Lab1

**Part1 PC Bootstrap**

2 manuals

QEMU Emulator & GDB

Kernel.img: bootloader+ kernel

JOS kernel --write--> virtual VGA & PC

Kerninfo:copy contents of image onto the sectors of a real hard disk

Exe1:

**The PC’s Physical Address Space:**

8086:1MB

<640K->RAM

A0000~FFFFF(384K)->BIOS,VGA

80286: + extended memory; very top:reserved by BIOS for PCI devices

>4GB:another hole

JOS:256M

**The ROM BIOS:**

2 terminal

First inst:ljmp

0xffff0:接近BIOS ROM末尾（0xf0000~0xfffff) jmp backward

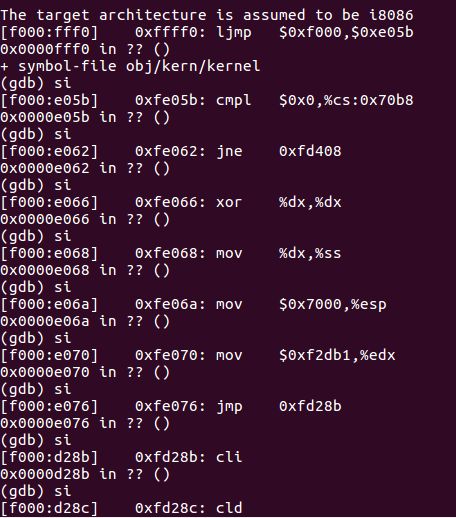
Start PC: cs=0xf000 ip=0xfff0

Jmp:cs=0xf000 ip=0xe05b

Real mode: CS\*16(16进制下左移一位） + IP

Exe2

{ 跳转 ss赋为0 栈顶指向0x7000 关中断 清除方向标志



}

VGABios: initialize devices such as VGA

**Part2 The Boot Loader**

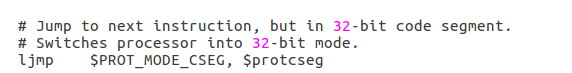
1st sector :boot sector(boot loader code resides here), loads 512B into 0x7c00 ~ 0x7dff, jmp set CS:IP -> 0000:7c00

Loader: assembly lg + c

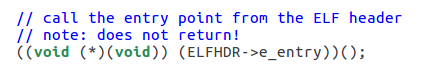
1. real mode->protected mode (VM > 1M)
2. Reads kernel from the hard disk by accessing the IDE disk device registers

Exe3{

At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

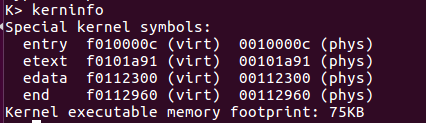
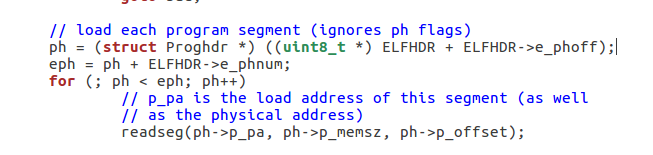
IMG_256

What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

The last inst:

The first inst:

How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?



}

**Loading the kernel**

Exe4{

1. a被正常赋值，但是c和a指向同一位置，修改c即修改a

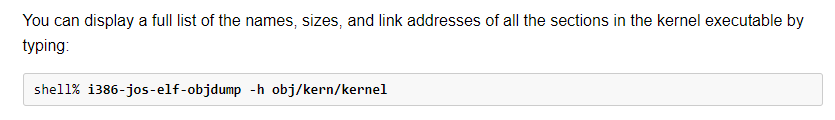
2. 3[c] = \*（3 + c）

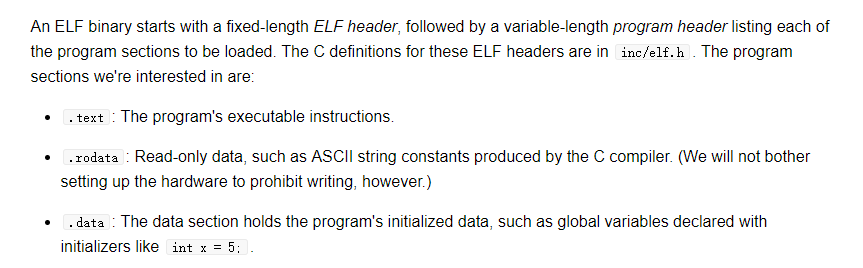
3. c+=1 向后移sizeof(int)

