

# Lab 1 Booting a PC

## Introduction

This lab is split into three parts. The first part concentrates on getting familiarized with x86 assembly language, the QEMU x86 emulator, and the PC's power-on bootstrap procedure. The second part examines the boot loader for our 6.828 kernel, which resides in the boot directory of the lab tree. Finally, the third part delves into the initial template for our 6.828 kernel itself, named JOS, which resides in the kernel directory.

## Experiment environment

Hardware:

Memory: 4G

Processor: Intel(R) Core(TM) i5-8250H CPU @ 1.60GHz × 2

OS Type: 64 bit

Disk: 50GB

Software:

OS: Ubuntu 20.04 LTS(x86\_64)

GCC: gcc 9.3.0 #Run gcc -v

Make: GNU Make 4.2.1 #Run make --version

GDB: GNU gdb 9.1.0 #gdb -v

实际上环境的配置还是比较麻烦的，按照实验页的步骤安装好 qemu 和 gdb 后，还需要一些其他的工具链，在配置时我没有记录详细过程，但大抵是参照参考文献的方法进行的，便不再回忆了。

## Part 1: PC Bootstrap

The purpose of the first exercise is to introduce you to x86 assembly language and the PC bootstrap process, and to get you started with QEMU and QEMU/GDB debugging. You will not have to write any code for this part of the lab, but you should go through it anyway for your own understanding and be prepared to answer the questions posed below.

## Exercises 1

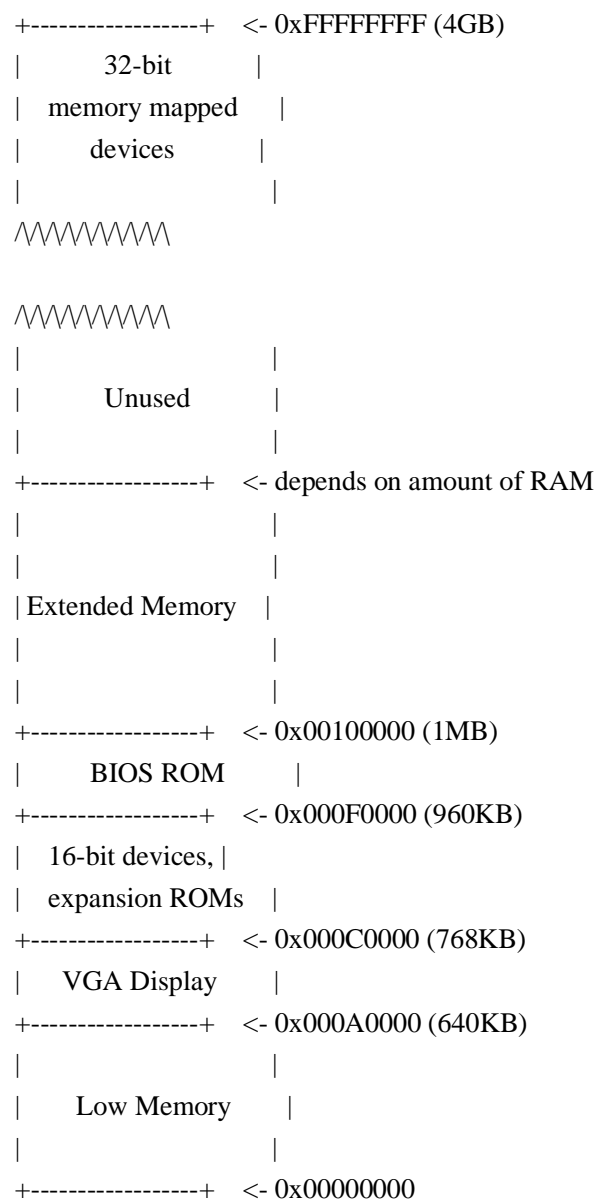
Familiarize yourself with the assembly language materials available on [the 6.828 reference page](#). You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.

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.

Exercise1 就是需要我们对链接中的参考资料有一些了解，没有什么实质性的内容。

## Exercises 2

We will now dive into a bit more detail about how a PC starts up. A PC's physical address space is hard-wired to have the following general layout:



The first PCs, which were based on the 16-bit Intel 8088 processor, were only capable of addressing 1MB of physical memory. The physical address space of an early PC would therefore start at 0x00000000 but end at 0x000FFFFF instead of 0xFFFFFFFF. The 640KB area marked

"Low Memory" was the *only* random-access memory (RAM) that an early PC could use; in fact the very earliest PCs only could be configured with 16KB, 32KB, or 64KB of RAM!

The 384KB area from 0x000A0000 through 0x000FFFFF was reserved by the hardware for special uses such as video display buffers and firmware held in non-volatile memory. The most important part of this reserved area is the Basic Input/Output System (BIOS), which occupies the 64KB region from 0x000F0000 through 0x000FFFFF. In early PCs the BIOS was held in true read-only memory (ROM), but current PCs store the BIOS in updateable flash memory. The BIOS is responsible for performing basic system initialization such as activating the video card and checking the amount of memory installed. After performing this initialization, the BIOS loads the operating system from some appropriate location such as floppy disk, hard disk, CD-ROM, or the network, and passes control of the machine to the operating system.

When Intel finally "broke the one megabyte barrier" with the 80286 and 80386 processors, which supported 16MB and 4GB physical address spaces respectively, the PC architects nevertheless preserved the original layout for the low 1MB of physical address space in order to ensure backward compatibility with existing software. Modern PCs therefore have a "hole" in physical memory from 0x000A0000 to 0x00100000, dividing RAM into "low" or "conventional memory" (the first 640KB) and "extended memory" (everything else). In addition, some space at the very top of the PC's 32-bit physical address space, above all physical RAM, is now commonly reserved by the BIOS for use by 32-bit PCI devices.

Recent x86 processors can support *more* than 4GB of physical RAM, so RAM can extend further above 0xFFFFFFFF. In this case the BIOS must arrange to leave a *second* hole in the system's RAM at the top of the 32-bit addressable region, to leave room for these 32-bit devices to be mapped. Because of design limitations JOS will use only the first 256MB of a PC's physical memory anyway, so for now we will pretend that all PCs have "only" a 32-bit physical address space. But dealing with complicated physical address spaces and other aspects of hardware organization that evolved over many years is one of the important practical challenges of OS development.

The following line:

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

is GDB's disassembly of the first instruction to be executed. From this output you can conclude a few things:

- The IBM PC starts executing at physical address 0x000ffff0, which is at the very top of the 64KB area reserved for the ROM BIOS.
- The PC starts executing with CS = 0xf000 and IP = 0xffff0.
- The first instruction to be executed is a jmp instruction, which jumps to the segmented address CS = 0xf000 and IP = 0xe05b.

Why does QEMU start like this? This is how Intel designed the 8088 processor, which IBM used in their original PC. Because the BIOS in a PC is "hard-wired" to the physical address range 0x000f0000-0x000fffff, this design ensures that the BIOS always gets control of the machine first after power-up or any system restart - which is crucial because on power-up there *is* no other software anywhere in the machine's RAM that the processor could execute. The QEMU emulator comes with its own BIOS, which it places at this location in the processor's simulated physical address space. On processor reset, the (simulated) processor enters real mode and sets CS to 0xf000 and the IP to 0xffff, so that execution begins at that (CS:IP) segment address. How does the segmented address 0xf000:ffff turn into a physical address?

To answer that we need to know a bit about real mode addressing. In real mode (the mode that PC starts off in), address translation works according to the formula: *physical address* = 16 \* *segment* + *offset*. So, when the PC sets CS to 0xf000 and IP to 0xffff, the physical address referenced is:

$$\begin{aligned} 16 * 0xf000 + 0xffff & \quad \# \text{ in hex multiplication by 16 is} \\ = 0xf0000 + 0xffff & \quad \# \text{ easy--just append a 0.} \\ = 0xffff0 & \end{aligned}$$

0xffff0 is 16 bytes before the end of the BIOS (0x100000). Therefore we shouldn't be surprised that the first thing that the BIOS does is `jmp` backwards to an earlier location in the BIOS; after all how much could it accomplish in just 16 bytes?

