Lab 1 实习报告: Booting a PC

赵睿哲
vincentzhaorz@pku.edu.cn

Abstract

本次 Lab 主要解决的问题是如何在硬件平台上启动一个操作系统，涉及到的概念很多，具体有硬件平台的知识，比如：80x86 处理器，内存和硬盘的结构等，还涉及到软件层面的细节，比如：C 和汇编语言的细节。本报告首先总结一下这次 Lab 中涉及到的知识点，之后分别对每个 exercise 和 challenge 进行详细的解答和分析，最后是本次实习的感悟和收获。

1 Introduction

Lab1 分为三个部分：bootstrap、bootloader 以及 kernel，分别对应于启动操作系统的三个关键步骤：首先，处理器从一个固定的内存地址开始，执行一系列指令以配置底层硬件；之后，处理器从硬盘中导入可执行文件，即操作系统，并跳转到程序的入口；最后，操作系统开始执行。下面分别对三个部分涉及到的知识进行总结和梳理：

1.1 bootstrap

如何在一个“空白”的硬件环境中启动一个软件？这是 bootstrap 需要解决的核心问题。为了理解这一点，首先需要知道 intel 8088（即 QEMU 模拟的环境下）的内存模型，并且知道在内存中有一块区域叫做 ROM，其中保存了 BIOS 程序。系统启动的时候首先执行的就是 BIOS 中的第一行代码。

BIOS 开始执行以后，就需要知道 BIOS 自己的“执行环境”是什么样的。BIOS 首先是运行在 16-bit 实模式的环境中，之后通过修改 CR0 寄存器进

入保护模式。实模式和保护模式除了在内存的分配机制上安全性不同以外，寻址模式也有很大的区别：实模式通过两个 16 进制寄存器计算得到地址，而保护模式则是通过全局描述符表来得到物理地址。前者支持的内存空间不超过 1MB，后者通过虚拟内存机制可以达到 4GB。

单纯运行 BIOS 并不是我们的目的，我们需要让处理器执行我们想要运行的软件中的代码。不像 BIOS 是存储在 ROM 中，软件一定是在磁盘上的，那么 BIOS 的下一步就是把磁盘上的程序导入到内存中。BIOS 每一次都是从磁盘的第一个扇区导入程序，因此我们可以把一个很小的程序放在第一个扇区中，让 BIOS 交出处理器的控制权，该程序就是 bootloader。因为 BIOS 导入程序以后，每次都跳转到 0x7C00 的地址开始执行程序，因此 bootloader 的第一行代码就要被导入到 0x7C00 的位置。

1.2 bootloader

尽管我们现在已经在执行自己想要运行的程序了，但是 bootloader 自身大小的限制肯定不能支持我们的需求，因此 bootloader 需要做的事就只有一件：导入体积更大的程序，即操作系统文件。

bootloader 通过与磁盘的 IO 操作把 kernel 导入进来以后，接下来的一件事就是跳转到 kernel 的入口函数。kernel 文件在这里是二进制格式的 ELF 文件，因此可以通过解析 ELF 的 header 得到 kernel 的入口。

1.3 kernel

一旦 kernel 被导入进来，我们就有更多的事可以做了。Lab1 提供的 kernel，与其说是一个操作系统，倒不如说是一个直接跑在硬件上的 shell，因此我们可以做的修改很有限：一方面是改进 VGA 上的输出效果，另一方面则是分析运行中的栈模型。后者在之后应该会起到更大的作用。

2 Exercises

2.1 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.*

该练习只是用于熟悉汇编语法，没有实际的内容

2.2 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.*

再次回忆一下 BIOS 的作用：BIOS 作为保存在 ROM 中的特定位置的程序，在 PC 启动时被首先执行；在初始化一系列硬件环境之后，把 bootloader 从硬盘中导入内存并转移控制权。因此，Exercise 2 中涉及到的代码即为“初始化硬件平台”相关的代码。

