第一章 Booting a PC

1.1 PC Bootstarp

这一部分主要介绍了 x86 语言以及 PC 的启动过程，并让我们熟悉了 QEMU 和 GDB 的调试方法。

1.1.1 Getting Started with x86 assembly

> Exercise 1
> 学习 x86 的汇编语法
>
> 可以阅读 PC Assembly Language Book 来学习 x86 的语法。注意这本书值使用的 NASM 汇编，但是实验中使用的是 GNU 的 AT&T 语法。

所以更建议阅读 Brennan 的内联汇编指南中的"语法"部分。它对我们将在 JOS 中与 GNU 汇编器一起使用的 AT&T 汇编语法进行了很好的(而且相当简短)描述。

1.1.2 Simulating the x86

首先安装 qemu 虚拟机以及计算所使用的工具链，以及调试工具 GDB。此次实验使用的为 Arch Linux，且在实验过程中发现 gcc7.3.0 不能够正确的对 jos 源码进行编译，因此所使用的的 gcc 为从源码编译的 i386-jos-elf-gcc。QEMU 则是使用的 Arch Linux 官方源中的 qemu。在安装完成后，进入目录并使用 make 进行编译。编译完成后使用 make qemu 命令启动 qemu。启动后的 QEMU 如图 1.1 所示。可以发 jos 内核已经正常启动，并且能够执行 help 命令。执行 help 命令以后显示最初始的 jOS 由两个命令：help 以及 kerninfo。输入 kerninfo 命令以后命令行中打印出了部分有关内核的信息。

```
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

图 1.1: 编译成功并启动的 qemu

1.1.3 The PC's Physical Address Space

x86 计算机的内存布局如图 1.2 所示

图 1.2: 内存布局

