Lab1实验报告

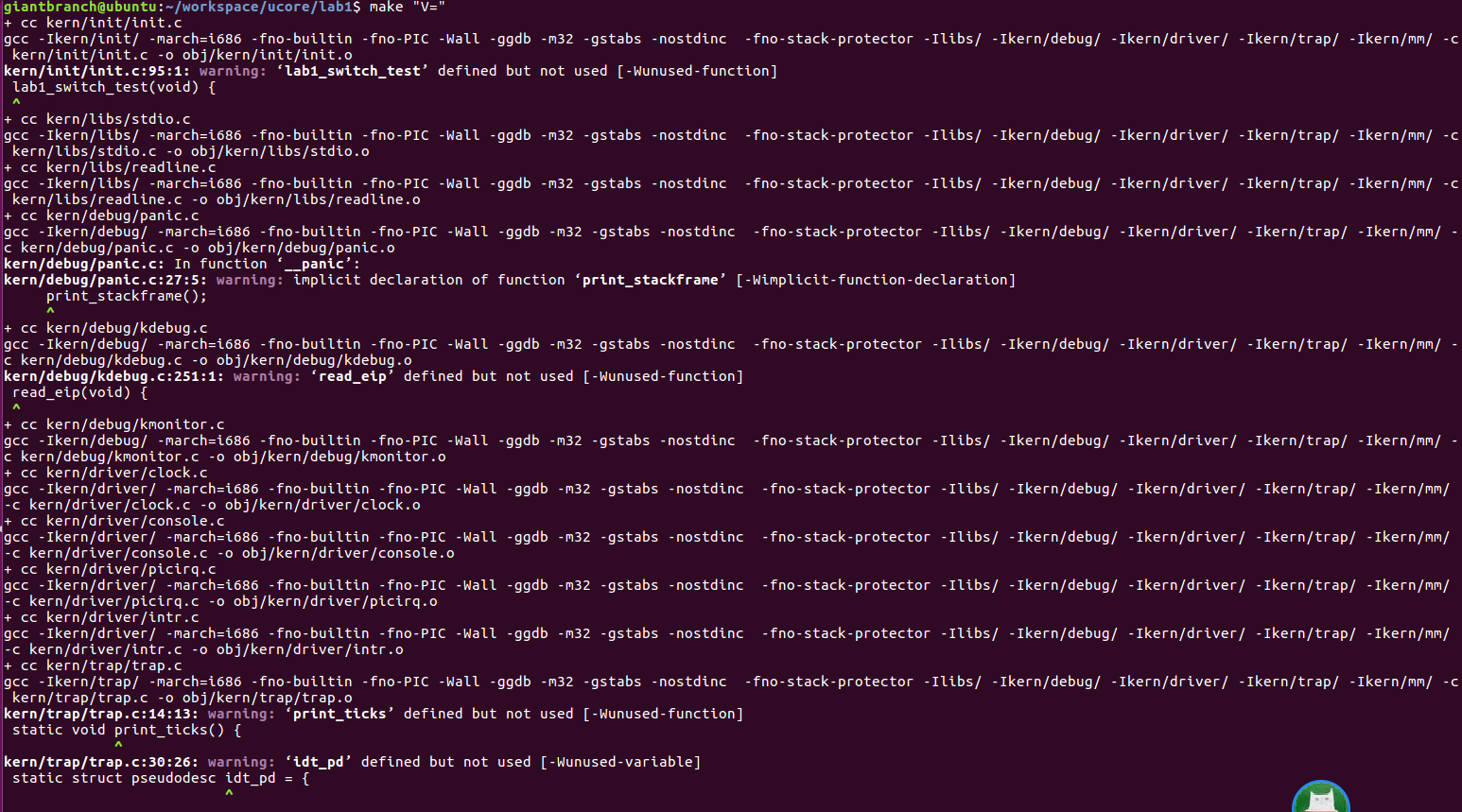
姓名：张沈芊芊

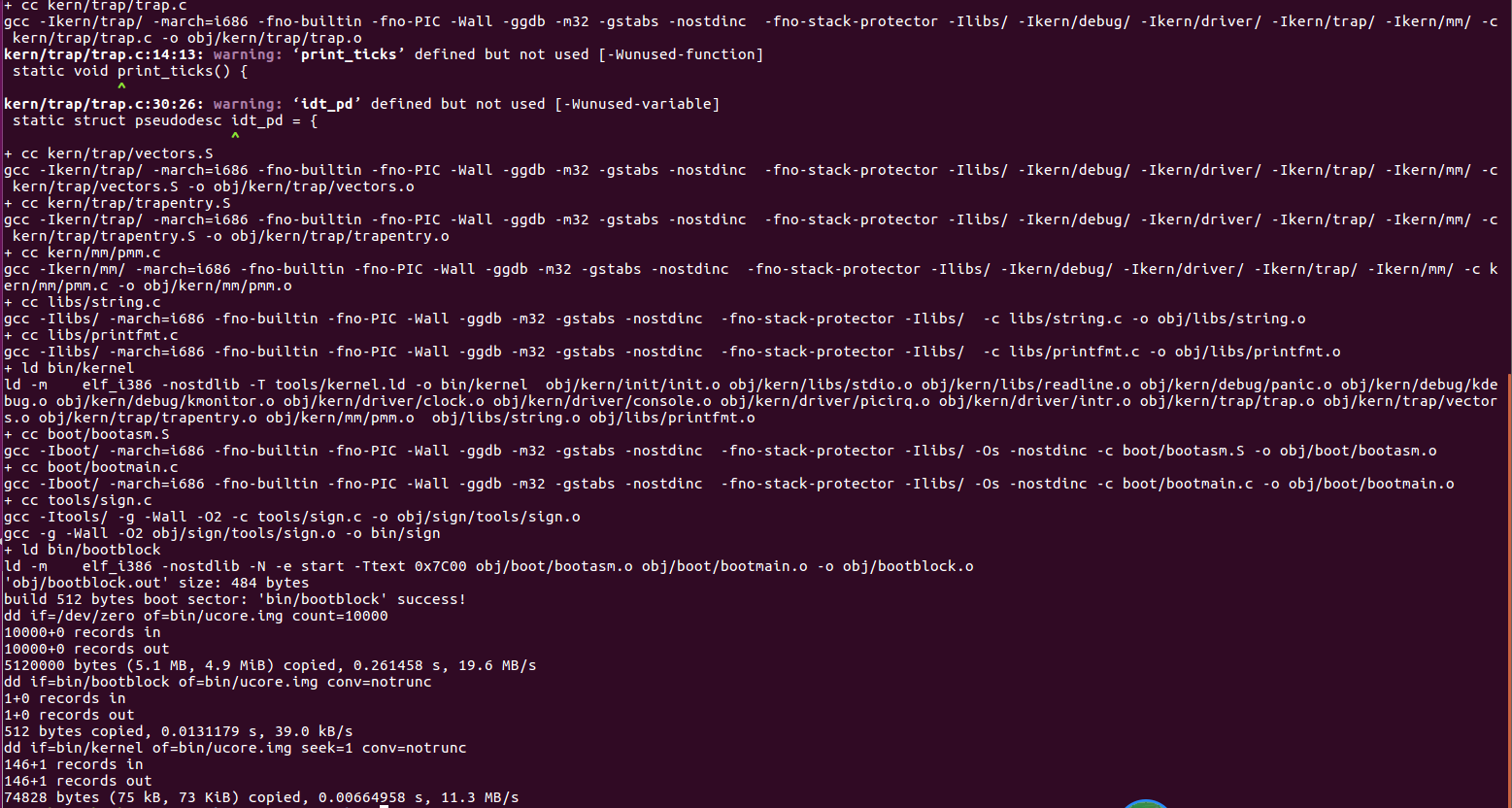
学号后三位：040

日期：2019年10月29日

**练习1：理解通过make生成执行文件的过程**

1. 调试makefile，了解make执行了哪些命令





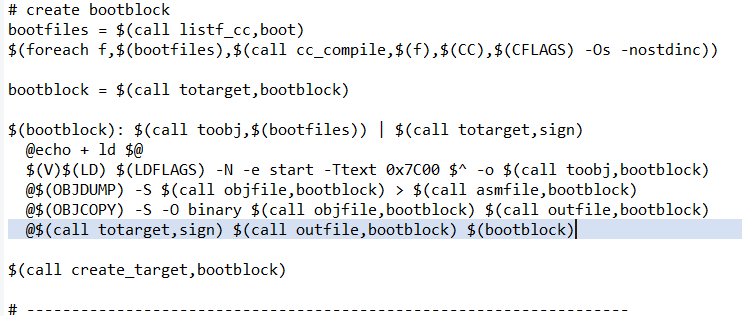
1. 提取主要命令，结合静态代码分析，可总结出ucore.img的生成过程
2. 编译生成bin/kernel所需要的所有文件，如：init.c stdio.c readline.c等。
3. 链接生成bin/kernel



1. 编译生成boot下的bootasm.S bootmain.c sign.c
2. 链接生成bin/bootblock，根据sign.c生成obj/bootblock.o

【这里没有看到根据sign规范生成bootblock的命令】

查看makefile文件create bootblock模块找到：

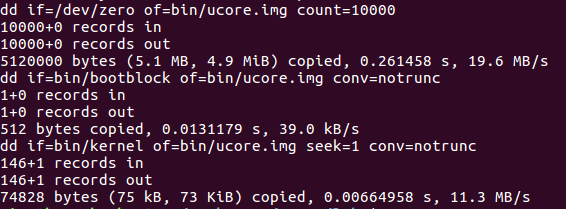


1. 生成ucore.img