由于 8088CPU 中寄存器都是 16 位，而 CPU 地址总线是 20 位的，我们怎么通过 16 位的寄存器去拼接 20 位的地址呢？所以我们需要采用下面的方法：把段寄存器中的值左移 4 位，形成 20 位段基址，然后和 16 位段内偏移相加，就得到了真实地址。比如上面的指令中段寄存器的内容为 0xf000，所以真实地址为  $0xf000 \ll 4 + 0xe05b = 0xfe05b$ 。

上面讲述了地址的计算方式，IBM PC 从物理地址 0x000ffff0 开始执行，它位于为 ROM BIOS 保留的 64KB 区域的最顶端。要执行的第一条指令是一条 `jmp` 指令，它跳转到分段地址 CS = 0xf000 和 IP = 0xe05b。

所以问题来了：

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.

Exercise2 需要我们在进入 QEMU 之后使用 `si` 来跟踪 ROM BIOS 中的更多指令，猜测他们在做些什么，我们可以参考一些资料，不需要弄清楚所有的细节。

所以我们接下来介绍一下前面的代码们：

1. 0xffff0: `ljmp $0xf000, $0xe05b`  
跳转到 0xf000:0xe05b.
2. 0xfe05b: `cml $0x0, $cs:0x6ac8`

3. 0xfe062: jne 0xfd2e1

0x0 与 cs:0x6ac8 这个位置的数字进行比较，如果不相等就跳到 0xfd2e1，指令决定了这是一次 reboot 还是 resume。如果不相等说明这是一次 resume，跳转到 resume 程序的入口地址。具体位置和 BIOS 有关，可以查看 [stackoverflow 上相关问题](#)。

4. 0xfe066: xor %dx, %dx

5. 0xfe068: mov %dx %ss

6. 0xfe06a: mov \$0x7000, %esp

7. 0xfe070: mov \$0xf34d2, %edx

8. 0xfe076: jmp 0xfd15c

9. 0xfd15c: mov %eax, %ecx

首先看地址说明没有跳转，这是一次 reboot。接着清零 %dx 寄存器。设置栈的段地址为 0x0000，偏移量 0x7000。跳转到 0xfd15c 继续执行。

10. 0xfd15f: cli

关闭中断指令，因为启动不能被中断。

11. [0xfd160] 0xfd160: cld

设置方向标志位为 0，表明串操作方向是从低地址到高地址。

12. 0xfd161: mov \$0x8f, %eax

13. 0xfd167: out %al, \$0x70

14. 0xfd169: in \$0x71, %al

这三个操作中涉及到两个新的指令 out, in。这两个操作是用于操作 IO 端口的。

CPU 与外部设备通讯时，通常是通过访问，修改设备控制器中的寄存器来实现的。那么这些位于设备控制器当中的寄存器也叫做 IO 端口。为了方便管理，80x86CPU 采用 IO 端口单独编址的方式，即所有设备的端口都被命名到一个 IO 端口地址空间中。这个空间是独立于内存地址空间的。所以必须采用和访问内存的指令不一样的指令来访问端口。

所以这里引入 in, out 操作：

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

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

标准规定端口操作必须要用 al 寄存器作为缓冲。

那么这三条命令就是要操作端口 0x70, 0x71，它们对应的是什么设备呢？根据下面的链接中所提供的清单：<http://bochs.sourceforge.net/techspec/PORTS.LST>

我们知道了，0x70 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。这个链接有详细介绍 <http://wiki.osdev.org/CMOS>

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 1 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 0b10000001 = 0x81。

即 `mov $0x81, %al`

`out %al, 0x70`

然后对于这个地址单元的操作，比如读或者写就可以由 0x71 端口完成，比如你现在想从 1 号存储单元里面读出它的值，在完成上面的两条指令后，就可以输入这条指令

`in $0x71, %al`

再回到我们的系统，这三条指令可以看出，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。

15. 0xfd16b: `in $0x92, %al`

16. 0xfd16d: `or $0x2, %al`

17. 0xfd16f: `out %al, $0x92`

这三条指令依然在控制外设。同样查询后得知 0x92 端口对应的是 PS/2 system control port A，第一位 1 使 A20 地址线被激活，用于测试可用的地址空间。

18. 0xfd171: `lidtw %cs:0x6ab8`

19. 0xfd177: `lgdtw %cs:0x6a74`

这两条指令分别用来加载 0xf6ab8 开始的 6 个字节到中断描述符表寄存器（IDTR）中和加载 0xf6a74 开始的 6 个字节到全局描述符表寄存器（GDTR）中。

20. 0xfd17d: `mov %cr0, %eax`

21. 0xfd180: `or $0x1, %eax`

22. 0xfd184: `mov %eax, %cr0`

这三条指令将 CR0 寄存器的最低位置为 1，代表着启动保护模式。

23. 0xfd187: `ljmpl $0x8, $0xfd18f`

一个长跳转结束保护模式。

其实很多地方还是不太理解的，不过先把搜集到的资料和自己的理解整合在这里就是了。

## Part 2: The Boot Loader

于 PC 来说，软盘，硬盘都可以被划分为一个个大小为 512 字节的区域，叫做扇区。一个扇区是一次磁盘操作的最小粒度。每一次读取或者写入操作都必须是一个或多个扇区。如果一个磁盘是可以被用来启动操作系统的，就把这个磁盘的第一个扇区叫做启动扇区。这一部分介绍的 boot loader 程序就位于这个启动扇区之中。当 BIOS 找到一个可以启动的软盘或硬盘后，它就会把这 512 字节的启动扇区加载到内存地址 0x7c00~0x7dff 这个区域内。然后使用 jmp 指令将 CS:IP 设置为 0000:7c00，并将控制权传递给引导加载程序。像 BIOS 加载地址一样，这些地址是相当任意的-但它们对于 PC 是固定的和标准化的。

对于 6.828，我们将采用传统的硬盘启动机制，这就意味着我们的 boot loader 程序的大小必须小于 512 字节。整个 boot loader 是由一个汇编文件，boot/boot.S，以及一个 C 语言文件，boot/main.c。Boot loader 必须完成两个主要的功能。

1. 首先，boot loader 要把处理器从实模式转换为 32bit 的保护模式，因为只有在这种模式下软件可以访问超过 1MB 空间的内容。

2. 然后，boot loader 可以通过使用 x86 的特定的 IO 指令，直接访问 IDE 磁盘设备寄存器，从磁盘中读取内核。

对于 boot loader 来说，有一个文件很重要，obj/boot/boot.asm。这个文件是我们真实运行的 boot loader 程序的反汇编版本。所以我们可以把它和它的源代码即 boot.S 以及 main.c 比较一下。

### 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.

Be able to answer the following questions:

- At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
- What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?
- *Where* is the first instruction of the kernel?
- 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?

