北京大学操作系统实习(实验班)报告

JOS-Lab 1: Booting a PC

黄睿哲 00948265
huangruizhe@pku.edu.cn

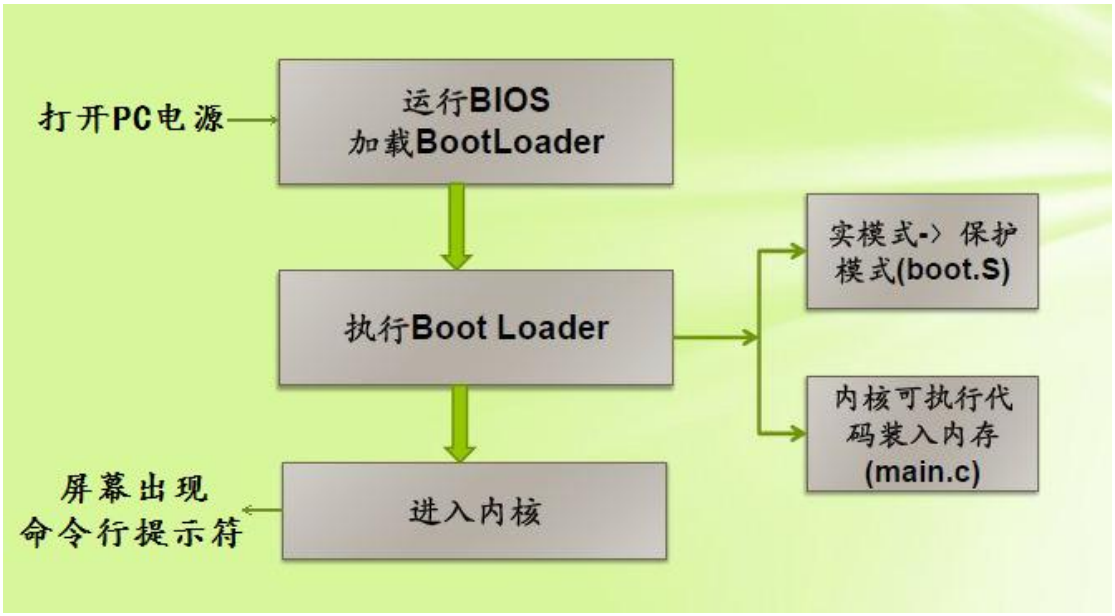
Feb 20, 2012

目录

内容一：总体概述

JOS-Lab1 实习的主要内容为 PC 上***系统的启动和初始化过程***：从用户打开电源直到屏幕出现命令行提示符的整个过程，如下图所示。实习包括三个部分：①熟悉 x86 汇编语言，QEMU x86 模拟器以及 PC 加电引导过程；②了解 JOS boot loader 及其功能；③了解内核可执行文件装入内存的过程。

看到一句挺有道理的话：“什么是系统启动？从 CPU 内核行为来看，启动就是从零地址取第一条指令并执行。”



I: PC Bootstrap

用户打开 PC 的电源时，内存中并没有任何其他的程序可以执行。于是，PC 令 CS=0xF000，IP=0xFFF0，物理地址为 0xFFFF0，恰为 ROM BIOS（本质是一段固件程序）在内存中存储的位置，是一条跳转指令，于是 BIOS 便开始在实模式下运行。BIOS 得到控制权后：①进行了一系列系统初始化工作，包括检测硬件信息（保存在主板芯片中）、初始化设备（如 CPU、内存、IO、VGA display 等）、初始化关键寄存器等；②BIOS 寻找和判断从什么设备（如软盘、硬盘、CD-ROM）启动操作系统；③BIOS 找到能够启动的磁盘后，从第一个扇区中将 Boot Loader 程序读入内存；④并将 CS:IP 置为相应的值，将控制权交给 Boot Loader 程序。

II: The Boot Loader

Boot Loader 程序的任务，是①将处理器从实模式切换到 32 位保护模式，扩大寻址空间（boot\boot.S）；②将内核的可执行代码从硬盘中读入内存，并将控制权交给内核，启动操作系统（boot\main.c）。Boot Loader 程序的可执行代码被保存在模拟硬盘的第一个扇区中，由于硬盘每个扇区大小为 512 字节，因此 Boot Loader 可执行代码的大小不能超过 510 个字节（剩下两个字节用于作特殊标记于扇区）。

当 BIOS 找到能够启动的磁盘后，BIOS 便将该磁盘第一个扇区 512 字节读入内存物理地址的 0x7c00 到 0x7dff，然后通过一条 jmp 指令将 CS:IP 置为 0000:7c00，将控制权交给 Boot Loader。

Boot Loader（boot\boot.S）会先进行一些初始化工作；然后，将 cr0 寄存器的最低位置 1，标志着系统进入保护模式；随后用一个跳转指令让系统真正开始使用 32 位的寻址模式；在进入保护模式后，程序重新初始化了相关寄存器，然后进入 bootmain 函数（boot\main.c）开始装入内核。内核的可执行文件（Kernel）为 ELF 格式，它被解压后装载入内存的 0x00100000 处；Kernel 第一条语句的虚拟地址位于 0xf010000c 处，经过地址转换的物理地址为 0x0010000c 处。

这部分中重要的概念有：保护模式和实模式，链接地址与装载地址，ELF 文件格式。这将会在报告中叙述。

III: The Kernel

Kernel 就是操作系统的核心部分，它的任务是管理进线程、存储器、文件、外设和其他系统资源。在 Lab1，JOS 的 Kernel 主要完成这些任务：①开启分页机制或加载新的 GDT 表（kern\entry.S）；②初始化堆栈指针（kern\entry.S）；③对内核中的一些变量（如保存全局变量的 BBS 节）进行初始化（kern\init.c）；④显存、键盘等控制台的初始化（kern\console.c）；⑤无限循环调用 monitor 函数，提示用户输入命令与操作系统进行交互（kern\init.c）。

进入 Kernel 后，我们知道 Kernel 在高物理地址 0x0010000c 处开始运行。事实上，Kernel 的代码一般被链接在更加高的虚拟地址，例如 0xf0100000，以便留下足够的低地址供用户程序使用。然而，物理内存一般没有这么大的空间，我们不能指望真的把 Kernel 加载到那个位置。因此，我们需要将虚拟地址 0xf0100000（Kernel 代码的链接地址，即 Kernel 代码事先认为它将要运行的地址，也是实际链接的地址）转换成物理地址 0x00100000（Kernel 代码的加载地址）。这样，Kernel 实际上运行在物理内存中 1MB 附近的位置。

在以前版本的 JOS 中，这是通过 GDT 来实现的；而现在这一版本，这是通过 kern\entry.S 中的 RELOC 宏来实现的一个简单分页机制。就是将虚拟地址[KERNBASE, KERNBASE+4MB)映射到物理地址[0, 4MB)，所要做的就是将 KERNBASE 偏移量减掉！

关于 cprintf 函数功能的实现，这里要了解系统将字符打印到屏幕的机制，以及相关输出函数之间的调用关系。

初始化堆栈指针，就是让 esp 指向内存中的一片堆栈空间的顶部。注意，堆栈是从高地址向下生长的。在此基础之上，我们可以实现一个回溯函数调用关系的功能，因此要了解堆栈的布局。

内容二：任务完成情况

I: 任务完成列表

Exercise	1~2	3~6	7~12	Challenge 1
第一周	Y	Y	Y	
第二周				Y

II: 具体 Exercise 完成情况

1. PC Bootstrap

1.1 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.

这一部分是 x86- AT&T 汇编语言的入门，区别于以前在微机原理课学过的 x86-Intel 汇编语言。两者只是格式不同，语义是几乎无异的：

	AT&T	Intel
操作数前缀	寄存器前”%”，立即数前”$”	无前缀
赋值方向	MOV (目的), (源)	movl (源), (目的)
寻址语法	%segreg: disp(base, index, scale)	segreg: [base + index*scale + disp]
操作码后缀	“l”长整数, “w”字, “b”字节	dword/word/byte ptr

此外我还学习了 AT&T 汇编中的一些汇编程序指令、GCC 内嵌汇编的方法。汇编程序指令都是以句点”.”开头，用于告诉 GNU 汇编程序一些必要信息，如分节、预留空间等，这些指令看懂即可。内嵌汇编的基本格式为：__asm__ __ volatile__(“statements”: output_regs :

input_regs : clobbered_regs);。

操作系统内核代码绝大部分使用 C 语言编写，只有一小部分使用汇编语言编写，例如与特定体系结构相关的代码和对性能影响很大的代码。要看懂 boot\boot.S, kern\entry.S, 反汇编得到的 obj/kern/kernel.asm 以及调试跟踪程序，都需要这部分的知识。

这部分材料参考 *Brennan's Guide to Inline Assembly* 和华中资料 Ch2.