在早期的 16 位 8088 处理器只能够操作 1MB 物理内存，因此物理地址空间为 0x00000000 到 0x000FFFFF。其中 0~640KB 为 Low Memory，只能被 RAM 所使用。而从 0x000A0000 到 0x000FFFFF 的 384KB 内存保留做特殊用途，包括显示缓存以及其他固件。从 0x000F0000 到 0x000FFFFF 这 64KB 区域为 BIOS。现代 x86 处理器支持 4GB 的 RAM，因此 RAM 扩展到了 0xFFFFFFFF，但 jos 只使用开始的 256MB。

1.1.4 The ROM BIOS

在一个终端下执行 make qemu-gdb 指令，然后新建一个 terminal，执行 make gdb 指令，结果如图 1.3 所示（此处使用的是 cgdb，便于调试，此后的调试也将使用 cgdb 而不是 gdb）。可以看到，gdb 已经开始调试并停止在 0xffff0 处。而这条指令就是 PC 启动后 BIOS 执行的第一条指令。

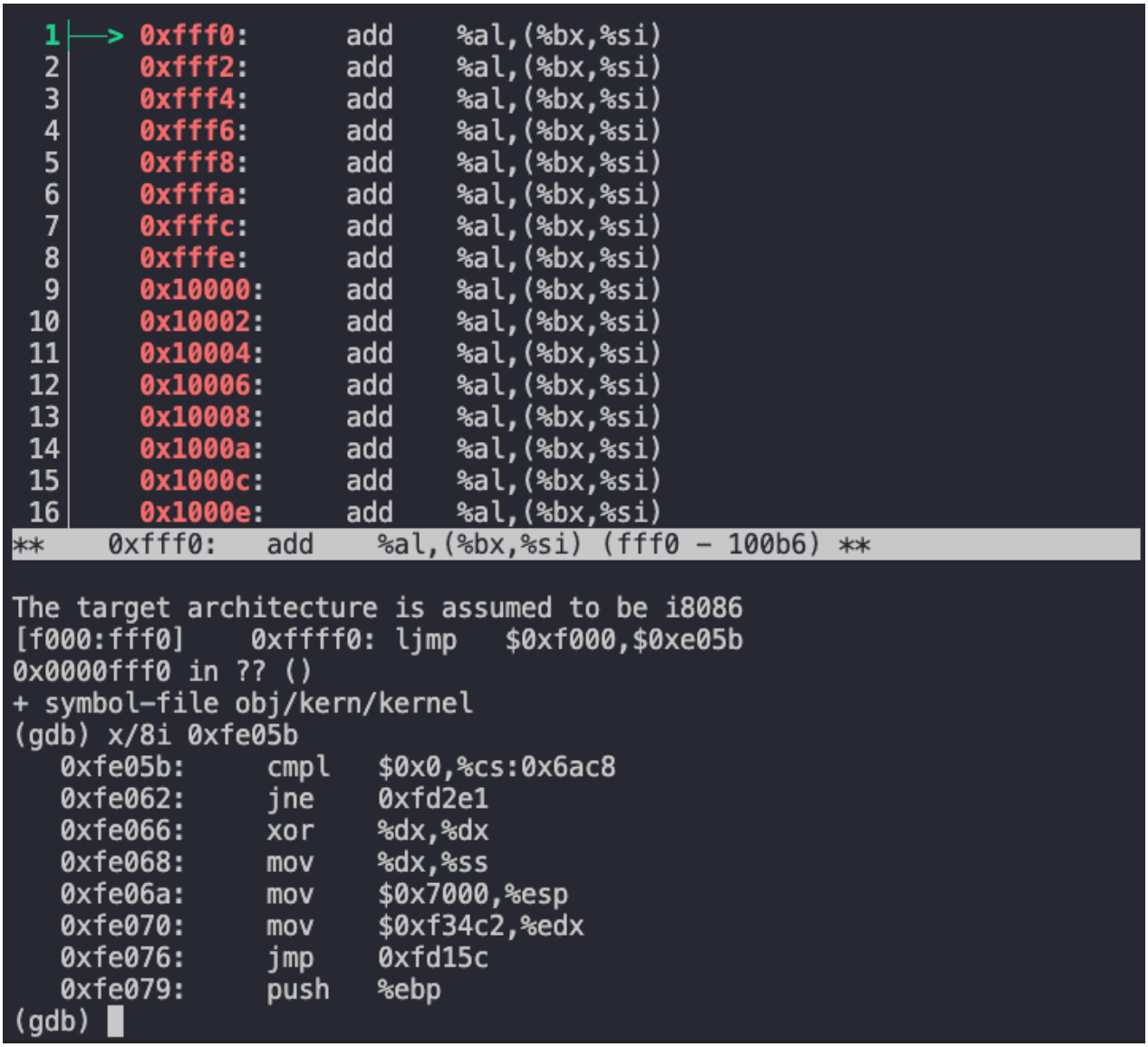

图 1.3: 初次调试界面

第一条指令表明：

- IBM PC 执行的起始物理地址为 0x000ffff0 (BIOS区域)，之后会把控制权交给硬盘第一扇区 0x7c00, 第一扇区运行到 0x7d6b 后，si 跳转到 0x10000c, 之后通过 cr0 和 cr3 开启分页后将虚拟地址 0xf0100000 映射到物理地址 0x00100000，总共 4MB 大小，内核就装载到 0xf0100000 这一位置
- PC 的偏移方式为 CS = 0xf000，IP = 0xfff0
- 第一条指令执行的是 jmp 指令，跳转到段地址 CS = 0xf000，IP = 0xe05b

> Exercise 2
>
> 提问： 使用 GDB 的 si 命令单步调试进入 ROM BIOS，然后猜测这些指令的作用。

解答： 在使用 make qemu-gdb 命令编译内核并启动 qemu，然后使用 make gdb 命令启动 gdb 后，使用 si 单步调试。连续执行 si 命令多步后，得到的前 30 条指令如下