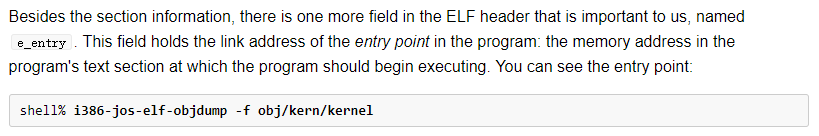
4. c + 1 向后移sizeof（char）然后写入sizeof（int）长度的数字

5. 思路同上

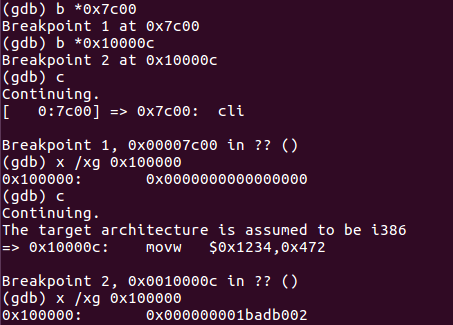
}







Exe5{



}

**Link vs. Load Address**



Exe6{

修改链接地址，loader仍从0x7c00开始，gdtr读取错误，ljmp跟着错误

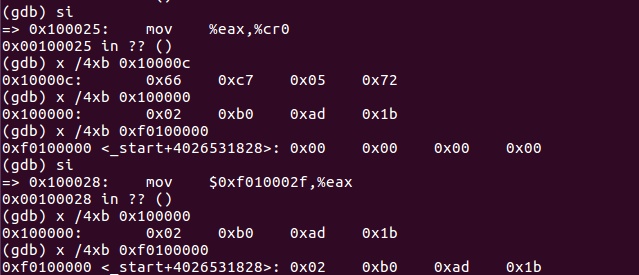
}

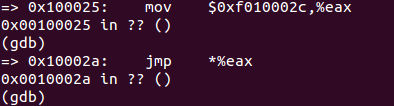
**Part 3: The Kernel**

**Using segmentation to work around position dependence**

EXE7{

修改CR0的时候切换至protected mode （movl %eax，%cr0）



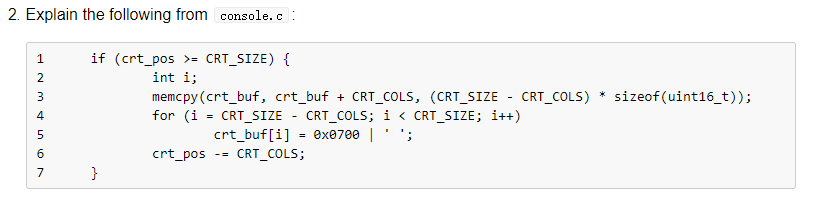
注释后在跳转的时候出现错误

}

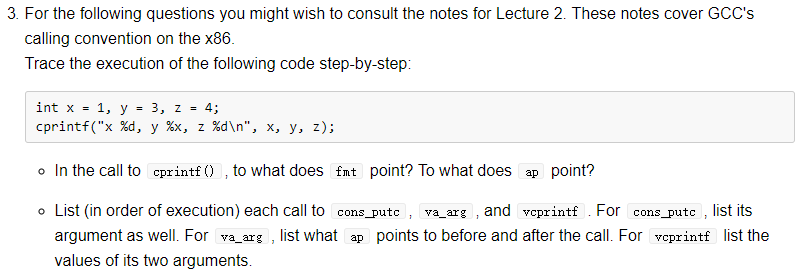
**Formatted Printing to the Console**

Exe8、9 {lib/printfmt.c}

Console.c定义了如何把字符显示在显示器上，提供了cons\_init, cputchar, getchar, iscons等函数接口，其中cputchar被readline和cprintf使用（即printf.c）。Printf.c调用该函数将字符传到显示器。



当当前已输入的字符位置大于最大size的时候，需要将屏幕向上滚动一行。方法是，将buf中下一行的数据（从crt\_buf + CRT\_COLS开始，大小为(CRT\_SIZE - CRT\_COLS)\*sizeof(uint16\_t)）全部向上一行赋值，然后在最后一行都变成空格，最后修改当前写入的位置。