Answers:

- 答: 可以阅读 boot.s 文件知道在如图所示的 `ljmp $PROT_MODE_CSEG, $protcseg`:

```
43
44 # Switch from real to protected mode, using a bootstrap GDT
45 # and segment translation that makes virtual addresses
46 # identical to their physical addresses, so that the
47 # effective memory map does not change during the switch.
48 lgdt    gdtDESC
49 movl    %cr0, %eax
50 orl     $CR0_PE_ON, %eax
51 movl    %eax, %cr0
52
53 # Jump to next instruction, but in 32-bit code segment.
54 # Switches processor into 32-bit mode.
55 ljmp     $PROT_MODE_CSEG, $protcseg
56
57 .code32                                # Assemble for 32-bit mode
58 protcseg:
```

图 3.1

- 答: boot loader 执行的最后一条语句是 bootmain 子程序中的最后一条语句

```
" ((void (*)(void)) (ELFHDR->e_entry))(); ",
306 7d71: ff 73 f4      pushl -0xc(%ebx)
307 7d74: ff 73 ec      pushl -0x14(%ebx)
308 7d77: e8 66 ff ff ff call 7ce2 <readseg>
309      for (; ph < eph; ph++)
310 7d7c: 83 c4 10      add $0x10,%esp
311 7d7f: eb e5         jmp 7d66 <bootmain+0x41>
312      ((void (*)(void)) (ELFHDR->e_entry))();
313 7d81: ff 15 18 00 01 00 call *0x10018
314 }
315
316 static inline void
317 outw(int port, uint16_t data)
318 {
319     outb(port, data >> 8);
320     outb(port, data & 0xff);
321 }
```

图 3.2

- 答: 即跳转到操作系统内核程序的起始指令处。这个第一条指令位于 `/kern/entry.S` 文件中, 第一句 `movw $0x1234, 0x472`.

```
Breakpoint 1, 0x00007d61 in ?? ()
(gdb) b* 0x7d81
Breakpoint 2 at 0x7d81
(gdb) c
Continuing.
=> 0x7d81:      call *0x10018

Breakpoint 2, 0x00007d81 in ?? ()
(gdb) x/10i $pc
=> 0x7d81:      call *0x10018
0x7d87:      mov $0x8a00,%edx
0x7d8c:      mov $0xffff8a00,%eax
0x7d91:      out %ax, (%dx)
0x7d93:      mov $0xffff8e00,%eax
0x7d98:      out %ax, (%dx)
0x7d9a:      jmp 0x7d9a
0x7d9c:      add %al, (%eax)
0x7d9e:      add %al, (%eax)
0x7da0:      add %al, (%eax)
(gdb) x/10i *0x10018
0x1000c:      movw $0x1234, 0x472
```

图 3.3

这个就是先在 `obj/boot/boot.asm` 中找到 `0x7d81` 位置会执行 bootmain 最后一条语句 (如图 3.2), 先设置断点然后执行到这里, 再进入, 看到 `0x10018` 开始后面的第一条语句。

- 答: 首先关于操作系统一共有多少个段, 每个段又有多少个扇区的信息位于操作系



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

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

## Exercise 4

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 printed lines 1 and 6 come from, how all the values in printed 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.

这个就是让你阅读一下 pointers.c 然后理解为什么会有这样的结果，这个其实并不难，在此复制一下其他人文档的结果：（非要说的话第三个和第五个最值得看一下吧）

首先程序声明了 3 个重要的数组，指针

int 型数组--int a[4];    int 类型指针--int \*b = malloc(16);    int 类型指针--int \*c;

打印的第一句：

```
printf("1: a = %p, b = %p, c = %p\n", a, b, c);
```

其中：

a 输出的是数组 a 的首地址，0xbfb4bf84。

b 输出的是指针 b 所指向的操作系统分配给它的空间的起始地址，0x8886008

c 输出的是未定义的指针变量 c 的值，0xb75d8255

打印第二句之前，完成的操作：

c = a; 让 c 指针和 a 指向同一个内存地址，0xbfb4bf84;

for(i = 0; i < 4; i++) a[i] = 100 + i;    这个操作会让数组 a 的四个单元的值变为，100,101,102,103。

c[0] = 200;    由于 c 和 a 指向同一个地方，c[0] = a[0] = 200;

printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d,", a[0], a[1], a[2], a[3]);    所以打印第二句，第一个元素的值会是 200。

打印第三句之前，完成操作：

```

c[1] = 300;
*(c+2) = 301;
3[c] = 302;

```

分别代表访问数组中的值的三种不同的方法。由于 c 和 a 相同，所以它们也是在修改数组 a 的值。把 a[1], a[2], a[3]全都被改变。printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d,", a[0], a[1], a[2], a[3]);

打印第四句前，完成操作：c = c+1;

```

将指针 c 指向数组中下一个单元，即 a[1]
*c = 400
修改 a[1]值为 400
printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d,", a[0], a[1], a[2], a[3]);

```

打印第五句前，完成操作

```

c = (int *)((char *)c + 1);

```

先把 c 强制转换为 char 类型指针，然后指针加 1，此时 c 的数值应该也加 1，由于之前 c 指向 a 数组中 1 号元素的起始地址，1 号元素起始地址为 0xbfb4bf88，之后加 1，变为 0xbfb4bf89。然后再把 c 强制转换回 int 类型指针，

此时 c 所操作的地址为 0xbfb4bf89~0xbfb4bf8c。

```

*c = 500

```

把地址地址为 0xbfb4bf89~0xbfb4bf8c 的替换为 500，此时会影响原来数组 a 的 1,2 号元素。原来一号元素在内存中存放在地址 0xbfb4bf88~0xbfb4bf8b 处，存放的值为 400，十六进制为 0x00000190

```

0xbfb4bf88  0xbfb4bf89  0xbfb4bf8a  0xbfb4bf8b
          90              01              00              00

```

原来二号元素在内存中存放在地址 0xbfb4bf8c~0xbfb4bf8f 处，存放的值为 301，十六进制为 0x0000012D。

```

0xbfb4bf8c  0xbfb4bf8d  0xbfb4bf8e  0xbfb4bf8f
          2D              01              00              00

```

而现在 c 操作的地址为 0xbfb4bf89~0xbfb4bf8c，并赋值 500，十六进制为 0x000001F4，所以内存单元变为

```

0xbfb4bf88  0xbfb4bf89  0xbfb4bf8a  0xbfb4bf8b
          90              F4              01              00
0xbfb4bf8c  0xbfb4bf8d  0xbfb4bf8e  0xbfb4bf8f
          00              01              00              00

```

所以此时 a[1]的值变为 0x0001F490 = 128144, a[2]的值变为 0x00000100 = 256.

打印第六句前，完成操作：

```

b = (int *)a + 1;

```

这步操作会把 b 的值加 1，由于 b 现在是 int 类型指针，所以数值上 b 的值增加 4，所以 b 的值变为 0xbfb4bf84 + 0x4 = 0xbfb4bf88

```

c = (int *)((char *)a+1);

```

这步还是先把 a 转换为 char 型指针，然后加 1，此时加 1 会让数值上只增加 1，所以 c 的值变为 0xbfb4bf84 + 0x1 = 0xbfb4bf85

## Exercise 5

对于 6.828, 可以将 ELF 可执行文件视为带有加载信息的标头, 然后是几个程序段, 每个程序段都是要在指定地址加载到内存中的连续代码或数据块。引导加载程序不会修改代码或数据; 它将其加载到内存中并开始执行它。

ELF 二进制文件以一个定长的 ELF 头开始, 后面是边长的程序头, 每个程序头列举了需要载入的程序节。ELF 头和程序头的定义详情可见 `inc/elf.h`。(以下我们来回忆 ICS 中讲的不多但很重要的编译链接那一章)

我们主要关心以下程序节:

- `.text` 程序的可执行指令
- `.rodata` 只读数据, 比如 C 编译器生成的 ASCII 字符串常量。
- `.data` 数据节包括了程序已经初始化的数据, 比如初始化了的全局变量。

在 `.data` 节之后是 `.bss` 节, 它为未初始化的全局变量保留空间。链接器只为 `.bss` 记录了地址和大小, 需要链接器或者程序自身来为这一节做零初始化。

在 `.text` 节中我们特别关心“VMA”(链接地址)和“LMA”(加载地址)。加载地址顾名思义指这一节应当被加载到内存中的位置。而链接地址则是这一程序节开始执行的期望位置。比如链接最重要的一步就是符号重定位。汇编代码中有一些常量在一开始编译生成可重定位目标文件的时候, 其值是不确定的, 因为编译器并不知道, 最后这些程序(数据)的位置; 只有在链接过程中, 我们知道了程序开始执行的期望位置, 然后链接器会对这些常量进行替换。在此基础上我们就可以做 Exercise 5 了。补充以下两个命令:

```
objdump -h obj/boot/boot.out #Used to check sections
```

```
objdump -x obj/kern/kernel #Used to inspect program headers
```

下面是问题:

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!

