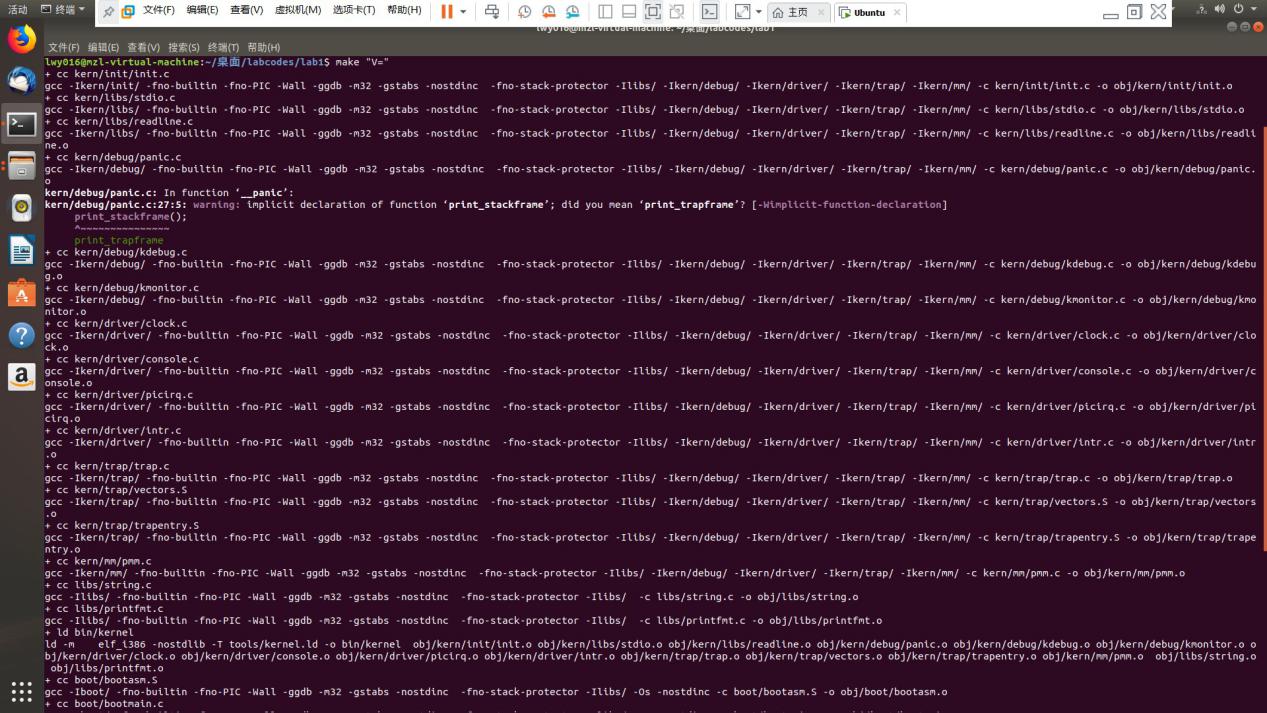
Ucore lab1实验报告

练习1：理解通过make生成执行文件的过程

执行 make "V="， 观察生成 ucore.img 的过程



1.操作系统镜像文件ucore.img是如何一步一步生成的？

编译16个内核文件，构建出内核bin/kernel 生成 bin/bootblock 引导程序

编译bootasm.S，bootmain.c，链接生成obj/bootblock.o

编译sign.c生成sign.o工具

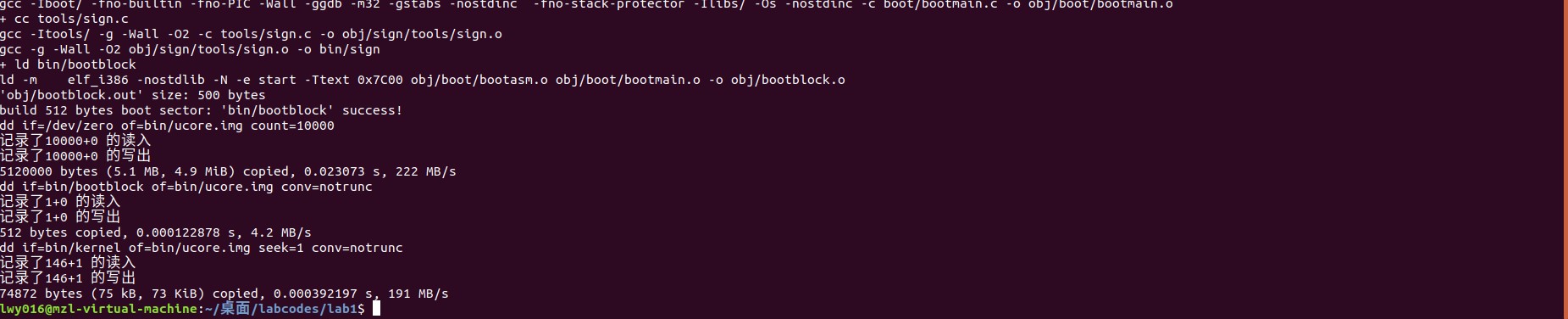
使用sign.o工具规范化bootblock.o，生成bin/bootblock引导扇区 生成 ucore.img 虚拟磁盘