```
1 1 0xffff0: ljmp $0xf000, $0xe05b
```

```
2 0xfe05b: cmpl $0x0, $cs:0x6ac8
3 0xfe062: jne 0xfd2e1
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
10 0xfd15f: cli
11 0xfd160: cld
12 0xfd161: mov $0x8f, %eax
13 0xfd167: out %al, $0x70
14 0xfd169: in $0x71, %al
15 0xfd16b: in $0x92, %al
16 0xfd16d: or $0x2, %al
17 0xfd16f: out %al, $0x92
18 0xfd171: lidtw %cs:0x6ab8
19 0xfd177: lgdtw %cs:0x6a74
20 0xfd17d: mov %cr0, %eax
21 0xfd180: or $0x1, %eax
22 0xfd184: mov %eax, %cr0
23 0xfd187: ljmpl $0x8, $0xfd18f
24 0xfd18f: mov $0x10, %eax
25 0xfd194: mov %eax, %ds
26 0xfd196: mov %eax, %es
27 0xfd198: mov %eax, %ss
28 0xfd19a: mov %eax, %fs
29 0xfd19c: mov %eax, %gs
30 0xfd19e: mov %ecx, %eax
```

- 第 1 条指令是一条跳转指令，跳转到 0xfe05b 处。
- 第 2 条和第 3 条指令共同构成了判断跳转，如果 $cs:0x6ac8 处的值不为 0 则跳转，而根据第 4 条指令的地址可以得知此处并没有发生跳转。
- 第 4 跳指令将%dx 清零。
- 第 5~8 条指令在设置完段寄存器、段指针以及 edx 寄存器后跳转到了 0xfd15c 处。然后第 10 跳指令关闭了中断。这应该是由于 boot 过程是比较关键的，因此屏蔽大部分的中断。
- 第 11 跳指令设置了方向标志位，表示后续操作的内存变化方向。
- 第 12~14 条指令涉及到 IO 操作。CPU 与外部设备通讯需要通过 IO 端口访问，而 x86CPU 使 用 IO 端口独立编址的方式。并且规定端口操作需要通过 al 或者 ax 进行。通过查询端口对应的设备2我们知道了 0x70 和 0x71 是用于操作 CMOS 的端口。而这三条指令是用于关闭不可屏蔽中断 (Non-Maskable Interrupt) 的。
- 第 18 和 19 条指令用于将 0xf6ab8 处的数据读入到中断向量表寄存器 (IDTR) 中，并将 0xf6a74的数据读入到全局描述符表格寄存器 (GDTR) 中，而后者是实现保护模式中较为重要的一部分。
- 第 20~21 用于将 CR0 寄存器的最低位置 1，进入保护模式，然而后面又从保护模式退出，继续在实模式下运行。
- 第 23~29 步用于重新加载段寄存器，在加载完 GDTR 寄存器后需要刷新所有的段寄存器的值。

1.2 The Bootloader

对于 PC 而言软盘和硬盘都被分为 512 字节的扇区。一个扇区是磁盘传输的最小粒度，也就是说读写都必须以扇区为单位执行。如果一个磁盘是可启动的，就把这个磁盘的第一个扇区叫做启动扇区，而 boot loader 就位于这一个扇区中。当 BIOS 找到一个可以启动的磁盘以后就会把这 512 字节的扇区加载到 0x7c00 到 0x7dff 中。jOS 采用传统的硬盘启动机制，也就是说 boot loader 的大小小于 512 字节。bootloader 只包含两个文件：一个汇编文件 boot/boot.S 和一个 C 文件 boot/main.c，并且主要有如下两个功能：

- 将处理器由实模式转换为保护模式。
- 通过 x86 的特殊 I/O 访问指令将内核从硬盘读入内存。

> Exercise 3
> 在 0x7c00 处设置断点，然后让程序运行到这个断点，然后使用 gdb 跟踪 boot.S 的每一条指令，并比较 obj/boot/boot.asm 以及 boot/boot.S。
>
> 追踪进入 bootmain() 中，然后追踪进入 readsect() 中，并找出每一条 readsect() 中 c 语句对应的汇编语句，然后找出把内核从磁盘读取到内存的 for 语句对应的汇编语句。找出 loop 运行完后那些语句会被执行，并在那里设置断点。继续运行到断点然后执行完接下来的语句。

完成 Exercise 3 ，首先使用 make qemu-gdb 命令启动 qemu，然后使用 make gdb 命令启动 gdb 并在 gdb 中使用 b *0x7c00 打上断点，然后使用 c 命令执行到此断点处。

从反汇编的代码 obj/boot/boot.asm 中可以看出，0x7c00 就是 boot loader 的起始地址。也就是说，实验要求我们从 boot loader 的开始处跟踪。

首先观察并分析 boot.S 以及 main.c 中的内容，分析每一个部分的作用，以便于后续的跟踪:

```
.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
  cld                         # String operations increment

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment
```

在 boot.S 的开始部分，首先关闭了在 BIOS 运行期间可能打开的中断，防止 boot loader 在运行期间被中断。cld 用于设置字符串处理时指针的移动方向。

从 xorw 开始的 4 句代码用于将段寄存器清零：首先将寄存器%ax 置为 0，然后将%ax 赋值给段寄存器%ds, %es, %ss。由于在 BIOS 阶段这些寄存器的值可能发生改变因此此处需要对其进行复位操作。

在上述准备代码完成后，执行从实模式到保护模式的代码：

```
  # Enable A20:
  #   For backwards compatibility with the earliest PCs, physical
  #   address line 20 is tied low, so that addresses higher than
  #   1MB wrap around to zero by default.  This code undoes this.
seta20.1:
```

```
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

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

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

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

这部分代码用于将 CPU 从实模式转换为保护模式：为了保留对于早期计算机的后向兼容性，第 20 根物理地址线保持为 0，因此所有大于 1MB 的地址全部会回卷，而这段代码解除了这一限制。

seta20.1 后面的 3 句指令不停地检测 0x64 端口的第 0x2 位，若其不为 0 则重复这一检测。这三句命令用于阻塞直到输入缓冲区为空。当缓冲区准备完成后，movb 和 outb 指令向 0x64 端口写入 0xd1。经过查询，0xd1 指令的作用是将下一次写入 0x60 端口的指令会被发送给键盘控制器 804x 的输入端口。

seta20.2 后面的三句与 seta20.1 后面的三句相同，再次等待缓冲区可以切入。在等待完成后的 movb 和 outb 指令向 0x60 端口写入数据 0xdf。0xdf 表示使能 A20 线，可以进入保护模式。

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

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

使能 A20 线后，首先把 gdtdesc 标识符送入 GDTR 中。GDTR 有 48 位长，其中高 32 位表示该表在内存中的起始地址，低 16 位表示该表的长度。这一指令就是将 GDT 表的起始地址以及长度送入 GDTR 中。gdtdsec 的标识符内容在 boot.S 尾部：