Exercise 5 要求修改 `boot/Makefrag` 中 `Ttext` 后面的链接地址, 重新编译 lab, 查看什么地方会出错。我们运行就会发现第一个用到链接时常量的指令 `lgdt gdt_desc` 会出错。

除了节中的信息外, ELF 头中的 `e_entry` 域也十分重要, 它是程序入口的链接地址, 即程序的 `.text` 节从内存的什么地址开始执行, 可以通过运行以下命令查看 `e_entry`:

```
objdump -f obj/kern/kernel
```

回答:

这道题希望我们能够去修改 boot loader 的链接地址, 在 lab 1 中, 作者引入了两个概念, 一个是链接地址, 一个是加载地址。链接地址可以理解为通过编译器链接器处理形成的可执行程序指令的地址, 即逻辑地址。加载地址则是可执行文件真正被装入内存后运行的地址, 即物理地址。

那么在 boot loader 中, 由于在 boot loader 运行时还没有任何的分段处理机制, 或分页处理机制, 所以 boot loader 可执行程序中的链接地址就应该等于加载地址。在 lab 中作者说, BIOS 默认把 boot loader 加载到 `0x7C00` 内存地址处, 所以就要求 boot loader 的链接地址也

要在 0x7C00 处。boot loader 地址的设定是在 boot/Makefrag 中完成的，所以根据题目的要求，我们需要改动这个文件的值。

首先按照题目要求，在 lab 目录下输入 make clean，清除掉之前编译出来的内核可执行文件，在清除之前你可以先把 obj/boot/boot.asm 文件拷贝出来，之后可以用来比较。然后打开这个 boot/Makefrag 文件，我们会发现下列语句：

```
26 $(OBJDIR)/boot/boot: $(BOOT_OBJS)
27     @echo + ld boot/boot
28     $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^
29     $(V)$(OBJDUMP) -S $@.out >$@.asm
30     $(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
31     $(V)perl boot/sign.pl $(OBJDIR)/boot/boot
32
```

图 5.1

其中的-Ttext 0x7C00，就是指定链接地址，我们可以把它修改为 0x7E00，然后保存退出。

然后在 lab 下输入 make，重新编译内核，首先查看一下 obj/boot/boot.asm，并且和之前的那个 obj/boot/boot.asm 文件做比较。下图是新编译出来的 boot.asm：

```
12 .code16                                # Assemble for 16-bit mode
13 cli                                    # Disable interrupts
14 7e00: fa                                cli
15 cld                                    # String operations increment
16 7e01: fc                                cld
17
```

图 5.2

然后我们还是按照原来的方式，调试一下内核：

由于 BIOS 会把 boot loader 程序默认装入到 0x7c00 处，所以我们还是再 0x7C00 处设置断点，并且运行到那里，结果发现如下：

```
(gdb) si
[ 0:7c2a] => 0x7c2a: mov    %eax,%cr0
0x00007c2a in ?? ()
(gdb) si
[ 0:7c2d] => 0x7c2d: ljmp   $0xb866,$0x87e32
0x00007c2d in ?? ()
(gdb) si
[f000:e05b] 0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062] 0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb) si
[f000:d0b0] 0xfd0b0: cli
0x0000d0b0 in ?? ()
```

图 5.3

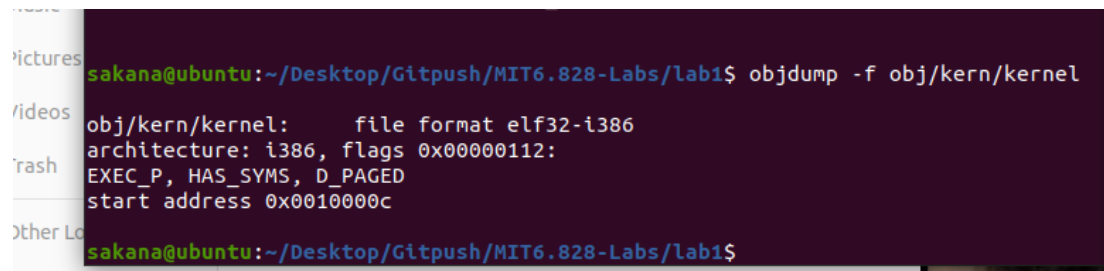
在 7c2d 之后就开始跳到不对的地方了，因为一开始把 7c00 变成 7e00 导致本来应该是 7c32 这里是 7e32 了。

## Exercise 6

Besides the section information, there is one more field in the ELF header that is important to us, named e\_entry. This field holds the link address of the *entry point* in the program: the memory

address in the program's text section at which the program should begin executing. You can see the entry point:

```
athena% objdump -f obj/kern/kernel
```



```
sakana@ubuntu:~/Desktop/Gitpush/MIT6.828-Labs/lab1$ objdump -f obj/kern/kernel
obj/kern/kernel:      file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
sakana@ubuntu:~/Desktop/Gitpush/MIT6.828-Labs/lab1$
```

图 6.1

所以这个 0x10000c 是 kernel 入口点。

下面是问题：

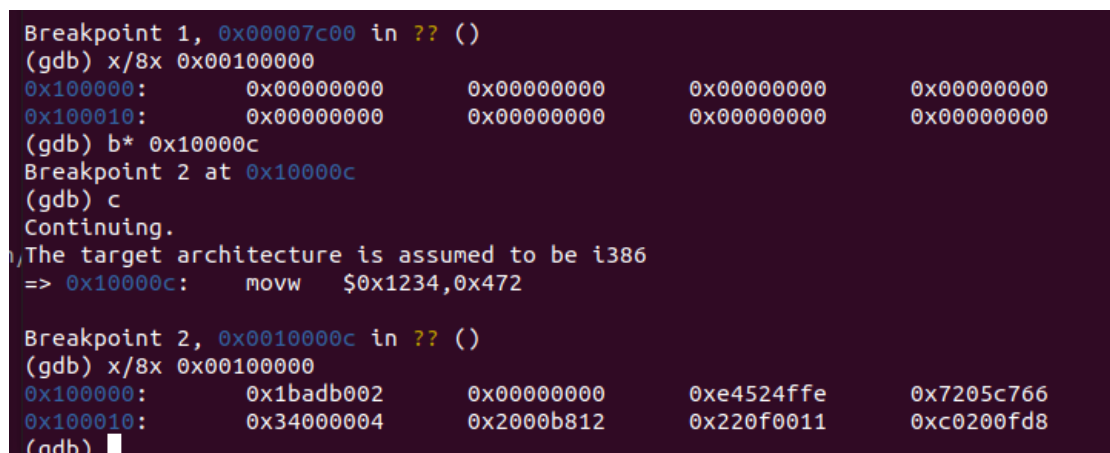
We can examine memory using GDB's `x` command. The [GDB manual](#) has full details, but for now, it is enough to know that the command `x/Nx ADDR` prints  $N$  words of memory at `ADDR`. (Note that both 'x's in the command are lowercase.) *Warning*: The size of a word is not a universal standard. In 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.)