dd初始化ucore.img为5120000 bytes，内容为0的文件

dd拷贝bin/bootblock到ucore.img第一个扇区

dd拷贝bin/kernel到ucore.img第二个扇区往后的空间

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？



规范的硬盘引导扇区的大小为512字节，硬盘结束标志位55AA

练习2：使用qemu执行并调试lab1中的软件

从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

在初始化位置0x7c00设置实地址断点,测试断点正常。

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。

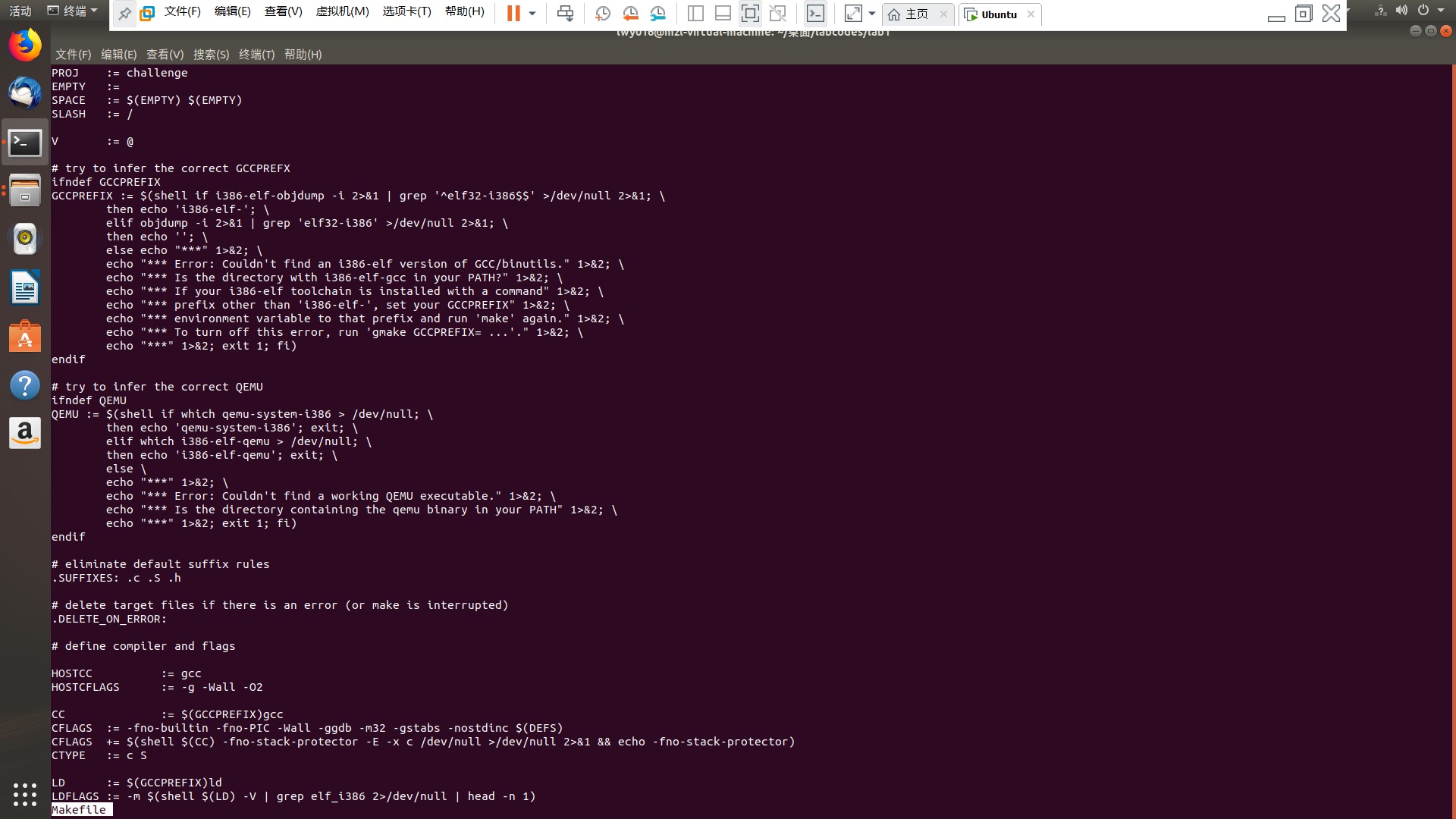
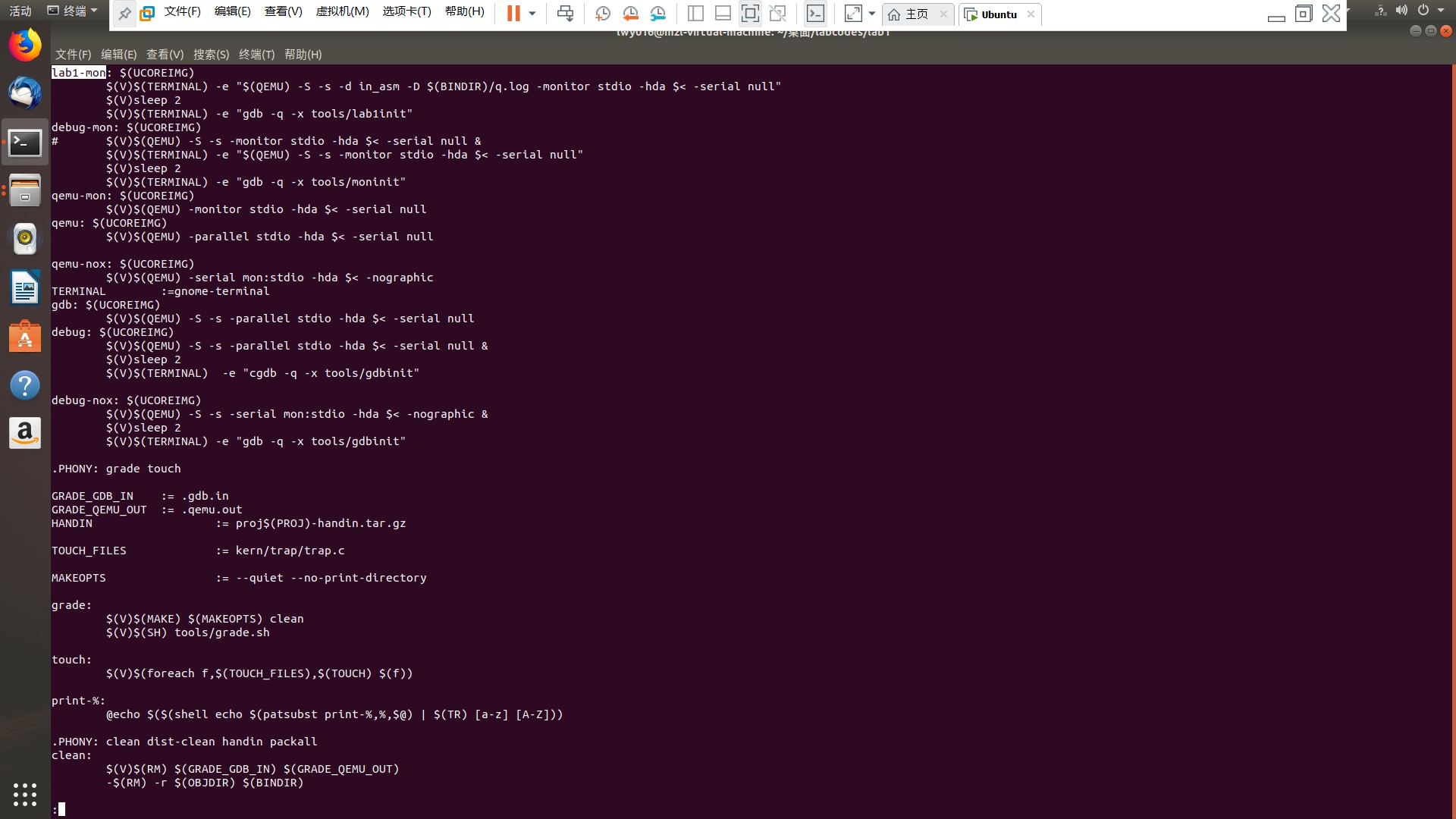
找一个bootloader或内核中的代码位置，设置断点并进行测试。

