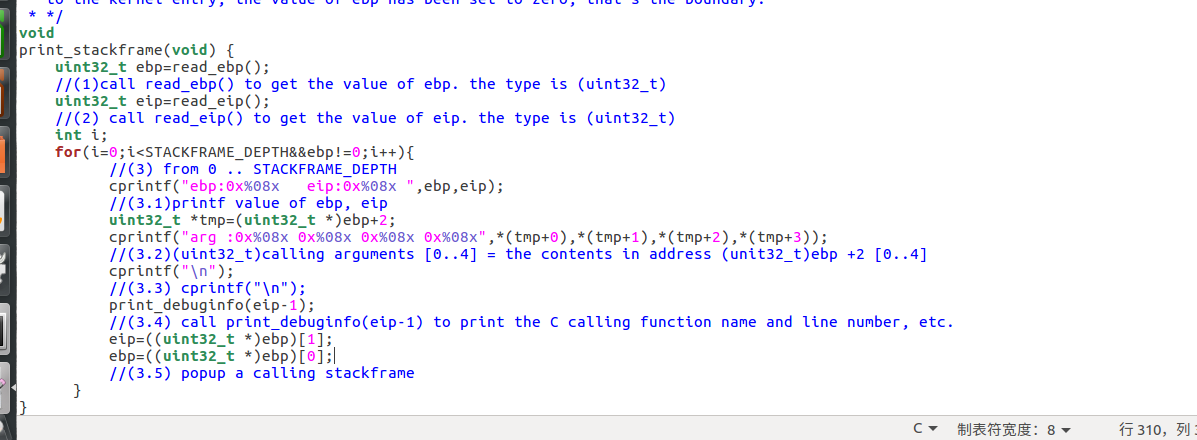
Bootmain函数，作为bootloader的入口，从硬盘读取8个扇区到内存0x10000处，并将其强制转换为elfhdr使用。然后根据头部的e\_magic是否等于ELF\_MAGIC判断ELF格式是否正确，之后根据描述表中的program header表的偏移量分别把程序段的程序读到内存中。具体代码如下：



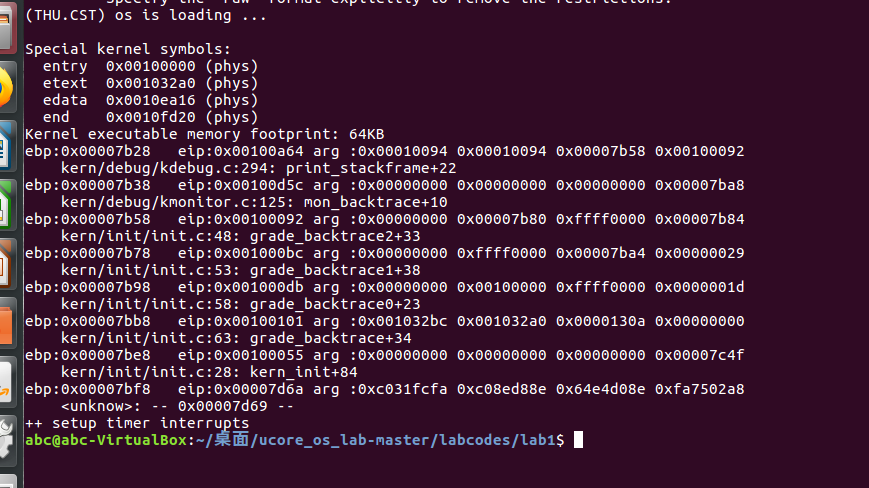
**练习5：实现函数调用堆栈跟踪函数**

[练习5]