Exercise 6 要求分别在 BIOS 进入 boot loader 和 boot loader 进入 kernel 的地方，检查从内存地址 0x00100000 开始的 8 个 word。回答以下两个问题：

- 为什么他们不一样？
- 在第二个断点停下来时，在该位置的是什么内容？

答：



```
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
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 0x00100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0x100010: 0x34000004 0x2000b812 0x220f0011 0xc0200fd8
(gdb)
```

图 6.2

为什么会产生这种变化，因为 bootmain 函数在最后会把内核的各个程序段送入到内存地

址 0x00100000 处, 所以这里现在存放的就是内核的某一个段的内容, 由于程序入口地址是 0x0010000C, 正好位于这个段中, 所以可以推测, 这里面存放的应该是指令段, 即.text 段的内容。

当 BIOS 进入 bootloader 时, 因为此时工作在实模式, 0x10000 以上的内存根本无法访问, 所以内存中的内容应该为空。bootloader 进入 kernel 时, 程序地址 0x100000 处加载的操作系统的代码。

## Part 3: The Kernel

We will now start to examine the minimal JOS kernel in a bit more detail. (And you will finally get to write some code!). Like the boot loader, the kernel begins with some assembly language code that sets things up so that C language code can execute properly.

在运行 boot loader 时，boot loader 中的链接地址（虚拟地址）和加载地址（物理地址）是一样的。但是当进入到内核程序后，这两种地址就不再相同了。

操作系统内核程序在虚拟地址空间通常会被链接到一个非常高的虚拟地址空间处，比如 0xf0100000，目的就是能够让处理器的虚拟地址空间的低地址部分能够被用户利用来进行编程。

但是许多的机器其实并没有能够支持 0xf0100000 这种地址那么大的物理内存，所以我们不能把内核的 0xf0100000 虚拟地址映射到物理地址 0xf0100000 的存储单元处。

这就造成了一个问题，在我们编程时，我们应该把操作系统放在高地址处，但是在实际的计算机内存中却没有那么高的地址，这该怎么办？

解决方案就是在虚拟地址空间中，我们还是把操作系统放在高地址处 0xf0100000，但是在实际的内存中我们把操作系统存放在一个低的物理地址空间处，如 0x00100000。那么当用户程序想访问一个操作系统内核的指令时，首先给出的是一个高的虚拟地址，然后计算机中通过某个机构把这个虚拟地址映射为真实的物理地址，这样就解决了上述的问题。那么这种机构通常是通过分段管理，分页管理来实现的。

在这个实验中，首先是采用分页管理的方法来实现上面所讲述的地址映射。但是设计者实现映射的方式并不是通常计算机所采用的分页管理机构，而是自己手写了一个程序 `lab/kern/entrygdir.c` 用于进行映射。既然是手写的，所以它的功能就很有限了，只能把虚拟地址空间的地址范围：0xf0000000~0xf0400000，映射到物理地址范围：0x00000000~0x00400000 上面。也可以把虚拟地址范围：0x00000000~0x00400000，同样映射到物理地址范围：0x00000000~0x00400000 上面。任何不在这两个虚拟地址范围内的地址都会引起一个硬件异常。虽然只能映射这两块很小的空间，但是已经足够刚启动程序的时候来使用了。

### 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.

使用 Qemu 和 GDB 去追踪 JOS 内核文件，并且停止在 `movl %eax, %cr0` 指令前。此时看一

下内存地址 0x00100000 以及 0xf0100000 处分别存放着什么。然后使用 stepi 命令执行完这条命令，再次检查这两个地址处的内容。确保你真的理解了发生了什么。如果这条指令 movl %eax, %cr0 并没有执行，而是被跳过，那么第一个会出现问题的指令是什么？我们可以通过把 entry.S 的这条语句加上注释来验证一下。

答：

按要求进行第一步：

```
=> 0x100025:  mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) x/4xb 0x00100000
0x100000:  0x02    0xb0    0xad    0x1b
(gdb) x/4xb 0xf0100000
0xf0100000 <_start-268435468>: 0x00    0x00    0x00    0x00
(gdb) si
=> 0x100028:  mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/4xb 0x00100000
0x100000:  0x02    0xb0    0xad    0x1b
(gdb) x/4xb 0xf0100000
0xf0100000 <_start-268435468>: 0x02    0xb0    0xad    0x1b
(gdb)
```

图 7.1

可以看到 0xf0100000 的值已经和 0x00100000 的值一样了，即原本存放在 0xf0100000 的内容现在已经被映射到 0x00100000 去了。

注释掉之后：

```
=> 0x100025:  mov    $0xf010002c,%eax
0x00100025 in ?? ()
(gdb) si
=> 0x10002a:  jmp    *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c <relocated>:  add    %al,(%eax)
relocated () at kern/entry.S:74
74      movl    $0x0,%ebp                                # nuke frame pointer
(gdb) si
=> 0xf010002e <relocated+2>:  add    %al,(%eax)
0xf010002e      74      movl    $0x0,%ebp                                # nuke
rame pointer
(gdb) si
=> 0xf0100030 <relocated+4>:  add    %al,(%eax)
0xf0100030      74      movl    $0x0,%ebp                                # nuke
rame pointer
(gdb) si
```

图 7.2

实际上是程序跑飞了，作为比较，我们可以看看正常的图 7.3



```

(gdb) si
=> 0xf010002f <relocated>: mov $0x0,%ebp
relocated () at kern/entry.S:74
74      movl $0x0,%ebp                # nuke fr
(gdb) si
=> 0xf0100034 <relocated+5>: mov $0xf0110000,%esp
relocated () at kern/entry.S:77
77      movl $(bootstacktop),%esp
(gdb) si
=> 0xf0100039 <relocated+10>: call 0xf01000aa <i386_init>
80      call i386_init
(gdb) si
=> 0xf01000aa <i386_init>: endbr32
i386_init () at kern/init.c:24
24      {
(gdb) si
=> 0xf01000ae <i386_init+4>: push %ebp
0xf01000ae 24      {
(gdb)

```

图 7.3

## Exercise 8

Most people take functions like `printf()` for granted, sometimes even thinking of them as "primitives" of the C language. But in an OS kernel, we have to implement all I/O ourselves. Read through `kern/printf.c`, `lib/printfmt.c`, and `kern/console.c`, and make sure you understand their relationship. It will become clear in later labs why `printfmt.c` is located in the separate `lib` directory.

接下来是我们的任务：

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.

首先回答一下介绍里面的问题，即 `printf.c`, `printfmt.c`, `console.c` 这三个文件的关联：

大致浏览三个源文件，其中粗略的观察到 3 点：

1. `kern/printf.c` 中的 `cprintf`, `vcprintf` 子程序调用了 `lib/printfmt.c` 中的 `vprintfmt` 子程序。
2. `kern/printf.c` 中的 `putch` 子程序调用了 `cputchar`，这个程序是定义在 `kern/console.c` 中的。
3. `lib/printfmt.c` 中的某些程序也依赖于 `cputchar` 子程序

所以得出结论，`kern/printf.c`, `lib/printfmt.c` 两个文件的功能依赖于 `kern/console.c` 的功能。

接着完成填充，其实很简单，模仿 case: 'u' 的十进制无符号进行就可：

```

// (unsigned) octal
case 'o':
    // Replace this with your code.
    num = getuint(&ap, lflag);
    base = 8;
    goto number;
// pointer

```

参照case 'u'对代码进行修改，修改之后make qemu显示结果

```

sakana@ubuntu:~/desktop/Gltipush/HIT6.828-Labs/Lab1$ make qemu
+ cc lib/printfmt.c
+ ld obj/kern/kernel
ld: warning: section '.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw
serial mon:stdio -gdb tcp::26000 -D qemu.log

```

图 8.1

对于这三个文件的更详细描述，可以见前人的文档，链接如下：

<https://www.cnblogs.com/fatsheep9146/p/5066690.html>

除此之外，Exercise8 还有额外的六个小问题：

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

解释 printf.c 和 console.c 之间的接口。特别的，console.c 提供了什么函数？它是如何被 printf.c 使用的？

printf.c 使用了 console.c 提供的 cputchar() 函数。由于 cprintf() 中参数长度是变长的，需要一个参数列表 va\_list 和格式字符串来决定参数格数。之后调用 vprintf() 函数，这个函数根据格式字符串从 va\_list 中读取参数，并使用 cputchar() 将参数打印出来。

Explain the following from console.c:

```
1      if (crt_pos >= CRT_SIZE) {
2          int i;
3          memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) *
sizeof(uint16_t));
4          for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5              crt_buf[i] = 0x0700 | ' ';
6          crt_pos -= CRT_COLS;
7      }
```

解释下面的 7 行代码：

crt\_pos 是当前光标位置，CRT\_SIZE 是屏幕上总共的可以输出的字符数(其值等于行数乘以每行的列数)，这段代码的意思是当屏幕输出满了以后，将屏幕上的内容都向上移一行，即将第一行移出屏幕，同时将最后一行用空格填充，最后将光标移动到屏幕最后一行的开始处。

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;
```

```
cstdio("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?

