JOS 2016 Lab 1实习报告

陈一茹 1400012976

October 3, 2016

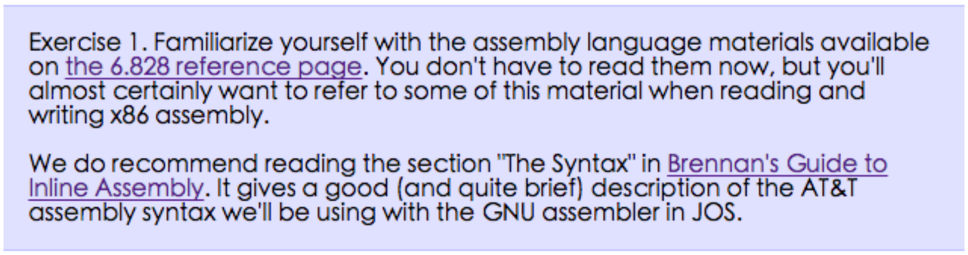
Contents

1 Introduction

Lab 1 分成三部分， 第一部分是熟悉x86汇编语言，QEMU x86模拟器， 和系统启动和加载内核的过程。第二部分是检测位于6.828内核的boot loader. 第三部分是 深入研究6.828内核—- JOS的初步模板。

2 Part 1: PC Bootstrap

2.1 Getting Started with x86 assembly


Exercise 1. Familiarize yourself with the assembly language materials available on the 6.828 reference page. You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.

We do recommend reading the section "The Syntax" in Brennan's Guide to Inline Assembly. It gives a good (and quite brief) description of the AT&T assembly syntax we'll be using with the GNU assembler in JOS.

这部分要求去熟悉一下汇编语言， 给出Intel 和 AT&T的区别。 幸运的是，我只在ICS上学过AT&T汇编，所以没什么好混淆的。 另外， 有助教发的《Intel® 64 and IA-32 Architectures Software Developer's Manual》， 我就没有重复下载， 80386 Reference网页版看起来会更加友好一些。

2.2 Simulating the x86

介绍了QEMU模拟器和GDB调试，我们之后就要在用这个去做Lab。 同时，我们要写的JOS也是可以在真实的机器上运行的。

命令行里输入$make，会编译好boot loader 和 kernel。 同时生成一个obj文件夹，包含 obj/kern/kernel.img内核镜像，这将是我的virtual hard disk。

在镜像运行我的qemu, $make qemu， 我的terminal和 QEMU都用输出， 鼠标没了， 然后Ctrl + Alt 鼠标会出现，但是不接受鼠标操作，鼠标点击一下就有没有了。 Ctrl+a x 是退出(我的虚拟机好像有问题， 退出的时候并不能退出，一点反应都没有，换了一台实验室的电脑继续，摆脱了不稳定的虚拟机)。 $make qemu-nox 可以不显示 the virtual VGA。

只支持两条命令help， kerninfo。

2.3 The PC's Physical Address Space

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

最早的PC机16位，只有1MB的物理内存 0x00000000 - 0x000FFFFF，低640KB是被划分成"Low Memory"， 但当时可能还用不到那么大的。 384KB 的空间从 0x000A0000 到 0x000FFFFF是为硬件保留的，像BIOS之类的。

（我就好奇了，为什么16位的机器能管20位的地址，然后查的8086是36根总线， 16根数据线， 20根地址线）

当现代的电脑突破了1MB的限制之后，为了保持向后兼容性。更多的PC机有了一个"洞"的概念（0x000A0000 to 0x00100000）， 把传统内存（低内存）和高内存分开。另外， 在32最高的地址区域， 可能没有RAM对应，通常是被保留给PCI设备的。所以，最近有超过4G的物理RAM，那么就有再有一个洞。 但是JOS只用256MB，所以我就不用管很麻烦了，直接假设只有32位物理地址。

2.4 The ROM BIOS

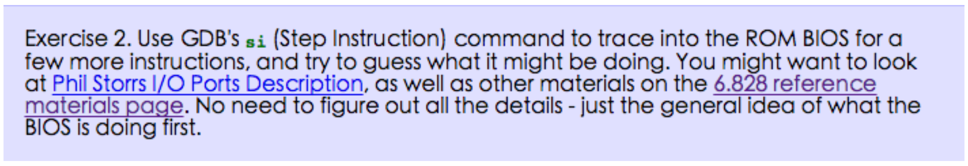
没学过计组的我默默去wiki了一下 BIOS 是什么——BIOS是个人电脑启动时加载的第一个软件。 BIOS用于电脑开机时运行系统各部分的的自我检测(Power On Self Test)，并加载引导程序(IPL)或存储在主存的作业系统，比如说这里就加载了boot loader。此外，BIOS还向作业系统提供一些系统参数。系统硬件的变化是由BIOS隐藏，程序使用BIOS功能而不是直接控制硬件。现代作业系统会忽略BIOS提供的抽象层并直接控制硬件组件。

用QEMU的调试工具， 按要求开两个terminal 一个$make qemu-gdb 启动qemu, 等待GDB的连接。 另一个 $make gdb。

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

出现了这一行，这是PC在启动执行的第一条语句，当前指令CS = 0xf000 and IP = 0xfff0即0xffff0，ljmp指令 跳转到CS = 0xf000 and IP = 0xe05b。0xffff0 只比结束0x100000少了16个字节，所以当然要跳转到一个叫靠前的地方。毕竟，16个字节能做什么呀?

另外，实模式的寻址是：physical address = 16 * segment + offset。（16*CS+IP）保护模式寻址方式是：段基址+偏移。当PC机启动时，CPU运行在实模式(real mode)下，而当进入操作系统内核后，将会运行在保护模式下(protected mode)。


Exercise 2. Use GDB's `si` (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at Phil Storrs I/O Ports Description, as well as other materials on the 6.828 reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

这里就是让我们熟悉一下GDB的si，发现就是ICS里面学习的stepi, 看几步之后的执行，可以看到从i8086模式一直进入到保护模式 i386，我贴一下跟踪出来的代码：

```
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6ac8
[f000:e062]    0xfe062: jne    0xfd2e1
[f000:e066]    0xfe066: xor    %dx,%dx
[f000:e068]    0xfe068: mov    %dx,%ss
[f000:e06a]    0xfe06a: mov    $0x7000,%esp
[f000:e070]    0xfe070: mov    $0xf34c2,%edx
[f000:e076]    0xfe076: jmp    0xfd15c
[f000:d15c]    0xfd15c: mov    %eax,%ecx
[f000:d15f]    0xfd15f: cli
[f000:d160]    0xfd160: cld
[f000:d161]    0xfd161: mov    $0x8f,%eax
[f000:d167]    0xfd167: out    %al,$0x70
[f000:d169]    0xfd169: in     $0x71,%al
[f000:d16b]    0xfd16b: in     $0x92,%al
[f000:d16d]    0xfd16d: or     $0x2,%al
[f000:d16f]    0xfd16f: out    %al,$0x92
[f000:d171]    0xfd171: lidtw  %cs:0x6ab8
[f000:d177]    0xfd177: lgdtw  %cs:0x6a74
[f000:d17d]    0xfd17d: mov    %cr0,%eax
[f000:d180]    0xfd180: or     $0x1,%eax
[f000:d184]    0xfd184: mov    %eax,%cr0
[f000:d187]    0xfd187: ljmpl  $0x8,$0xfd18f
The target architecture is assumed to be i386
=> 0xfd18f: mov    $0x10,%eax
=> 0xfd194: mov    %eax,%ds
=> 0xfd196: mov    %eax,%es
=> 0xfd198: mov    %eax,%ss
=> 0xfd19a: mov    %eax,%fs
=> 0xfd19c: mov    %eax,%gs
=> 0xfd19e: mov    %ecx,%eax
=> 0xfd1a0: jmp    *%edx
=> 0xf34c2: push   %ebx
=> 0xf34c3: sub    $0x2c,%esp
=> 0xf34c6: movl   $0xf5b5c,0x4(%esp)
=> 0xf34ce: movl   $0xf447b,(%esp)
=> 0xf34d5: call   0xf099e
=> 0xf099e: lea    0x8(%esp),%ecx
=> 0xf09a2: mov    0x4(%esp),%edx
=> 0xf09a6: mov    $0xf5b58,%eax
=> 0xf09ab: call   0xf0574
=> 0xf0574: push   %ebp
=> 0xf0575: push   %edi
=> 0xf0576: push   %esi
=> 0xf0577: push   %ebx
=> 0xf0578: sub    $0xc,%esp
=> 0xf057b: mov    %eax,0x4(%esp)
=> 0xf057f: mov    %edx,%ebp
=> 0xf0581: mov    %ecx,%esi
=> 0xf0583: movsbl 0x0(%ebp),%edx
=> 0xf0587: test   %dl,%dl
=> 0xf0589: je     0xf0758
=> 0xf058f: cmp    $0x25,%dl
=> 0xf0592: jne    0xf0741
=> 0xf0741: mov    0x4(%esp),%eax
=> 0xf0745: call   0xefc70
=> 0xefc70: mov    %eax,%ecx
=> 0xefc72: movsbl %dl,%edx
=> 0xefc75: call   *(%ecx)
=> 0xefc65: mov    %edx,%eax
=> 0xefc67: mov    0xf693c,%dx
=> 0xefc6e: out    %al,(%dx)
=> 0xefc6f: ret
=> 0xefc77: ret
```