1.2 Simulating the x86

这部分介绍了 QEMU 模拟器及其使用。QEMU 是一台真实 PC 的模拟。要使 QEMU 这台“机器”运行起来，需要提供一个 kernel.img 映像文件作为 QEMU 的磁盘文件，存放在模拟硬盘的特定位置。obj\kern\kernel.img 映像文件通过 make 得到，它是将 JOS 一系列源文件经过预编译、编译、汇编（得到*.o 文件）、链接（得到 obj\boot\boot 和 obj\kern\kernel 文件）、修整（得到 obj\kern\kernel.img）而来。

通过 make qemu 运行 QEMU 模拟器后，出现命令行，我们就可以和模拟的 PC 交互了。

1.3 The PC's Physical Address Space

这部分介绍了 PC 物理内存的区域和功能布局。

PC 物理内存空间的布局是由硬件规定的。从 0x0 到 0x9FFFF 的 640KB 被称为 Low Memory，是程序员（尤其是早期）可用的内存区域；从 0xA0000 到 0xEFFFF 的 320KB 用作 VGA 显示缓存、16 位外设扩展 ROMs 等；从 0xF0000 到 0xFFFFF(1MB)的 64KB 为 BIOS，是用于 PC 启动初始化的固化的程序。早期物理内存只有 1MB，后来内存空间被扩展到 16MB 乃至 4GB 时，但为了前后兼容，早期的内存布局被延续下来。高于 1MB 的空间被称为扩展内存，此外在 32 为地址空间的顶部被保留供 32 位 PCI 外设使用。

1.4 The ROM BIOS

这部分我学习了使用 GDB 对 QEMU 进行 Debug 的方法，以及对开机后 BIOS 初始化系统的过程有了一个初步的了解。

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.

BIOS 是用于 PC 启动初始化的固化的程序。用户打开 PC 的电源时，内存中并没有任何其他的程序可以执行。于是，PC 令 CS=0xF000，IP=0xFFF0，物理地址为 0xFFFF0 开始执行第一条指令。对 QEMU 进行单步跟踪，发现正如讲义所述，第一句指令位于 F000:FFF0 即物理地址的 0xFFFF0 处（BIOS 空间的顶部），是一条跳转指令，它跳转到 0xFE05B 处：

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

接下来 BIOS 工作只能看到汇编码，反正也看不懂，那就继续按照讲义所说的好了。于是 BIOS 就开始建立中断描述符表和进行一些设备初始化的工作（包括 VGA，PCI 总线等）。随后 BIOS 寻找能够启动的磁盘，从磁盘第一个扇区中将 Boot Loader 程序读入内存，并将 CS:IP 置为相应的值，将控制权交给 Boot Loader 程序。

2. The Boot Loader

Boot loader 完成两项任务，一是将处理器从实模式切换到 32 位保护模式，即将控制寄

存器 cr0 的 PE 位置 1，并且声明之后的为 32 位代码；二是将内核的可执行代码从硬盘中读入内存，并将控制权交给内核。

在 boot\main.c 文件的注释中看到如下叙述，对我们了解 boot loader 以及整个启动过程很有帮助：①Boot loader 包含两个文件 boot\boot.S 和 boot\main.c，它们编译链接后的可执行代码保存在磁盘第一扇区；②磁盘第二个扇区开始保存 ELF 格式的 Kernel 映像文件；③Boot loader 获得控制权后，在 boot\boot.S 中将处理器从实模式切换到 32 位保护模式，并为 boot\main.c 中 C 代码建立了临时堆栈，随后 boot\main.c 中 bootmain()函数读入 Kernel 并转交控制权。

关于代码阅读可以参看华中资料 Ch3，里面有较为详尽的解释。

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.

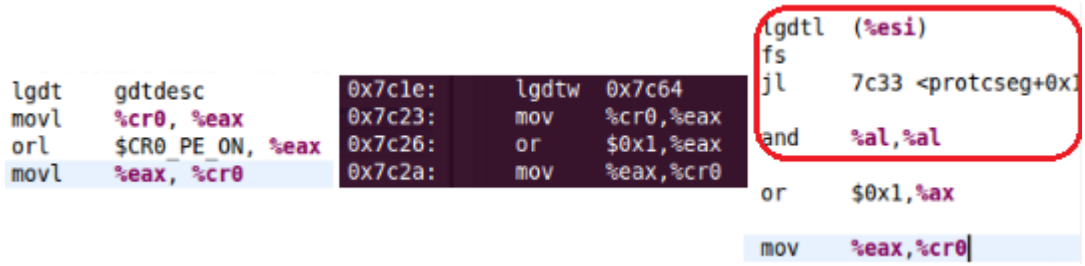
Trace into bootmain() in boot/main.c, and then into readsect(). Identify the exact assembly instructions that correspond to each of the statements in readsect(). Trace through the rest of readsect() and back out into bootmain(), and identify the begin and end of the for loop that reads the remaining 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.

一些基本的 GDB 调试命令：

• si N	表示单步调试 N 步；
• x/Nx	列出接下来的 N 条 16 进制代码；
• x/Ni 或 x/i	列出接下来的 N 条汇编代码；
• b address	在 address 处设置断点；
• c	连续执行到断点；
• info registers	显示寄存器信息；
• where	显示当前函数调用堆栈情况。

打开 GDB 调试，将断点设置为 0x7c00（Boot loader 被装在到内存中的起始位置）并运行到断点，便进入 boot\boot.S。我们对比 boot\boot.S 源文件、obj\boot\boot.asm 反汇编文件、GDB 单步输出的指令，发现源文件使用的是 AT&T 格式的汇编，而反汇编和 GDB 的结果采用 Intel 格式的汇编。我认为出现这个不同是反汇编器导致的，并无大碍。

其次，在 boot\boot.S 第 48~51 行发现，对于下面这条指令（左图）在 gdb 中语义一致（中图），而在 obj\boot\boot.asm 中却如右图所示，和前两者明显不同。这是出错了么？



事实上，这确实是反汇编程序弄错了。反汇编程序是按照 32 位模式反汇编的，而这里这条语句 lgdt gdtdesc 却包含了一个 16 位模式的地址！因此出现了反汇编结果与原义不同的现象。

Be able to answer the following questions:

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

在 boot\boot.S 代码 48~55 行，首先 CPU 载入 GDT 表，然后将控制寄存器 cr0 的 PE 位置为 1，随后调用 ljmp 指令对 CS:IP 寄存器进行相应设置，至此 CPU 才真正切换到 32 位保护模式。第 57 行.code32 声明了以下均为 32 为代码，因此 CPU 从第 60 行开始执行第一条 32 位代码。

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

Boot loader 执行的最后一条指令位于 boot\main.c 第 60 行：

```
// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry))();
```

其中 e_entry 是 Kernel 可执行程序的入口地址，也即 Kernel 第一条指令的地址。Kernel 执行的第一条指令是：movw $0x1234,0x472。

3. Where is the first instruction of the kernel?

通过 GDB 跟踪发现该指令位于物理内存的 0x10000c 处：

```
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
```

根据 objdump –x obj/kern/kernel 的结果也可以得知内核可执行代码的入口地址为 0x10000c。两者是一样的！

```
eraser@ubuntu:~/eclipseWorkSpace/lab1/obj/kern$ objdump -x kernel

kernel:     file format elf32-i386
kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Program Header:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
         filesz 0x00008036 memsz 0x00008036 flags r-x
    LOAD off    0x0000a000 vaddr 0xf0109000 paddr 0x00109000 align 2**12
         filesz 0x0000a300 memsz 0x0000a960 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rwx
```

需要说明的是，这和之前的 JOS 版本是不一样的。之前的 JOS 版本 Kernel 的链接地址为 0xf010000c，需要通过调用 e_entry 时和 0xffffffff 相与得到物理地址。这里免去了这个麻烦。

这条指令是 movw $0x1234,0x472，我们在 entry.S 文件的开始找到了这条语句。

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?