1. fmt指向字符串，ap指向参数列表
2. vcprintf，fmt包括了输入的字符串"x %d, y %x, z %d\n"，ap指向变量x，值为1.

cons\_putc （int）’x’ cons\_putc （int）’ ’

va\_arg 运行前指向x，运行结束后指向y。

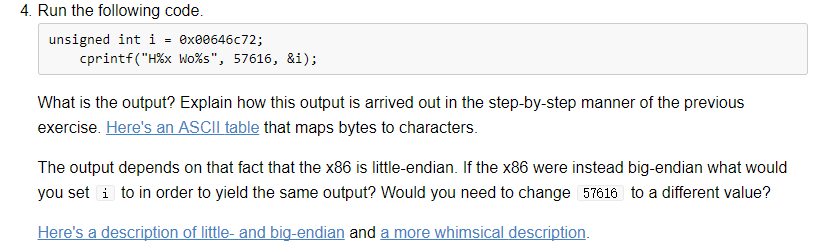
cons\_putc （int）’1’ cons\_putc （int）’,’ cons\_putc （int）’ ’ cons\_putc （int）’y’ cons\_putc （int）’ ’

va\_arg 运行前指向y，运行结束后指向z。

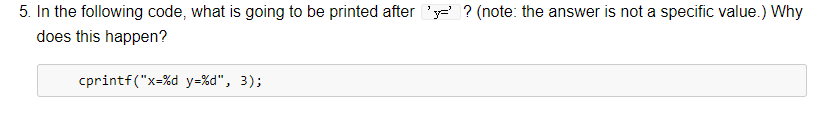
cons\_putc （int）’3’ cons\_putc （int）’,’ cons\_putc （int）’ ’ cons\_putc （int）’z’ cons\_putc （int）’ ’

va\_arg 运行前指向z，运行结束后指向z之后。

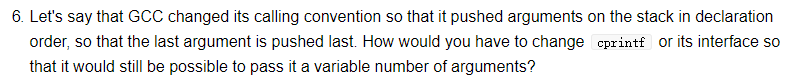
cons\_putc （int）’4’ cons\_putc （int）’\n’



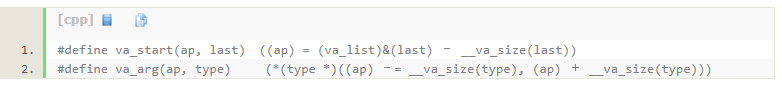
He110 World（57616-> 0xe110, 72->r 6c->l 64->d 00->’\0’)



Ap指向垃圾地址，对应的值不能确定

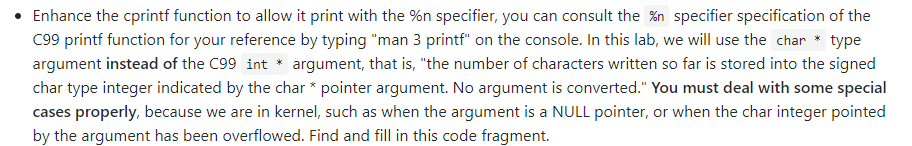


修改两个宏，使va\_start指向最后一个变量开始的位置，va\_arg将ap向前移动



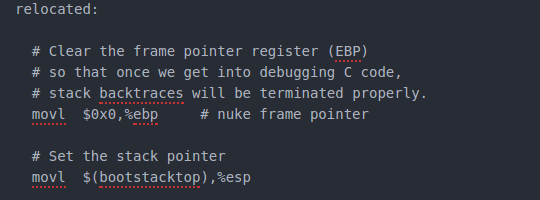
EXE10{’-’已经处理（padc） 在已经输出的数字后面看情况加空格

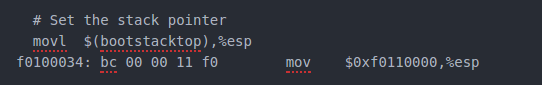
}

->

**The Stack**

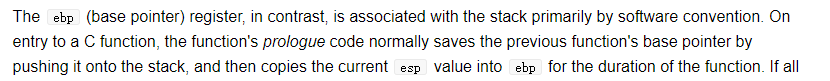
Exe11{

（entry.S)