创建大小为10000个块的ucore.img，初始化为0，每个块大小为512B。

把bookblock中的内容写到第一个块。

从第二个块开始写入kernel中的内容，共75kB。



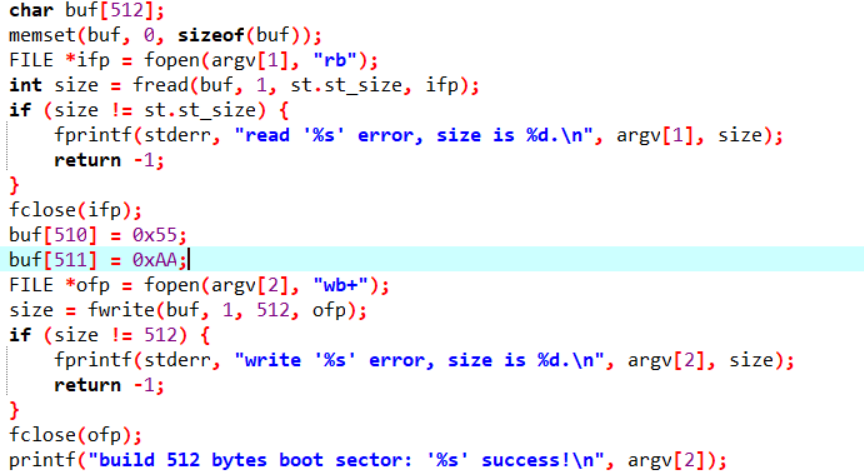
1. 一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？

//这里涉及到内存知识，还没学，以后补

先简单说下我的直观印象吧：

1. 有一个根目录bin，其他文件夹都在它下面
2. 需要有kernel和boot（bootblock）

//查资料后得知，这里问的是sign.c约束的bootblock规范



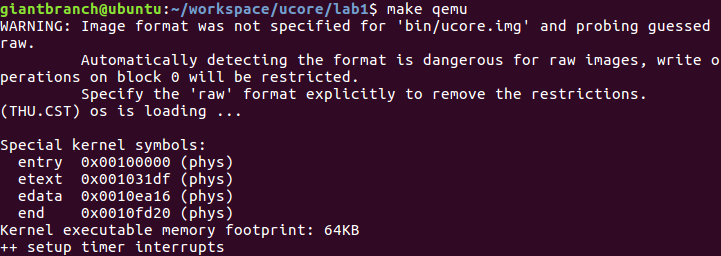
特征：

1. 磁盘主引导扇区内存为512字节
2. 磁盘最后两个字节固定为0x55AA

#### 练习2：使用qemu执行并调试lab1中的软件。（要求在报告中简要写出练习过程）

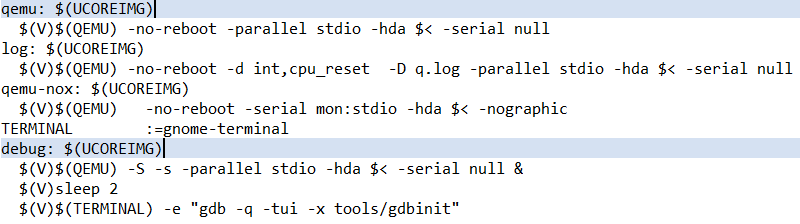
### 练习2.1 从 CPU 加电后执行的第一条指令开始,单步跟踪 BIOS 的执行。