Boot loader 通过读取 Kernel 的 ELF 文件头知道该 ELF 文件被分成了多少个部分、每个部分的信息。在 boot\main.c 的 bootmain()函数中，这是通过读取 ph 结构体的 ph->p_offset 知道每一块的起始地址，和 ph->p_memsz 知道每块的长度。

```
// load each program segment (ignores ph flags)
ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
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);
```

2.1 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, though not as strongly recommended. A tutorial by Ted Jensen that cites K&R heavily is available in the course readings.

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.

这部分的目的是让我们深入理解 C 语言的指针及其运算。事实上我们可以这样理解：①指针指向内存地址，单位是字节；②指针的“步长”是由指针类型决定的，如 int*指针的步长是 4 字节，而 char*指针的步长是 1 字节，若某一结构体 struct m 的大小为 27 字节，则 struct m*指针的步长就是 27 字节；③对指针进行加减运算时，单位“1”要按照“步长”来计算；④数组和结构体存放在连续的内存中。这样我们就很容易理解 JOS 代码中涉及的指针的问题了。

如下是 pointer.c 的运行结果，稍微计算一下还是很好理解的。

```
eraser@ubuntu:~/桌面$ ./pointers
1: a = 0xbfa677e4, b = 0x9991008, c = 0xb94324
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0xbfa677e4, b = 0xbfa677e8, c = 0xbfa677e5
```

2.2 Real vs. Protected Mode, Link vs. Load Address, ELF File Format

实模式和保护模式。①实模式。在 8086 时代，CPU 中设置了四个段寄存器 CS、DS、SS、ES，每个段寄存器都是 16 位的，对应于 20 位地址总线的高 16 位。当访存指令进行寻址时，CPU 将段寄存器中地址左移四位再加上指令中的地址，就实现了从 16 位内存地址到 20 位实际地址的转换，即：实际物理地址 = (段寄存器 << 4) + 偏移地址。②保护模式。到了 80286 时代，地址总线位数增加到 24 位，并且访问内存时不能直接从段寄存器中获得段的起始地址了，而需要进行额外的转换和检查。一般保护模式段式寻址可用 xxxx：yyyyyyyy 表示，其中 xxxx 是段寄存器中的值，表示在 GDT 中对应描述符的索引，yyyyyyyy 用于和描述符中的该段的起始地址相加，得到最终的物理地址。总的来说，80286 为了和早期的 8086 兼容可以使用实模式或保护模式寻址。

链接地址和装载地址。①链接地址。链接地址是程序在链接时假定的执行起始地址，是程序自己假设在内存中存放的起始位置，即编译器在编译时候会认定程序将会连续地存放在从起始处的链接地址开始的内存空间。②装载地址。装载地址是程序在物理内存中实际存放的位置，它和链接地址可能不相同，这时就需要进行一些转换。

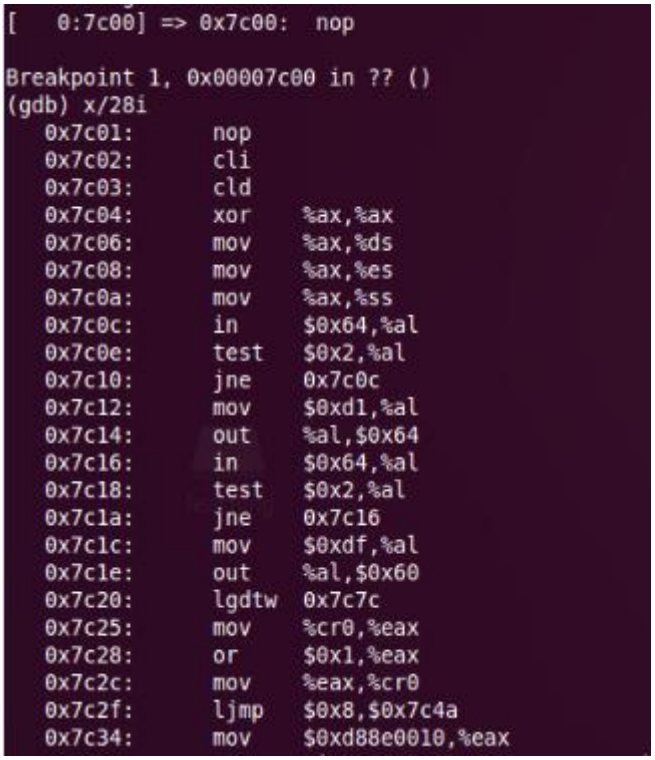
ELF 文件格式。当我们编译和链接一个 C 可执行程序时，编译器首先将每个“.c”源文件编译成“.o”目标文件，随后链接器将所有的目标文件“链接”起来，也就是将他们之间互相引用的地址进行修整，然后合并出一个单独的二进制映像文件。这个二进制的映像文件就是以 ELF 格式保存的。ELF 文件由 ELF 文件头和后续的程序段构成，我们根据 ELF 文件头中的信息就能够正确解析 ELF 文件中可执行代码的内容。借此我们就能了解 boot\main.c 是如何将 Kernel 映像读入内存的。

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

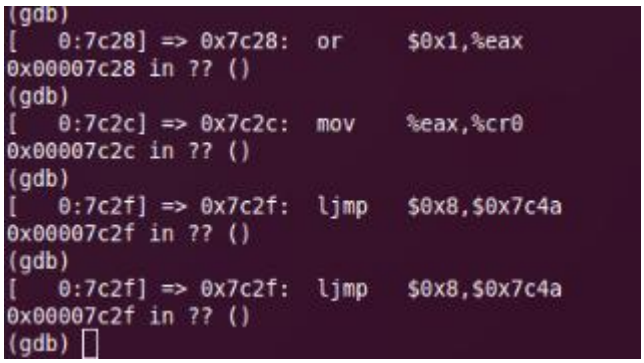
happens. Don't forget to change the link address back and make clean again afterward!

首先，我将 boot\Makefrag 中的-Ttext 0x7c00 修改为-Ttext 0x7c16，这就修改了 boot loader 代码的链接地址，而它的加载地址不变。现在，链接地址比加载地址要高 16 个字节，也就是出现了 16 字节的错位。

打开 GDB 调试，b *0x7c00 将断点设为 Boot Loader 第一条指令的地址并连续执行到那里，Boot Loader 还是能够装载入正确的位置，即改变链接地址的确不影响装载地址。这时，我们用 x/Ni 看看 boot loader 即将执行哪些指令：



显然，只有涉及地址跳转的语句才有可能在链接地址被修改时发生问题。我们发现 boot loader 将要执行到三条跳转指令，依次位于 0x7c10, 0x7c1a, 0x7c2f 处。最终，boot loader 在最后一条跳转指令出错了，执行不下去了！



```
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     00f57540 0000f883
IDT=     00000000 000003ff
CR0=00000011 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault.  Halting for inspection via QEMU monitor.
```

这是因为，前两句跳转指令都是进行设备忙等待的轮询语句，设备在检测时已经准备好了，因此不会条件跳转到其目标地址而是继续执行下一条指令。而到最后一条跳转语句是一定会跳转的，链接地址改变了其跳转的目标地址，因此导致出错了。

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.)