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

可以看出 gdt 表的大小为 0x17，表的起始地址为标号 gdt 所在的位置。

在加载完 GDT 表的信息到 GDTR 后，使用了三个指令修改给%cr0 寄存器的内容。从文件首可以找到：

```
.set PROT_MODE_CSEG, 0x8         # kernel code segment selector
.set PROT_MODE_DSEG, 0x10        # kernel data segment selector
.set CR0_PE_ON,      0x1         # protected mode enable flag
```

也就是说，这几句将寄存器%cr0 的最低位置为 1，而%cr0 的最低位是保护模式的启动位，将 CR0 的这一位置为 1 则表示保护模式启动。

在正式启动保护模式后，执行了一个 ljmp 指令，将当前的运行模式切换为 32 位地址模式。接下来运行的就是 32 位的代码：

```
  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call bootmain
```

在 32 位的代码开头，修改了 5 个段寄存器的值。与前述 BIOS 部分一样，当加载完 GDTR 后必须重新加载段寄存器的值，而加载 CS 的值必须通过 ljmp 加载（也就是将运行模式切换为 32 位地址模式的指令）。从而使 GDTR 生效。然后设置完栈指针的值后，跳转到 bootmain 中去。

在 main.c 中，bootmain 函数如下：

```
void
bootmain(void)
{
  struct Proghdr *ph, *eph;

  // read 1st page off disk
  readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

  // is this a valid ELF?
  if (ELFHDR->e_magic != ELF_MAGIC)
    goto bad;

  // 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);

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

bad:
  outw(0x8A00, 0x8A00);
  outw(0x8A00, 0x8E00);
  while (1)
    /* do nothing */;
}
```

从代码中可以看出，首先 bootmain 调用了 readseg 函数：

```
// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
// Might copy more than asked
void
readseg(uint32_t pa, uint32_t count, uint32_t offset)
```

而从 readseg 函数定义处的注释可以看出，这个函数的作用是：将从 offset 开始的 count 个字节读到 pa 所在的位置。因此在 bootmain 中，readseg((uint32_t) ELFHDR, SECTSIZE*8, 0); 这条语句的作用是将内核的第一页 (SECT SIZE × 8 = 4096) 的内容读取到 ELFHDR 处，也就是说，把内核的 elf 头部放入内存开始部分。

然后，下一条语句验证这是否是一个合法的 elf 文件头部。如果不是则跳转到 bad 标签处。

然后，在 elf 头部找到程序的段信息。而 e_phoff 表示的是 ProgramHeaderTable 距离 elf 起始地址的偏移量。将这一偏移量加上 elf 的起始地址存入 ph 中，作为 ProgramHeaderTable 的起始地址，然后通过偏移量将 eph 作为其结束地址。

接下来的 for 循环就是加载所有的段进入内存中。p_offset 值得是这一段开头相对于 elf 文件开头的偏移量，ph_memsz 指的是这个段被装入内存后的大小，而 p_pa 指的则是装入内存的起始地址。

最后 ((void (*)(void)) (ELFHDR->e_entry))(); 语句执行 e_entry。e_entry 是执行的入口地址，也就是说，通过执行这个地址的方式把控制权从 bootloader 转移给内核。

接下来，开始使用 gdb 进行跟踪。首先设置断点，然后运行到断点处。gdb 反汇编代码如 1.4 所示，图中显示了 GDB 反汇编的起始 17 条指令

```
  1 ---> 0x7c00:      cli
  2      0x7c01:      cld
  3      0x7c02:      xor    %ax,%ax
  4      0x7c04:      mov    %ax,%ds
  5      0x7c06:      mov    %ax,%es
  6      0x7c08:      mov    %ax,%ss
  7      0x7c0a:      in     $0x64,%al
  8      0x7c0c:      test   $0x2,%al
  9      0x7c0e:      jne    0x7c0a
 10      0x7c10:      mov    $0xd1,%al
 11      0x7c12:      out    %al,$0x64
 12      0x7c14:      in     $0x64,%al
 13      0x7c16:      test   $0x2,%al
 14      0x7c18:      jne    0x7c14
 15      0x7c1a:      mov    $0xdf,%al
 16      0x7c1c:      out    %al,$0x60
**    0x7c00:   cli     (7c00 - 7cda) **
nfiguration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
---Type <return> to continue, or q <return> to quit---
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
```

