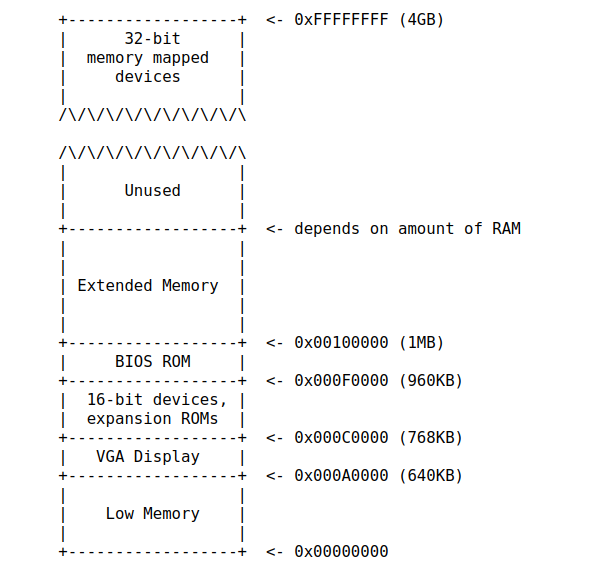
**MIT lab 1实验报告**

**1.1、PC bootstrap**

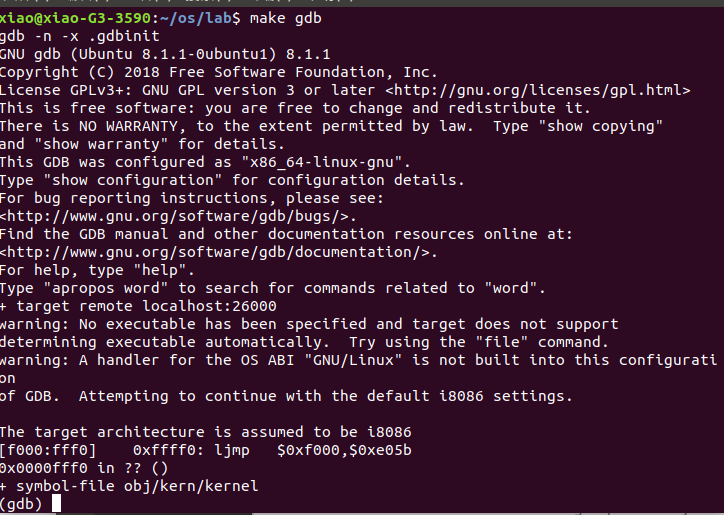
图1

如图1所示，上图为PC机的物理地址空间，其中最重要的是0x000F0000到0x0010000的(Basic Input/Output System) BIOS，BIOS负责进行一些基本的系统初始化任务，比如开启显卡，检测该系统的内存大小等等工作。在初始化完成后，BIOS就会从某个合适的地方加载操作系统。

接下来是用qemu和gdb去探索PC的启动过程。在一个terminal中输入make qemu-gdb后，在另一个terminal中输入make gdb，结果如图2所示。可以看到指令

[f000:fff0] 0xffff0: ljmp $0xf000, $0xe05b

这条指令是整个PC启动后，执行BIOS的第一条指令。从中我们可以知道，PC在物理地址0x000ffff0 处启动，这个物理地址位于BIOS的64kb中的高位置，是一条跳转指令。另外，在实模式下，物理地址的计算是通过段基址\*16+偏移量获得的，即将段基址左移一位再加上偏移量，如0xffff0=0xf000\*16+0xfff0。

图2

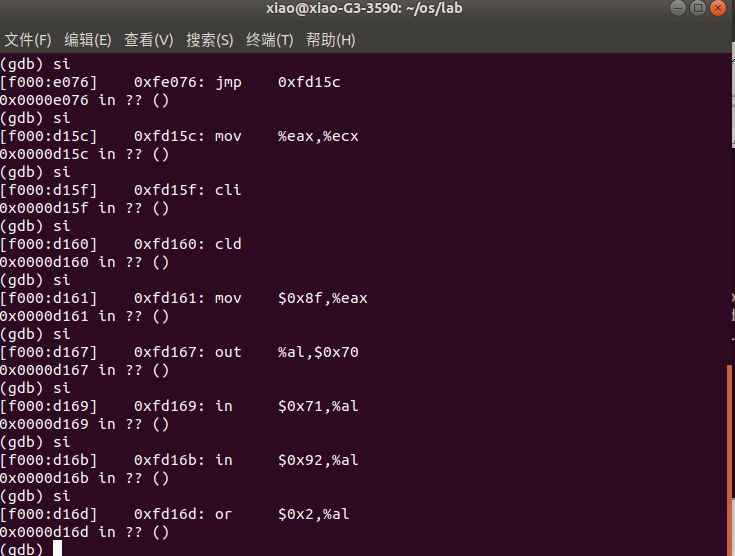
**Exercise 2 用gdb的si指令探索更多的指令，猜测一下他们在做什么**