1. 出现问题：输入make qemu闪退

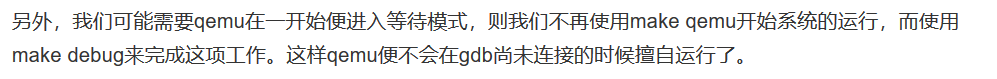


解决方案：尝试输入make debug，正常运行

原因：查看makefile文件，得到两条命令的具体定义



事实上，在实验手册中也有指出：



1. 输入qemu debug后，进行配置并调试

qemu和gdb之间使用网络端口1234进行通讯。

输入target remote localhost:1234

连接出错：

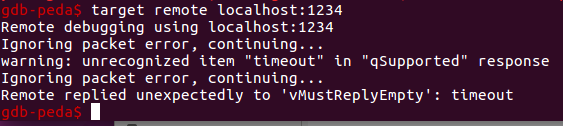


过了一段时间后重试，超时：



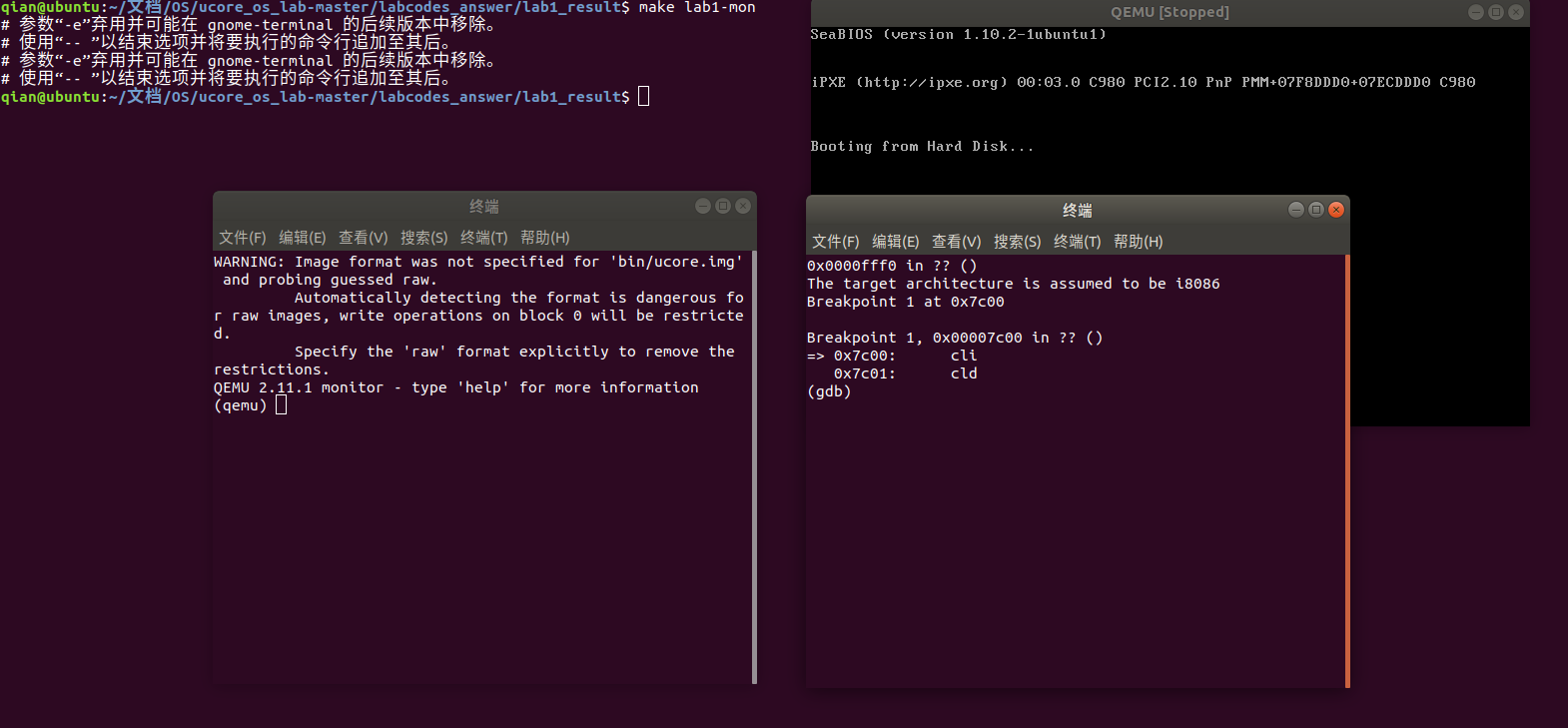
解决方案：不是在qemu中输入，而是在ubuntu的终端中先进入gdb，再输入target remote localhost:1234

连接超时：



### 【由于出现了一些莫名其妙的问题，我重装了ubuntu18.04，继续实验】

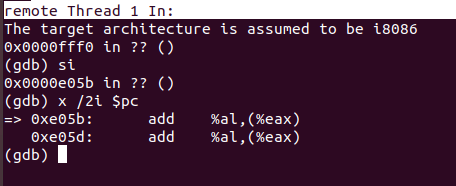
3、输入make lab1-mon，指明调试的是lab1的init文件，从bootloader第一条指令开始。



发现这里是运行到0x7c00处，触发断点，需要修改gdbinit文件：



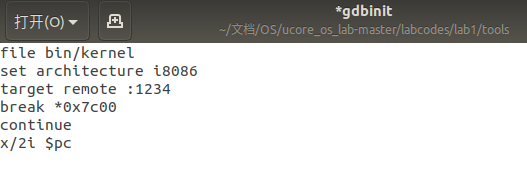
单步跟踪，查看BIOS代码：



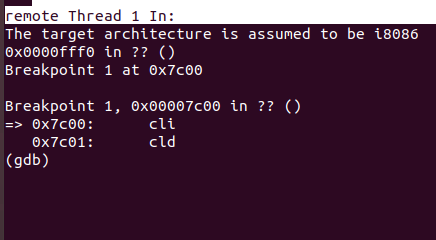
发现这里从0xfff0开始后，输入si进行单步跟踪后，发现跳转到0xe05b处，当前eip处汇编指令是两个add指令。