在 i386-init.c 中加入测试代码即可 @@ -34,6 +34,11 @@ i386\_init(void)

```
    cons_init();
    cprintf("6828 decimal is %o octal!\n", 6828);
+   {
+       int x = 1, y = 3, z = 4;
+       Lab1_exercise8_3:
+       cprintf("x %d, y %x, z %d\n", x, y, z);
+   }
    // Test the stack backtrace function (lab 1 only)
    test_backtrace(5);
```

```

entering test_backtrace 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.
x 1, y 3, z 4
He110 WorldK>

```

图 8.2

回答这个问题：

fmt 指向格式字符串，ap 指向 x 的地址。

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

这个问题我不是很理解，但是可以看看以下两个链接的内容：

<https://github.com/clpsz/mit-jos-2014/tree/master/Lab1/Exercise08>

[https://blog.csdn.net/weixin\\_43908091/article/details/108934336](https://blog.csdn.net/weixin_43908091/article/details/108934336)

Run the following code.

```

unsigned int i = 0x00646c72;
printf("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.

结果实际上就是图 8.2 的结果，He110 World

57616 的 16 进制表示是 e110，而 ascii 表中 0x72,0x6c,0x64 分别代表 r,l,d 三个字母。因此 57616 用%x 打印出来就是 e110，可以看作是 ello，而 0x00646c72 在内存中的布局是 0x7c 0x6c 0x64 0x00，可以看作是字符串"rld0"，所以面这行代码恰好可以打印出 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 i to in order to yield the same output? Would you need to change 57616 to a different value?

57176 自然是不用的，通过上面的分析我们知道受到影响的只有 0x646c72，所以我们改成 0x726c6400 即可

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

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

Why does this happen?

```

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

```

这个采用同样的方法，测试结果得到：

```

2073 //leaving test_backtrace 4
16 //leaving test_backtrace 5
17 //Welcome to the JOS kernel monitor!
18 //Type 'help' for a list of commands.
19 //x=3 y=-267321448K> _
20
21 //if (buf != NULL)

```

图 8.3

因为这个函数没有边界检查，默认是 3 这个数对应的地址后面一个位置的数。若 3 对应的地址为 x，那么第二个输出指向的地址就是 x+4。这是一个未定义行为，将输出 x+4 开始的四个字节的内容。

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?

这样的话把参数倒过来定义，需要把格式字符串放在最后，参数列表也需要反过来写。

这个我也不太懂，看链接吧：<https://github.com/clps/2014-mit-jos-lab1-exercise08>

## Exercise 9

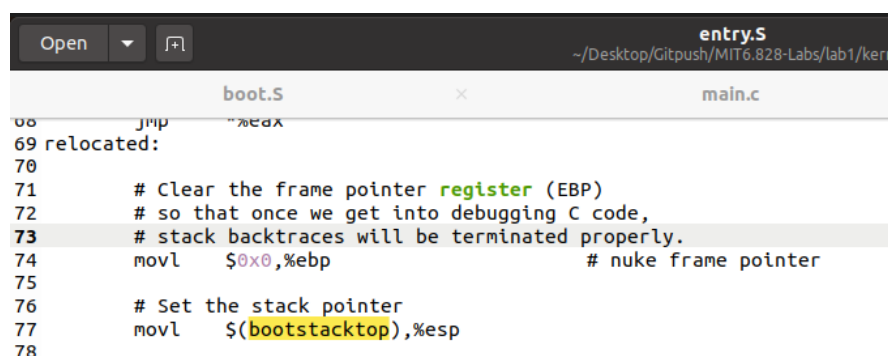
In the final exercise of this lab, we will explore in more detail the way the C language uses the stack on the x86, and in the process write a useful new kernel monitor function that prints a *backtrace* of the stack: a list of the saved Instruction Pointer (IP) values from the nested call instructions that led to the current point of execution.

所以问题来了：

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?

答：

这个在 `entry.s` 里：



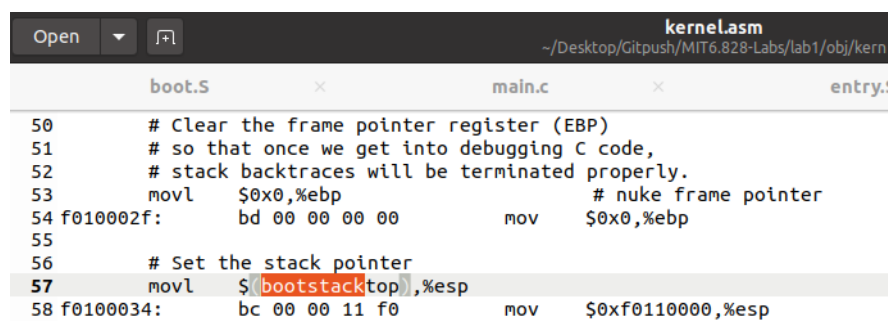
```
entry.s
~/Desktop/Gitpush/MIT6.828-Labs/lab1/kern

boot.s  x  main.c

68      jmp     *%eax
69 relocated:
70
71      # Clear the frame pointer register (EBP)
72      # so that once we get into debugging C code,
73      # stack backtraces will be terminated properly.
74      movl    $0x0,%ebp          # nuke frame pointer
75
76      # Set the stack pointer
77      movl    $(bootstacktop),%esp
78
```

图 9.1

然后再 `kernel.asm` 里发现地址即为 0xf0110000



```
kernel.asm
~/Desktop/Gitpush/MIT6.828-Labs/lab1/obj/kern

boot.s  x  main.c  x  entry.s

50      # Clear the frame pointer register (EBP)
51      # so that once we get into debugging C code,
52      # stack backtraces will be terminated properly.
53      movl    $0x0,%ebp          # nuke frame pointer
54 f010002f:  bd 00 00 00 00          mov     $0x0,%ebp
55
56      # Set the stack pointer
57      movl    $(bootstacktop),%esp
58 f0100034:  bc 00 00 11 f0          mov     $0xf0110000,%esp
--
```

图 9.2

再来看看保留多大的空间吧：

```

86 .data
87 #####
88 # boot stack
89 #####
90     .p2align      PGSHIFT          # force page alignment
91     .globl        bootstack
92 bootstack:
93     .space        KSTACKSIZE
94     .globl        bootstacktop
95 bootstacktop:
96

```

图 9.3

同样是在 entry.s 的, KSTACKSIZE 大小, 然后, 在 inc/memlayout 里得到清晰的答案: KSTACKSIZE 大小是 8\*PGSIZE, 如图 9.4:

```

93 #define EXPHYSMEM    0x100000
94
95 // Kernel stack.
96 #define KSTACKTOP    KERNBASE
97 #define KSTACKSIZE   (8*PGSIZE)           // size of a kernel stack
98 #define KSTKGAP      (8*PGSIZE)           // size of a kernel stack guard
99
100 // Memory-mapped IO.
101 #define MMIOLIM      (KSTACKTOP - PTSIZE)
102 #define MMIIOBASE    (MMIOLIM - PTSIZE)
103
104 #define ULIM          (MMIOBASE)
105

```

图 9.4

可以看出, 栈的设置方法是在数据段中预留出一些空间来用作栈空间。memlayout.h 97 行定义的栈的大小:

```

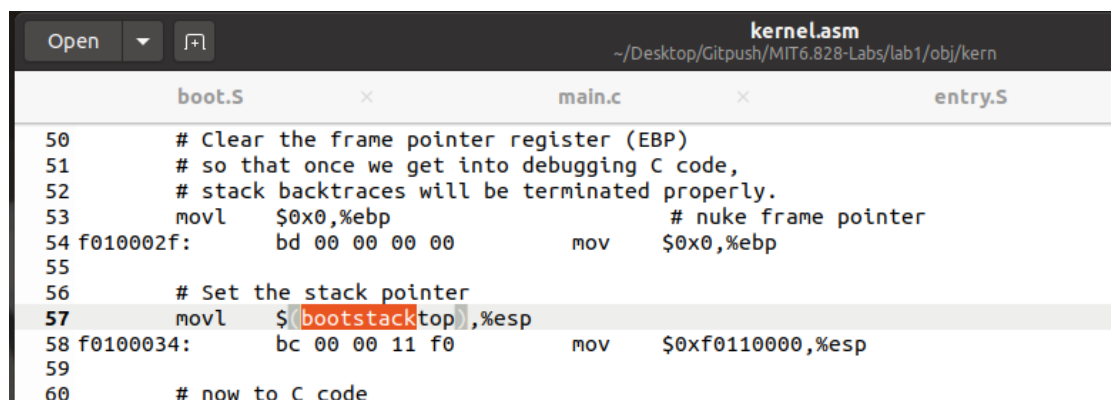
#define PGSIZE    4096    // bytes mapped by a page
#define KSTACKSIZE (8*PGSIZE)    // size of a kernel stack

```

因此栈大小为 32KB, 栈的位置为 0xf0108000-0xf0110000

堆栈指针又是指向这块被保留的区域的哪一端的呢?

堆栈由于是向下生长的, 所以堆栈指针自然要指向最高地址了。最高地址就是我们之前看到的 bootstacktop 的值。所以将会把这个值赋给堆栈指针寄存器。



```

kernel.asm
~/Desktop/Gitpush/MIT6.828-Labs/lab1/obj/kern

boot.S    ×    main.c    ×    entry.S
50      # Clear the frame pointer register (EBP)
51      # so that once we get into debugging C code,
52      # stack backtraces will be terminated properly.
53      movl    $0x0,%ebp          # nuke frame pointer
54 f010002f:    bd 00 00 00 00      mov     $0x0,%ebp
55
56      # Set the stack pointer
57      movl    $(bootstacktop),%esp
58 f0100034:    bc 00 00 11 f0      mov     $0xf0110000,%esp
59
60      # now to C code

```

图 9.5

或者我们在图 9.5 其实也能看出来的

## Exercise 10

The x86 stack pointer (esp register) points to the lowest location on the stack that is currently in use. Everything *below* that location in the region reserved for the stack is free. Pushing a value onto the stack involves decreasing the stack pointer and then writing the value to the place the stack pointer

points to. Popping a value from the stack involves reading the value the stack pointer points to and then increasing the stack pointer. In 32-bit mode, the stack can only hold 32-bit values, and esp is always divisible by four. Various x86 instructions, such as call, are "hard-wired" to use the stack pointer register.

The ebp (base pointer) register, in contrast, is associated with the stack primarily by software convention. On entry to a C function, the function's *prologue* code normally saves the previous function's base pointer by pushing it onto the stack, and then copies the current esp value into ebp for the duration of the function. If all the functions in a program obey this convention, then at any given point during the program's execution, it is possible to trace back through the stack by following the chain of saved ebp pointers and determining exactly what nested sequence of function calls caused this particular point in the program to be reached. This capability can be particularly useful, for example, when a particular function causes an assert failure or panic because bad arguments were passed to it, but you aren't sure *who* passed the bad arguments. A stack backtrace lets you find the offending function.

x86 堆栈指针(esp 寄存器)指向当前正在使用的堆栈上的最低位置。在为堆栈保留的区域中, 该位置以下的所有内容都是空闲的。将值压入堆栈涉及到减少堆栈指针, 然后将该值写入堆栈指针所指向的位置。从堆栈取出一个值涉及读取堆栈指针所指向的值, 然后增加堆栈指针。在 32 位模式下, 堆栈只能容纳 32 位的值, esp 总是能被 4 整除。各种 x86 指令, 比如调用, 对我们来说是“硬连接的”

相反, ebp(基指针)寄存器主要根据软件约定与堆栈相关联。在进入 C 函数时, 函数的序言代码通常通过将前一个函数的基指针压入堆栈来保存它, 然后在函数运行期间将当前的 esp 值复制到 ebp 中。如果所有的功能在程序遵守本公约, 在任何给定的点在程序的执行期间, 可以追溯链后通过堆栈保存 ebp 指针和决定什么嵌套的函数调用序列使这个程序中的特定点。此功能可能特别有用, 例如, 当某个特定函数由于传入了错误的参数而导致断言失败或恐慌时, 但您不确定是谁传递了错误的参数。堆栈回溯让您找到出错的函数。

那么问题来了:

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.

这个问的是, 先找到 test\_backtrace 的地址(在 kernel.asm 中发现是 0xf0100040), 查看在 kernel 启动后每次这个函数被 call 到时会发生什么, 每次 test\_backtrace 每一次递归嵌套向栈里压了多少个 32 位的 word 以及这些 word 都是什么?

答:

首先要知道:

查看寄存器

```
(gdb) i r
```

```
(gdb) i r a          # 查看所有寄存器 (包括浮点、多媒体)
```

```
(gdb) i r esp
```

```
(gdb) i r pc
```

```
U=> 0xf0100040 <test_backtrace>: endbr32
Breakpoint 1, test_backtrace (x=5) at kern/init.c:13
13 {
(gdb) i r
eax            0x0            0
ecx            0x3d4          980
edx            0x3d5          981
ebx            0xf0111308      -267316472
esp            0xf010ffdc      0xf010ffdc
ebp            0xf010fff8      0xf010fff8
esi            0x10094         65684
edi            0x0            0
eip            0xf0100040      0xf0100040 <test_backtrace>
eflags         0x46           [ IOPL=0 ZF PF ]
cs             0x8            8
ss             0x10           16
ds             0x10           16
es             0x10           16
fs             0x10           16
gs             0x10           16
```

图 10.1

所以我们进行 `i r` 查看寄存器，`esp` 指向的地址 `0xf010ffdc`，  
一个栈帧(stack frame)的大小计算如下：

1. 在执行 `call test_backtrace` 时有一个副作用就是压入这条指令下一条指令的地址，压入 4 字节返回地址
2. `push %ebp`，将上一个栈帧的地址压入，增加 4 字节
3. `push %ebx`，保存 `ebx` 寄存器的值，增加 4 字节
4. `sub $0x14,%esp`，开辟 20 字节的栈空间，后面的函数调用传参直接操作这个栈空间中的数，而不是用 `push` 的方式压入栈中

加起来一共是 32 字节，也就是 8 个 `int`。可知压入了 8 个 32 位的 `word`。