图3

如图3所示，这些指令大多都是一些初始化操作，如设置寄存器的值，关闭中断，加载中断向量表寄存器(IDTR)，但是有很多指令还是很迷惑的，比如像下面3条指令

[f000:d16b] 0xfd16b: in $0x92,%al

[f000:d16d] 0xfd16d: or $0x2,%al

[f000:d16f] 0xfd16f: out %al,$0x92

他们将端口号0x92的1号bit位置1，查阅资料后我发现bit 1= 1 indicates A20 active，这个位是使能A20号线，如果A20地址线被激活，那么系统工作在保护模式下。但是之后将操作系统加载进内存中是要在实模式下进行的，我也不是很清楚这几条指令有什么用。

总而言之，在初始化总线和重要设备后，BIOS会搜索可引导设备，如软盘、硬盘驱动器或CD-ROM。最终，当它找到可引导磁盘时，BIOS会从磁盘读取引导boot loader，并将控制权转交给它。

**1.2、****The Boot Loader**

如果磁盘是可引导的，那么磁盘的第一个sector即前512个字节就是boot loader，BIOS会将boot loader加载至物理地址0x7c00到0x7cff。boot loader有两个主要的功能：

1.将处理器从实模式转化为32位的保护模式，

2通过使用x86的特定的IO指令，直接访问IDE磁盘设备寄存器，从磁盘中读取内核

**Exercise 3 在地址0x7c00处设置一个断点，跟踪/boot/boot.S文件的每一条指令，和反汇编文件obj/boot/boot.asm进行比较；跟踪bootmain函数中的readsect()子函数，回答四个问题。**

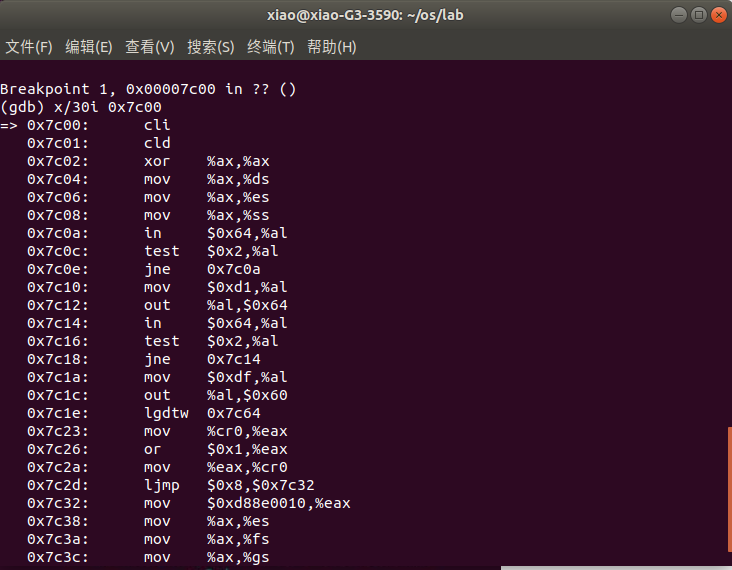


图4

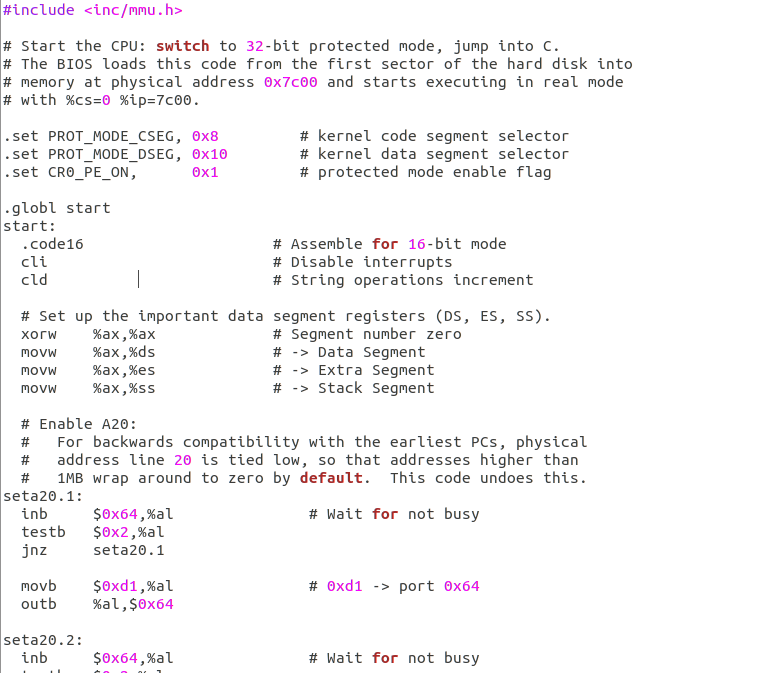
图5

图6

如图所示，图4是设置断点后的30条指令的结果，图5是boot.S的代码，图6是boot.asm的代码，三者在指令上并没有什么区别，只不过在boot.S中多了很多标识符，如seta20.1，而boot.asm中都转化成了真实地址。

注意到boot.S中的最后一行代码是call bootmain，就是调用bootmain函数，下面对bootmain函数进行分析。

1.// read 1st page off disk

**readseg((uint32\_t) ELFHDR, SECTSIZE\*8, 0);**

将内核中的前(4MB = SECTSIZE\*8 = 512\*8)的内容读取的内存地址ELFHDR(0x10000)处

2.// is this a valid ELF?

**if (ELFHDR->e\_magic != ELF\_MAGIC)**

**goto bad;**

判断是否是合法的ELF文件

3.// load each program segment (ignores ph flags)