练习2.2 在初始化位置0x7c00设置实地址断点,测试断点正常。

修改gdbinit文件：

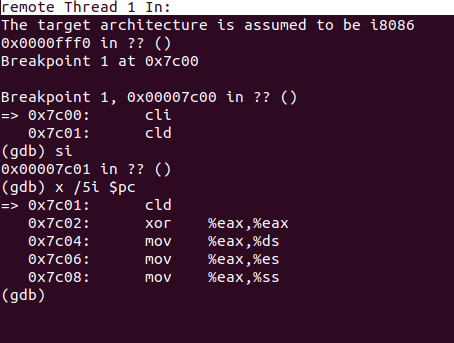


make debug后，得到结果，说明断点正常：

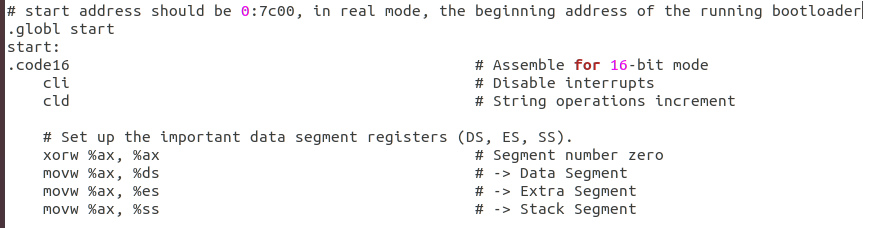


练习2.3 从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较。

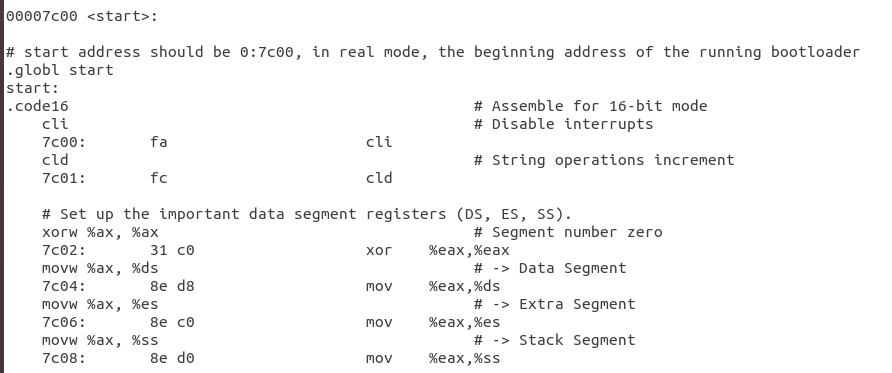
单步跟踪反汇编得到的代码：



查看bootasm.S文件：



查看bootblock.asm文件：



对比发现，反汇编得到的代码和bootasm.S、 bootblock.asm两个文件中的代码基本相同。有两处细微差别：

1. 汇编指令的xor与xorw、mov与movw：

**movw** ，即mov word：字传送，16位。

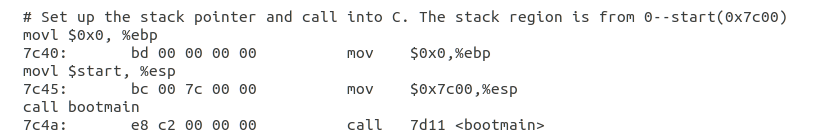
1. 寄存器的ax与eax：

|============eax=============|--32位,4个字节，2个字，1个双字  
|======ax======|--16位,2个字节，1个字

由此得出，xorw/movw是和ax配套的，而xor/mov是和eax配套的。

练习2.4：自己找一个bootloader或内核中的代码位置，设置断点并进行测试。

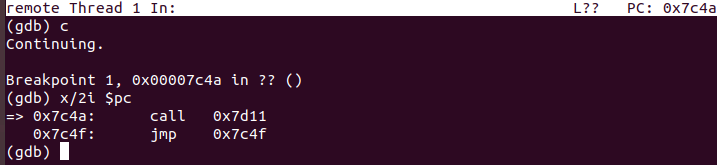
查看bootblock.asm文件，发现bootmain入口处为0x7c4a：



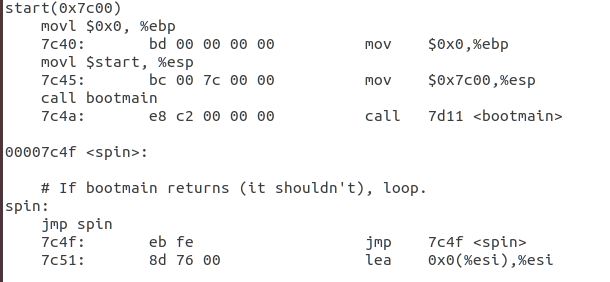
故在此处设置断点：



测试结果：



到bootblock.asm文件中，验证：

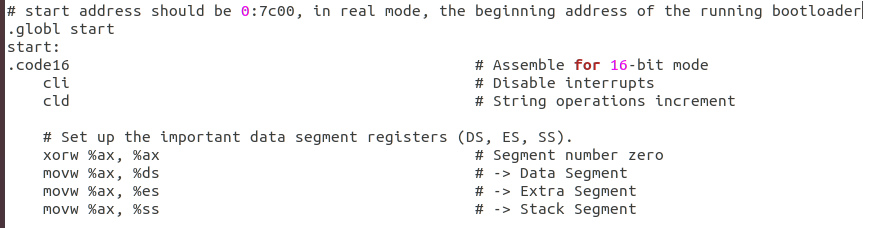


断点设置正常。

#### **练习3：分析bootloader进入保护模式的过程。**

查看bootasm.S文件：

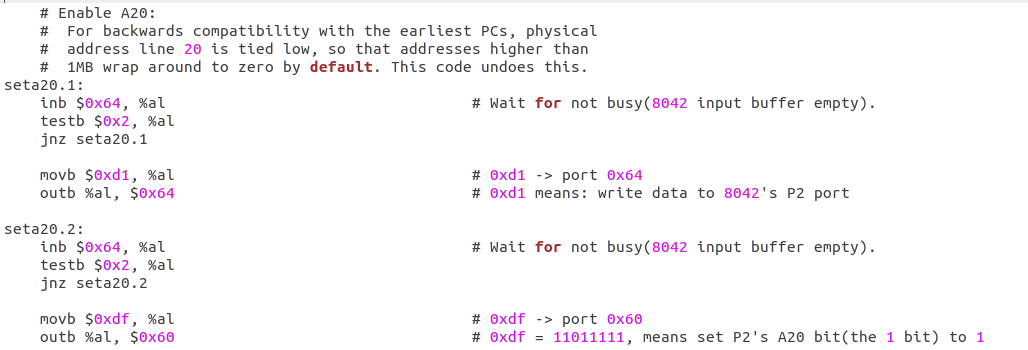
实模式下，起始地址0x7c00。



关闭中断/ 清除方向标志 /段寄存器ds es ss清零。

### 练习3.1 为何开启A20，以及如何开启A20？

初始时A20为0，访问超过1MB的地址时，就会从0开始循环计数。将A20地址线置为1之后，才可以访问4G内存。A20地址位由8042控制，8042有2个有两个I/O端口：0x60和0x64。



开启A20的流程：

Seta20.1:

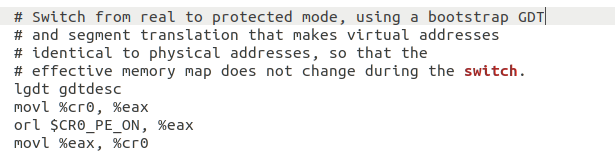
1. 等待8042 input buffer为空
2. 从0x64读入一个字节到al中
3. 检测al的第2位，若为0，则跳出seta20.1；若为1，则发送write data to 8042’s P2 port命令到8042 Input buffer,即将0xd1写入al中，将al写入0x64端口中；

Seta20.2:

1. 等待8042 input buffer为空
2. 从0x64读入一个字节到al中
3. 检测al的第2位，若为0，则跳出seta20.2；若为1，将8042 Output Port（P2）得到字节的第2位置为1，然后写入8042 Input buffer，即将0xdf写入al中，将al写入0x60端口中；

A20打开。

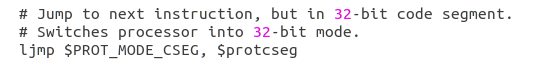
练习3.2 如何初始化GDT表？



1. 载入GDT表
2. 进入保护模式

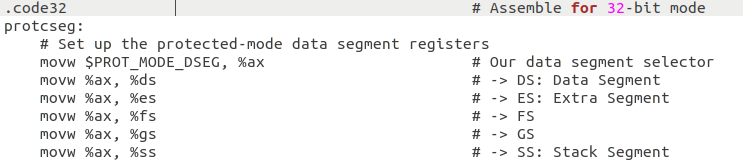
将cr0寄存器PE位置为1。

1. 通过长跳转更新cs的基地址



由于已经进入保护模式，所以这里需要用到逻辑地址。$PROT\_MODE\_CSEG的值为0x80。

1. 设置段寄存器，建立堆栈



设置帧指针、栈指针：



1. 转到保护模式完毕，进入bootmain主函数



练习3.3 如何使能和进入保护模式

将cr0的PE位置1.

PE：CR0的位0是启用保护（Protection Enable）标志。当设置该位时即开启了保护模式；当复位时即进入实地址模式。这个标志仅开启段级保护，而并没有启用[分页](http://baike.baidu.com/view/159980.htm" \t "https://www.cnblogs.com/cplover/archive/2013/10/16/_blank)机制。若要启用[分页](http://baike.baidu.com/view/159980.htm" \t "https://www.cnblogs.com/cplover/archive/2013/10/16/_blank)机制，那么PE和PG标志都要置位。

## **练习4：分析bootloader加载ELF格式的OS的过程。**

练习4.1 bootloader如何读取硬盘扇区的？

通过指导书“硬盘访问概述”一节，知道：bootloader让CPU进入保护模式后，下一步的工作就是从硬盘上加载并运行OS。考虑到实现的简单性，bootloader的访问硬盘都是LBA模式的PIO（Program IO）方式，即所有的IO操作是通过CPU访问硬盘的IO地址寄存器完成。

当前硬盘数据是储存到硬盘扇区中，一个扇区大小为512字节。读一个扇区的流程大致如下：

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

如何具体地从硬盘读取数据呢？因为我们所要读取的操作系统文件是存在0号硬盘上的，所以，我们来看一下关于0号硬盘的I/O端口：



进一步解释如下：

首先，CHS和LAB模式对应的是IO读取磁盘的两种不同方式：

CHS是 C柱面（10位） H 磁头（8位） S扇区（6位），LBA则是将硬盘划分成一个一个扇区。LBA方式访问是使用data寄存器，LBA寄存器（总共3个），device寄存器，command寄存器来完成的。

那么，LBA如何实现寻址：

1. LBA是一个整数，通过转换成CHS格式完成磁盘具体寻址。

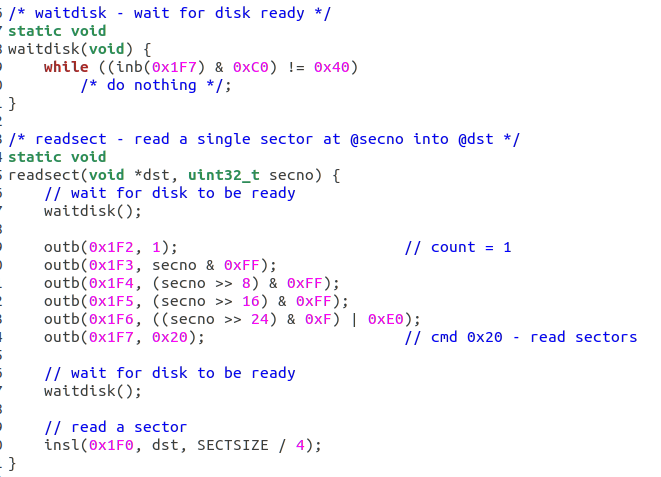
2. LBA采用48个bit位寻址，最大寻址空间128PB。

附：

LBA28的寄存器

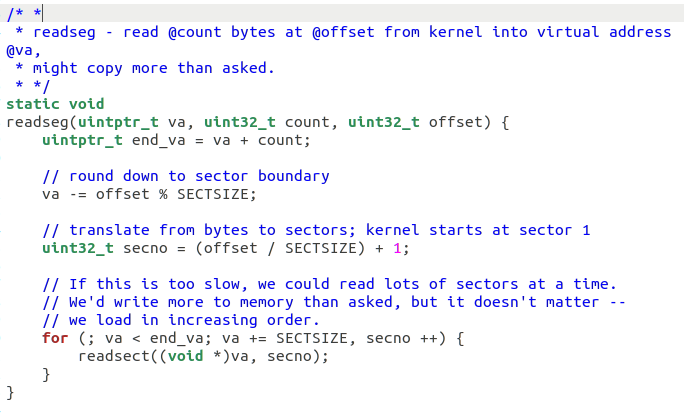
| **寄存器** | **端口** | **作用** |
| --- | --- | --- |
| data寄存器 | 0x1F0 | 已经读取或写入的数据，大小为两个字节（16位数据) 每次读取1个word,反复循环，直到读完所有数据 |
| features寄存器 | 0x1F1 | 读取时的错误信息 写入时的额外参数 |
| sector count寄存器 | 0x1F2 | 指定读取或写入的扇区数 |
| LBA low寄存器 | 0x1F3 | lba地址的低8位 |
| LBA mid寄存器 | 0x1F4 | lba地址的中8位 |
| LBA high寄存器 | 0x1F5 | lba地址的高8位 |
| device寄存器 | 0x1F6 | lba地址的前4位（占用device寄存器的低4位） 主盘值为0（占用device寄存器的第5位） 第6位值为1 LBA模式为1，CHS模式为0（占用device寄存器的第7位） 第8位值为1 |
| command寄存器 | 0x1F7 | 读取写入的命令，返回磁盘状态  读取扇区:0x20 写入扇区:0x30  磁盘识别:0xEC |

参看boot/bootmain.c中的readsect函数实现：



1. 调用waitdisk()函数：0xc0:11000000h,0x40:01000000h,01和11相与得到的还是01，故在waitdisk()函数中，可保证0x1F7端口command寄存器中最高两位是01。
2. 0x1F2端口，是sector count寄存器，表示每次读取1个扇区
3. 0x1F3、0x1F4、0x1F5：分别读取lba地址低8位、中8位、高8位
4. 0x1F6：读取lba地址的前4位（占用device寄存器的低4位）
5. 0x1F7：0x20表明命令是读取扇区。
6. insl：获取数据