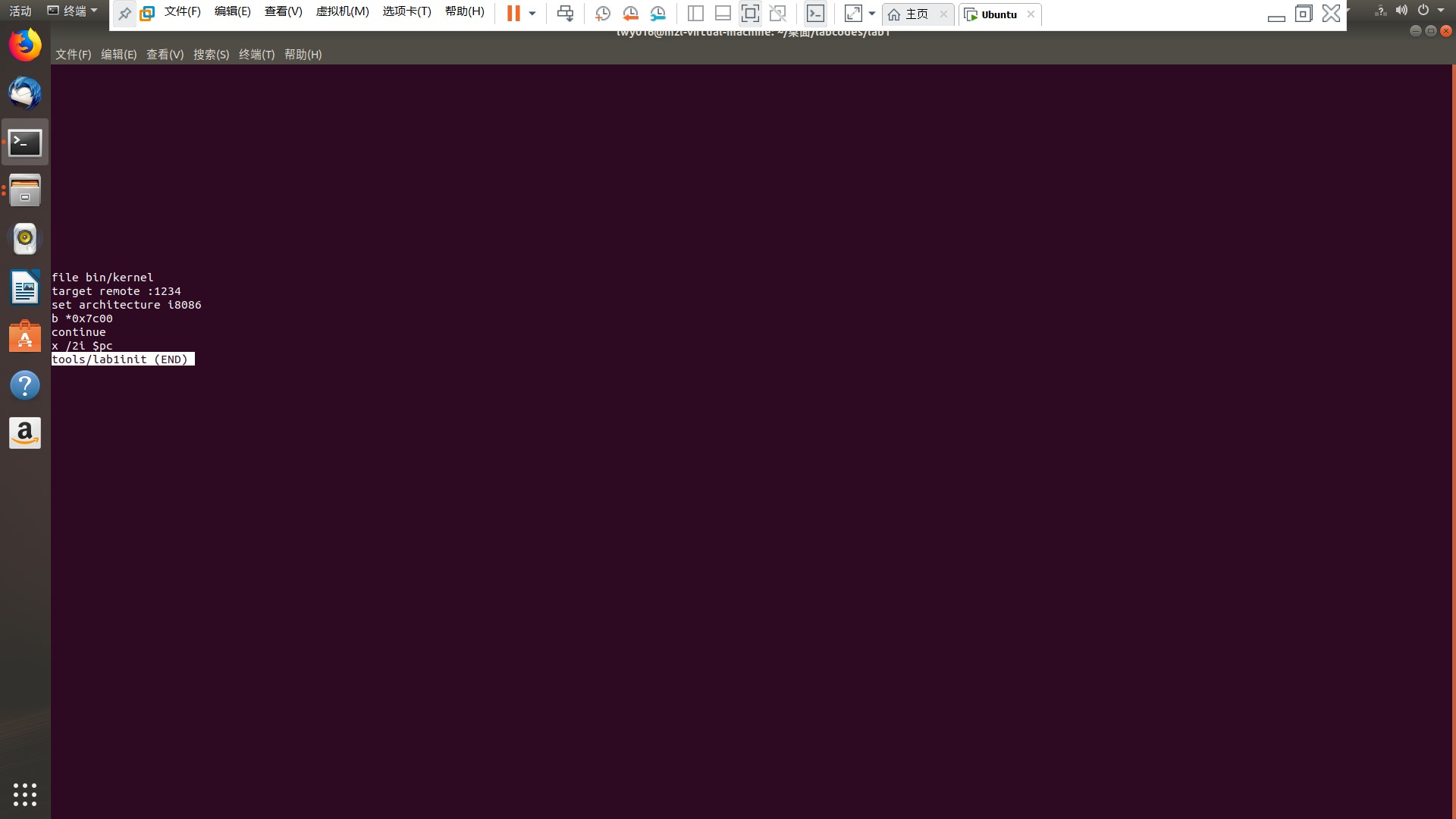
执行less Makefile命令去浏览 Makefile 文件中的内容，过 /lab1-mon 去定位到相应行数的代码