单就在 QEMU 中执行的代码而言，代码执行序列从指令 ljmp 开始，跳转到 BIOS ROM 中的低位指令并执行，一直到导入 bootloader 代码并跳转到其入口 (0x7c00) 为止。

Trace 到的代码如下：

```
1 0xfe05b:          cmpl   $0x0,%cs:0x65b4
2 0xfe062:          jne    0xfd3aa
```

```
0xfe066:        xor    %ax,%ax
0xfe068:        mov    %ax,%ss
0xfe06a:        mov    $0x7000,%esp
0xfe070:        mov    $0xf431f,%edx
0xfe076:        jmp    0xfd233
0xfd233:        mov    %eax,%ecx
0xfd236:        cli
0xfd237:        cld
0xfd238:        mov    $0x8f,%eax
0xfd23e:        out    %al,$0x70
0xfd240:        in     $0x71,%al
0xfd242:        in     $0x92,%al
0xfd244:        or     $0x2,%al
0xfd246:        out    %al,$0x92
0xfd248:        lidtw  %cs:0x68f8
0xfd24e:        lgdtw  %cs:0x68b4
0xfd254:        mov    %cr0,%eax
0xfd257:        or     $0x1,%eax
0xfd25b:        mov    %eax,%cr0
0xfd25e:        ljmpl  $0x8,$0xfd266
```

首先，这段代码运行在 16-bit 实模式下。开始的几条指令只是简单的初始化一些寄存器的值，比如把栈指针%esp 的值赋成 0x7000，以及禁用中断等操作。之后最关键的是：

1. I/O 操作: 其中 0x70 与 0x71 是与 CMOS 交互，0x92 是 System Control Port；
2. 初始化 GDT: 该指令是即将进入保护模式的前奏，因为保护模式的寻址方式与实模式完全不同，保护模式需要 GDT；
3. 转换为保护模式: 直接把%cr0 设置为 0x1 即可

有趣的一点是最后一条指令，为什么这里需要 long jump？主要原因是，保护模式下无法按照原来的寻址模式找到应该要执行的段，必须查 GDT 表。

进入保护模式后的代码更加复杂，在这里不加赘述。

2.3 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/icommand 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.

2.3.1 Part 1

执行调用 bootmain() 之前的代码。0x7c00 是 bootloader 在内存中执行的起始地址，这时要执行的执行是在 16-bit 实模式下的 cli 指令：

```
The target architecture is assumed to be i8086
[   0:7c00] => 0x7c00:  cli
```

之后，在 0x7c23 的指令之前，所执行的操作包括：

1. 初始化段寄存器：CS，DS，SS
2. 启用 A20 线，这里采用的方法是用 keyboard controller（加脚注）
3. 初始化全局描述符表 (GDT)，为转入保护模式做准备

```
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
ljmp    $PROT_MODE_CSEG, $protcseg
```

当 CR0 被设置为 0x1 时，保护模式开启。注意这里的 ljmp 依然跟以前一样，为了跳转需要得到存储在全局描述符表中的段地址。

2.3.2 Part 2

对 bootmain() 的调用始于指令：

```
0x7c45: call   0x7d0b
```

经过调用 gdb 的 x/i 指令，以及分析 boot.asm 代码，发现 0x7d0b 是 bootmain 的起始地址，因此从值了开始调用 bootmain()

bootmain() 中主要进行的操作主要是把内核 ELF 文件 load 到指定的位置。为了完成这样一个功能，除了 ELF 格式提供的种种接口以外（包含在 inc/elf.h) 中，还需要 readseg() 函数来读取硬盘中的 sector 中的内容。readseg 函数的定义：

```
void readseg(uint32_t pa, uint32_t count, uint32_t offset) {
        uint32_t end_pa;
        end_pa = pa + count;
        pa &= ~(SECTSIZE - 1);
        offset = (offset / SECTSIZE) + 1;
        while (pa < end_pa) {
                readsect((uint8_t*) pa, offset);
                pa += SECTSIZE;
                offset++;
        }
}
```