贴的比较多就调点主要的说：

f000:d15f 0xfd15f: cli 关中断，启动时的操作是比较关键的，所以肯定是不能被中断的。这个关中断指令用于关闭那些可以屏蔽的中断。比如大部分硬件中断。

f000:d160 0xfd160: cld CLD是用来操作方向标志位DF（Direction Flag）。CLD使DF复位，即DF=0。 若为字节操作MOVSB： SI←SI ± 1, DI←DI ± 1，当方向标志DF = 0时，用"+"；当方向标志DF = 1时，用"−"。 若为字操作MOVSW：SI←SI ± 2, DI←DI ± 2，当方向标志DF = 0时，用"+"；当方向标志DF = 1时，用"−"。

- 这个有三步指令，

```
f000:d161]     0xfd161:  mov     $0x8f,%eax
[f000:d167]     0xfd167: out     %al,$0x70
[f000:d169]     0xfd169: in      $0x71,%al
```

out %al, PortAddress 向端口地址为PortAddress的端口写入值，值为al寄存器中的值

in PortAddres,%al 把端口地址为PortAddress的端口中的值读入寄存器al中

所以上述指令把 0x70 端口输出 0x8f，经查找得知，x70端口和0x71端口是用于控制系统中一个叫做CMOS的设备，这个设备是一个低功耗的存储设备，它可以用于在计算机关闭时存储一些信息，它是由独立的电池供电的。这个CMOS中可以控制跟PC相关的多个功能，其中最重要的就是时钟设备（Real Time Clock）的，它还可以控制是否响应不可屏蔽中断NMI(Non-Maskable Interrupt)。操作CMOS存储器中的内容需要两个端口，一个是0x70另一个就是0x71。其中0x70可以叫做索引寄存器，这个8位寄存器的最高位是不可屏蔽中断(NMI)使能位。如果你把这个位置1，则NMI不会被响应。低7位用于指定CMOS存储器中的存储单元地址，所以如果你想访问第0xF号存储单元，并且在访问时，我要使能NMI，那么你就应该向端口0x70里面送入0b10001111= 0x8F。三条指令可以看出，它首先关闭了NMI中断，并且要访问存储单元0xF的值，并且把值读到al中，但是在后面我们发现这个值并没有被利用，所以可以认为这三条指令是用来关闭NMI中断的。

- 接下来的三步指令

```
[f000:d16b]     0xfd16b: in      $0x92,%al
[f000:d16d]     0xfd16d: or      $0x2,%al
[f000:d16f]     0xfd16f: out     %al,$0x92
```

0x92控制的是 PS/2系统控制端口A，而第16，17步的操作明显是在把这个端口的1号bit置为1。这个端口的bit1的功能是(经查找)

bit 1= 1 indicates A20 active

即A20位，即第21个地址线被使能， google可知，如果A20地址线被激活，那么系统工作在保护模式下。但是在之后的boot loader程序之前，计算机首先要工作在实模式下啊， 到boot loader才会进入保护模式啊。后来查找得知， 这个时候开启32位模式是去去测试可用内存空间。在boot loader之前，它肯定还会转换回实模式。

- lidtw 这条是 load 0xf68f8 into IDTR 载入中断描述符表寄存器,应该是中断描述符表的 物理地址。

- lgdtw 同理是把地址载入全局描述符表寄存器,应该是 GDTR 的物理地址。

-
  ```
  [f000:d17d]    0xfd17d: mov    %cr0,%eax
  [f000:d180]    0xfd180: or     $0x1,%eax
  [f000:d184]    0xfd184: mov    %eax,%cr0
  ```

  计算机中包含CR0 CR3四个控制寄存器，用来控制和确定处理器的操作模式。其中这三个语句的操作明显是要把CR0寄存器的最低位(0bit)置1。CR0寄存器的0bit是PE位，启动保护位，当该位被置1，代表开启了保护模式。觉得很奇怪， 我想查看cr0的PG值，但是好像被保护，并不能显示出值来。姑且把它当成事未开启分页的保护模式。**为什么这里BIOS就进入了保护模式呢，之后boot loader又一次开启了保护模式，这里有点奇怪** 我搜了一下stack，居然有人和我问的问题一样，他说这只是利用一下保护模式的特性，毕竟保护模式比实模式的功能多好多，然后在进入bootloader之前又会换回实模式。

- The target architecture is assumed to be i386

  ```
  => 0xfd18f: mov    $0x10,%eax
  => 0xfd194: mov    %eax,%ds
  => 0xfd196: mov    %eax,%es
  => 0xfd198: mov    %eax,%ss
  => 0xfd19a: mov    %eax,%fs
  => 0xfd19c: mov    %eax,%gs
  ```

  之后修改这些寄存器的值。这些寄存器都是段寄存器。如果刚刚加载完GDTR寄存器我们必须要重新加载所有的段寄存器的值，而其中CS段寄存器必须通过长跳转指令，即23号指令来进行加载。所以这些步骤是在第19步完成后必须要做的。这样才能是GDTR的值生效。

- 最后单步跟踪进入了循环，上述代码只显示了循环中的一次, 猜测这是在把磁盘上的 boot loader 文件拷贝到内存中。

3 Part 2: The Boot Loader

PC 的软盘和硬盘都被分成512字节， 每512字节叫做一个sector, 一个sector是磁盘的最小传送单元。 BIOS完成一系列的初始化工作后， 会把512字节的boot sector 放到 0x7c00开始的一段空间里面， 然后用jmp 命令 跳转到0x7c00，即将控制转移给boot loader。

Boot loader包含汇编语言和C语言: boot/boot.S, boot/main.c。然后，我就去详细地看了一下这两个文件。这两个文件中boot/boot.S首先执行。

boot.S刚开始的代码都是实模式下的，之后呢，设置一些常数，为了转换成保护模式做准备 CR0_PE_ON为1。PROT_MODE_CSEG，PROT_MODE_DSEG设置数据段，代码段基址。 设立全局的start标志，执行cli,cld。 初始化重要的寄存器如DS, ES, SS。

```
# Enable A20:
  #   For backwards compatibility with the earliest PCs, physical
  #   address line 20 is tied low, so that addresses higher than
  #   1MB wrap around to zero by default.  This code undoes this.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

这一段Enable A20。 好吧，我又来补计算机组成的课了。说是为了向后兼容，Line 20现在是关闭的，所以超过1M的地址都转向了对1M求模的地址， 现在这些代码将A20打开。