这条命令干了两件事情：

第一个是让 qemu 把它执行的指令给记录下来，放到 q.log 这个地方

第二个是和 gdb 结合来调试正在执行的 Bootloader

* 
* 

1. 看初始化执行指令中都有哪些内容，使用less tools/lab1init
2. 

第一条指令是加载 bin/kernel。（加载符号信息，事实上是ucore的信息）。第二条指令是与 qemu 进行连接，通过这个TRP进行连接

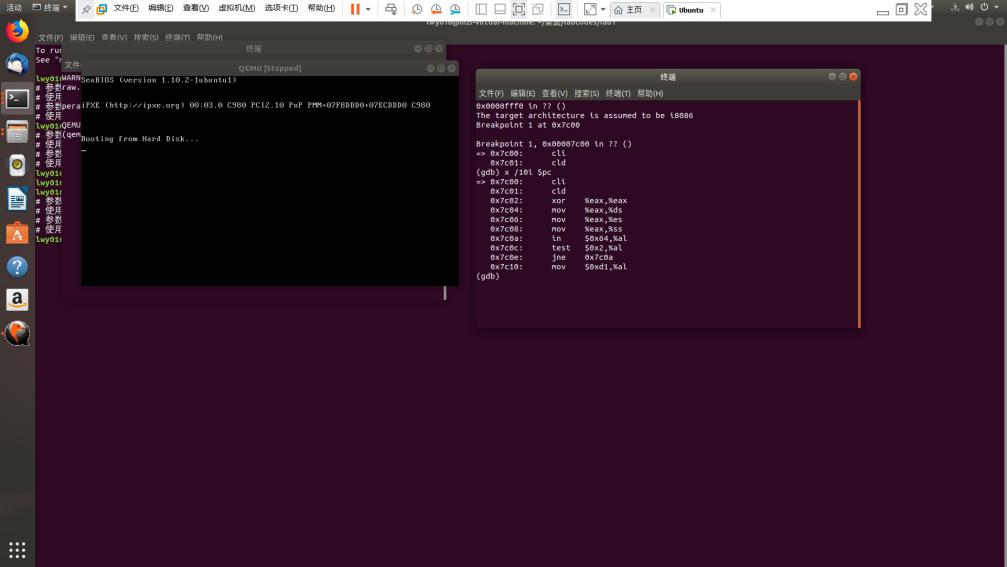
刚开始的时候，BIOS是进入8086的16位实模式方式，一直到0x7c00。在BIOS这个阶段，启动，最后把Bootloader加载进去，把控制权交给Bootloader，那么Bootloader第一条指令就是在0x7c00处，所以我们在这个地方设置一个断点，break 0x7c00

然后让这个系统继续运行，它会在这个断点处停下来，可以把相应的这个指令给打印出来。

最后一条指令的意思是把PC(也就是EIP，即指令指针寄存器)，它存在当前正在执行这个指令的地址，那么x是显示的意思，／2i是显示两条，i是指令。

执行 bootloader第一条指令make lab1-mon

可以看到，qemu 已经启动起来了。但是它断下来了，我们可以看到断点箭头指向 0x7c00 处。我们还可以显示更多的条数信息，比如我们可以执行 x /10i $pc ，可以把当前的10条指令都显示出来。



可以将执行的汇编代码与bootasm.S 和 bootblock.asm 进行比较，看看二者是否一致。

练习3：分析bootloader进入保护模式的过程

为何开启A20，以及如何开启A20

如何初始化GDT表

如何使能和进入保护模式

Bootloader 完成了一些最基本的功能，比如它能够把80386的保护模式给开启，使得现在的软件进入了一个32位的寻址空间，这就是我们的寻址方式发生了变化。事实上，A20就是第21根线，用来控制是否允许对 0x10FFEF 以上的实际内存寻址。称为A20 Gate默认情况下，A20地址线是关闭的(20bit以上的地址线限制为0)，因此在进入保护模式(需要访问超过1MB的内存空间)前，我们需要开启A20地址线(20bit以上的地址线可为0或者1)。具体代码如下：

seta20.1:

inb $0x64, %al # 等待8042键盘控制器不忙 testb $0x2, %al

jnz seta20.1

movb $0xd1, %al

outb %al, $0x64

seta20.2:

inb $0x64, %al # 等待8042键盘控制器不忙

testb $0x2, %al

jnz seta20.2

movb $0xdf, %al # 打开A20

outb %al, $0x60

# 初始化gdt

lgdt gdtdesc

# 进入保护模式

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

# 长跳转

ljmp $PROT\_MODE\_CSEG, $protcseg

.code32

protcseg:

# 设置段寄存器，并建立堆栈

movw $PROT\_MODE\_DSEG, %ax

movw %ax, %ds # -> DS: Data Segment

movw %ax, %es # -> ES: Extra Segment

movw %ax, %fs # -> FS

movw %ax, %gs # -> GS

movw %ax, %ss # -> SS: Stack Segment

# 设置堆栈

movl $0x0, %ebp

movl $start, %esp # 栈顶为0x7c00

# 进入bootmain，不再返回

call bootmain

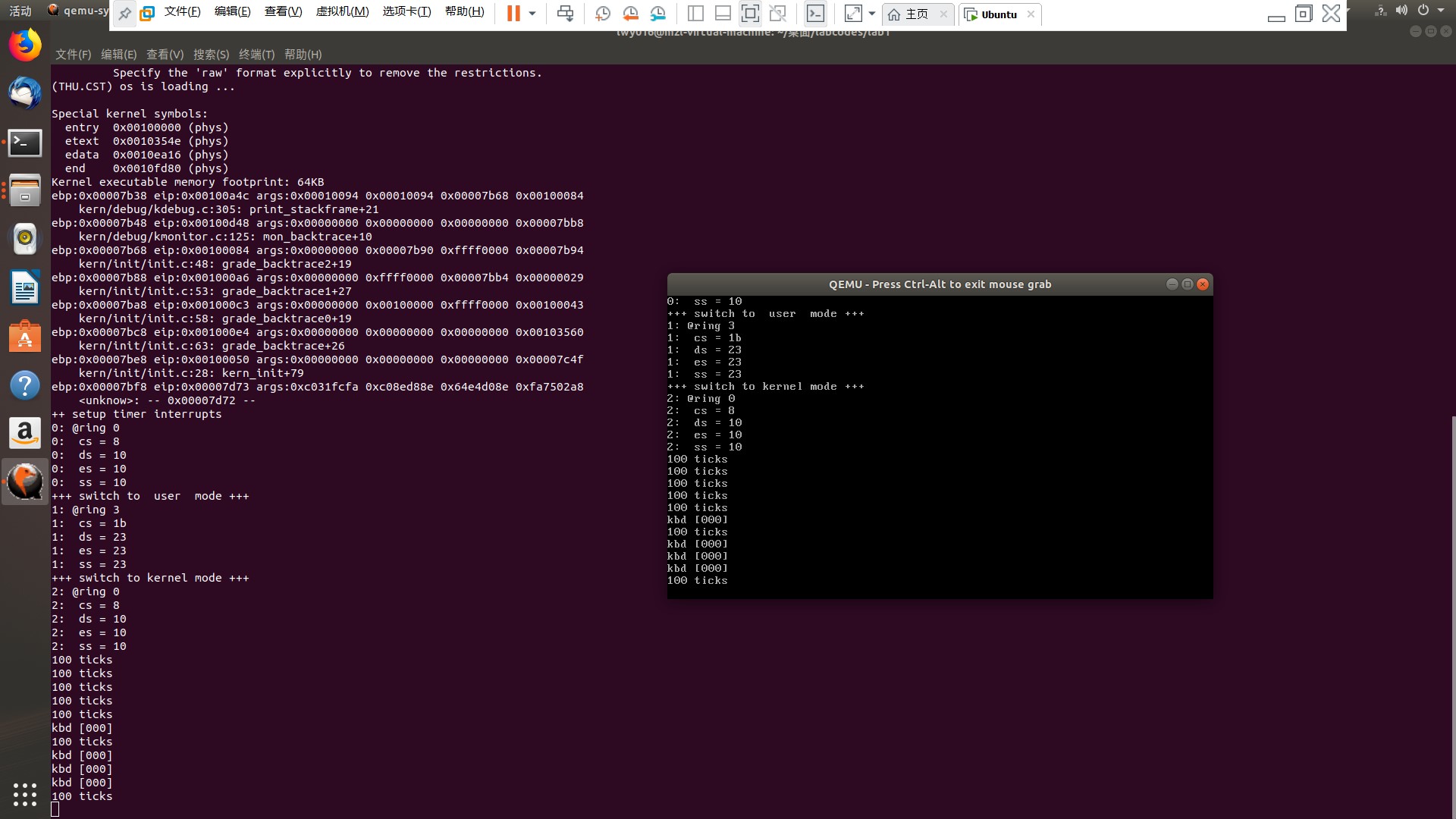
spin:

jmp spin

生成合格的BootLoader主引导扇区



在Protected Mode下，一个重要的必不可少的数据结构就是GDT。想要进行GDT的访问，要根据CPU给的逻辑地址分离出段选择子。利用段选择子查找到对应的段描述符。将段描述符里的Base Address和EIP相加而得到线性地址。



### 练习4：分析bootloader加载ELF格式的OS的过程

bootloader如何读取硬盘扇区的

\* bootloader进入保护模式并载入c程序bootmain

\* bootmain中readsect函数完成读取磁盘扇区的工作，函数传入一个指针和一个uint\_32类型secno，函数将secno对应的扇区内容拷贝至指针处

\* 调用waitdisk函数等待地址0x1F7中低8、7位变为0,1，准备好磁盘

\* 向0x1F2输出1，表示读1个扇区，0x1F3输出secno低8位，0x1F4输出secno的8~15位，0x1F5输出secno的16~23位，0x1F6输出0xe+secno的24~27位，第四位0表示主盘，第六位1表示LBA模式，0x1F7输出0x20