该函数的了哟及并不复杂，给定要读取的数据大小 count 和起始位置 offset 以后，接下来需要的就是不断读取 disk 的一个 sector 的内容（因为 sector 是磁盘传输数据的最小单元)，调用的函数是 readsect()，定义如下：

```
1  void readsect(void *dst, uint32_t offset) {
2          waitdisk();
3
4          outb(0x1F2, 1);         // count = 1
5          outb(0x1F3, offset);
6          outb(0x1F4, offset >> 8);
7          outb(0x1F5, offset >> 16);
8          outb(0x1F6, (offset >> 24) | 0xE0);
9          outb(0x1F7, 0x20);      // cmd 0x20 - read sectors
10
11         waitdisk();
12
13         insl(0x1F0, dst, SECTSIZE/4);
14 }
```

该函数有很明显的“汇编特征”，下面是对其汇编代码的分析：

首先，waitdisk() 函数被调用，目的是让程序等待磁盘可用，那如何得到该信息呢？通过观察 waitdisk 的 asm 代码，发现其首先调用 inb 从端口 0x1f7 读取一个 byte 的磁盘数据。该数据之后与 0xC0 求 and，得到的是 BSY 和 RDY 位的信息，如果 BSY 位为真，则磁盘正处于忙状态。

当 waitdisk() 运行结束之后，此时 disk 可用，因此需要告诉磁盘要读的数据地址。关键是所有 outb 语句的最后一句 outb(0x1F7, 0x20)，其中 0x1F7 是磁盘 I/O 的端口，0x20 是一条命令，用于读取 sector(引用)

再次等待 disk 可用时，即可把 disk 对应位置的数据读入了，使用的指令是 insl。这里要注意，被读入的地址在 dst 中。我在这里 trace 到了 insl 函数搜集到的数据，insl() 内部包含一个循环，每次循环使用 insl 指令来读一个双字：

```
1  repnz insl (\%dx),\%es:(\%edi)
```

DX 是端口号，ES 是基址，EDI 是偏移量寄存器。根据对循环的 trace，发现 ES 在每次循环都不变，EDI 增加 4，即增加 4 个 byte 的偏移。即，读入了一个 double word string。

bootloader 首先读取的是 ELF 文件头，之后根据 ELF 文件的信息，把剩余的 kernel 代码导入。读入的目标内存地址存放在 ELF 的 pa 成员中。一旦 ELF 的所有数据都进入内存以后，执行 kernel 的入口函数：

```
((void (*)(void)) (ELFHDR->e_entry))();
```

调用的是函数指针指向的函数。

2.3.3 Part 3

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

.code32 之后的指令均为 32 位模式下的代码，但是之前当 CR0 被置 0x1 时已经进入 32-bit 的保护模式了

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

在调用内核入口函数的位置设置断点 b *0x7d63 后，得到 bootloader 的最后一条指令：

```
0x7d63: call   *0x10018
```

很明显，0x10018 是 kernel 入口函数的地址，之后跳转到这个函数。直接 si 就可以得到内核的第一条语句：

```
0x10000c:       movw   $0x1234,0x472
```

Question 3 *Where is the first instruction of the kernel?*

位置在 0x10000c，不仅通过 gdb 可以 trace 到，在 ELF Header 中的 start address 也有提示。

Question 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?*

根据 boot loader 的程序逻辑：

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

```
4 eph = ph + ELFHDR->e_phnum;
5 for (; ph < eph; ph++)
6         readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

由此可见，每次是对一个 code segment 调用 readseg 函数。ELF 文件中清楚地指明了每个 code segment 的大小 (p_memsz) 和起始位置 (p_offset)。

2.4 Exercise 4

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.

输出结果如下：