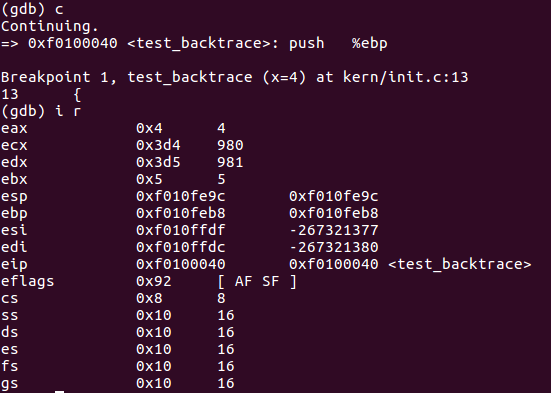
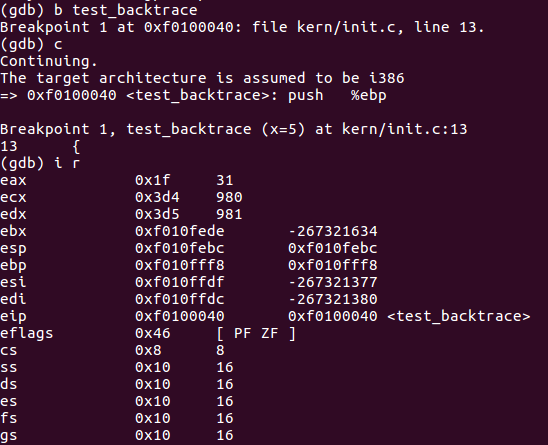
(kernel.asm)

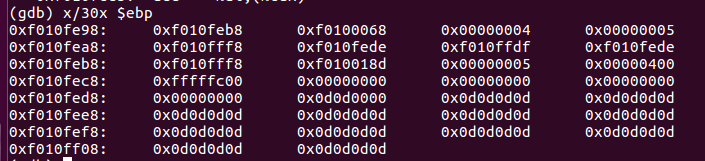
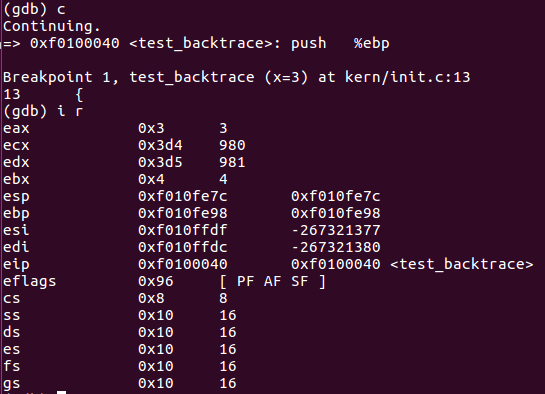
Ebp初始为0，esp指向栈底