**ph = (struct Proghdr \*) ((uint8\_t \*) ELFHDR + ELFHDR→e\_phoff);**

查阅资料发现elf头部文件中包含Program Header Table。这个表格存放着程序中所有段的信息。通过这个表我们才能找到要执行的代码段，数据段等等。这行代码就是获得这个表。ph被赋值为指向Program Header Table的表头。

4.**for (; ph < eph; ph++)**

// p\_pa is the load address of this segment (as well

// as the physical address)

**readseg(ph->p\_pa, ph->p\_memsz, ph→p\_offset);**

readseg(uint32\_t pa, uint32\_t count, uint32\_t offset)函数的作用是Read 'count' bytes at 'offset' from kernel into physical address 'pa'. 所以这个循环的作用就是将各个段的内容读入内存中

5.**((void (\*)(void)) (ELFHDR->e\_entry))();**

e\_entry字段指向的是这个文件的执行入口地址。自此就把控制权从boot loader转交给了操作系统的内核。

对于之前提到的四个问题，下面进行一些我的理解：

1. 在什么时候处理器开始运行于32bit模式？什么导致CPU从16位切换为32位模式？

在boot.S文件中，计算机首先工作于实模式，此时是16bit模式。当运行完 " ljmp $PROT\_MODE\_CSEG, $protcseg " 语句后，正式进入32位模式。

2. boot loader中执行的最后一条语句是什么？内核被加载到内存后执行的第一条语句又是什么？

boot loader执行的最后一条语句是bootmain子程序中的最后一条语句 " ((void (\*)(void)) (ELFHDR->e\_entry))();"，即跳转到操作系统内核程序的起始指令处。

执行的第一条语句位于/kern/entry.S文件中，第一句 movw $0x1234, 0x472

3. 内核的第一条指令在哪里？

上一个问题中已经回答过这个问题，第一条指令位于/kern/entry.S文件中。

4. boot loader是如何知道它要读取多少个扇区才能把整个内核都送入内存的呢？在哪里找到这些信息？

关于操作系统一共有多少个段，每个段又有多少个扇区的信息位于操作系统文件中的Program Header Table中。这个表中的每个表项分别对应操作系统的一个段。并且每个表项的内容包括这个段的大小，段起始地址偏移等等信息。所以如果我们能够找到这个表，那么就能够通过表项所提供的信息来确定内核占用多少个扇区。