```
 # Switch from real to protected mode, using a bootstrap GDT
  # and segment translation that makes virtual addresses
  # identical to their physical addresses, so that the
  # effective memory map does not change during the switch.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE_ON, %eax
  movl    %eax, %cr0
```

这段首先是利用lgdt将全局描述符的地址、长度信息加载到GDTR寄存器中，为打开保护模式先做准备。 然后又将CR0寄存器中PE位设为1。从而就打开了保护模式。

GDT定义在这个文件的末尾部分：

```
# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULL        # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
  SEG(STA_W, 0x0, 0xffffffff)         # data seg

gdtdesc:
  .word   0x17                            # sizeof(gdt) - 1
  .long   gdt                              # address gdt
```

.word 的作用可以理解为设置一个大小为 word 的数据，由于 0x17 = 23，所以 GDTR 中的 Size 为 24，即存在 3 个 GDT Entry .long 的作用与 .word 同理

前面几行，分别设置了NULL段，代码段，数据段，这里调用了 SEG 函数，相关定义可以从 mmu.h 中找到 上面几行执行的结果大概可以视作如下：

GDT[0] = base = 0x0, limit = 0x0, type = 0x0 null段

GDT[1] = base = 0x0, limit = 0xfffffff, type = 0xA 代码段可执行可写

GDT[2] = base = 0x0, limit = 0xfffffff, type = 0x2 数据段可读可写

```
# Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg
```

这段代码ljmp及跳转到代码段的protcseg段，也就是.code32的位置，从而在保护模式下运行。$PROT_MODE_CSEG定义在本文件的开始，为0x8， 正好落在GDT代码段的位置里。我看了一下我在boot.asm中的地址，首先我发现x/10i是不可以的，因为刚开始执行的是BIOS，和这个boot.S 的执行代码不在一个段里，所以x/10i就找不到正确的代码了，后来改用b * , 但是我以为0x7c24是它的地址，但是后来也一直continue，并没有找到，后来发现，0x7c24是无效的代码，是不执行的， 因该是0x7c23。

```
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7c32
0x00007c2d in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c32:  mov    $0x10,%ax
0x00007c32 in ?? ()
(gdb)
=> 0x7c36:  mov    %eax,%ds
```

跟踪出来，我发现， 虽然在执行ljmp之前已经改了cr0的最后一位PE值，但是并没有真正进入保护模式，ljmp之后，才显示进入了保护模式。**很奇怪啊，我不得不查找相应的资料，后来，还是在 stack 上找到了答案，感觉很靠谱啊： ljmp is one of the only ways to change the cs register. This needs to be done to activate protected mode, as the cs register needs to contain the selector for a code segment in the GDT. (In case you want to know, the other ways to change cs are far call, far return, and interrupt return)**

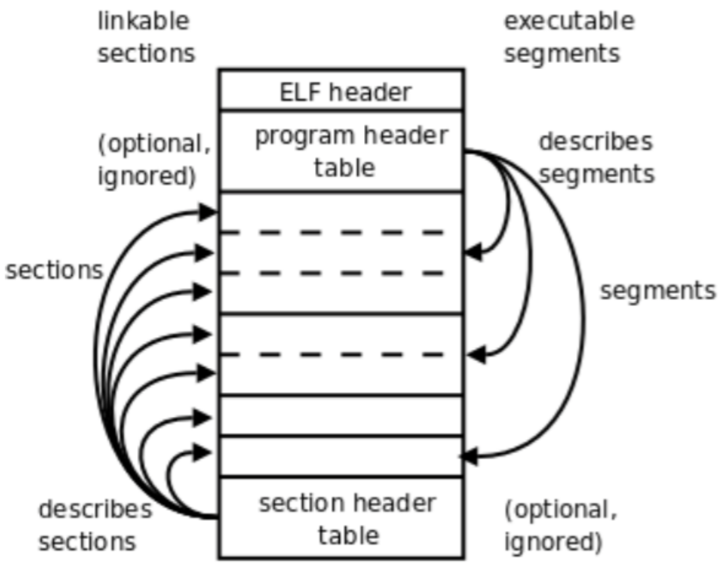
另外有一点需要注意的是： 保护模式下的CS值是不能被设成0x0的，以防进入空字段。但是这里能够正确的执行，这是因为进入的时候就是0x0了，并不非法。