图 1.4: gdb 反汇编代码

下面为 boot/boot.S 的前 17 条指令

```
1  .set PROT_MODE_CSEG, 0x8         # kernel code segment selector
2  .set PROT_MODE_DSEG, 0x10        # kernel data segment selector
3  .set CR0_PE_ON,      0x1         # protected mode enable flag
4
5  .globl start
6  start:
7    .code16                     # Assemble for 16-bit mode
8    cli                         # Disable interrupts
9    cld                         # String operations increment
10
11   # Set up the important data segment registers (DS, ES, SS).
12   xorw    %ax,%ax             # Segment number zero
13   movw    %ax,%ds             # -> Data Segment
14   movw    %ax,%es             # -> Extra Segment
15   movw    %ax,%ss             # -> Stack Segment
```

```
16
17    # Enable A20:
18    #   For backwards compatibility with the earliest PCs, physical
19    #   address line 20 is tied low, so that addresses higher than
20    #   1MB wrap around to zero by default.  This code undoes this.
21  seta20.1:
22    inb     $0x64,%al               # Wait for not busy
23    testb   $0x2,%al
24    jnz     seta20.1
25
26    movb    $0xd1,%al               # 0xd1 -> port 0x64
```

然后是 `obj/boot/boot.asm` 前 10 条指令：

```
1   .globl start
2   start:
3     .code16                     # Assemble for 16-bit mode
4     cli                         # Disable interrupts
5       7c00: fa                    cli
6     cld                         # String operations increment
7       7c01: fc                    cld
8
9     # Set up the important data segment registers (DS, ES, SS).
10    xorw    %ax,%ax             # Segment number zero
11      7c02: 31 c0                 xor    %eax,%eax
12    movw    %ax,%ds             # -> Data Segment
13      7c04: 8e d8                 mov    %eax,%ds
14    movw    %ax,%es             # -> Extra Segment
15      7c06: 8e c0                 mov    %eax,%es
16    movw    %ax,%ss             # -> Stack Segment
17      7c08: 8e d0                 mov    %eax,%ss
18
19  00007c0a <seta20.1>:
20    # Enable A20:
21    #   For backwards compatibility with the earliest PCs, physical
22    #   address line 20 is tied low, so that addresses higher than
23    #   1MB wrap around to zero by default.  This code undoes this.
24  seta20.1:
25    inb     $0x64,%al               # Wait for not busy
26      7c0a: e4 64                 in     $0x64,%al
27    testb   $0x2,%al
28      7c0c: a8 02                 test   $0x2,%al
29    jnz     seta20.1
30      7c0e: 75 fa                 jne    7c0a <seta20.1>
31
32    movb    $0xd1,%al               # 0xd1 -> port 0x64
33      7c10: b0 d1                 mov    $0xd1,%al
```

通过比较可以发现，源程序中如 movb，inb 等指令在 gdb 中实际执行的是 mov，in 指令，用于标识操

作数长度的后缀没有了，但是指令的实际功能是相同的。在 boot/boot.S 中的标识在 gdb 中变为了实际的物理地址。而在 boot.asm 中则同时包含了这两项，即转换前使用标识标示地址以及编译后使用真实地址表示地址的。此外 boot/boot.S 中的注释是不被编译的。

继续执行，一直执行到 call bootmain 处。执行完后 gdb 反汇编如1.5所示。

```
 78     0x7d0a:      pop    %edx
 79     0x7d0b:      jmp    0x7cf7
 80     0x7d0d:      lea    -0xc(%ebp),%esp
 81     0x7d10:      pop    %ebx
 82     0x7d11:      pop    %esi
 83     0x7d12:      pop    %edi
 84     0x7d13:      pop    %ebp
 85     0x7d14:      ret
 86 --> 0x7d15:      push   %ebp
 87     0x7d16:      mov    %esp,%ebp
 88     0x7d18:      push   %esi
 89     0x7d19:      push   %ebx
 90     0x7d1a:      push   $0x0
 91     0x7d1c:      push   $0x1000
 92     0x7d21:      push   $0x10000
 93     0x7d26:      call   0x7cdc
 94     0x7d2b:      add    $0xc,%esp
**    0x7c45:   call   0x7d15 (7c45 - 7d4c) **
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb)
(gdb) b *0x7c45
Breakpoint 1 at 0x7c45
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7c45:      call   0x7d15
```

图 1.5: gdb 中 bootmain 的反汇编

bootmain 中，7d15 到 7d19 处的 4 条语句是 C 语言执行函数调用的时候必须要进行的一些任务，包括修改栈帧信息以及保护寄存器%esi，%ebx 的值。在进行了这些必要的任务后。7d1a 到 7d1c 处的指令是对于参数的压栈，以将参数传递给 regadseg 函数。压栈完成后使用 call 语句调用 readseg 函数。