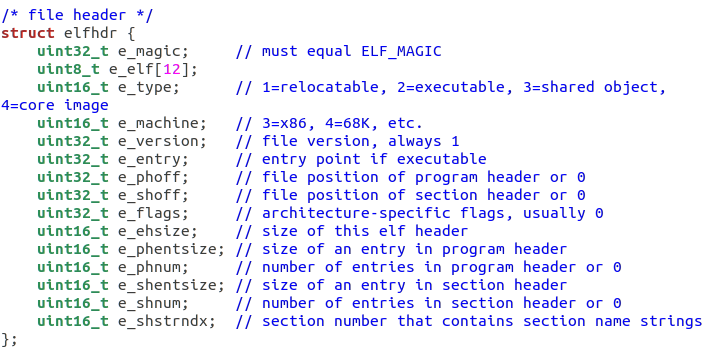
查看readseg函数：



在readsect()的基础上，设置成可以读取任意长度的内容。注意ELF从1扇区开始，因为0扇区被引导占用了。

练习4.2 bootloader是如何加载ELF格式的OS？

在bootmain.c文件中，发现有头文件<elf.h>，查看该文件得到ELF头部定义：





ELF文件由4部分组成，分别是ELF头（ELF header）、程序头表（Program header table）、节（Section）和节头表（Section header table）。

主要变量含义：

e\_magic：判断读取的ELF文件是否为正确格式

e\_entry：程序入口所对应的虚拟地址

e\_phoff：程序头表的位置偏移

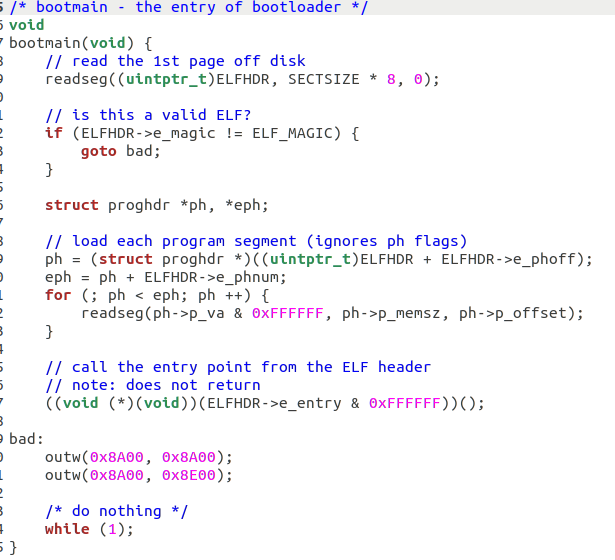
e\_shoff：节头表的位置偏移

e\_phnum：程序头表的入口数量

继续查看bootmain.c文件：

对扇区大小和ELF头部大小进行宏定义：

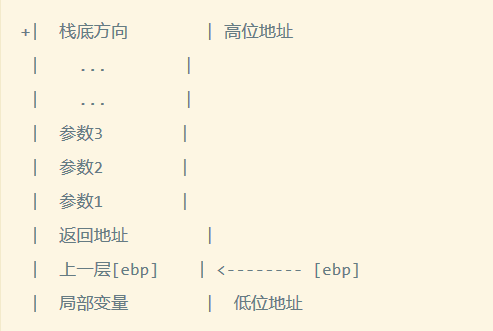
bootmain.c函数：



1. 读取硬盘的第一页，8个扇区，将数据存放在0x10000处。
2. 校验e\_magic字段，判断是否是合法的ELF文件
3. 若不合法：转bad（向寄存器0x8A00写入0x8A00后，再写0x8E00。这里是什么意思？）
4. 若合法：按照ELF头部的描述表，将每个程序段加载到相应的内存位置：先将描述表的首地址存放到ph中，eph是结束地址。然后按照描述表把数据读取到内存中。
5. 根据ELF头部的入口信息，找到内核的入口点。

#### **练习5：实现函数调用堆栈跟踪函数**

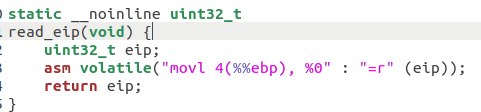
函数调用栈结构：



查看lab1/obj/bootblock.asm，了解bootloader源码与机器码的语句和地址等的对应关系；查看lab1/obj/kernel.asm，了解 ucore OS源码与机器码的语句和地址等的对应关系。

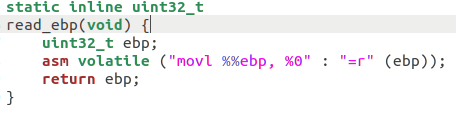
可通过read\_ebp()和read\_eip()函数来获取当前ebp寄存器和eip 寄存器的信息。然后通过ebp+12,ebp+16,ebp+20,ebp+24来输出4个参数的值，最后更新ebp：ebp=ebp[0],更新eip：eip=ebp[1]。直到ebp 对应地址的值为0（表示当前函数为bootmain）。