```
# Set up the stack pointer and call into C.
```

```
  movl    $start, %esp
  call bootmain

  # If bootmain returns (it shouldn't), loop.
spin:
  jmp spin
```

设置相应的寄存器，和C的堆栈，执行bootmain函数。

好了，现在该看boot/main.c文件,这个文件的唯一工作就是将JOS内核读入内存，然后将控制权交给JOS内核。JOS内核就是编译的出来的kernel.img，这是一个ELF格式的文件。 先来看一下elf的文件格式：



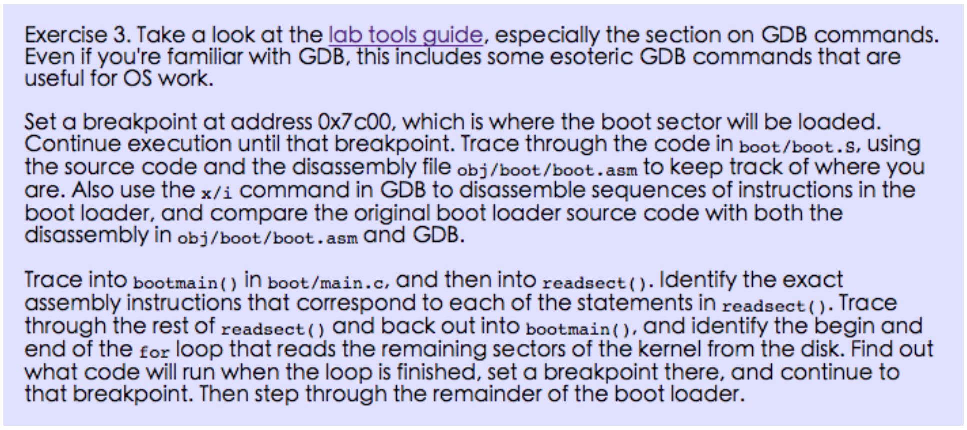
图片来源：Linux C 编程作者 宋劲斌

另外还有一些概念：

- uint32_t e_magic; must equal ELF_MAGIC
- uint32_t e_entry; 程序入口地址
- uint32_t e_phoff; program headers的偏移
- uint16_t e_phnum; program headers的数量
- uint32_t p_offset; 段偏移
- uint32_t p_pa; 给出segment的物理地址
- uint32_t p_memsz; segment 的size

有了这些基础知识作为准备，我就能看懂这段main.c代码了。首先，bootmain()前半部分先将前8个sector的信息从硬盘中读入，放置在ELFHDR指向的空间内，#define ELFHDR ((struct Elf *) 0x10000) ，有这个可知，这个是从0x10000开始的区栈，大概是当个临时的区栈。然后根据其magic 校验码，判断是否合法。 接着，根据ELFHDR的e_phoff字段和e_phnum字段找到程序头的起始地址和个数，找到程序头表。 最后， 通过程序头表的相关信息，通过的readseg()函数将JOS内核以Segment为单位读入内存。最后，跳转到e_entry字段指向的位置，将控制权交给JOS内核。JOS内核加载完毕，Bootloader的工作加载完毕。

到现在为止，我们已经进入到了JOS内核了。


Exercise 3. Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `boot/boot.S`, using the source code and the disassembly file `obj/boot/boot.asm` to keep track of where you are. Also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.

Trace into `bootmain()` in `boot/main.c`, and then into `readsect()`. Identify the exact assembly instructions that correspond to each of the statements in `readsect()`. Trace through the rest of `readsect()` and back out into `bootmain()`, and identify the begin and end of the `for` loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

根据他的要求，设置断点，跟踪，之前，代码有不解的地方的时候已经设置过了，轻车熟路了，但是看到这里才后悔，没早点看到这段话，他说要在0x7c00设置断点，我刚刚还因为0x7c24设断点，没有正确的结果而纠结了好久，要是早看到这个就不纠结了= =。。

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

- 从 boot.S 可以看出,进入 boot loader 之后,置%cr0 的 PE 为 1,然后ljmp$PROT_MODE_CSEG, $protcseg 的作用是设置 cs: eip(因为没有其他方式能改变cs 寄存器),开始进入了保护模式。导致实模式切换到保护模式的最根本原因应该就是把 PE 位置为 1,允许保护模式。

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

- 最后一条指令是(ELFHDR->e_entry))();对应 call *0x10018。这是用来存放着 e_entry, 我设置断点并跟踪到了 0x10000c, movw $0x1234, 0x472这就是 kernel 的第一条指令的位置。

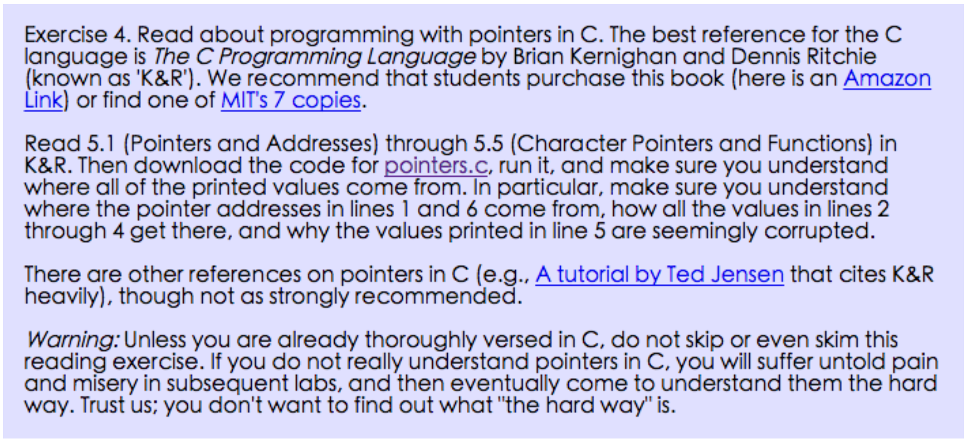
3. Where is the first instruction of the kernel?

- 由上题 0x10000c, movw $0x1234, 0x472

4. 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?

- 前面讲ELF的格式的时候已经说过了， 从bootmain() 函数中就可以看出来了，boot loader 先读取 kernel 的 ELF 文件的文件头,然后通过文件头信息直接可以知道 e_phnum 为 segment 的数量。

3.1 Loading the Kernel


Exercise 4. Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an Amazon Link) or find one of MIT's 7 copies.

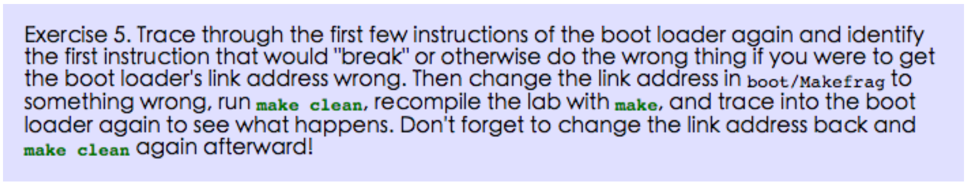
Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for pointers.c, run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in lines 1 and 6 come from, how all the values in lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.

There are other references on pointers in C (e.g., A tutorial by Ted Jensen that cites K&R heavily), though not as strongly recommended.

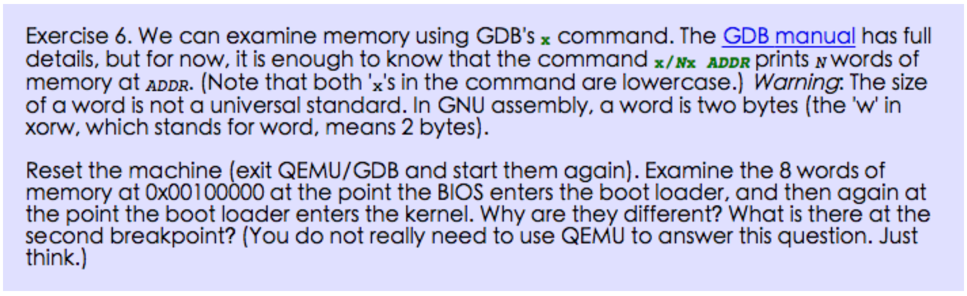
Warning: Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

回顾C指针的用法， 下载了print.c 看了看， 有几点要注意的是没见到的用法3[c] = 302，这里相当于c[3]，内存是小端法， 比较难的是 c = (int *) ((char *) c + 1); *c = 500; 表示500这个数会覆盖a[1]的最高三个byte和a[2]的最低一个byte。

接着， 要求用objdump看看文件信息。就不展示了。


Exercise 5. Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in `boot/Makefrag` to something wrong, run `make clean`, recompile the lab with `make`, and trace into the boot loader again to see what happens. Don't forget to change the link address back and `make clean` again afterward!

我把Makefrag 里面的-Ttext 参数改成了 0x7c10, 结果程序异常退出。输入b*0x7c00发现,bootloader 还是被正常加载到 0x7c00 位置, 开始几条指令都没有问题。一直到置%cr0 的 PE 位进入保护模式都可以,而在 ljmp 时 出错了。原来的指令是 ljmp $0x8, $0x7c32 现在是 ljmp $0x8, $0x7c42,地址不同,结果就出现错误。分析原因是， 之前的是不涉及相对位置的， 但是这句话涉及相对位置， 所以改了就会出错了。



Exercise 6. We can examine memory using GDB's `x` command. The GDB manual has full details, but for now, it is enough to know that the command `x/Nx ADDR` prints *N* words of memory at *ADDR*. (Note that both 'x's in the command are lowercase.) *Warning*: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).

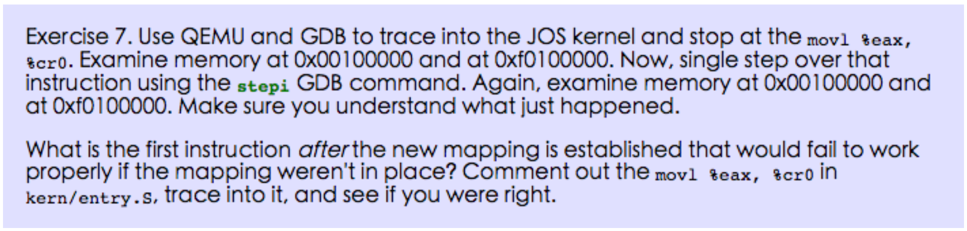
Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

```
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000: 0x00000000  0x00000000  0x00000000  0x00000000
0x100010: 0x00000000  0x00000000  0x00000000  0x00000000
(gdb) b* 0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:  movw   $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000: 0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010: 0x34000004  0x0000b812  0x220f0011  0xc0200fd8
```

这个发生了变化，在刚进入 bootloader 时,0x100000 处的内容全为 0; 但是在进入kernel 之时, 此处的内容就被修改了。不同的原因就在于刚进入bootloader时还没有进行拷贝工作所以全零, 而之后 bootloader 把 kernel 里的内容从磁盘拷贝到了这片存储区域,由 e_entry 可知 kernel 的第一条语句位于 0x10000c,所以说明这里存放着 kernel 的 ELF 文件内容。

4 Part 3: The Kernel

现在我们进入JOS内核了。 在页表存储设备中，程序的链接地址和加载地址必须要一样，但我们发现，JOS内核加载和链接的地址都是不同的，这样设计是因为操作系统内核一般习惯在高地址运行，这样可以把低地址留给用户的程序。 另外要注意，开启保护模式，不是意味着是用虚拟地址，实际上要开启CR0_PG才能从物理地址转换成使用虚拟地址寻找。



Exercise 7. Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the `stepi` GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.

What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the `movl %eax, %cr0` in `kern/entry.S`, trace into it, and see if you were right.