　　那么关于这个表存放在哪里的信息，则是存放在操作系统内核映像文件的ELF头部信息中。

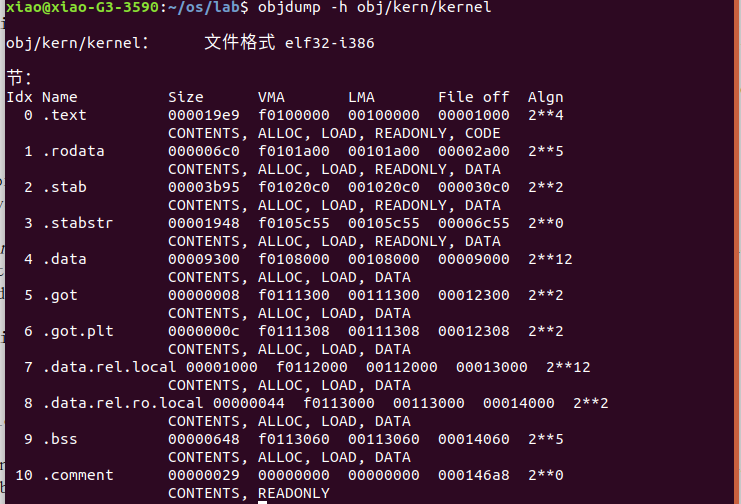
**Exercise 4是对c语言指针的练习，这里略过**

当我们编译和链接一个c程序(比如JOS内核)时，编译器会将每个c源文件(‘.c’)转换成一个对象文件(‘.o’)，该对象文件包含以硬件所需的二进制格式编码的汇编语言指令。然后链接器将所有已编译的目标文件组合成一个二进制映像，例如obj/kern/kernel，在这里是ELF格式的二进制映像，即“Executable and Linkable Format”。The boot loader不会修改ELF文件中的代码或数据，而是将它加载进内存中并执行它。ELF二进制文件以固定长度的ELF头部开始，由需要加载的每个程序段的变长程序头部组成。我们比较关心的是以下几个段

.text:程序的可执行指令。

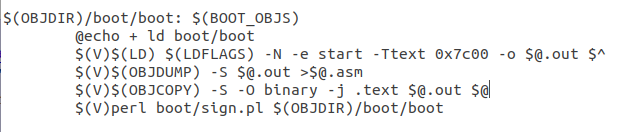
.rodata:只读数据，例如C编译器生成的ASCII字符串常量。

.data: data部分保存程序初始化后的数据，比如用初始化器声明的全局变量，如int x = 5;