```
1 1: a = 0x7fff576bf100, b = 0x7ffdd1c04bc0, c = 0x108540000
```

```
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 = 0x7fff576bf100,b = 0x7fff576bf104,c = 0x7fff576bf101
```

第 1 行和第 6 行是简单的 C printf %p 标识符的用法：输出指针地址。2,3,4 行很简单，主要考察的是指针指向同一个位置时，修改任意指针指向的值，都会影响其他指针指向的值（囧）。

第 5 行实际上是把 c 先类型转换为 char 型的指针，char 型指针加 1 的结果，是指向的地址值加 1(1byte)。之后再转换为 int* 类型后，c 指向的位置是原始地址 +1 后的位置，对 c 的值的改变，会影响到 a[1] 的后 3 个 byte 和 a[2] 的第一个 byte。

具体的计算：首先 400 的二进制形式为 0b110010000，500 的二进制形式为 111110100，根据 big-endian 的规则，a[1] 会变为 0b11111010010010000，即低 8 位来自 400，之后 24 位来自 500 的低 24 位。

2.5 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/Makefragto 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 cleanagain after*

如果链接地址不是在装载地址，那么很可能有些依赖于相对地址的跳转语句会跳转到奇怪的位置而出错。

把 Makefrag 中的 -Ttext 的值改为 0x7C01，之后从 0x7C00 开始执行，可以发现：

1. 初始的几条指令均为 nop；
2. continue 之后在`0x7c30:ljmp $0x8,$0x7c36`上出现错误了；

2.6 Exercise 6

We can examine memory using GDB's xcommand. The GDB manual has full details, but for now, it is enough to know that the command x/Nx ADDRprints Nwords 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.)

在运行 QEMU 测试之前，首先想到运行到 bootloader 时和运行到 kernel 时的最大区别是：bootloader 是加载到 0x7C00 的内存地址中，而 kernel 是加载到 0x10000C 的位置。因此前者不会影响 0x100000 之后的内存，而 kernel 会。

测试结果也是如此：

bootloader address:

```
0x100000:    0x00000000    0x00000000    0x00000000    0x00000000
```

kernel address:

```
0x100000:    0x1badb002    0x00000000    0xe4524ffe    0x7205c766
0x100010:    0x34000004    0x7000b812    0x220f0011    0xc0200fd8
0x100020:    0x0100010d    0xc0220f80    0x10002fb8    0xbde0fff0
0x100030:    0x00000000    0x117000bc    0x005fe8f0    0xfeeb0000
0x100040:    0x53e58955    0x8b14ec83    0x5c89085d    0x04c70424
```

2.7 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 stepiGDB 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.

该指令的作用是启用 32-bit 保护模式，在此之前，`0xf0100000`直接访问物理地址，而这已经超出了内存空间的范围，因此结果都是 0；启用以后有了 virtual memory 机制，因此会把地址值`0xf0100000`直接 map 到`0x100000`的物理地址，所以读取改地址得到的内容和读取`0x100000`的内容一样。

跟之前的 Exercise 5 很类似，这里注释掉以后，第一个利用了 0xf01XXXXXX 的地址的指令就会出错，这里是 jmp(`0x10002a`) 指令，因为其参数是`0xf010002c`

2.8 Exercise 7

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

直接按照 10 进制的写法修改即可。

即先用`getuint`获取数值，然后设置 base 值为 8，最后跳转到 number 标签进行处理：

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

输出：`6828 decimal is 15254 octal!`

对后续问题的回答：

Question 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?*

printf.c 使用了 console.c 提供的 cputchar 的实现。

根据注释，该函数的作用是提供一个 high-level 的对输出一个字符的功能的实现。

Question 2 *Explain the following from console.c*