```
=> 0x100025:  mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) x/x 0x00100000
0x100000: 0x1badb002
(gdb) x/x 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000
(gdb) si
=> 0x100028:  mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/x 0x00100000
0x100000: 0x1badb002
(gdb) x/x 0xf0100000
0xf0100000 <_start+4026531828>: 0x1badb002
```

在movl %eax, %cr0 指令执行之前,利用 x/x 查看 0x00100000 有内容,但是 0xF0100000 全为 0。在执行这条语句之后,0xf0100000 的内容和 0x00100000 处的内容 就一样了。 因为这句话开启了CR0_PG 位，所以两个地址对应到了同一个的物理地址。

如果把kern/entry.S 中的 movl %eax, %cr0 注释掉之后, 再运行就 Aborted 了。原因是没有开启 CR0_PG 位,所以虚拟地址没有被正确翻译成物理地址,导致找不到正确的指令,于是程序崩溃退出。

4.1 Formatted Printing to the Console

这里要求我们研究一下printf()函数，涉及三个文件 kern/printf.c, lib/printfmt.c, and kern/console.c。 实现的是printf.c里面的cprintf()函数:

```c
int cprintf(const char *fmt, ...)
{
  va_list ap;
  int cnt;
  va_start(ap, fmt);
  cnt = vcprintf(fmt, ap);
  va_end(ap);
  return cnt;
}
```

其中，va_list、va_start、va_end是变长参数的类型和宏。 cprintf() 调用了 vcprintf() 进行输出，vcprintf() 进一步又调用了vprintfmt(), 其中putch函数指针真正在终端显示字符了

```c
static void putch(int ch, int *cnt)
{
  cputchar(ch);
  *cnt++;
}
int vcprintf(const char *fmt, va_list ap)
{
  int cnt = 0;

  vprintfmt((void*)putch, &cnt, fmt, ap);
  return cnt;
}
```

其中，cputchar()函数是在console.c中的。

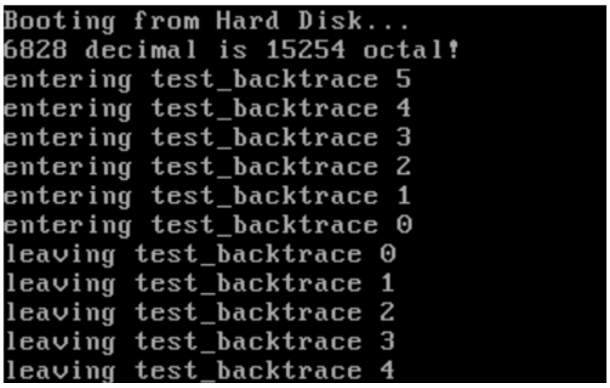
有个Exercise：

Exercise 8. We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

解答：

```
// (unsigned) octal
case 'o':
  num = getuint(&ap,lflag);
  base = 8;
  goto number;
```

他原来是输出XXX，现在替换之后运行：



```
Booting from Hard Disk...
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
```

这个很简单。接下来一连串问题：

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?

- 正如我上面所说，putch() 函数调用了 console.c的cputchar()函数，相当于对cputchar()函数进行封装，加上了 int *cnt 作为输出计数。

2. Explain the following from console.c:

```
if (crt_pos >= CRT_SIZE) {
  int i;
  memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) *
      sizeof(uint16_t));
  for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
    crt_buf[i] = 0x0700 | ' ';
    crt_pos -= CRT_COLS;
}
```

- 我用grep -r "CRT_SIZE"搜索，在console.h 中找到

```
#define CRT_ROWS   25
#define CRT_COLS   80
#define CRT_SIZE   (CRT_ROWS * CRT_COLS)
```

所以CRT_SIZE 就是屏幕能够容纳的字符数，CRT_COLS一行能显示的字符数，crt_pos光标位置。 然后为什么是0x0700 和 uint16_t呢，查资料知道： 高 8 位应该用于指定颜色,而低 8 位才是真正的字符。所以这段代码的意思就是：如果光标位置 crt_pos 大于 CRT_SIZE, 就把输出缓冲区整体向上移动一行, 最下面空出来的一行用空格填满,颜色是默认的 0x07,然后光标定位到末尾行的最左端。

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);
```

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

- fmt 指向的就是 format string,即"x %d, y %x, z %d \n",查了之后我发现，ap的流程是这样的，将所有参数从右到左压栈，然后va_start()指向后面参数列表的第一个参数，每次需要读取，用va_arg()宏ap从栈中弹出一个参数。所以，ap 指向第一个参数 x,调用完之后用va_end(ap)关闭 ap,使它指向 null(因为没有参数了)。

(2)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.

- vcprintf("x %d, y %x, z %d \n", x, y, z);
- cons_putc('x')
- cons_putc(' ')
- va_arg():ap从栈里弹出x
- cons_putc('1')
- cons_putc(',')
- cons_putc(' ')
- cons_putc('y')

- cons_putc(' ')
- va_arg():ap从栈里弹出y
- cons_putc('3')
- cons_putc(',')
- cons_putc(' ')
- cons_putc('z')
- cons_putc(' ')
- va_arg():ap从栈里弹出z
- cons_putc('4')
- cons_putc('\n')

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. Here's an ASCII table that maps bytes to characters. The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

Here's a description of little- and big-endian and a more whimsical description.

- 输出是 He110 World, 这里的57616 用十六进制表示就是 e110, 然后把 i 作为 字符串输出,小端法机器中的表示是 72 6c 64 00 对应 ASCII 码是 r l d (null), 正好打印出这个字符串。 如果是大端法机器,不用改57616，因为这本来就是一个数。因为是按照字符串输出,所以要想输出正确的值，需要把修改成0x726c6400。

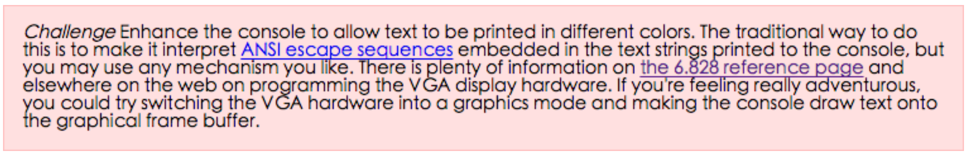
5. In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen? cprintf("x=

- y之后的结果会是一个不确定数。这是因为， 在输入完3之后，ap已经指向了不确定的内存空间，这部分空间的值是随机的。

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

- 如果更改了将参数压栈的顺序，那么只需要更改va_start(), va_arg(), 将ap 一开始就指向这些元素的最底部，然后每次读取参数的时候从栈底向栈顶移动ap指针，输出后清空参数。

4.2 Challenge


Challenge Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret ANSI escape sequences embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on the 6.828 reference page and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

- 之前提到，控制台打印字符的时候高 8 位为颜色,低 8 位为字符，为了实现不同的颜色， if (!(c & 0xFF)) c |= COLOR; 我们希望能够在格式字符串 fmt 中指定不同的字符串打印不同 的颜色,所以这就需要改进一下 fmt 的格式。