\* 调用waitdisk函数等待磁盘准备好

\* 调用insl函数把磁盘扇区数据读到指定内存

bootloader是如何加载ELF格式的OS

bootloader通过bootmain函数完成ELF格式OS的加载。

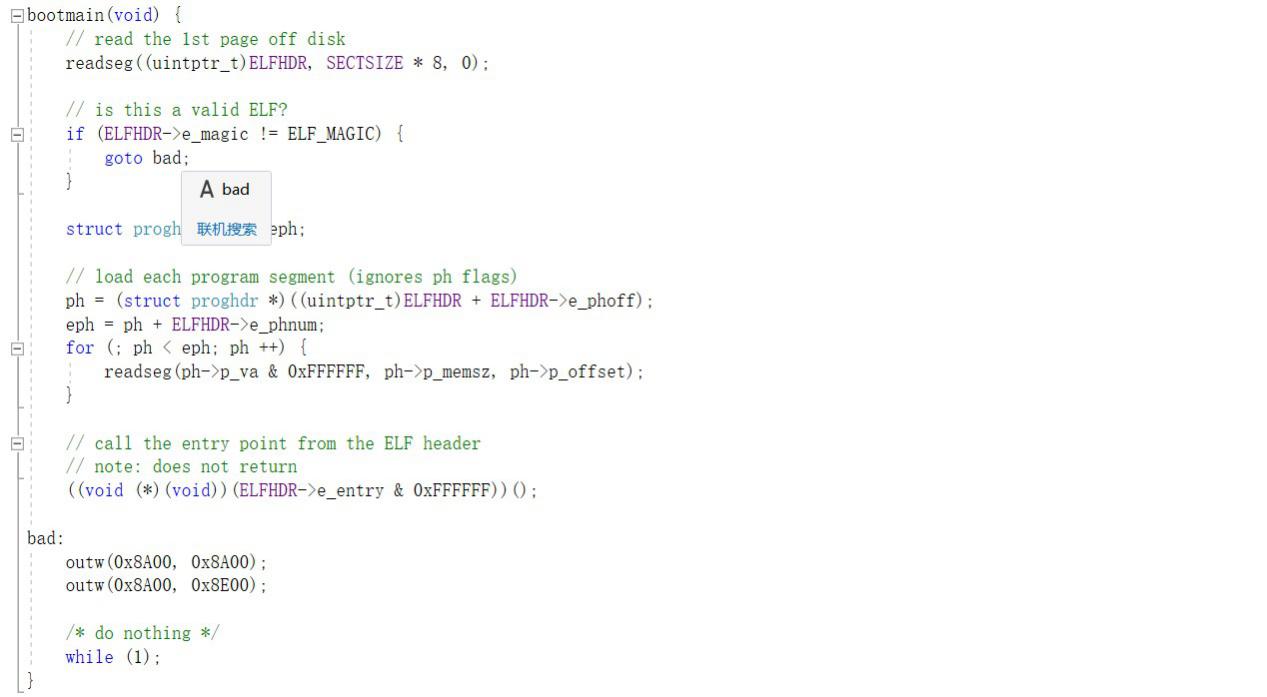
\* 调用readseg函数从kernel头读取8个扇区得到elfher

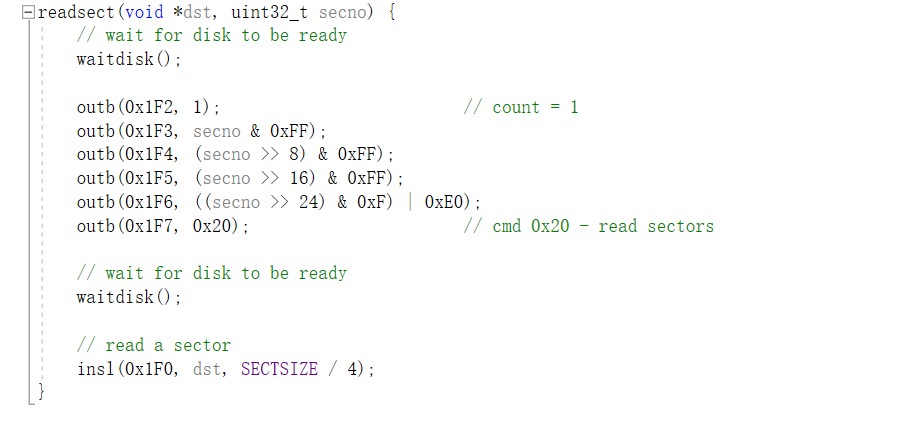
\* 判断elfher的成员变量magic是否等于ELF\_MAGIC，不等则进入bad死循环

\* 相等表明是符合格式的ELF文件，循环调用readseg函数加载每一个程序段

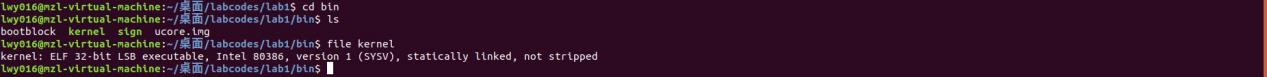
\* 调用elfher的入口指针进入OS

需要调用的两个函数bootmain()与readsect()