```
1  // What is the purpose of this?
2  if (crt_pos >= CRT_SIZE) {
3      int i;
4
5      memmove(crt_buf, crt_buf + CRT_COLS,
6          (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
7      for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
8          crt_buf[i] = 0x0700 | '␣';
9      crt_pos -= CRT_COLS;
10 }
```

该段代码是在一个往 VGA buffer 中填充一个字符的函数中。crt_buf 是缓冲区，CRT_COLS 是屏幕宽度，CRT_SIZE 是屏幕面积。因此，假如要输出的字符的位置已经超出当前屏幕大小限制时（即条件判断），要删掉上面一行并读取下面一行（即 memmove），然后把屏幕上剩下的位置初始化（即最后的 for 循环）。

Question 3 *Trace the execution of the following code step-by-step*

```
1  int x = 1, y = 3, z = 4;
2  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 是格式化字符串，ap 是参数列表。

   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.

cons_putc 的参数是参数列表的地址（即 ap)
va_arg 用于根据类型移动位置
vcprintf 的参数即为格式化字符串，以及参数列表（列表的地址可与 cons_putc 的参数比较）

Question 3 *Run the following code.*
unsigned int i = 0x00646c72; cprintf("H%x Wo%s", 57616, &i);

结果是：He110 World

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set ito in order to yield the same output? Would you need to change 57616 to a different value?

57616 的 16 进制表示即为 e110，i 的地址指针本来的类型为 int *，这里因为%s 而转换为 char *，造成的结果是分别对 i 的每个字节进行 ASCII 码的转换。和之前的问题一样，这里大端法小端法的结果一定不同。

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

解释到 y 的时候已经超出参数列表的范围了，因此指向未分配的内存区域了。

Question 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 cprintfor its interface so that it would still be possible to pass it a variable number of arguments?*

该问题肯定跟参数列表，或者说 va_arg 的实现有关。通过 grep 找到在 incstdarg.h 中有对应的实现：

```
1 #define va_arg(ap, type) __builtin_va_arg(ap, type)
```

如果编译器改变了参数的顺序的话，在这里直接把函数的内容改成反方向的即可；比如原先是按照地址增长的顺序获得参数，现在可以按照减少的顺序获得

2.9 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.

根据之前的题目分析可以发现，所有的VGA输出最后都要经过`cga_putc`这个入口函数。该函数的参数是要输出的字符，加上格式化信息，即颜色。根据这一段代码：

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

switch (c & 0xff) {
    case '\b':
    if (crt_pos > 0) {
        crt_pos--;
        crt_buf[crt_pos] = (c & ~0xff) | '␣';
    }
    break;
    ...
```

很明显，因为后面的**`switch`**取得仅有低 8 位，因此低 8 位是 ASCII 字符信息。那么颜色判断应该保存在前 8 位中。原代码中对颜色的处理很简单，只要当前位置没有字符，那么就输出黑色，否则不加颜色。

现在设置一个全局变量:user_vga_color，该值初始为 0，用户可以用一些颜色宏来修改该值。定义如下颜色宏:

```
#define VGA_COLOR_BLACK 1
#define VGA_COLOR_RED 4
#define VGA_COLOR_GREEN 2
```

并且修改cga_putc函数:

同时在cga_init中要初始化user_vga_color为 0

```
c |= (user_vga_color << 8);

if (!(c & ~0xFF))
    c |= 0x0700;
```

想要输出颜色的时候可以直接:

```
user_vga_color = VGA_COLOR_RED;
cprintf("RED");
```

输出结果如下图所示，在图片的最下方是输出的一条红字符串和一条绿字符串:

2.10 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?*

初始化栈一般是跟 ESP(栈顶指针) 寄存器有关。在 entry.S 中有如下语句:

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

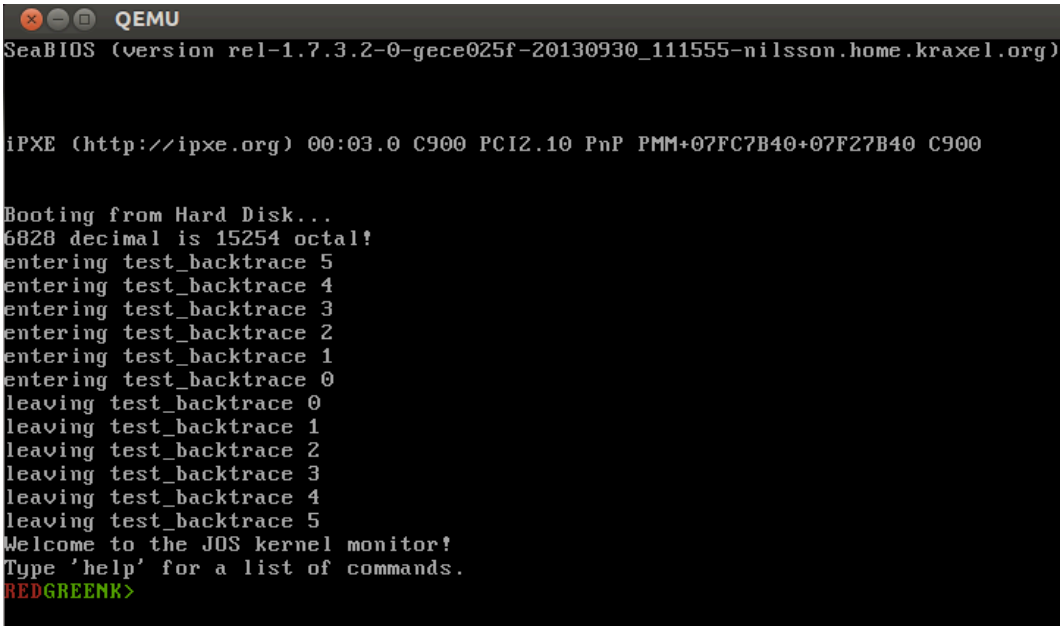

Figure 1: Challenge 效果图

bootstacktop是一个很神奇的值，观察下面一段在 entry.S 的结尾的代码:

```
1  .data
2  ###################################################
3  # boot stack
4  ###################################################
5      .p2align    PGSHIFT         # force page alignment
6      .globl              bootstack
7  bootstack:
8      .space              KSTKSIZE
9      .globl              bootstacktop
10 bootstacktop:
```

这里声明了.data 段，然后把栈底跟 page 的边界对齐 (PGSHIFT 是对齐变量)，栈顶相对于栈底偏移了栈的大小 (KSTKSIZE)。在 inc/memlayout.h 和 inc/mem.h 中定义了

1. PGSHIFT 的大小为 12，即 $log_2(PGSIZE)$，PGSIZE 为 4096B，即 4KB
2. KSTKSIZE 的大小为 page 大小的 8 倍，即 32KB

因此bootstacktop就是从 data 段开始的，对齐页边界以后，一个长度为 32KB 的栈的栈顶地址。把 ESP 初始化为该值以后，相当于得到了一个空栈。

下面检查一下bootstacktop的值：首先 objdump kernel 发现.data 的地址为0xf0110000，经过计算，bootstacktop的值应该为0xf0118000（该值在 elf 文件中也存在)。最后 gdb 调试时也发现bootstacktop的值为0xf01180000。

2.11 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.*

test_backtrace的起始地址在 0xf0100040, 之后执行的语句如下：

首先是对栈的操作和参数的处理：

```
1 f0100040:       55                      push   %ebp
2 f0100041:       89 e5                   mov    %esp,%ebp
3 f0100043:       53                      push   %ebx
4 f0100044:       83 ec 14                sub    $0x14,%esp
5 f0100047:       8b 5d 08                mov    0x8(%ebp),%ebx
```

ebp 中存储的是基址，因此要将其入栈，然后把当前的栈顶位置 esp 保存到 ebp 中。由于寄存器 ebx 的值在 call 之后也要被保存，因此这里多了一次 push。之后减少栈指针值，然后把参数 x(即存放在 0x8(ebp) 处的数据) 传给 ebx。

下面计算栈空间：

首先，因为在函数刚被调用时，执行了两次push操作，因此实际上使用了 8byte 的栈空间；之后，因为要保存传给test_backtrace的参数和 eip 的值 (即返回地址)，又需要 8 个 byte 的空间；除此以外，test_backtrace还调用了打印函数cprintf，该函数有两个参数，一个是格式化字符串，一个是参数列表。因为编译器之前已经确定，这里调用cprintf的参数列表只有一个值 x，因此要为调用cprintf预留 8byte 的空间用于传递参数。因此，传递参数最多用 8byte。但是因为 esp 在代码中直接减去了 0x14，因此在这里的预留空间应该为 20bytes。

综上，总共需要的空间为：8+20+4=32，即为 32KB 大小的空间。

举例，第一次调用cprintf的参数准备：