图7

可以通过命令i386-jos-elf-objdump -h obj/kern/kernel查看所有段的信息，如图7所示

**Exercise 5 修改boot/Makefrag中boot loader的链接地址，看看会发生什么**

图8

如图8所示，-Ttext即为boot loader的链接地址，可以看到，默认的地址是0x7c00，现在将其改为0x7f00,重新make

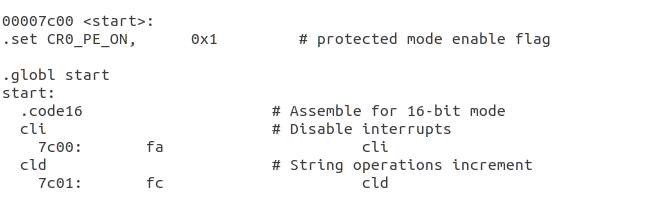
图9

图9是重新make之前boot.asm中的内容

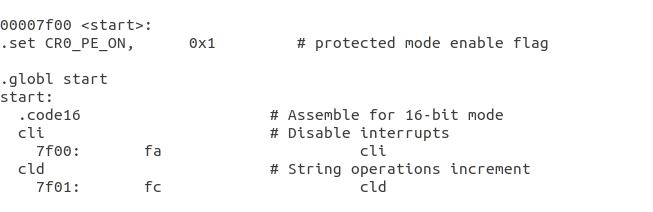
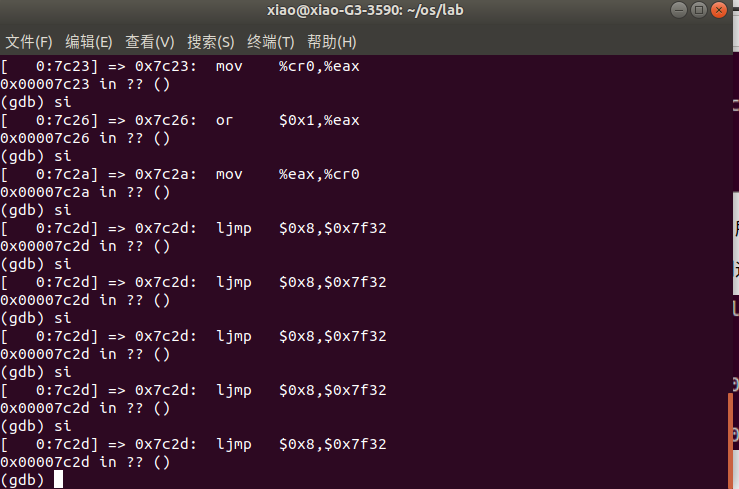
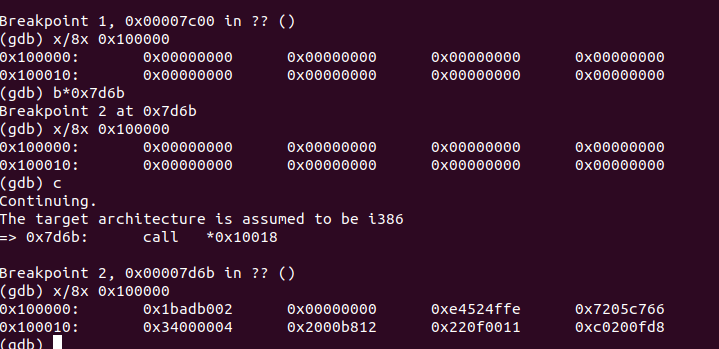
图10

图10是make之后boot.asm中的内容

图11

在0x7c00处设置断点后，可以看到，一开始进行的指令都是没什么问题的，直到进行到一条跳转指令ljmp $0x8,$0x7f32,本来是要跳转到$0x7c32，但由于地址改变，程序也一直卡在这个跳转指令

**Exercise 6 比较**BIOS进入boot loader时0x00100000处的八个字和the boot loader进入内核时0x00100000处的八个字的内容****

图12

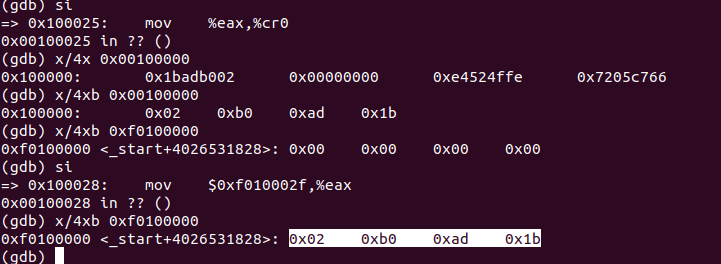
**BIOS进入boot loader时地址为0x7c00，**在boot.asm中查到((void (\*)(void)) (ELFHDR->e\_entry))();的地址为0x7d6b，即boot loader进入内核程序的入口，如图12所示，在这两个地址处设置断点。可以看到，**BIOS进入boot loader时0x10000是没有内容的，当**boot loader进入内核程序时就出现了内容，且第二个断点后的下一条指令地址是0x10000c,是内核程序的起始地址（通过objdump -f obj/kern/kernel指令查看），说明是内核程序的指令被存放到这里。

**1.3、The kernel**

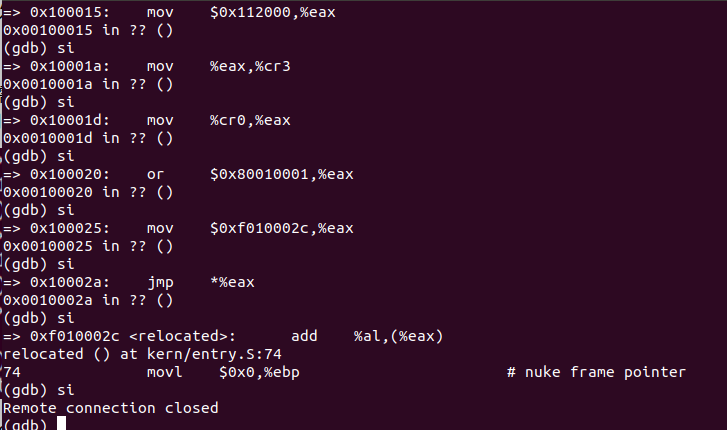
在运行boot loader时，boot loader中的链接地址（虚拟地址）和加载地址（物理地址）是一样的。但是当进入到内核程序后，这两种地址就不再相同了。通常操作系统被链接到一个高地址处（虚拟地址），但在实际的内存中会被放到一个低地址处，如高地址0xf0100000会被映射到低地址0x00100000。