加载过程展现



### 练习5：实现函数调用堆栈跟踪函数

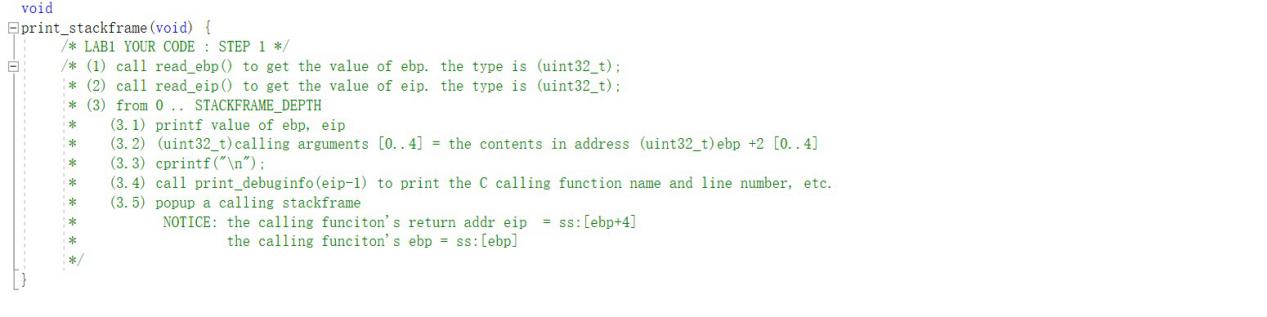
完成kdebug.c中函数print\_stackframe的实现

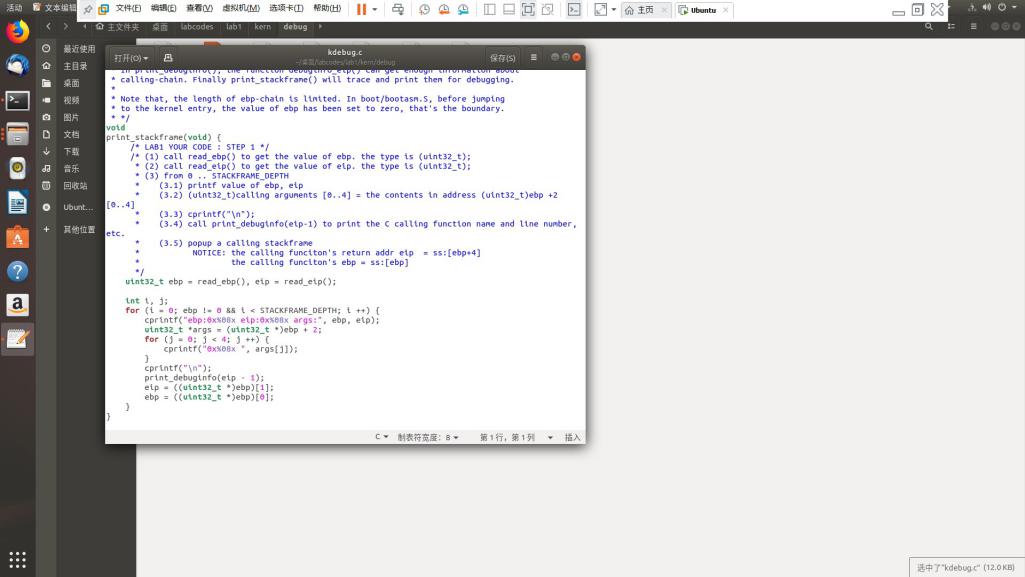
ebp为基址指针寄存器 esp为堆栈指针寄存器(指向栈顶)

ebp寄存器处于一个非常重要的地位，该寄存器中存储着栈中的一个地址（原ebp入栈后的栈顶），从该地址为基准，向上（栈底方向）能获取返回地址、参数值，向下（栈顶方向）能获取函数局部变量值，而该地址处又存储着上一层函数调用时的ebp值