}



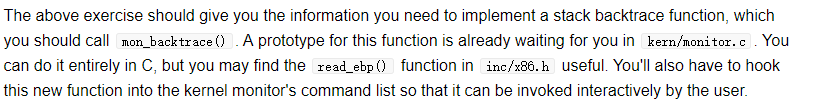
Exe12{

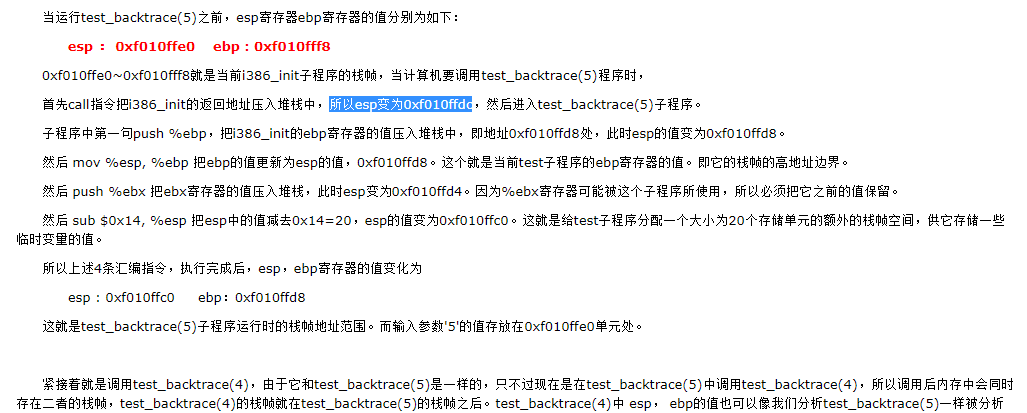


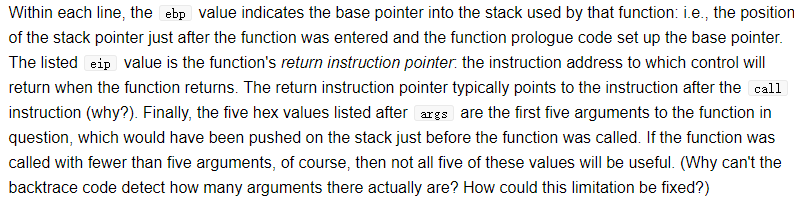


返回地址 保存ebp 保存ebx

}

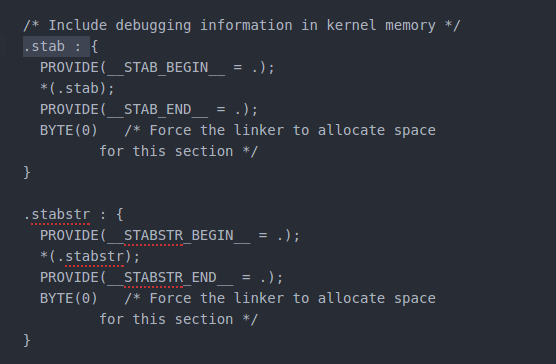




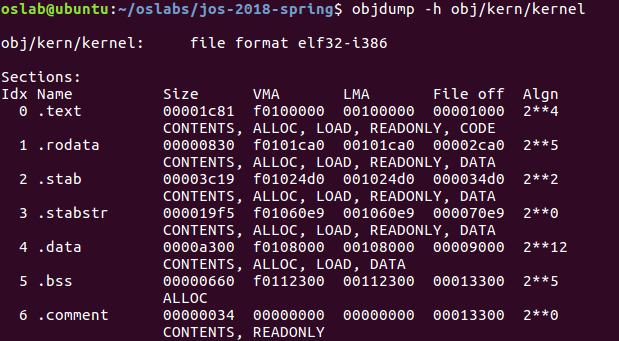


Exe14{

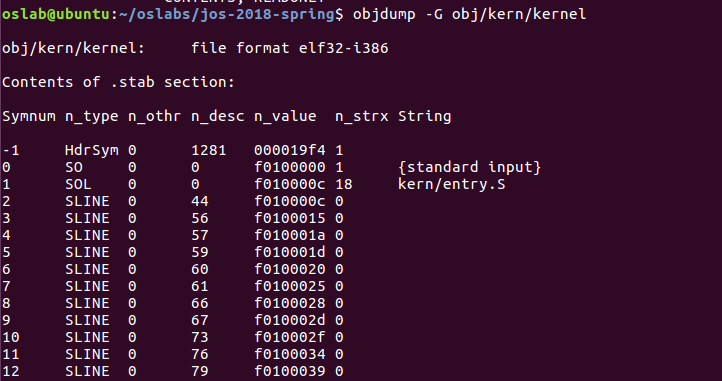
In kernel.ld:



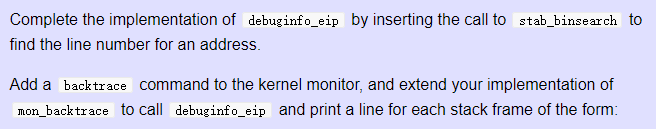
run i386-jos-elf-objdump -h obj/kern/kernel



run i386-jos-elf-objdump -G obj/kern/kernel

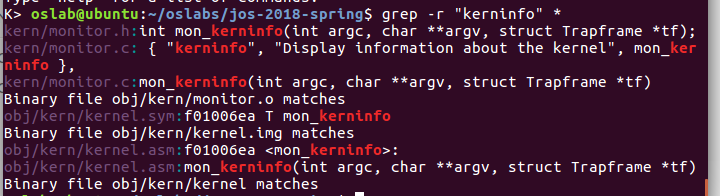


run i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS\_KERNEL -gstabs -c -S kern/init.c, and look at init.s.



}

EXE15{



}