Excercise 3 要求追踪到 readsect 函数中，因此继续向下追踪。执行到 readseg 函数时，反汇编如图 1.6 所示。

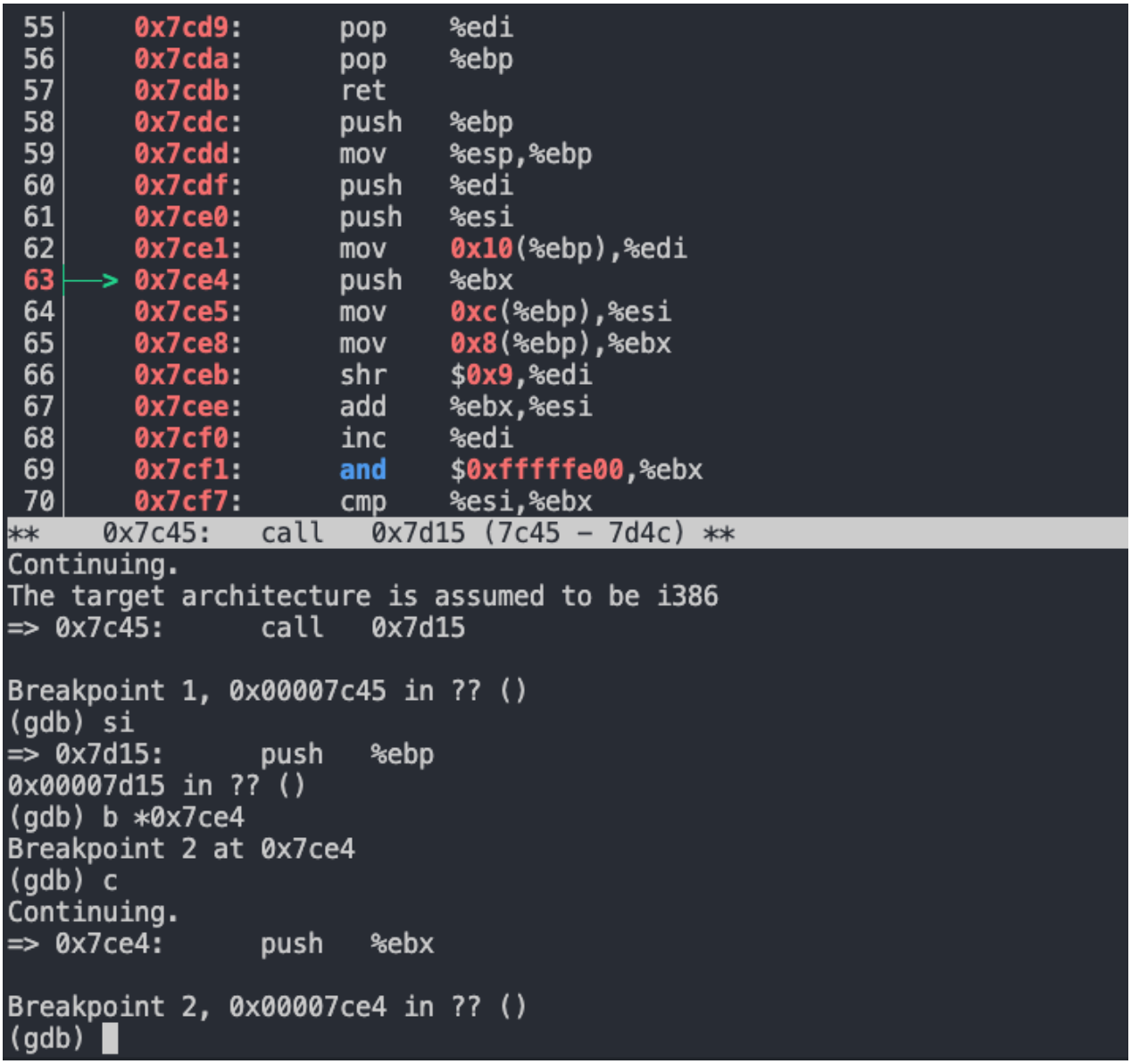

图 1.6: gdb 中 readseg 函数反汇编

在 readseg 函数中，7cdc 到 7ce1 处的程序依然是例行执行的修改栈帧、保护寄存器、初始化局部变量等。7ce5 到 7cf0 处的汇编语句用于计算 end_pa、pa、offset，为后面读取内核做准备。从这三句汇编语言可以看出一句 C 语言语句可能对应不止一句汇编语句。

在计算完起始地址，结束地址，偏移量后，进入 while 循环。在进入循环的第一条指令为 jmp 7cf7，进行了一个绝对跳转。实际上，C 语言实现 while 循环一般是先进行跳转，将判断的语句置于循环体之后，也就是将 while 实现为 do-while 的形式。进行判断的语句为：

```
7cf7: 39 f3                 cmp    %esi,%ebx
7cf9: 73 12                 jae    7d0d <readseg+0x31>
```

在循环体中，一共执行了 3 条 C 语言语句：

```
readsect((uint8_t*) pa, offset);
pa += SECTSIZE;
offset++;
```