我上网查了查，C 语言有这样的 \033[字背景颜色;字体颜色 字符串的用法，我就模仿一个，写一下

主要就是改一下fmt格式，我定义了一个#include <inc/color.h>的头文件

```
int FG_COLOR;
int BG_COLOR;
int COLOR;
```

只有三行，用来定义一下背景色和字体色和合起来的颜色。

有了这些变量， 我接着在printfmt.c里面添加了对于颜色的识别和赋值语句，如下

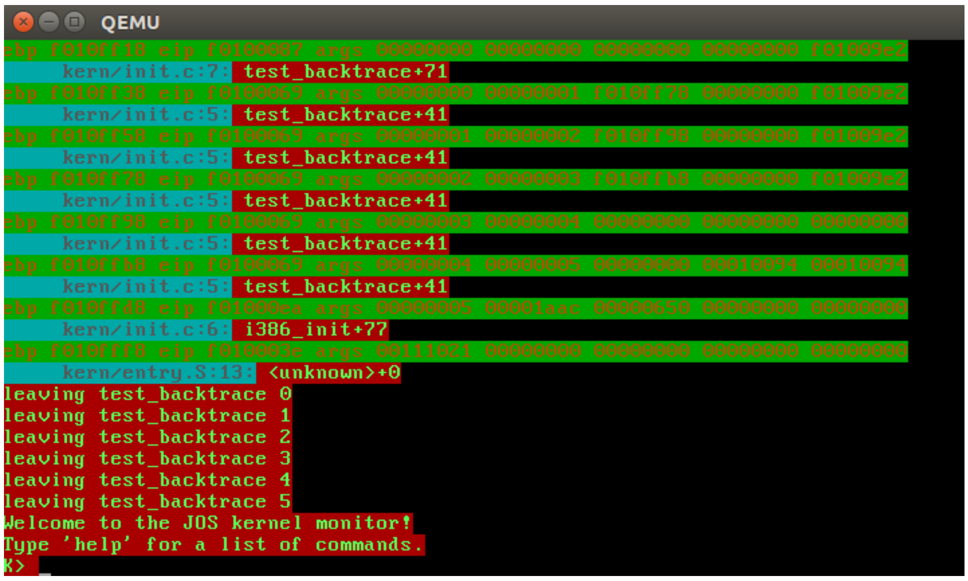
```
while (1) {
    while ((ch = *(unsigned char *) fmt++) != '%') {
      if (ch == '\0')
        return;
      else if(ch == '\033'){
        if((ch = *(unsigned char *) fmt++) != '[') {
            putch(ch, putdat);
            continue;
        }
        BG_COLOR = *(unsigned char *) fmt++;
        FG_COLOR = *(unsigned char *) fmt++;

        if(BG_COLOR >= '0' && BG_COLOR <= '9')
            BG_COLOR -= '0';
        else if(BG_COLOR >= 'a' && BG_COLOR <= 'f')
            BG_COLOR = BG_COLOR - 'a' + 10;
        else if(BG_COLOR >= 'A' && BG_COLOR <= 'F')
            BG_COLOR = BG_COLOR - 'A' + 10;
        else BG_COLOR = 0;

        if(FG_COLOR >= '0' && FG_COLOR <= '9')
            FG_COLOR -= '0';
        else if(FG_COLOR >= 'a' && FG_COLOR <= 'f')
            FG_COLOR = FG_COLOR - 'a' + 10;
        else if(FG_COLOR >= 'A' && FG_COLOR <= 'F')
```

```
                FG_COLOR = FG_COLOR - 'A' + 10;
            else BG_COLOR = 7;

            COLOR = (BG_COLOR << 12) | (FG_COLOR << 8);
            continue;
          }
          putch(ch, putdat);
        }
```

如果有 \033[字背景颜色;字体颜色, 就会显示用户自定义的颜色，没有写的话，紧跟上文的颜色。

比如说，我对mon_backtrace()附这样的值，效果是这样的

```
    cprintf("\033[26ebp %08x eip %08x args %08x %08x %08x %08x
        %08x\n",ebp,eip,ary[0],ary[1],ary[2],ary[3],ary[4]);
    cprintf("\033[38  %s:%d:", eip_info.eip_file, eip_info.
        eip_line);
    cprintf("\033[4a %.*s+%d\n", eip_info.eip_fn_namelen,
        eip_info.eip_fn_name, eip - eip_info.eip_fn_addr);
```



```
ebp f010ff18 eip f0100087 args 00000000 00000000 00000000 00000000 f01009e2
       kern/init.c:7: test_backtrace+71
ebp f010ff38 eip f0100069 args 00000000 00000001 f010ff78 00000000 f01009e2
       kern/init.c:5: test_backtrace+41
ebp f010ff58 eip f0100069 args 00000001 00000002 f010ff98 00000000 f01009e2
       kern/init.c:5: test_backtrace+41
ebp f010ff78 eip f0100069 args 00000002 00000003 f010ffb8 00000000 f01009e2
       kern/init.c:5: test_backtrace+41
ebp f010ff98 eip f0100069 args 00000003 00000004 00000000 00000000 00000000
       kern/init.c:5: test_backtrace+41
ebp f010ffb8 eip f0100069 args 00000004 00000005 00000000 00010094 00010094
       kern/init.c:5: test_backtrace+41
ebp f010ffd8 eip f01000ea args 00000005 00001aac 00000650 00000000 00000000
       kern/init.c:6: i386_init+77
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
       kern/entry.S:13: <unknown>+0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _
```

这样就完成了challenge啦!

4.3 The Stack

x86内核栈指针从高往低增长。

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?

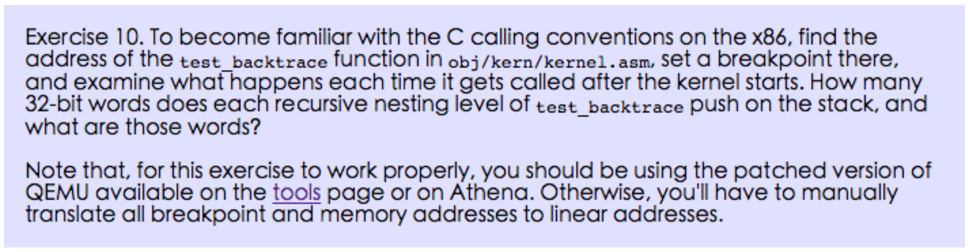
内核栈在kern/entry.S中被初始化。

```
movl  $0x0,%ebp     # nuke frame pointer
# Set the stack pointer
movl  $(bootstacktop),%esp
```

栈定义在kern/entry.S的尾部：

```
.data
####################################################
# boot stack
####################################################
  .p2align  PGSHIFT   # force page alignment
  .globl    bootstack
bootstack:
  .space    KSTKSIZE
  .globl    bootstacktop
bootstacktop:
```

PGSIZE 可以在 mmu.h 中找到,为 4096(4KB)。所以 kernel 的栈的大小为 8*PGSIZE =32KB, %esp初始化为栈的最高地址处(bootstacktop),然后栈是从高地址向 低地址方向“增长”的。


Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the `test_backtrace` function in `obj/kern/kernel.asm`, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words?

Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the tools page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

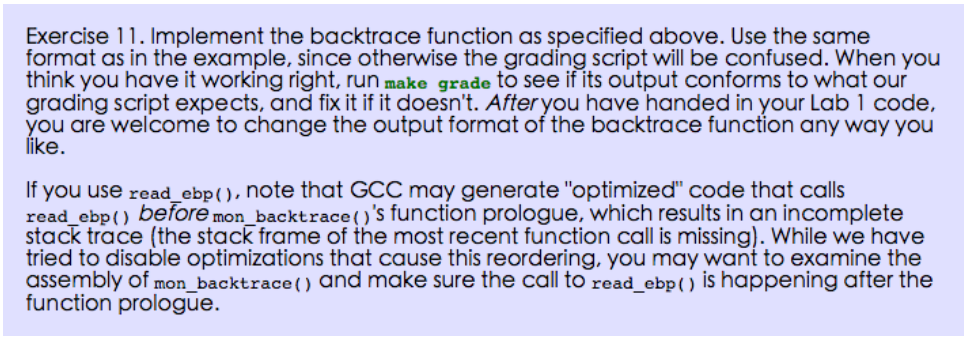
打开 kernel.asm 查找可知 test_backtrace 从 0xf0100040 开始,于是在此设断点。

1. 每次进入 test_backtrace 函数时都会先 push %ebp 和 push %ebx 保存两个 32-bit 寄存器。

2. 之后 sub $0x14, %esp 开辟了 20-byte 的空间。

3. 在执行call指令时也会自动把返回地址(%eip)压栈

于是 test_backtrace 函数的栈帧就一共有 4+4+20+4=32 bytes 入栈。



Exercise 11. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run `make grade` to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

If you use `read_ebp()`, note that GCC may generate "optimized" code that calls `read_ebp()` *before* `mon_backtrace()`'s function prologue, which results in an incomplete stack trace (the stack frame of the most recent function call is missing). While we have tried to disable optimizations that cause this reordering, you may want to examine the assembly of `mon_backtrace()` and make sure the call to `read_ebp()` is happening after the function prologue.

JOS内核会在启动时候调用test_backtrace(),当函数递归的执行若干次之后，会执行mon_trace()函数。mon_backtrace()的作用是逆序讲当前函数的调用链输出，即输出当前函数（mon_trace()本身）的信息，然后在输出调用的上层的信息。一直到JOS内核启动为止。输出信息：栈的基址ebp, 函数返回后继续执行的指令地址eip, 以及函数调用时的前五个参数（不足者补0）。

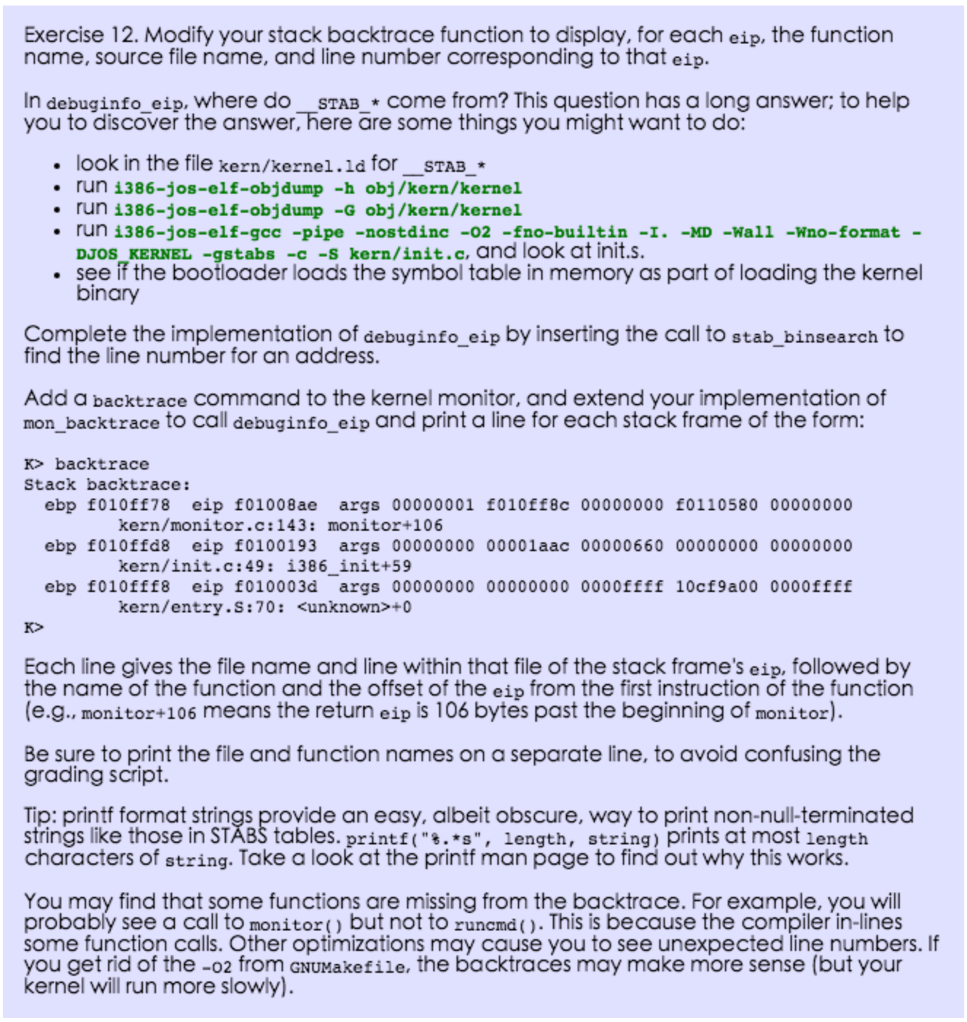
test_backtrace()中的32个字节，我们在之前已经说过了。现在我们按顺序输出这些信息。结束的条件可以在entry.S 中找到，就是%ebp等于零的时候。那么这段代码就不难写了：

```c
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
  int *ebp, eip, *old_ebp;
  int ary[5]={};

  cprintf("Stack backtrace:\n");

  ebp=(int *)read_ebp();
  while((int)ebp!=0)
  {
    old_ebp=(int *)*(ebp);
    eip=*(ebp+1);
    for(int i=0;i<5;++i)
    {
      int j=i+2;
      ary[i]=*(ebp+j);
    }
    cprintf("ebp %08x eip %08x args %08x %08x %08x %08x %08x\n"
        ,ebp,eip,ary[0],ary[1],ary[2],ary[3],ary[4]);
    ebp=old_ebp;
  }
  return 0;
}
```

按照顺序调用，很简单，搞清楚栈里参数的位置就可以，注意 arg[4]在最高地址，所以参数依次像上面找就可以了。写完之后，再在Struct Command中添加一一个backtrace的命令。完成了！

```c
static struct Command commands[] = {
  { "help", "Display this list of commands", mon_help },
  { "kerninfo", "Display information about the kernel",
     mon_kerninfo },
  { "backtrace","trace_lab 1", mon_backtrace},
};
```

最后一个exercise, 目前我的backtrace可以给出调用backtrace的所有函数调用的堆栈参数信息。现在我们要向这个加入每个地址的函数名。有一个debuginfo_eip()来帮助我，可以返回这个个地址的调试信息。



Exercise 12. Modify your stack backtrace function to display, for each `eip`, the function name, source file name, and line number corresponding to that `eip`.

In `debuginfo_eip`, where do `__STAB_*` come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- look in the file `kern/kernel.ld` for `__STAB_*`
- 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.
- see if the bootloader loads the symbol table in memory as part of loading the kernel binary

Complete the implementation of `debuginfo_eip` by inserting the call to `stab_binsearch` to find the line number for an address.

Add a `backtrace` command to the kernel monitor, and extend your implementation of `mon_backtrace` to call `debuginfo_eip` and print a line for each stack frame of the form:

```
K> backtrace
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143: monitor+106
  ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000 00000000
         kern/init.c:49: i386_init+59
  ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
         kern/entry.S:70: <unknown>+0
K>
```

Each line gives the file name and line within that file of the stack frame's `eip`, followed by the name of the function and the offset of the `eip` from the first instruction of the function (e.g., `monitor+106` means the return `eip` is 106 bytes past the beginning of `monitor`).

Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

Tip: printf format strings provide an easy, albeit obscure, way to print non-null-terminated strings like those in STABS tables. `printf("%.*s", length, string)` prints at most `length` characters of `string`. Take a look at the printf man page to find out why this works.

You may find that some functions are missing from the backtrace. For example, you will probably see a call to `monitor()` but not to `runcmd()`. This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the `-O2` from `GNUMakefile`, the backtraces may make more sense (but your kernel will run more slowly).

就是要显示每个eip的函数名， 原文件名，行号。

inc中有Stab的结构体：

```c
// Entries in the STABS table are formatted as follows.
struct Stab {
  uint32_t n_strx;  // index into string table of name
  uint8_t n_type;         // type of symbol
  uint8_t n_other;        // misc info (usually empty)
  uint16_t n_desc;        // description field
  uintptr_t n_value;  // value of symbol
};
```

搜一搜资料，了解了Stab结构体：stab是ELF众多节之一，用于存放调试信息，每条信息都会以一个结构体的形式存放在Stab节中。n_type表示的是Stab的信息种类，有可能是一个函数、一个文件、一个行号的信息。一个Stab结构对应的字符串（比如文件的文件名、函数的函数名等）放在另外一个节中：stabstr节。stab结构体只存放一个下标n_strx, 可以通过这个下标在stabstr节中寻找字符串。n_value是当前表项对应符号值.，n_desc：符号描述,在调试的角度,我们只要知道它可以表示源文件中的行号。

按照objdump -G obj/kern/kernel命令执行，看到这些，这里摘要一点内容：

```
Symnum n_type n_othr n_desc n_value  n_strx String
11     SLINE  0      77     f0100034 0
94     FUN    0      0      f0100040 2723   test_backtrace:F
    (0,18)
66     SO     0      2      f0100040 2639   kern/init.c
```

- SLINE表示代码段的行号在那个函数的第77行
- 0xf0100040 是 test_backtrace:F(0,18)的函数地址
- 0xf0100040 是 kern/init.c文件地址

我现在要做的就是补全kdebug.c中的寻找行号。

那么 debuginfo_eip 是如何运转的呢?首先它使用 N_SO 格式查找全局的 stabs 列表, 确认 addr 所在的文件;接下来利用缩小的区间查找 N_FUN 格式,确定 addr 所在的函数。调用函数stab binsearch的两个参数int *region left 和int *region right需 要传入的是表项序号,不是内存地址,这里直接以stab end 减去stabs即可。这样可以防止二分查找不存在的情况。另外按照注释，我还得知道particular stabs type，在stab.h 文件中找到#define N_SLINE 0x44。于是补全代码如下。

```
// Search within [lline, rline] for the line number stab.
// If found, set info->eip_line to the right line number.
// If not found, return -1.
//
// Hint:
//   There's a particular stabs type used for line numbers.
//   Look at the STABS documentation and <inc/stab.h> to find
//   which one.
// Your code here.
  stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
```

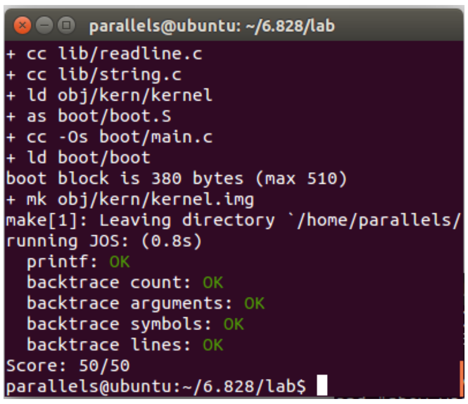
```
    if (lline <= rline) info->eip_line = lline - lfun;
    else return -1;
```

再完善输出就很简单了：

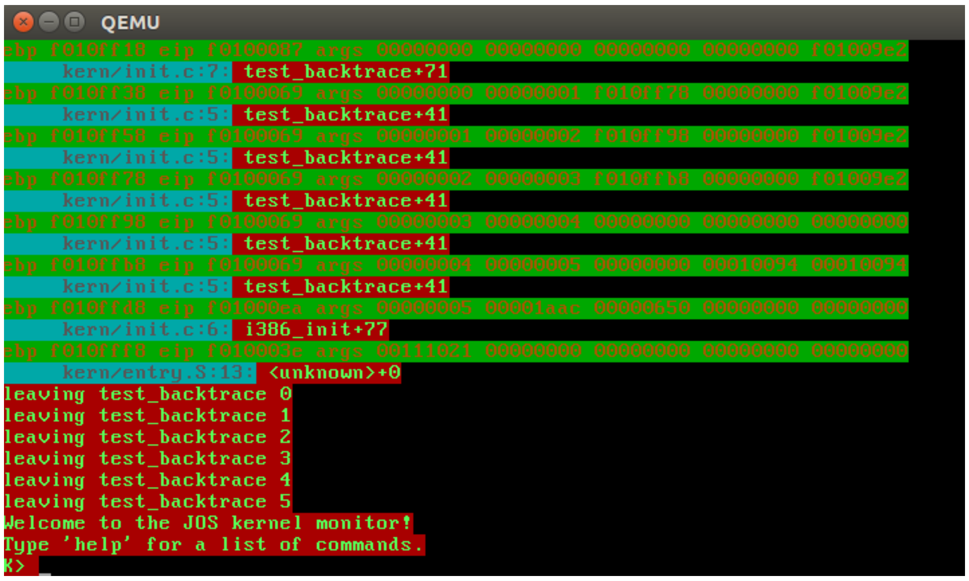
```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
  int *ebp, eip, *old_ebp;
  int ary[5]={};

  cprintf("Stack backtrace:\n");

  ebp=(int *)read_ebp();
  while((int)ebp!=0)
  {
    old_ebp=(int *)*(ebp);
    eip=*(ebp+1);
    for(int i=0;i<5;++i)
    {
      int j=i+2;
      ary[i]=*(ebp+j);
    }
    struct Eipdebuginfo eip_info;
    debuginfo_eip((uintptr_t)eip, &eip_info);
    cprintf("ebp %08x eip %08x args %08x %08x %08x %08x %08x\n"
        ,ebp,eip,ary[0],ary[1],ary[2],ary[3],ary[4]);
    cprintf(" %s:%d:", eip_info.eip_file, eip_info.eip_line);
    cprintf(" %.*s+%d\n", eip_info.eip_fn_namelen, eip_info.
        eip_fn_name, eip - eip_info.eip_fn_addr);
    ebp=old_ebp;
  }

  return 0;
}
```

%.*s是因为字符串是non-null-terminated strings，所以必须得这样输出。

- 放两张图,This completes the lab.



```
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 380 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory `/home/parallels/
running JOS: (0.8s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
parallels@ubuntu:~/6.828/lab$
```



```
ebp f010ff18 eip f0100087 args 00000000 00000000 00000000 00000000 f01009e2
      kern/init.c:7: test_backtrace+71
ebp f010ff38 eip f0100069 args 00000000 00000001 f010ff78 00000000 f01009e2
      kern/init.c:5: test_backtrace+41
ebp f010ff58 eip f0100069 args 00000001 00000002 f010ff98 00000000 f01009e2
      kern/init.c:5: test_backtrace+41
ebp f010ff78 eip f0100069 args 00000002 00000003 f010ffb8 00000000 f01009e2
      kern/init.c:5: test_backtrace+41
ebp f010ff98 eip f0100069 args 00000003 00000004 00000000 00000000 00000000
      kern/init.c:5: test_backtrace+41
ebp f010ffb8 eip f0100069 args 00000004 00000005 00000000 00010094 00010094
      kern/init.c:5: test_backtrace+41
ebp f010ffd8 eip f01000ea args 00000005 00001aac 00000650 00000000 00000000
      kern/init.c:6: i386_init+77
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
      kern/entry.S:13: <unknown>+0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

祝国庆快乐！