**Exercise 7：使用Qemu和GDB去追踪JOS内核文件，并且停止在movl %eax, %cr0指令前。此时看一下内存地址0x00100000以及0xf0100000处分别存放着什么。然后使用stepi命令执行完这条命令，再次检查这两个地址处的内容。**

如果这条指令movl %eax, %cr0并没有执行，而是被跳过，那么第一个会出现问题的指令是什么？我们可以通过把entry.S的这条语句加上注释来验证一下。

图13

如图13所示，将断点设置在0x10000c上，单步执行，直到执行mov %eax，%cr0，此时0x100000中是有内容的，而0xf0100000中没有内容，执行完后0xf0100000中内容与0x100000中内容相同。说明在0xf0100000处的内容在该条指令执行后就会被映射到0x00100000处。

图14

如图14所示，在注释掉指令movl %eax, %cr0后，在0x10002a处的jmp指令，要跳转的位置是0xf010002C，但是没有进行虚拟地址到物理地址的映射，因此发生了错误。

**Exercise 8是对代码的修改，要求补全对八进制的输出实现，我直接在源代码上进行了修改，这里不再赘述。**

文中提到的几个问题：

**1.**Explain the interface between** printf.c **and** console.c**. Specifically, what function does** console.c **export? How is this function used by** printf.c**?****

**Console.c输出了cputchar，这个函数**在printf.c中的putch函数中被调用

**2.Explain the following from console.c:**

**1 if (crt\_pos >= CRT\_SIZE) {**

**2 int i;**

**3 memmove(crt\_buf, crt\_buf + CRT\_COLS, (CRT\_SIZE - CRT\_COLS) \* sizeof(uint16\_t));**

**4 for (i = CRT\_SIZE - CRT\_COLS; i < CRT\_SIZE; i++)**

**5 crt\_buf[i] = 0x0700 | ' ';**

**6 crt\_pos -= CRT\_COLS;**

**7 }**

if的判断条件是当前的输出内容大于屏幕的一页，如果成立，那么就要将整个内容往上移一行，空出最后一行用于输出新的内容，for循环就是在将最后一行都置为空，最后再修改当前的输出位置。

**3.For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.**

**Trace the execution of the following code step-by-step:**

**int x = 1, y = 3, z = 4;**

**cprintf("x %d, y %x, z %d\n", x, y, z);**

**In the call to cprintf(), to what does fmt point? To what does ap point?**

**List (in order of execution) each call to cons\_putc, va\_arg, and vcprintf. For cons\_putc, list its argument as well. For va\_arg, list what ap points to before and after the call. For vcprintf list the values of its two arguments.**

fmt指的就是字符串，即 x %d, y %x, z %d\n，ap指的是输入参数的集合，包括x，y，z。

cons\_putc参数是字符串中的每个字符。在cprintf中调用了 vcprintf，这个函数的输入参数是fmt和ap，即字符串和输入参数的集合，与cprintf相同，在 vcprintf中，又调用了vcprintfmt，第一个参数是函数指针，能够显示字符，在这里调用了自定义的putch函数，第二个参数原本是一个内存地址，但是在这里用做计数器，第三个参数和第四个参数是fmt和ap，同上。在vcprintfmt函数中的getint函数中调用了va\_arg函数，va\_arg函数会从ap列表中取出下一个参数，每次调用后ap就会指向下一个参数。

**4.Run the following code.**

**unsigned int i = 0x00646c72;**

**cprintf("H%x Wo%s", 57616, &i);**

**What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise.**

输出是He110，World首先是按十六进制输出57616，为e110，然后是按字符串输出变量i的地址，x86是小端模式，所以从低处到高处依次是0x72,0x6c,0x64,0x00，即为rld‘/0’