从 BIOS 进入 boot loader 是在地址 0x7c00 处，从 boot loader 即将进入 kernel 是在地址 0x7d63 处，我们在这两处设置断点。分别执行到两个断点并查看 0x100000 处的内存，情况如下：

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x7d63
Breakpoint 2 at 0x7d63
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d63:      call   *0x10018

Breakpoint 2, 0x00007d63 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x1000b812      0x220f0011      0xc0200fd8
(gdb)
```

内存 0x100000 是内核的最终载入地址，内核由 boot loader 负责载入内存。当从 BIOS 进入 boot loader 时，内核还未被加载，因此这时 0x10000 以及后续内存为全零；当 boot loader

即将进入内核运行时，这时显然内核已经被装载好了，因此 0x10000 开始就是内核可执行代码的内容了。

3. The Kernel

Kernel 主体代码的阅读顺序是：kern\entry.S，kern\init.c，kern\console.c，kern\printf.c，kern\monitor.c。这也是 Kernel 的执行顺序，从文件名我们甚至大致可以猜测 Kernel 干了什么事情。

3.1 Using virtual memory to work around position dependence

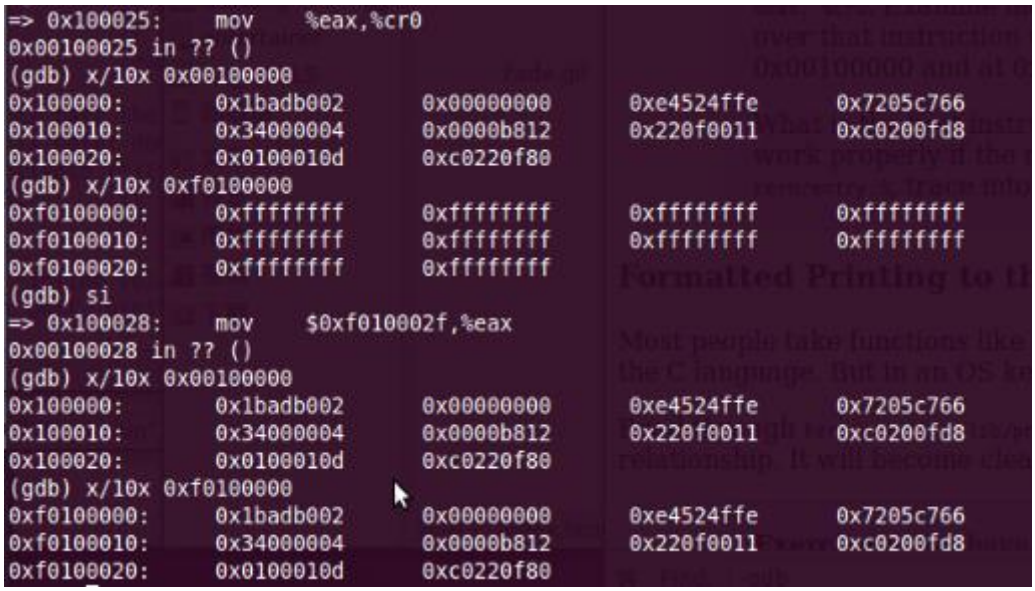
Boot loader 的链接地址和装载地址是一致的，但是对于 Kernel，它的链接地址和装载地址不一致，这就带来了地址转换的问题。

操作系统的 kernel 常常希望在内存空间的高地址运行，如 0xf0100000。然而，物理内存一般没有这么大的空间，我们不能指望真的把 Kernel 加载到那个位置。因此，我们需要将虚拟地址 0xf0100000（Kernel 代码的链接地址，即 Kernel 代码事先认为它将要运行的地址，也是实际链接的地址）转换成物理地址 0x00100000（Kernel 代码的加载地址）。这样，Kernel 实际上运行在物理内存中 1MB 附近的位置。

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.

用 GDB 跟踪到 kernel 中 movl %eax, %cr0 这条语句，执行前后内存的情况如下图所示。



我们发现在执行完 movl %eax, %cr0 语句后，内存中地址 0xf0100000 和 0x00100000 处的内容就变得完全一样了。这是因为通过执行这条语句，操作系统打开了一个简单的分页机制，目的是解决 kernel 链接地址与装载地址不一致的问题。这是 kern\entry.S 中对分页机制的叙述：

```
# We haven't set up virtual memory yet, so we're running from
# the physical address the boot loader loaded the kernel at: 1MB
# (plus a few bytes).    However, the C code is linked to run at
# KERNBASE+1MB.    Hence, we set up a trivial page directory that
# translates virtual addresses [KERNBASE, KERNBASE+4MB) to
# physical addresses [0, 4MB).    This 4MB region will be suffice
# until we set up our real page table in i386_vm_init in lab 2.
```

从以上叙述我们了解到，kernel 将[KERNBASE, KERNBASE+4MB)的内存空间映射到了物理内存的[0, 4MB)，这样在 kernel 中凡是引用到了[KERNBASE, KERNBASE+4MB)的地址都将被 RELOC 宏先减去一个 KERNBASE 的偏移量，使之能够引用到正确的位置。如下所示：

```
#####################################################################
# The kernel (this code) is linked at address ~(KERNBASE + 1 Meg),
# but the bootloader loads it at address ~1 Meg.
#
# RELOC(x) maps a symbol x from its link address to its actual
# location in physical memory (its load address).
#####################################################################

#define    RELOC(x) ((x) - KERNBASE)
```

如果分页机制没有打开，也就是注释掉 movl %eax, %cr0 这行代码，会发生什么情况呢？我们用 GDB 单步跟踪到分页机制开启前，如 b *0x00100020。继续执行，发现执行到注释掉的代码后面的跳转指令就出错了。这是很好理解的，因为它要跳转到一个高地址 0xf010002c，而现在分页没有开启，这样的高地址在物理内存中是不存在的。所以跳转指令出错了。

```
(gdb) b *0x00100020
Breakpoint 1 at 0x100020
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x100020:    or     $0x80010001,%eax

Breakpoint 1, 0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    $0xf010002c,%eax
0x00100025 in ?? ()
(gdb)
=> 0x10002a:    jmp    *%eax
0x0010002a in ?? ()
(gdb)
=> 0xf010002c:  (bad)
74              movl   $0x0,%ebp                       # nuke frame pointer
(gdb)
=> 0xf010002c:  (bad)
74              movl   $0x0,%ebp                       # nuke frame pointer
(gdb)
```

3.2 Formatted Printing to the Console

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.

在 JOS 中，所有底层的操作都需要操作系统的代码来实现，正如 cprintf()函数。cprintf()在 kern\printf.c 中定义，调用了 vcprintf()，它们主要做了一些变量和参数的传递，非常有意思的是在这里传递的参数是个数可变的，不过我不是很清楚为什么要把调用关系弄得那么复杂；然后会调用 lib\printfmt.c 中的 vprintfmt()函数，这个函数是一个非常强大的“状态机”，能够根据给定的格式化参数将变量转化成格式化的字符；任何输出的内容最后都会转化成字符，然后调用 kern\printf.c 中的 putch()函数一个一个字符地输出，值得注意的是，虽然是输出字符，但是 putch()的参数居然是 8 位的 int 类型，原来高位用于保存用于屏幕输出的其他控制信息；putch()调用 console.c 中的 cputchar()函数，cputchar()调用 cons_putc()，cons_putc()调用 serial_putc()、lpt_putc()和 cga_putc()，其中前两者用于初始化，cga_putc()完成主要的功能，包括设置字符默认格式、换行回车滚屏等。

主要的输出函数调用路径：

cprintf → vcprintf → vprintfmt → putch → cputchar → cons_putc → cga_putc

---

我们要实现 cprintf()函数输出八进制数字的功能，只需要在 vprintfmt()中将”%o”仿照”%d”实现即可。完成后屏幕输出：

Be able to answer the following questions:

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?

kern\console.c 向下面向硬件，向上提供了供其他函数调用的、和硬件交互的接口。其中，在 kern\printf.c 中，主要调用了 cputchar 函数。

2. Explain the following from console.c:

```
1       if (crt_pos >= CRT_SIZE) {
2           int i;
3           memcpy(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       }
```

这段代码的功能，是用于在屏幕输出时控制“屏幕向上滚动”。即当输出字符的行数超过屏幕的行数时（满屏），屏幕中的每一行字符将向上平移一行，以使最后一行可用。

crt_buf 指向物理内存的 0xa0000 到 0xc0000 的 128KB，即该空间是留给于显示屏相关联的 VGA 显示缓存的。显示屏规定为 25 行，每行宽度为 80 个字符。当屏幕输出的总字符数 crt_pos 超过 CRT_SIZE 时，屏幕要向上滚动一行。首先，memcpy 将第二行到末尾的输出内容复制到 crt_buf 开始处，即覆盖掉第一行；然后，for 循环将最后一行全部输出为空格，给我们一个“清空”的感觉；最后将光标移动到新一行的行首。

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.

在 cprintf 中，fmt 指向的是格式字符串，即"x %d, y %x, z %d\n"；ap 指向的是即将要输出的、不定参数表的第一个参数地址，在上例为变量 x 的地址。

在上例中的调用顺序和参数变化如下所示：

调用函数	相关参数
Vcprintf	fmt = "x %d, y %x, z %d\n\0" , ap = &x
cons_putc	ch = (register int)‘x’
cons_putc	ch = (register int)‘ ’
va_arg	(before) ap = &x, (after) ap = &y
cons_putc	ch = (register int)‘1’
cons_putc	ch = (register int) ‘,’
cons_putc	ch = (register int) ‘ ’
cons_putc	ch = (register int) ‘y’
cons_putc	ch = (register int) ‘ ’
va_arg	(before) ap = &y, (after) ap = &z
cons_putc	ch = (register int)‘3’
cons_putc	ch = (register int) ‘,’
cons_putc	ch = (register int) ‘ ’
cons_putc	ch = (register int) ‘z’
cons_putc	ch = (register int) ‘ ’
va_arg	(before) ap = &z, (after) ap = &z + 1
cons_putc	ch = (register int)‘4’
cons_putc	ch = (register int)‘\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?

输出结果如图所示：

57616 的十六进制表示是“E110”。将 unsigned int 型的 i 按照字符串来输出，由于 x86 是小尾表示，则 i 在内存中实际保存为“0x72, 0x6c, 0x64, 0x00”，转换成 char *的 ascii 码就是是“’r’, ‘l’, ‘d’, ‘\0’”。

如果改为大尾表示法，则应该令 i = 0x726c6400；而 57616 当作数字输出，与具体的存储方式没有关系，所以不需要更改。

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);

输出如下：

y 的输出结果是不可预料的。因为根据 vprintfmt 的机制，要打印的内容是存放在从可变参数指针指向的地址向后的连续空间。如果没有足够的参数供打印，那么 vprintfmt 将从连续空间向后继续取值，而这个位置的内容我们无法确定，因而打印的结果不可预料。

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?

如果改变参数入栈的顺序，那么可变参数首地址的指针 ap，将指向最后一个参数。我们可以让 cprintf 在传入参数时，增加一个参数表示可变参数的个数 cnt，这样通过 ap – cnt 就可以求出第一个参数的地址，cprintf 就可以按照原来的方法操作了。我们也可以这样：既然 ap 指向最后一个参数，那么 cprintf 就从最后一个参数按照 fmt 格式开始打印，每次 ap 自减，不过要像后序遍历一样，是要等前一个参数输出了后一个才能输出。这时候，如果参数个数和 fmt 规定个数不一致的话，第 5 题中出现不可预料结果的就是前面的参数而不是后面的。

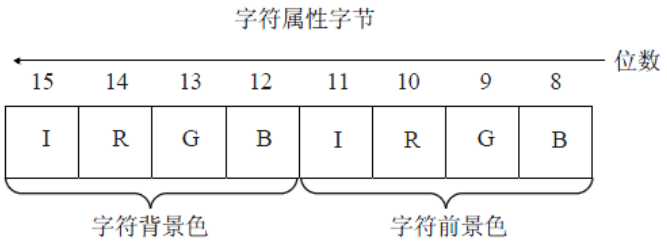
Challenge Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret ANSI escape sequences embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on the 6.828 reference 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.

在屏幕中输出的内容都是按照字符一个一个输出的，我们在 kern/console.c 的 cga_putc() 函数中看到控制颜色输出的机制露出了破绽：

```
static void
cga_putc(int c)
{
    // if no attribute given, then use black on white
    if (!(c & ~0xFF))
        c |= 0x0700;
```

这就是为什么在输出一个字符时，传入的参数是一个 int 型而不只是 char 型，原来高位保存了用于打印的格式信息。通过查阅资料，我们了解到 int 型参数 c 的 0 到 7 位是字符的 ASCII 码，而 8 到 15 位则是字符输出的格式，如下图所示：



因此，要让屏幕出处有颜色的字符，我们只要想办法将传给 cga_putc 的参数 c 的高 8 位进行相应的置位即可。这个参数从哪里开始传递呢？经过阅读我发现，从 vprintfmt 开始，对于字符”%c”的处理就是是当作整型来对待的：

```
// character
case 'c':
    putch(va_arg(ap, int), putdat);
    break;
```

因此，在调用 cprintf 函数输出”%c”时，可以给可变参数列表传入一个 int 型变量，它的第 8 位是字符的 ASCII 码，而高 8 位则是字符输出的格式：

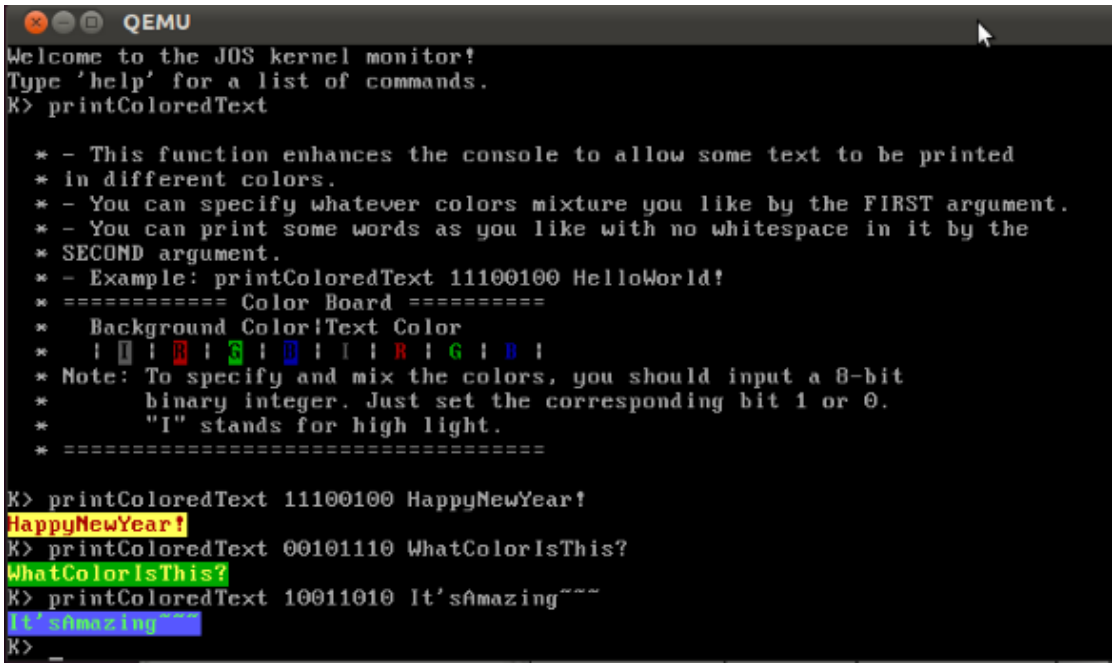
```
int ch = 'a';
ch |= 0x0x4200;
cprintf("%c", ch);
```

嗟夫！经尝试，余方知余之想法可行。

于是我在控制台加入一个输出彩色字符串的命令 printColoredText，实现方法仿照 kern/monitor.c 中的其他命令，见 mon_printColoredText()函数。为了和用户更好的交互，让用户能够指定打印的字符串和颜色，我通过控制台传入命令和参数 argv。argv 第一个参数是命令本身；第二个参数让用户输入一个 8 位二进制数，设置字符的前景和背景颜色；第三个参数让用户输出指定的字符串。mon_printColoredText()实现的伪代码如下：

```
1   Int
2   mon_printColoredText(int argc, char **argv, struct Trapframe *tf){
3           if(argc == 1){
4                   输出帮助信息;
5           }else{
6                   将 argv[1]字符串转换成整数，即 color;
7                   for(用 pchar 遍历 argv[2]的每个字符){
8                           int ch_32 = *pchar;         // get a char
9                           ch_32 |= textcolor;         // specify the color
10                          cprintf("%c", ch_32);
11                  }
12          }
13  }
```

例如，用户指定输出：加亮黄色背景、红色前景的字符串“HappyNewYear!”，命令行和打印结果截图如下：



3.3 The Stack

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?

初始化堆栈要在调用 C 函数之前，因此观察 kern/entry.S，发现内核初始化堆栈的代码段如下：