从这三条语句中可以看出，readseg 是通过调用 readsect 来逐个扇区将数据读入内存从而达到加载整个段的效果。下面使用 gdb 继续追踪进入 readsect 中。此时 gdb 的反汇编如1.7所示。

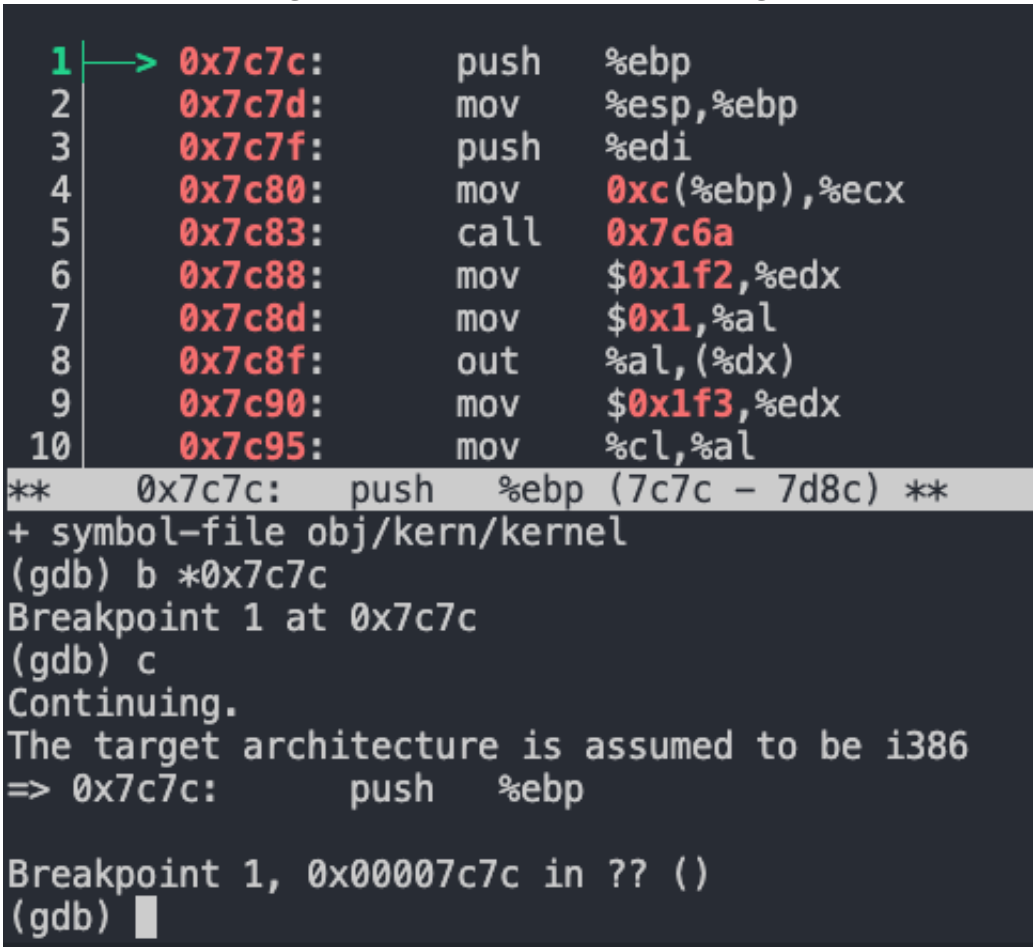

图 1.7: readsect 在 gdb 中的反汇编

从反汇编中可以看出，readsect 函数体执行的第一个语句为 waitdisk。而根据注释可以知道这一语句用于等待磁盘空闲。在磁盘空闲后，执行了一系列 outb 语句，这一系列 outb 语句用于向端口写值。第一个 outb 语句对应的汇编为：

```
7c88: ba f2 01 00 00        mov    $0x1f2,%edx
7c8d: b0 01                 mov    $0x1,%al
7c8f: ee                    out    %al,(%dx)
```

可以看出，C 语言中 outb 的实现就是使用汇编的 out 语句实现的端口 IO。从所有 outb 语句对应的 7c88 到 7cb8 之间的汇编语句可以看出：outb 语句中第一个参数是端口号，第二个参数是送入端口的值。然后经过查询得知，这一函数首先向端口 0x1f2 送入值 1，这表示取出一个扇区；然后将要取出的扇区的 32 位编号分为 4 段，分别送入 0x1f3 到 0x1f6，最后向 0x1f7 端口送入 0x20 指令，表示要读取这个扇区。

在上述的一系列动作完成后，调用 waitdisk() 阻塞并等待磁盘读取完成。读取完成后继续执行下一条语句，即 insl 语句。insl 函数对应的汇编指令如下。

```
7cc9: 8b 7d 08              mov    0x8(%ebp),%edi
7ccc: b9 80 00 00 00        mov    $0x80,%ecx
7cd1: ba f0 01 00 00        mov    $0x1f0,%edx
7cd6: fc                    cld
7cd7: f2 6d                 repnz insl (%dx),%es:(%edi)
```