```
=> 0xf010004a:  mov    %ebx,0x4(%esp)
14 cprintf("entering␣test_backtrace␣%d\n", x);
(gdb)
=> 0xf010004e:  movl   $0xf01017a0,(%esp)
0xf010004e 14 cprintf("entering␣test_backtrace␣%d\n", x);
(gdb) x/s 0xf01017a0
0xf01017a0:      "entering␣test_backtrace␣%d\n"
```

注意0xf01017a0中存储的数据 (即为x/s 0xf01017a0的输出)。

2.12 Exercise 11.

Implement the backtrace function as specified above.

实现代码如下：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    int i;
    uint32_t ebp, eip, args[5];
    cprintf("Stack␣backtrace:\n");
    ebp = read_ebp();
```

```
8     while (ebp != 0x0)
9     {
10        eip = *((uint32_t *)ebp+0x1);
11        for (i = 0; i < 5; i++)
12            args[i] = *((uint32_t*) ebp+2+i);
13        cprintf("␣␣ebp␣%08x␣␣eip␣%08x", ebp, eip);
14        cprintf("␣␣args␣%08x␣%08x␣%08x␣%08x␣%08x\n",
15                args[0],
16                args[1],
17                args[2],
18                args[3],
19                args[4]);
20        ebp = *((uint32_t *)ebp);
21    }
22    return 0;
23 }
```

基本思路是，ebp 中保存的是函数刚被调用时，esp 指向的位置，即栈顶；栈顶再往地址高位依次保存：上层过程调用的的 ebp，返回的地址 eip，以及最大 5 个的参数列表。

2.13 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.*

首先应该解决__STAB_*是什么的问题
在 kernel.ld 中，存在对.stab 段的描述：

```
1 /* Include debugging information in kernel memory */
2 .stab : {
3     PROVIDE(__STAB_BEGIN__ = .);
4     *(.stab);
5     PROVIDE(__STAB_END__ = .);
6     BYTE(0)     /* Force the linker to allocate space
7                    for this section */
8 }
```

这是一个 linker script 文件，目的是对程序的不同段进行位置分配。那么在这里分配一个.stab 段的目的，正是为了在其中嵌入 debug 信息。用 objdump 的 -G 选项可以输出该 ELF 文件的 stab 信息。比如我想知道和代码文件 console.c 相关的 stab 信息，可以直接：

```
>i386-jos-elf-objdump -G obj/kern/kernel | grep "console.c"
137    SO     0      2      f010019c 2816   kern/console.c
167    SOL    0      0      f01001a8 2816   kern/console.c
173    SOL    0      0      f01001b3 2816   kern/console.c
177    SOL    0      0      f01001ba 2816   kern/console.c
200    SOL    0      0      f010021e 2816   kern/console.c
204    SOL    0      0      f0100226 2816   kern/console.c
211    SOL    0      0      f0100246 2816   kern/console.c
215    SOL    0      0      f010024e 2816   kern/console.c
219    SOL    0      0      f0100266 2816   kern/console.c
245    SOL    0      0      f01003a6 2816   kern/console.c
249    SOL    0      0      f01003c3 2816   kern/console.c
261    SOL    0      0      f01003d8 2816   kern/console.c
265    SOL    0      0      f01003e5 2816   kern/console.c
290    SOL    0      0      f01004b8 2816   kern/console.c
334    SOL    0      0      f010059a 2816   kern/console.c
338    SOL    0      0      f01005a0 2816   kern/console.c
343    SOL    0      0      f01005ae 2816   kern/console.c
350    SOL    0      0      f01005f0 2816   kern/console.c
354    SOL    0      0      f0100604 2816   kern/console.c
```