```
# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
movl    $0x0,%ebp           # nuke frame pointer

# Set the stack pointer
movl    $(bootstacktop),%esp
```

内核首先将 ebp 寄存器置为 0，作为堆栈回溯结束的标志；然后将 esp 寄存器置为一个内存地址 bootstacktop，我们看到 bootstacktop 是这样定义的：

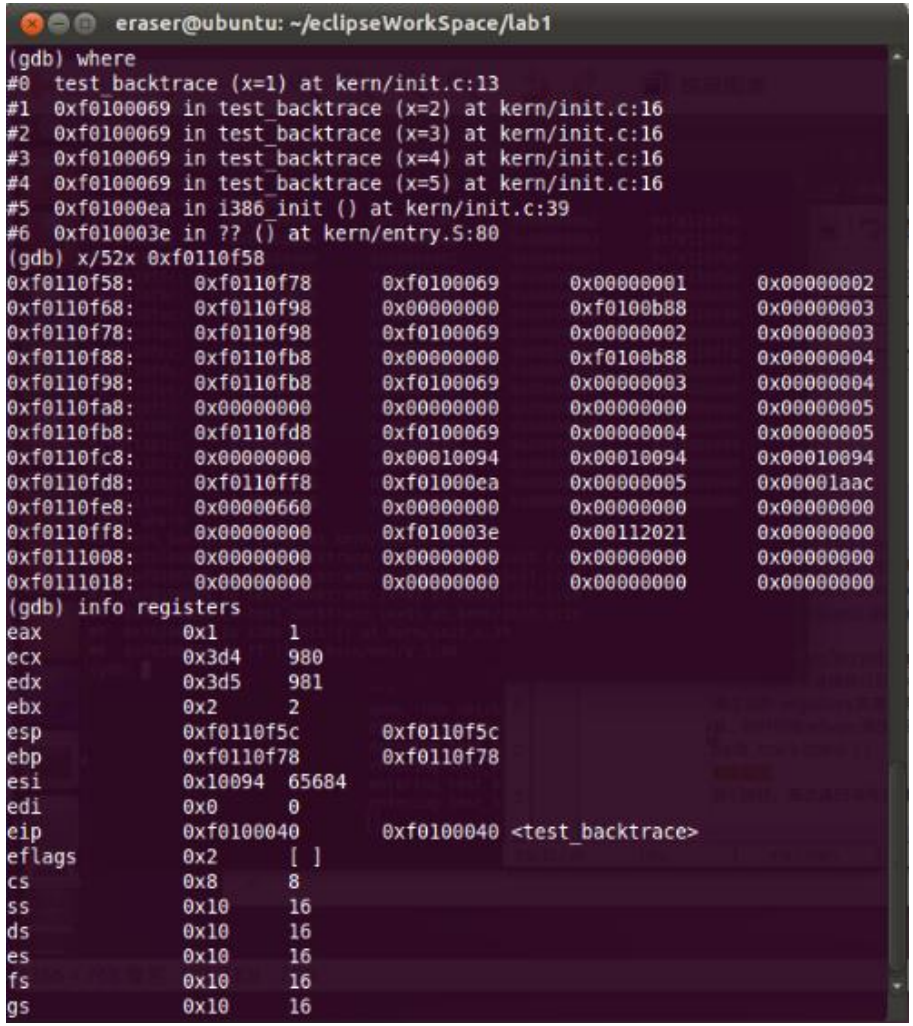
```
###################################################################
# boot stack
###################################################################
    .p2align    PGSHIFT     # force page alignment
    .globl      bootstack
bootstack:
    .space      KSTKSIZE
    .globl      bootstacktop
bootstacktop: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

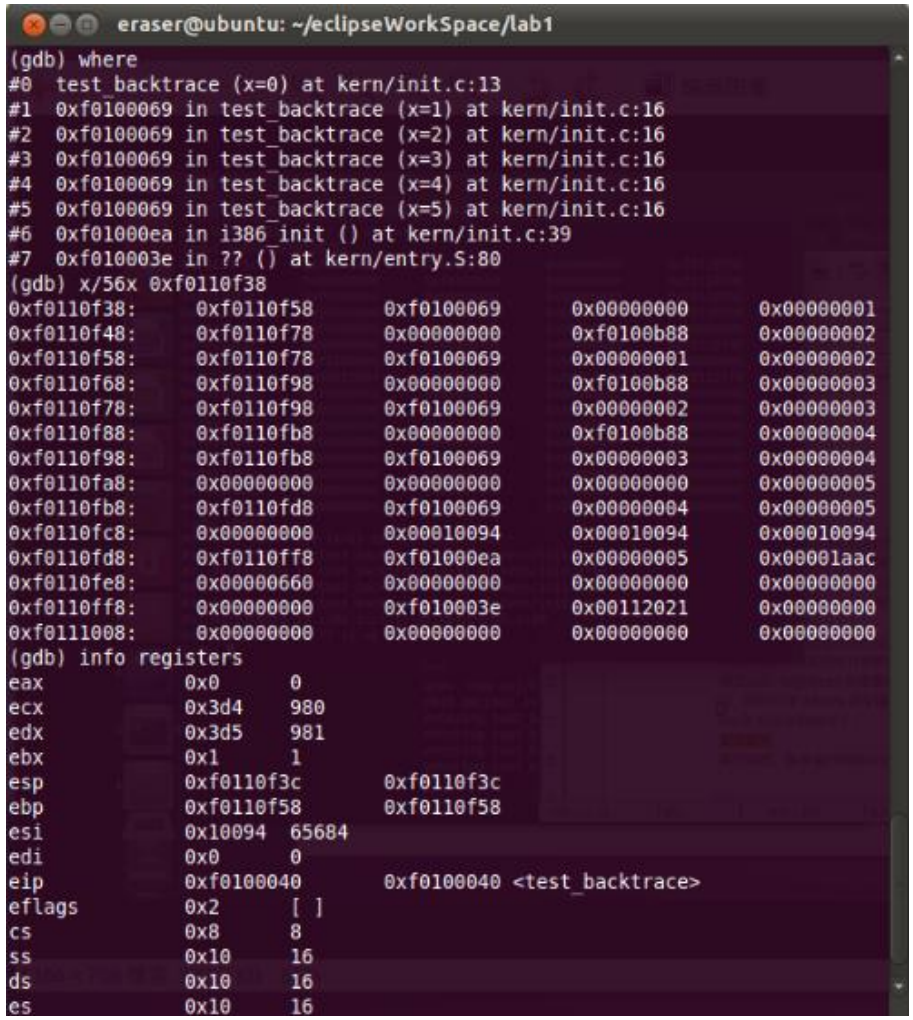
首先，先在内存预留 KSTKSIZE 大小的空间，然后将这片空间的顶部的高地址赋值给 bootstacktop，因此，我们便可以知道内核中堆栈是自高地址向低地址生长的，因为初始时栈底 esp 指向的是高地址 bootstacktop。在 obj/kern/kernel.asm 中观察到 bootstacktop = 0xf0111000。

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.