read\_eip()函数定义在kdebug.c中：



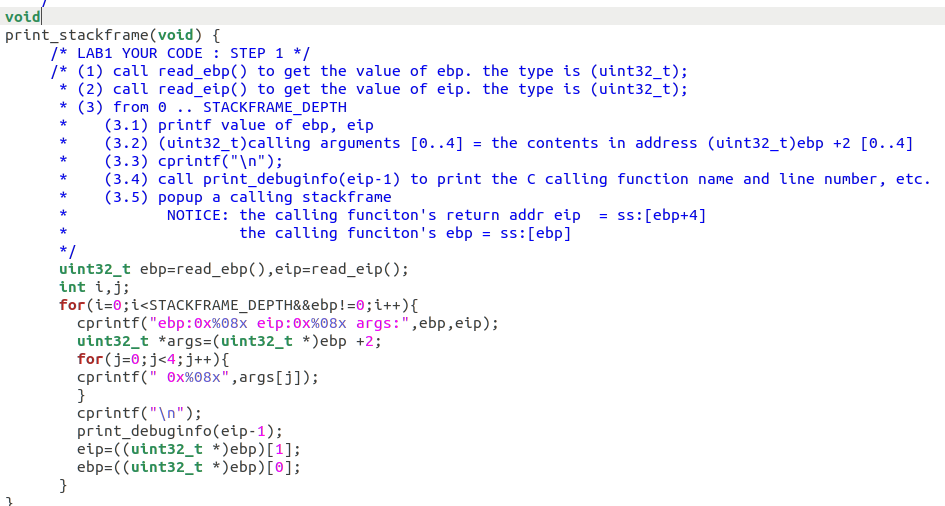
内联汇编，读取（ebp-4）的值到变量eip中。

read\_ebp() 函数定义在x86.h中：

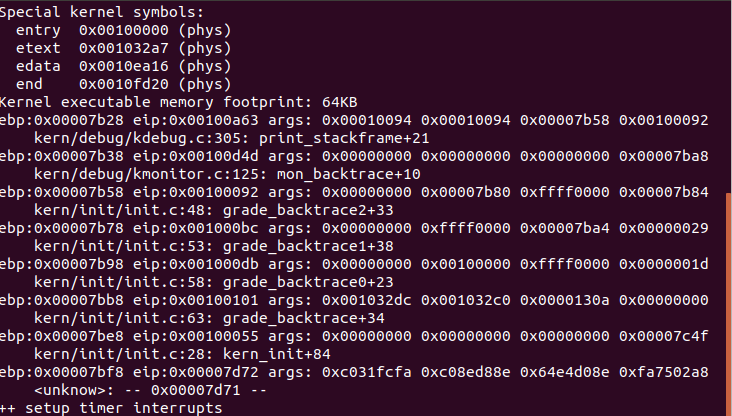


内联汇编，读取ebp寄存器的值到变量ebp中。

结合以上分析与原本的注释，完成函数print\_stackframe()：



在lab1中执行make qemu后，在qemu模拟器中得到输出：



除了eip和args的个别值外，与实验手册中给的示例基本一致。

最后一行各个数值的含义：

ebp：bootloader设置的堆栈从0x7c00开始，使用call bootmain转入bootmain函数，由于call指令压栈，所以ebp的值为0x7bf8。

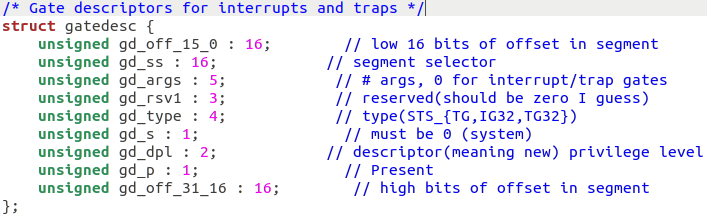
eip：调用的函数的返回地址。

args是四个参数。

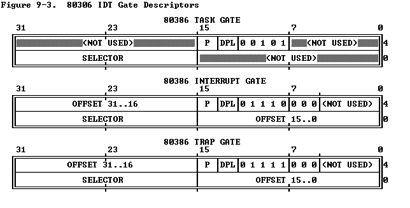
**练习6：完善中断初始化和处理**

练习6.1 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？

参见kern/mm/mmu.h中的struct gatedesc数据结构对中断描述符的具体定义：



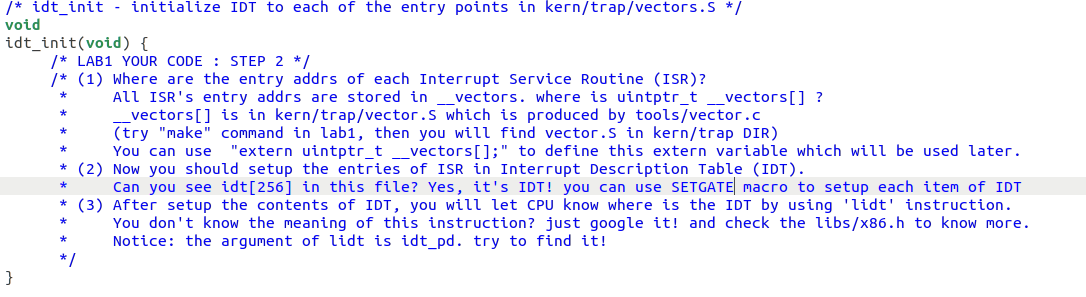
80386的任务门描述符、中断门描述符、陷阱门描述符的格式：



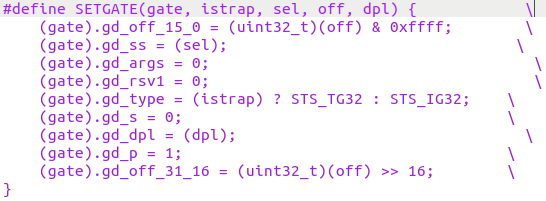
所以，IDT中一个表项占8个字节，2-3字节是段选择子SELECTOR，0-1字节和6-7字节拼成偏移量OFFSET，通过段选择子去GDT中取得相应的段描述符，段描述符里保存了中断服务例程的段基址和属性信息，段基址加上偏移地址就是中断服务例程的起始地址。

32-37位代表中断处理代码的入口。

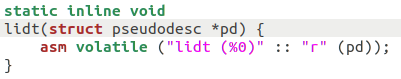
### 练习6.2 请编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt\_init。



根据上述注释，在mmu.h中查看SETGATE函数的实现，作用是对IDT表项赋值：



在x86.h中查看lidt命令定义：在操作系统创建IDT时，设定IDT的起始地址



内联汇编。LIDT（Load IDT Register）指令：使用一个包含线性地址基址和界限的内存操作数来加载IDT。

idt\_init函数的实现：

1. 引用保存在vectors.S中的256个中断处理例程的入口地址数组
2. 使用SETGATE宏，对中断描述符表中的每个表项进行设置。

参数说明：

SETGATE(gate, istrap, sel, off, dpl)

gate是门描述符，可通过其间接取得中断处理例程的入口地址；

istrap为0，表示是中断入口（若为1，则为陷阱入口）；

sel是段选择子，用于定位interrupt/trap handler；

off是偏移量，用于定位代码段；

dpl是描述符优先级；

宏定义：

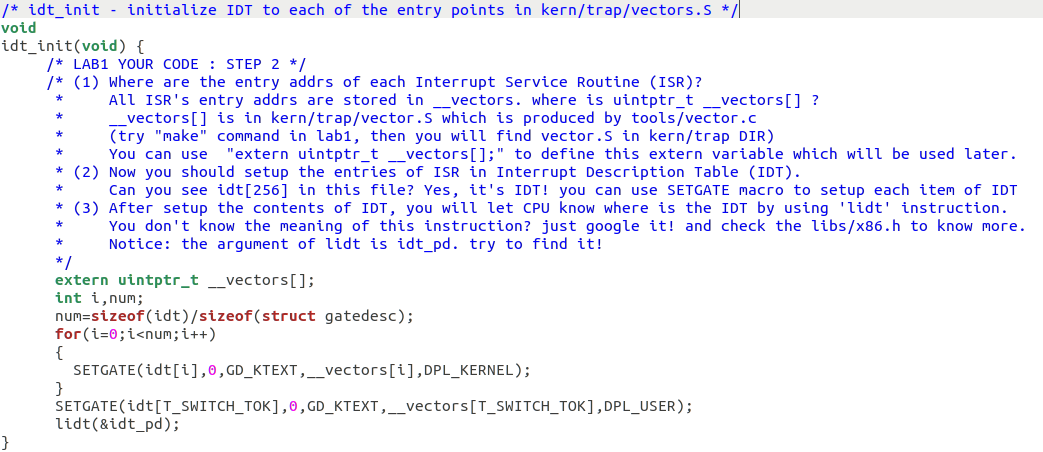
#define GD\_KTEXT ((SEG\_KTEXT) << 3) // kernel text