其中最终要的是最后一列，即 stabstr，可以用来输出我们需要的调试信息；第二列指的是该 stab 的类型，如果是 SO 则代表文件名，SLINE 代表代码行号，FUN 代表的是函数名称。更进一步的分析可以发现，与一个函数相关的 stab 信息，都是连续的：

```
435 FUN    0      0      f0100717 3928   mon_help:F(0,1)
436 PSYM   0      0      00000008 3853   argc:p(0,1)
437 PSYM   0      0      0000000c 3944   argv:p(0,19)
438 PSYM   0      0      00000010 3957   tf:p(0,20)
439 SLINE  0      34     00000000 0
```

```
440 SLINE  0      38     00000006 0
441 SLINE  0      40     0000003e 0
442 FUN    0      0      f010075c 3968  mon_backtrace:F(0,1)
443 PSYM   0      0      00000008 3853  argc:p(0,1)
444 PSYM   0      0      0000000c 3944  argv:p(0,19)
445 PSYM   0      0      00000010 3957  tf:p(0,20)
446 SLINE  0      60     00000000 0
447 SLINE  0      64     00000007 0
448 SOL    0      0      f010076f 2831  ./inc/x86.h
```

在代码文件中,mon_help 与 mon_backtrace 距离就很近(这里 mon_kerninfo 为什么会被提前我也不知道)，在 stab 段中也是紧挨着的。同时，第 5 列的值，对于 FUN 类型的 stab 来说，是汇编代码中的地址，而且这些地址是非降序的，所以我们可以通过 eip 在第 5 列上的二分查找来得到需要的信息(相关的功能已经在 stab_binsearch 中实现了，因此这里只需要调用 API 即可)。

明白了原理以后，后续的修改就很容易了:

首先，在 debuginfo_eip 中增加对行号的支持:

```
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
if (lline <= rline)
    info->eip_line = stabs[lline].n_desc;
else
    return -1;
```

然后，在mon_backtrace中增加格式化输出的支持:

```
if (debuginfo_eip(eip, &info) != -1)
{
    cprintf("␣␣␣␣␣␣␣␣%s:", info.eip_file);
    cprintf("%d:␣",         info.eip_line);
    cprintf("%.*s",         info.eip_fn_namelen,
                            info.eip_fn_name);
    cprintf("+%u\n",        eip-(uint32_t)info.eip_fn_addr);
}
```

最后，增加backtrace的命令:

```
static struct Command commands[] = {
{ "help",
  "Display␣this␣list␣of␣commands",
   mon_help },
{ "kerninfo",
  "Display␣information␣about␣the␣kernel",
  mon_kerninfo },
{ "backtrace",
  "Display␣information␣about␣the␣current␣call␣stack",
  mon_backtrace},
};
```

3 Conclusion

综上，Lab1 是一个代码量很少，但是涉及硬件底层细节很多的项目。作为一个没有学过微机原理的同学，通过这个 Lab 学到了很多东西：比如实模式和保护模式究竟是什么，x86 的寄存器模型与指令编码和体系结构书上的 MIPS 到底有多大的区别。更重要的是明白了，体会到了一个概念，即操作系统本身也只不过是一个程序而已。这个程序的运行环境，和编码框架都跟一般的 C 语言算法程序不一样，也更能展现 C 语言的真正的魅力。大家都说 C 语言更适合系统底层的编程，当我看到了在 C 代码中嵌入的 asm 指令，以及 C 函数到汇编代码的直接翻译的能力时才真切地体会到了这一点。