5.In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

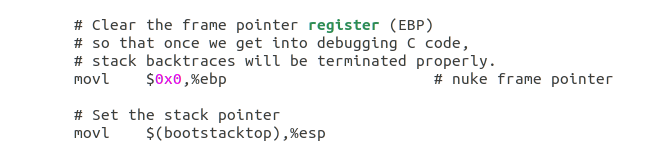
cprintf("x=%d y=%d", 3);

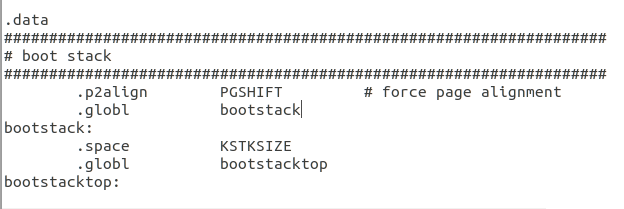


图15

如图15，输出的是一个随机数，这是因为参数y没有被指定。

**Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?**

图16

图17

esp和ebp寄存器是跟堆栈相关的两个非常重要的寄存器，前者指向的是整个堆栈中正在被使用的部分的最低地址，后者用于记录每一个程序的栈帧的相关信息。因此，初始化栈肯定要初始化这两个寄存器。在bootmain函数进行完后跳到entry，即进入到entry.S，其中如图16所示对这两个寄存器进行了初始化.在entry.S的末尾声明了bootstack的大小为KSTKSIZE=8\*PGSIZE=32kb，如图17所示，这个大小即为内核对栈保留的空间。栈向下添加数据，因此栈指针指向最高地址，初始化为bootstacktop。

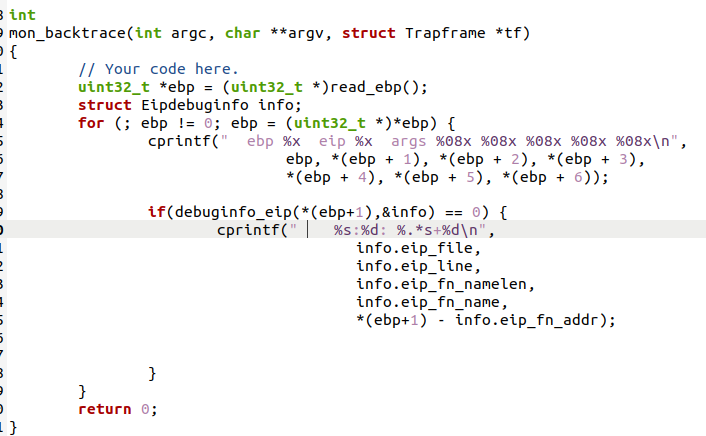


图18

图18是对mon\_backtrace的修改，图19是输出的结果。

debuginfo\_eip中的\_\_STAB\_\*在kernel.ld中定义，定义了\_STAB\_BEGIN\_，STAB\_END\_，STABSTR\_BEGIN\_，\_STABSTR\_END\_等。

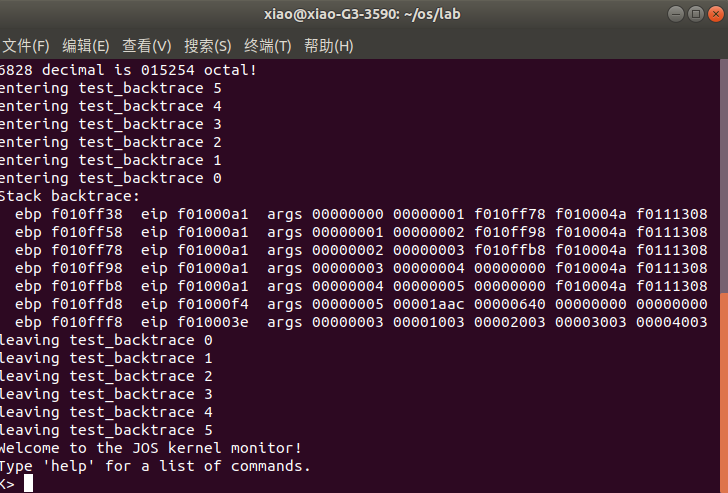
图19

图20

图20是最后的输出结果