```
0x0, 0x0, 0x0, 0x0, 0x1f80, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x1f80}, v2_int64 = {0
mxcsr          0x1f80          [ IM DM ZM OM UM PM ]
(gdb) x/8x 0xf010ffdc
0xf010ffdc:  0xf01000fc  0x00000005  0x00001aac  0x00000640
0xf010ffec:  0x00000000  0x00000000  0x00010094  0x00000000
(gdb) █
```

图 10.2

再这样查看，但不知道具体含义是什么。

对于 Exercise10，github 的解答更为好理解，但 csdn 的得到了预期的结果，cnblog 的虽然没告诉怎么操作但结果应该是最清晰最易懂的：

<https://github.com/clpsz/mit-jos-2014/tree/master/Lab1/Exercise10>

[https://blog.csdn.net/weixin\\_43908091/article/details/108934336](https://blog.csdn.net/weixin_43908091/article/details/108934336)

<https://www.cnblogs.com/fatsheep9146/p/5079930.html>

## 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. 这个就是要我们自己实现以下 `mon_backtrace` 函数, 这个在实验页面里是有提示的:

The above exercise should give you the information you need to implement a stack backtrace function, which you should call `mon_backtrace()`. A prototype for this function is already waiting for you in `kern/monitor.c`. You can do it entirely in C, but you may find the `read_ebp()` function in `inc/x86.h` useful. You'll also have to hook this new function into the kernel monitor's command list so that it can be invoked interactively by the user.

这个告诉我们几个需要看的文件和函数

The backtrace function should display a listing of function call frames in the following format:

Stack backtrace:

```
ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061
...
```

这个给了我们输出的例子, 可以看到  $eip = ebp + 0x4$

Each line contains an `ebp`, `eip`, and `args`. The `ebp` value indicates the base pointer into the stack used by that function: i.e., the position of the stack pointer just after the function was entered and the function prologue code set up the base pointer. The listed `eip` value is the function's *return instruction pointer*: the instruction address to which control will return when the function returns. The return instruction pointer typically points to the instruction after the call instruction (why?).

从上面两句(The listed...call instruction)知道写 `p = (uint32_t *) ebp;`

Finally, the five hex values listed after `args` are the first five arguments to the function in question, which would have been pushed on the stack just before the function was called. If the function was called with fewer than five arguments, of course, then not all five of these values will be useful. (Why can't the backtrace code detect how many arguments there actually are? How could this limitation be fixed?)

The first line printed reflects the *currently executing* function, namely `mon_backtrace` itself, the second line reflects the function that called `mon_backtrace`, the third line reflects the function that called that one, and so on.

这里告诉我们, `ebp=p[0]`, 因为时要知道返回 call 自己的那个, 也就是 `ebp=(uint32_t*)ebp`. You should print *all* the outstanding stack frames. By studying `kern/entry.S` you'll find that there is an easy way to tell when to stop.

这里告诉我们一个: 什么时候终止呢? 我们看了 `kernel.s` 之后发现这两行:

```
# stack backtraces will be terminated properly.
movl    $0x0,%ebp          # nuke frame pointer
```

也就是 `ebp = 0` 时候终止

Here are a few specific points you read about in K&R Chapter 5 that are worth remembering for the following exercise and for future labs.

- If `int *p = (int*)100`, then `(int)p + 1` and `(int)(p + 1)` are different numbers: the first is 101 but the second is 104. When adding an integer to a pointer, as in the second case, the integer is implicitly multiplied by the size of the object the pointer points to.
- `p[i]` is defined to be the same as `*(p+i)`, referring to the *i*'th object in the memory pointed to by `p`. The above rule for addition helps this definition work when the objects are larger



than one byte.

- `&p[i]` is the same as `(p+i)`, yielding the address of the *i*'th object in the memory pointed to by *p*.

代码是这样的，实际是 github 的代码：

```
@@ -58,8 +58,17 @@ mon_kerninfo(int argc, char **argv, struct Trapframe
*tf)
    int
    mon_backtrace(int argc, char **argv, struct Trapframe *tf)
    {
-        // Your code here.
-        return 0;
+        uint32_t ebp, *p;
+
+        ebp = read_ebp();
+        while (ebp != 0)
+        {
+            p = (uint32_t *) ebp;
+            cprintf("ebp %x eip %x args %08x %08x %08x %08x %08x\n", ebp, p[1],
p[2], p[3], p[4], p[5], p[6]);
+            ebp = p[0];
+        }
+
+        return 0;
    }
```

## Exercise 12

At this point, your backtrace function should give you the addresses of the function callers on the stack that lead to `mon_backtrace()` being executed. However, in practice you often want to know the function names corresponding to those addresses. For instance, you may want to know which functions could contain a bug that's causing your kernel to crash.

To help you implement this functionality, we have provided the function `debuginfo_eip()`, which looks up `eip` in the symbol table and returns the debugging information for that address. This function is defined in `kern/kdebug.c`.

就是引入了一种情况，我们想知道与地址有关的函数名，怎么办呢？`debuginfo_eip()`可能会帮到我们，然后引出了 Exercise 12 的问题：

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

修改你的 backtrace 函数，对每个 `eip`，输出 functionname, filename, line number

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 `objdump -h obj/kern/kernel`
- run `objdump -G obj/kern/kernel`

- `run 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
- 总之就是提示我们看 `kernel.ld`, 找到 `Stab` 结构

`struct Stab`

`Symnum` 是符号索引, 整个符号表看作一个数组, `Symnum` 是当前符号在数组中的下标  
`n_type` 是符号类型, `FUN` 指函数名, `SLINE` 指在 `text` 段中的行号  
`n_othr` 目前没被使用, 其值固定为 0  
`n_desc` 表示在文件中的行号  
`n_value` 表示地址。特别要注意的是, 这里只有 `FUN` 类型的符号的地址是绝对地址, `SLINE` 符号的地址是偏移量, 其实际地址为函数入口地址加上偏移量。比如第 3 行的含义是地址 `f01000b8(=0xf01000a6+0x00000012)` 对应文件第 34 行。

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).

首先是找到 `kdebug.c` 里面的 `stab_binsearch` 把它填好, 模仿它上面的 `lfun` 那个以及根据 `hint` 来写, 如图 12.1 所示:

```

// 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 = stabs[lline].n_desc;
}
else{
    cprintf("line not found\n");
}
}

```

图 12.1

接下来补全 mon\_backtrace:

```

int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    uint32_t ebp, *p, eip;
    struct Eipdebuginfo info;
    ebp = read_ebp();
    cprintf("Stack backtrace:\n");
    while (ebp != 0)
    {
        p = (uint32_t *) ebp;
        eip = p[1];
        cprintf("ebp %x eip %x args %08x %08x %08x %08x\n", ebp, eip, p[2], p[3], p[4], p[5], p[6]);
        if (debuginfo_eip(eip, &info) == 0)
        {
            int fn_offset = eip - info.eip_fn_addr;
            cprintf("%s:%d: %.s+%d\n", info.eip_file, info.eip_line, info.eip_fn_namelen,
                    info.eip_fn_name, fn_offset);
        }
        ebp = p[0];
    }
    return 0;
}

```

图 12.2

这里那个%.s 就是上面讲解的东西，这里感觉如果完全靠自己想来写还是很难的...

这部分留个问号吧，比如 info 是什么我不知道，cprintf 那里倒是大致理解。

再修改这个函数上面不远的 command 的内容：

```

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

```

评分一下吧，最后的最后：

```

+ cc kern/monitor.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
ld: warning: section '.bss' type changed to PROGBITS
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 412 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/sakana/Desktop/Gitpush/MIT6.828-Labs/lab1'
running JOS: (1.3s)
printf: OK
backtrace count: OK
backtrace arguments: OK
backtrace symbols: OK
backtrace lines: OK
Score: 50/50
sakana@sakana:~/Desktop/Gitpush/MIT6.828-Labs/lab1$
} else {

```

## Reference

<https://pdos.csail.mit.edu/6.828/2018/labs/lab1/>

<https://github.com/clpsz/mit-jos-2014/tree/master/Lab1>

[https://blog.csdn.net/weixin\\_43908091/article/details/108934336](https://blog.csdn.net/weixin_43908091/article/details/108934336)

<https://blog.csdn.net/a747979985/article/details/94334901>