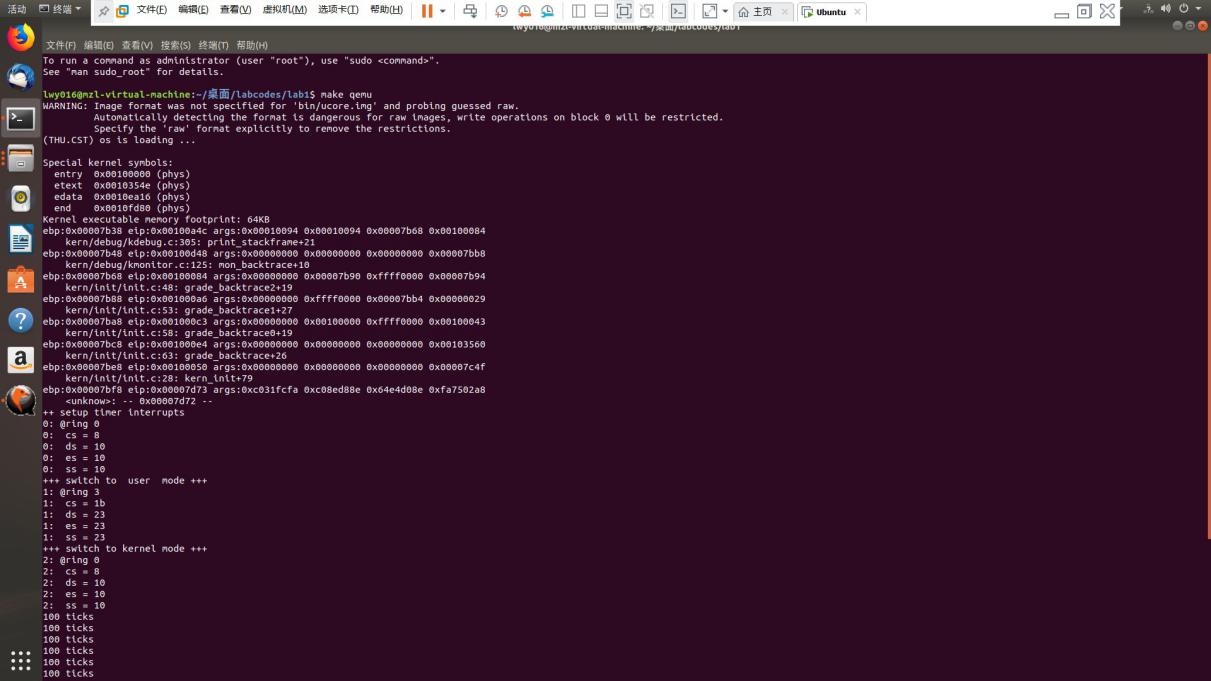
举一个实际的例子查看ebp与esp两个寄存器如何构建出完整的函数栈：

leave等同于movl %ebp, %esp，popl %ebp两条指令





效果如下：



### 练习6：完善中断初始化和处理

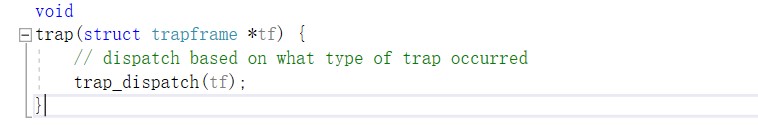
中断描述符表（也可简称为保护模式下的中断向量表）中一个表项占多少字节？其中哪几位代表中断处理代码的入口？

当CPU收到中断时，会查找对应的中断描述符表（IDT），确定对应的中断服务例程。

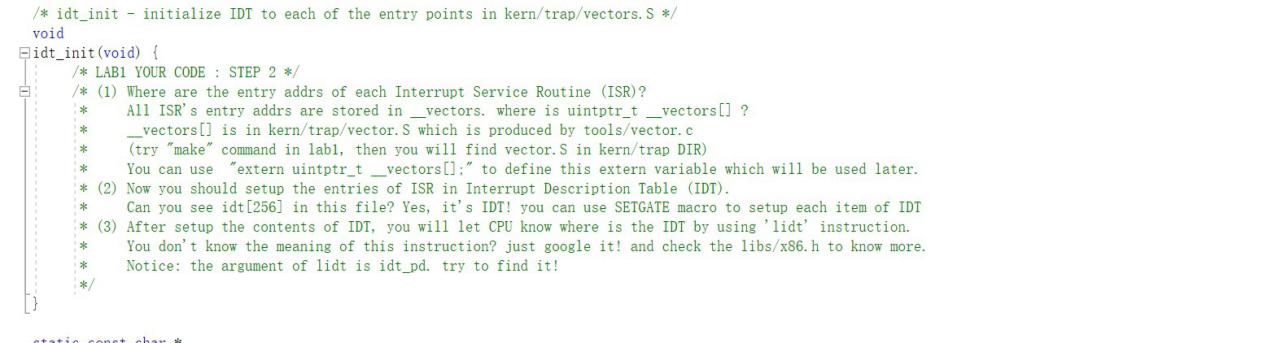
IDT是一个8字节的描述符数组，IDT 可以位于内存的任意位置，CPU 通过IDT寄存器（IDTR）的内容来寻址IDT的起始地址。指令LIDT和SIDT用来操作IDTR。

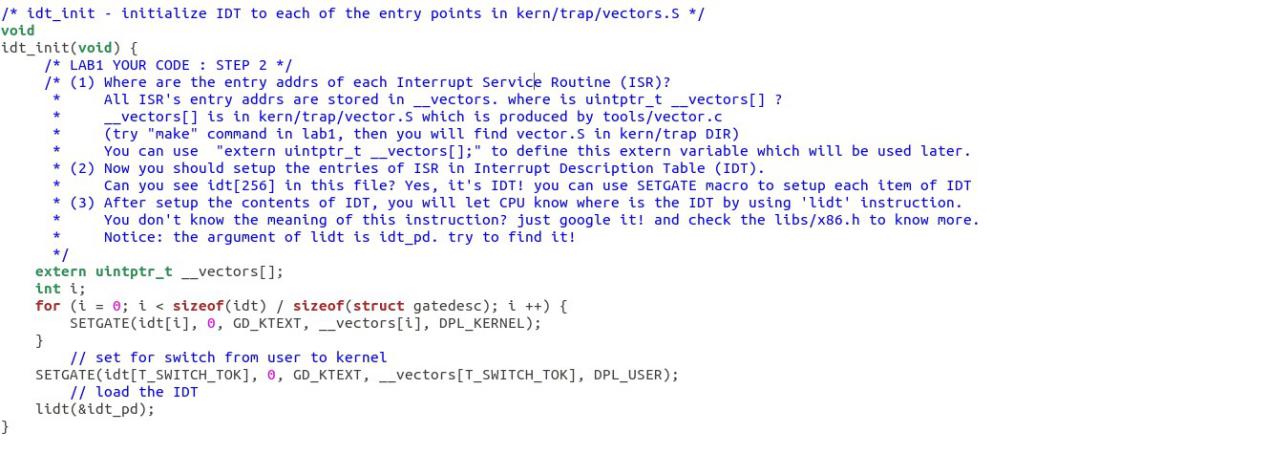
DT的一个表项如下，4个字节分别存储offset的高位地址、段选择子和offset低位地址

需要调用的函数



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。





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”。