在 obj/kern/kernel.asm 中定位到 test_backtrace 的地址为 0xf0100040，打开 GDB 设置断点连续执行到断点。为观察每次调用 test_backtrace 时的堆栈情况，我们先通过 info registers 来查看寄存器 esp，ebp 的值，然后用 x/Nx 定位到内存中的 esp 指向的相应地址查看栈的内容，同时可用 where 命令辅助我们知道现在执行到哪里了（这是 GDB 提供的类似 back_trace 的命令）：


eraser@ubuntu: ~/eclipseWorkSpace/lab1
(gdb) where
#0 test_backtrace (x=1) at kern/init.c:13
#1 0xf0100069 in test_backtrace (x=2) at kern/init.c:16
#2 0xf0100069 in test_backtrace (x=3) at kern/init.c:16
#3 0xf0100069 in test_backtrace (x=4) at kern/init.c:16
#4 0xf0100069 in test_backtrace (x=5) at kern/init.c:16
#5 0xf01000ea in i386_init () at kern/init.c:39
#6 0xf010003e in ?? () at kern/entry.S:80
(gdb) x/52x 0xf0110f58
0xf0110f58: 0xf0110f78 0xf0100069 0x00000001 0x00000002
0xf0110f68: 0xf0110f98 0x00000000 0xf0100b88 0x00000003
0xf0110f78: 0xf0110f98 0xf0100069 0x00000002 0x00000003
0xf0110f88: 0xf0110fb8 0x00000000 0xf0100b88 0x00000004
0xf0110f98: 0xf0110fb8 0xf0100069 0x00000003 0x00000004
0xf0110fa8: 0x00000000 0x00000000 0x00000000 0x00000005
0xf0110fb8: 0xf0110fd8 0xf0100069 0x00000004 0x00000005
0xf0110fc8: 0x00000000 0x00010094 0x00010094 0x00010094
0xf0110fd8: 0xf0110ff8 0xf01000ea 0x00000005 0x00001aac
0xf0110fe8: 0x00000660 0x00000000 0x00000000 0x00000000
0xf0110ff8: 0x00000000 0xf010003e 0x00112021 0x00000000
0xf0111008: 0x00000000 0x00000000 0x00000000 0x00000000
0xf0111018: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) info registers
eax   0x1   1
ecx   0x3d4   980
edx   0x3d5   981
ebx   0x2   2
esp   0xf0110f5c   0xf0110f5c
ebp   0xf0110f78   0xf0110f78
esi   0x10094 65684
edi   0x0   0
eip   0xf0100040   0xf0100040 <test_backtrace>
eflags   0x2   [ ]
cs   0x8   8
ss   0x10   16
ds   0x10   16
es   0x10   16
fs   0x10   16
gs   0x10   16


eraser@ubuntu: ~/eclipseWorkSpace/lab1
(gdb) where
#0 test_backtrace (x=0) at kern/init.c:13
#1 0xf0100069 in test_backtrace (x=1) at kern/init.c:16
#2 0xf0100069 in test_backtrace (x=2) at kern/init.c:16
#3 0xf0100069 in test_backtrace (x=3) at kern/init.c:16
#4 0xf0100069 in test_backtrace (x=4) at kern/init.c:16
#5 0xf0100069 in test_backtrace (x=5) at kern/init.c:16
#6 0xf01000ea in i386_init () at kern/init.c:39
#7 0xf010003e in ?? () at kern/entry.S:80
(gdb) x/56x 0xf0110f38
0xf0110f38: 0xf0110f58 0xf0100069 0x00000000 0x00000001
0xf0110f48: 0xf0110f78 0x00000000 0xf0100b88 0x00000002
0xf0110f58: 0xf0110f78 0xf0100069 0x00000001 0x00000002
0xf0110f68: 0xf0110f98 0x00000000 0xf0100b88 0x00000003
0xf0110f78: 0xf0110f98 0xf0100069 0x00000002 0x00000003
0xf0110f88: 0xf0110fb8 0x00000000 0xf0100b88 0x00000004
0xf0110f98: 0xf0110fb8 0xf0100069 0x00000003 0x00000004
0xf0110fa8: 0x00000000 0x00000000 0x00000000 0x00000005
0xf0110fb8: 0xf0110fd8 0xf0100069 0x00000004 0x00000005
0xf0110fc8: 0x00000000 0x00010094 0x00010094 0x00010094
0xf0110fd8: 0xf0110ff8 0xf01000ea 0x00000005 0x00001aac
0xf0110fe8: 0x00000660 0x00000000 0x00000000 0x00000000
0xf0110ff8: 0x00000000 0xf010003e 0x00112021 0x00000000
0xf0111008: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) info registers
eax   0x0   0
ecx   0x3d4   980
edx   0x3d5   981
ebx   0x1   1
esp   0xf0110f3c   0xf0110f3c
ebp   0xf0110f58   0xf0110f58
esi   0x10094 65684
edi   0x0   0
eip   0xf0100040   0xf0100040 <test_backtrace>
eflags   0x2   [ ]
cs   0x8   8
ss   0x10   16
ds   0x10   16
es   0x10   16

在上面两幅图中，左图是 test_backtrace 参数为 1 时的情况，右图是 test_backtrace 参数为 0 时的情况，对比两幅图的 esp，ebp，栈的内容，我们发现：

- 栈是自顶向下生长；
- ebp 指向的地址的内容是上一层函数调用时的 ebp；
- (ebp 指向的地址+1)的内容，是函数返回地址；
- (ebp 指向的地址+2)的内容，是当前函数调用的传入参数；
- (ebp 指向的地址+3)的内容，不知道是什么……orz。

每次递归调用 test_backtrace，它向栈中压入 8 个 32 位字，依次为 ebp，eip(返回地址)，参数 1，参数 2……。

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.

需要实现的 mon_backtrace()函数在 kern/monitor.c 中，这个函数的功能是，能够根据寄存器 esp 和 ebp 的值在堆栈上追溯函数一层一层的调用，并显示函数调用的返回地址和参数等信息。在上一个 exercise 中，我们已经了解了栈的布局。因此，我们只要通过调用 read_ebp()一次读取当前的 ebp 指针，就可以根据 ebp 和栈的布局输出相应的信息，然后通过迭代 ebp 进行回溯。那么我们怎么判断 ebp 指向了栈底呢？我们注意到，在 kern/enrty.S 初始化堆栈时，曾经有一句代码令 ebp=0，因此只要发现 ebp=0 时我们就可以停止回溯了。

函数实现的伪代码和截图见 exercise 12。

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 the 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).

在上一个 exercise 中，我们实现了回溯功能，能够根据寄存器 esp 和 ebp 的值在堆栈上追溯函数一层一层的调用，并给出调用函数的返回地址。在这个 exercise 中，要求我们将这个“返回地址”对应到相应的“函数名”“源文件名”“在源文件中相应代码的行号”。根据编译时关于对应关系的记录，我们就可以完成这个功能，而 kern/kdebug.c 已经帮我们完成了一大部分了。因此只要参照已有的代码，调用 stab_binsearch 找到行号即可。

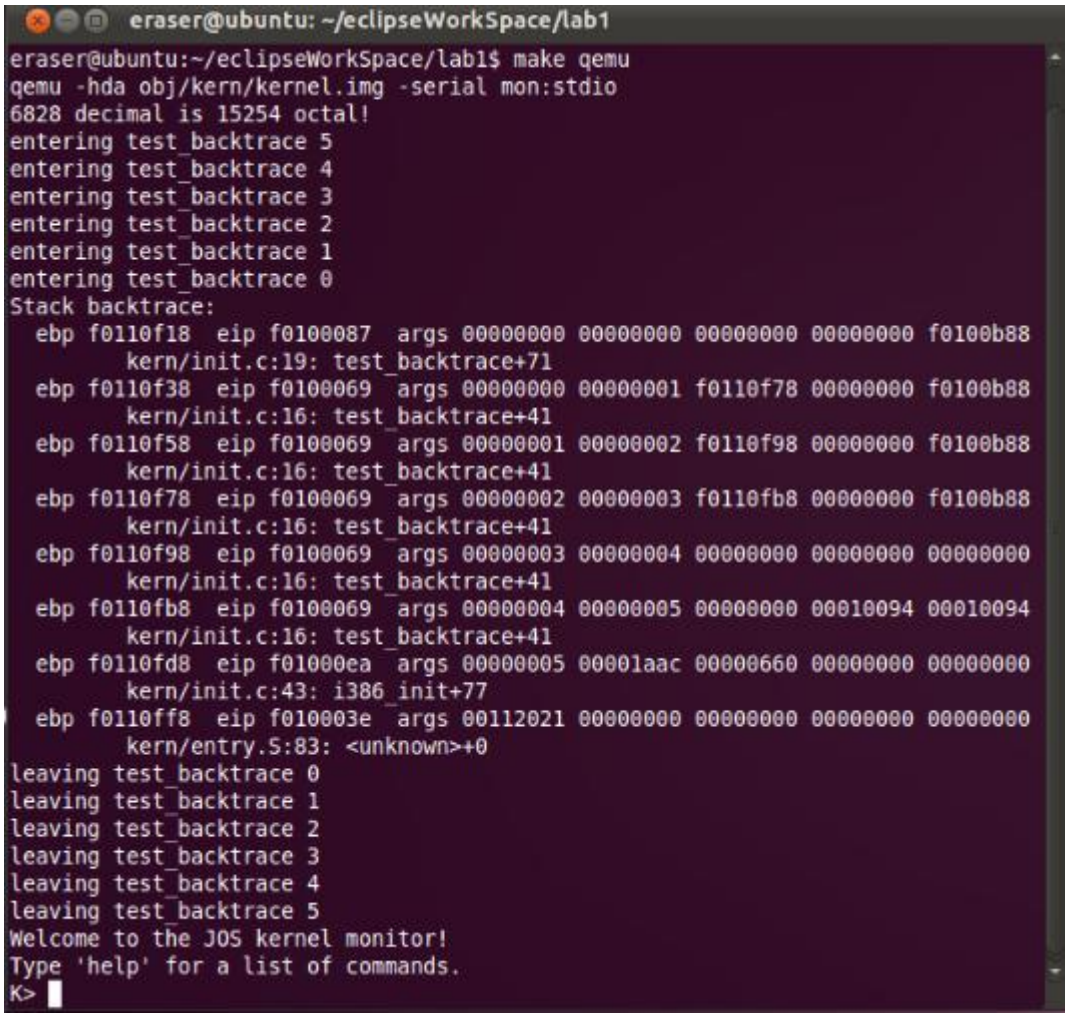
这里实现有一个细节，stab_binsearch 返回的结果 lline 不是行号，而是那一行在 stab 数组中的下标！因此，真正的行号是 stabs[lline].n_desc。