按照注释的提示编写代码如下：



结果截图如下：



最后一行各个数值解释：

对应于bootmain.c中的bootmain函数，即第一个调用堆栈的函数。

bootloader设置的堆栈从0x7c00开始，然后调用call bootmain，向栈中压入参数。

edp=0x00007bf8,eip=0x00007d6a，压入的参数为：0xc031fcfa,0xc08ed88e,0x64e4d08e,0xfa7502a8

**练习6：完善中断初始化和处理**

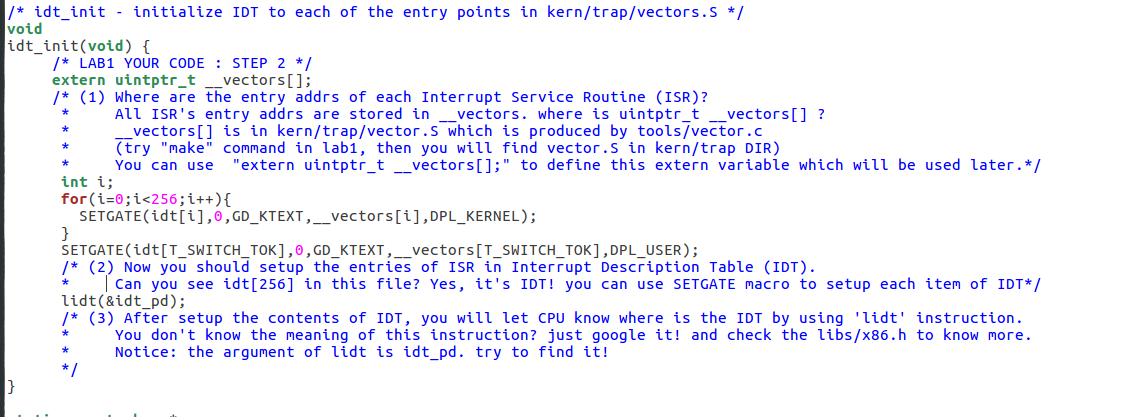
[练习6.1]

一个表项占8个字节。

通过段选择子获得段基址，加上段内偏移量可得到中断处理代码的入口。其中16~31位位段选择子，0~15位、48~63位为偏移量

[练习6.2]

按照注释提示，编写代码如下：



分析：

ucore启动后，通过idt\_init函数初始化IDT表，就是初始化每个中断向量对应的中断处理函数的段选择子、偏移量和属性。

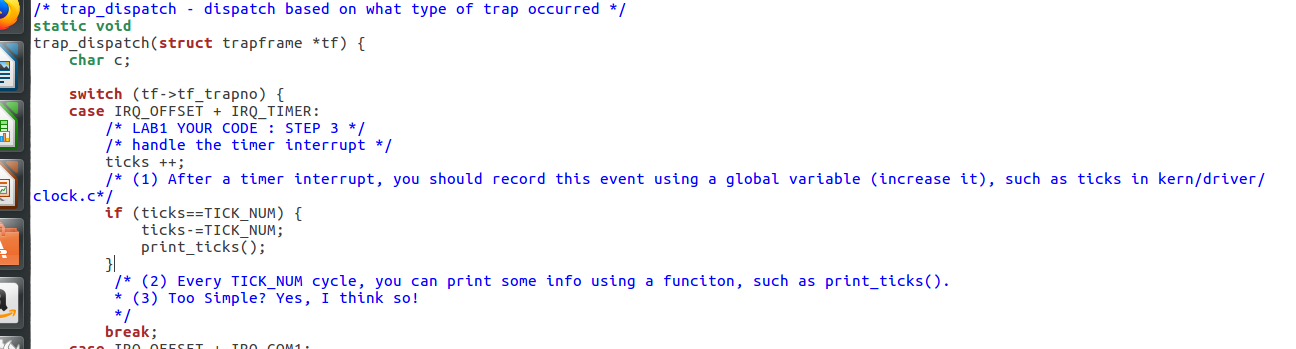
vectors中存储了中断处理程序的入口程序和入口地址。vectors定义在vector.S文件中，通过一个工具程序vector.c生成。由vector.S文件开头可知，中断处理函数属于.text的内容，所以中断处理函数的段选择子选择子GD\_KTEXT。

使用mmu.h中的SETGATE宏来填充idt数组的内容。

完成初始化后，执行lidt命令将IDT表的起始地址加载到IDTR寄存器中。

[练习6.3]

按照注释，完善代码如下：



分析：可以用kern/driver/clock.c中的全局变量ticks记录当前中断次数，每次发生中断ticks加一，如果ticks=100，就调用print\_ticks函数打印信息，并将ticks置为0

结果如下：