#define DPL\_KERNEL (0)

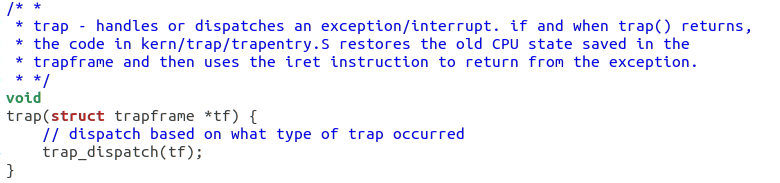
#define DPL\_USER (3)

#define T\_SWITCH\_TOK 121 // user/kernel switch

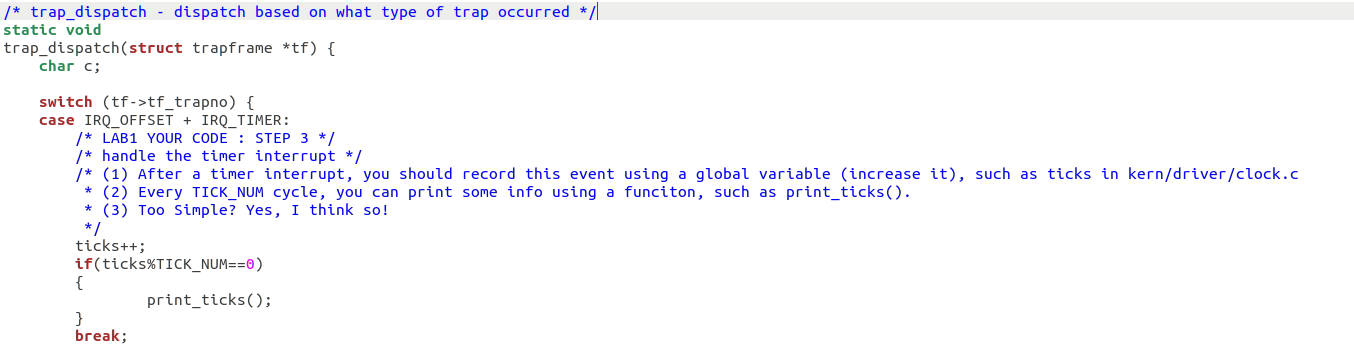
3、通过指令lidt把IDT的起始地址装入IDTR寄存器中，IDT初始化完成。



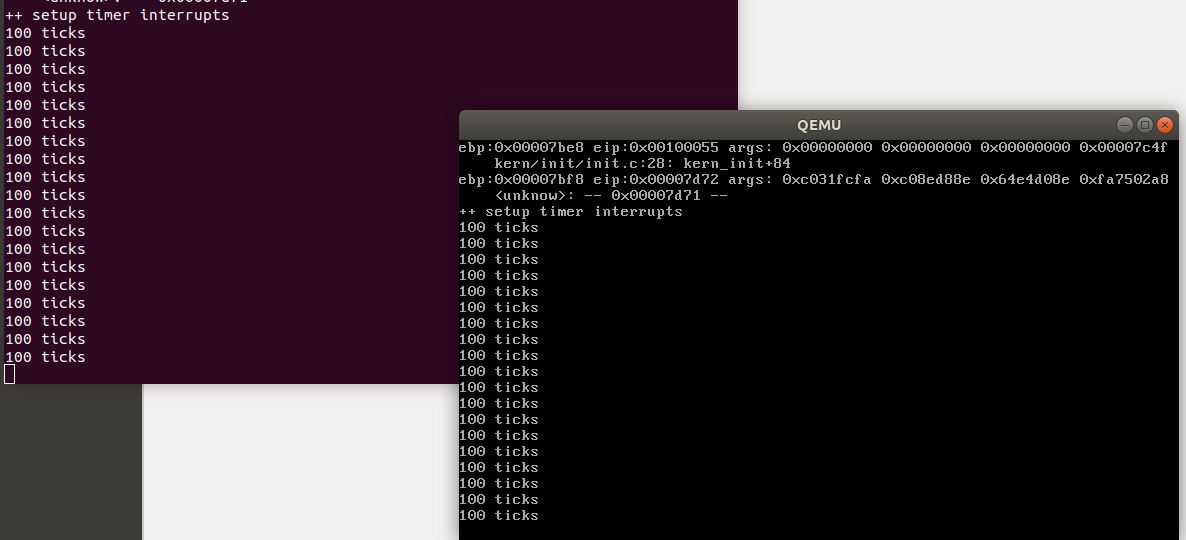
### 练习6.3 请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写trap函数，使操作系统每遇到100次时钟中断后，调用print\_ticks子程序，向屏幕上打印一行文字“100 ticks”。



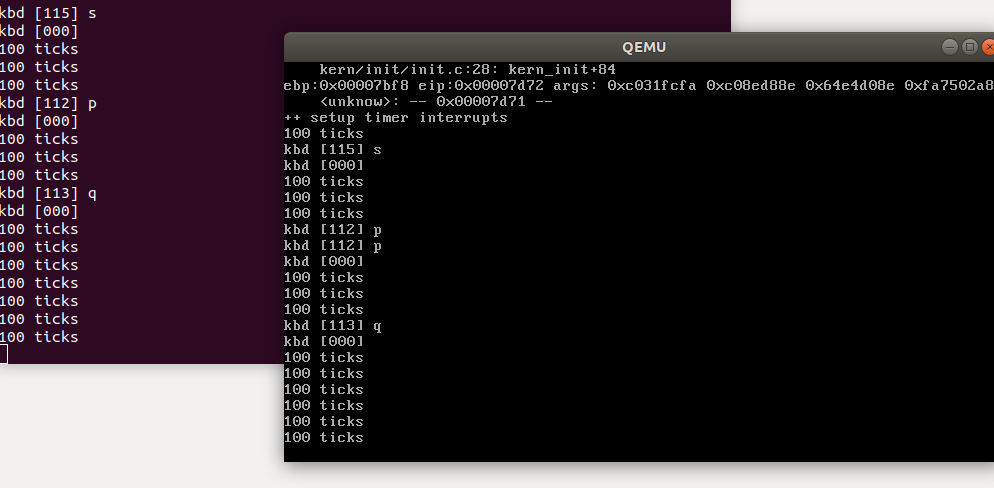
真正需要补全的代码部分在函数trap\_dispach()中：



在QEMU中运行结果如下：



且按下的键也会在屏幕上显示：



--END--