mon_backtrace 函数实现的伪代码如下：

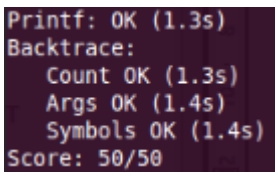
```
1   int mon_backtrace(int argc, char **argv, struct Trapframe *tf){
2       // 读取当前 ebp
3       uint32_t cur_ebp = read_ebp();
4       while(cur_ebp != 0){
5           // ebp 内容赋给指针，指针的内容是上一个 ebp 的值
6           cur_ebp_p = (uint32_t *)cur_ebp;
7           // eip 的值
8           cur_eip = *(cur_ebp_p + 1);
9           // 5 个参数
10          cprintf("%08x ", *(cur_ebp_p + 2));
```

```
11              cprintf("%08x ", *(cur_ebp_p + 3));
12              cprintf("%08x ", *(cur_ebp_p + 4));
13              cprintf("%08x ", *(cur_ebp_p + 5));
14              cprintf("%08x\n", *(cur_ebp_p + 6));
15              // 将 eip 传入获得函数名、文件名、行号
16              info_ret = debuginfo_eip(cur_eip, &info);
17
18              // 在 ebp 上进行迭代，直到 ebp=0
19              cur_ebp = *cur_ebp_p;
20          }
21      }
```

backtrace 功能的截图如下：



Lab1 到此结束，右边是 make grade 截图。



内容三：遇到的困难以及解决方法

困难一：Linux 环境新手上路

万事开头难。刚开始时对 Linux 环境下的命令和操作几乎是一片空白，就好像被丢进一个神秘的部落，这种感觉可以用一个“慌”字来形容。想想也只是换一个环境而已，不至于这样，二话不说要赶鸭子上树，硬着头皮做下去。路是走出来的，问同学，查资料。非常感谢刘驰、王仲禹、林舒同学，在一开始的时候耐心教我入门，至少让我知道“问题是什么”，不至于干着急。然后幸好有丰富的网络资源，那两天我阅读了很多关于 Linux 的百科、博客、资料，对什么 make、makefile、gcc、gdb、linux 启动过程、linux 常用命令等等这些很基本的东西有了一个初步的了解，在熟悉环境的同时也建立了信心。

困难二：粗心大意/对工具和命令不熟悉

刚开始时做事情有点心急，不是在 gdb 把地址打错了，就是忘记加*号，或者多加了*号，导致输出结果和预期不符合，反而更加心急。要解决这个问题，我觉得首先要耐心把资料看清楚，这样才不会犯那些低级错误，白白浪费时间。对于工具和命令不熟悉的问题，我觉得也不能强迫自己在一天之内就成为老手，所以只能在平时多积累，多总结。

困难三：想不清楚就画出来

在做到最后 Stack 的时候，要完成一个 backtrace 的功能。其实在上个学期做 Minijava->MIPS 编译器时，栈的布局是类似的，我觉得理解起来没什么困难。于是就开始写代码了。但是越写越糊涂，就好像陷到泥潭里了，怎么也写不对。我记得那天是中午写的，写到两点半越写越心烦。还好后来我决定把栈的布局给认认真真地画出来，把每个地址的内容、内容指向的内容都标出来，最后才把代码写对了。上个学期的操作系统原理课也是直到最后面把 UNIX、FAT、NTFS 文件系统布局给画出来，才有了深刻的理解。所以不能偷懒，想不清楚的东西要画出来。

在做 exercise12 的时候（也就是把每个 eip 对应到函数名、文件名、行号），我原以为做完了，但是刘驰提醒我 make grade 打满分了不一定对。于是我再重新阅读代码，发现

stab_binsearch 返回的结果 lline 不是行号，而是那一行在 stab 数组中的下标！因此，真正的行号是 stabs[lline].n_desc。

困难四：报告难写难于写代码，表达能力不足

可能是表达能力还不够吧，总之这次写报告写得很纠结。我花了做 exercise 三倍的时间来写报告。一方面原因是因为当我们下笔的时候，总会把每个要写的东西确认清楚，这就导致了不少节外生枝的事情；另一方面原因源于有限的表达能力，我不能像一些学长和同学熟练地玩转各种术语，也还不能够以最简练的语言把问题切中要害。所以，这个学期要努力提高自己的表达素养，值得期待。

内容四：收获和感想

首先就是阅读量：大！除了之前为了入门 linux 读的那些和具体实习无关的资料，和实习相关的资料也让人望而生畏。我觉得这很能锻炼我的阅读能力，如何快速地把握住文章的要义。总的来说我觉得我们课程提供的材料还是比较“友好”的，逻辑框架非常清晰，基本能够根据每段的第一句把握住思路，为快速阅读提供了帮助。另外，有时候在做完一个 exercise 之后再回头看前面的内容，发现原来作者表述是如此精确，那些名词和动词的比喻含义非常生动，不禁让我产生一种精神上的愉悦。

上个学期做编译实习，我曾经一行一行调试过那些冗长枯燥的中间代码；在实验室里写程序，我曾经认认真真阅读二进制文件找出错误。这些积累让我有信心把 JOS 代码读下去。我突然觉得以前“浪费”的时间是多么的值得。不管怎么样，耐耐心心地把事情做下去，总比在那里纠结要好。

通过这次 Lab，我终于知道了在我们打开电脑的时候从按下电源键的一瞬间到最后系统启动完毕，到底发生了什么事情。以前对于 BIOS、Booting、Kernel 这些东西几乎没有概念，通过这次 Lab，终于有了一次比较深入的接触。另外我发现大一大二学过的很多东西，例如堆栈布局、VGA 显示缓存等，在这次实习中都有所涉及，感觉就是一种知识的综合调用。尽管最后写报告的时候非常纠结，花了挺多时间，但是写报告有助于深入思考，之前忽略的一些东西在写报告时才注意到，并最终弄清楚。

总的来说，操作系统实习是以前学习内容、以前实践积累的一种延续，在学习新知识的同时，还能够体验到巩固和提升。

最后，我要感谢刘驰、王仲禹、林舒同学，他们给予了我很大的帮助。遇到很多不明白的地方，我向刘驰同学提出了很多问题，他都给我耐心地解释，让我收获很大。

内容五：对课程的意见和建议

暂无。

内容六：参考文献

[1] MIT 6.828 课程资料，MIT，2011；
[2] 邵志远，JOS 实验讲义第一章、第二章、第三章，华中科技大学，2010；
[3] 王仲禹，系统的启动和初始化实习报告，北京大学，2011；
[4] Debian Booting Process：http://wiki.debian.org/BootProcess。

备忘：
1\Ex3 后面问题第 3 题?地址为什么是一样的呢？
2\Cprintf 为什么要把调用关系弄得那么复杂
3\在 ex10 那幅图中看出 Esp 指向哪里？