可以看出，insl 函数有 3 个参数，而在第一条指令中，将 insl 的第一个参数送入 edi。第一个参数是 dst，也就是目的地址。0x7ccc 到 0x7cc1 处的两条指令指令将寄存器 %ecx 和 %edx 分别赋值为 0x80 和 0x1f0。在执行 cld，即清除方向标识指令后，执行了一个 repnz 指令，repnz 指令重复执行紧随其后的指令直到寄存器 %ecx 的值为 0 且 ZF 标志位为 1。在这里，被重复的指令是 insl (%dx), %es:(%edi)。在这个指令中，%dx 为端口号，此处为 0x1f0，%edi 为起始地址 pa，此时为 0x10000。单步执行这一指令一次，前后 0x10000 处的数据如1.8所示。

```
 29      0x7cc9:      mov    0x8(%ebp),%edi
 30      0x7ccc:      mov    $0x80,%ecx
 31      0x7cd1:      mov    $0x1f0,%edx
 32      0x7cd6:      cld
 33  --> 0x7cd7:      repnz insl (%dx),%es:(%edi)
 34      0x7cd9:      pop    %edi
 35      0x7cda:      pop    %ebp
 36      0x7cdb:      ret
 37      0x7cdc:      push   %ebp
**    0x7c7c:    push   %ebp (7c7c - 7d8c) **
(gdb) c
Continuing.
=> 0x7cd6:      cld

Breakpoint 2, 0x00007cd6 in ?? ()
(gdb) print/x $edi
$1 = 0x10000
(gdb) x/8bx 0x10000
0x10000:        0x00    0x00    0x00    0x00    0x00    0x00    0x00    0
x00
(gdb) si
=> 0x7cd7:      repnz insl (%dx),%es:(%edi)
0x00007cd7 in ?? ()
(gdb) x/8bx 0x10000
0x10000:        0x00    0x00    0x00    0x00    0x00    0x00    0x00    0
x00
(gdb) si
=> 0x7cd7:      repnz insl (%dx),%es:(%edi)
0x00007cd7 in ?? ()
(gdb) x/8bx 0x10000
0x10000:        0x7f    0x45    0x4c    0x46    0x00    0x00    0x00    0
x00
(gdb)
```

图 1.8: 执行 repnz 指令后 0x10000 处的数据

可以看到，一条 insl 指令读取 4 字节的数据。因此读取一个扇区（512）字节的读取需要重复 128次 insl 指令，执行到下一条指令时 %edi 的值也验证了这一点。

当读取完成后，执行如下三条指令，其作用是还原之前保护的寄存器的值并返回到调用者处。

```
7cd9: 5f                    pop    %edi
7cda: 5d                    pop    %ebp
7cdb: c3                    ret
```

回到 readseg 函数后，执行调用 readsect 完成后接下来的指令：

```
pa &= ~(SECTSIZE - 1);
  7cf1: 81 e3 00 fe ff ff     and    $0xfffffe00,%ebx
offset = (offset / SECTSIZE) + 1;

// If this is too slow, we could read lots of sectors at a time.
// We'd write more to memory than asked, but it doesn't matter --
// we load in increasing order.
while (pa < end_pa) {
  7cf7: 39 f3                 cmp    %esi,%ebx
  7cf9: 73 12                 jae    7d0d <readseg+0x31>
```

这些代码的作用在分析部分已经提到，其中 0x7cf6 和 0x7cfc 处的代码就是 pa+=SECTSIZE 以及 offset++ 两条 C 语言代码的直接对应，紧随其后的两条 pop 则是释放先前传入 readsect 的参数。最后的 cmp 以及 jb 则实现了 while 循环。

在 while 循环结束后的第一条语句打上断点，此处 (0x7d03~0x7d0a) 处的 5 条语句就是例行的恢复保护的寄存器以及返回到调用者处。之后，追踪返回到 bootmain 中。通过 boot.asm 可以判断出，在判断完 ELF 头部是否合法后，接下来的 for 语句的起始和结束位置分别为 0x7d5b 以及 0x7d69

```
for (; ph < eph; ph++)
  7d5b: 83 c3 20              add    $0x20,%ebx
  // 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);
  7d5e: ff 73 ec              pushl  -0x14(%ebx)
  7d61: e8 76 ff ff ff        call   7cdc <readseg>
  goto bad;

// 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++)
  7d66: 83 c4 0c              add    $0xc,%esp
  7d69: eb e6                 jmp    7d51 <bootmain+0x3c>
```

0x7d5e~0x7d66 执行的就是 for 循环的循环体，也仅对应一条 C 语言语句，即 readseg(ph->p_pa,ph->p_memsz, ph->p_offset); 前面已经分析过 readseg 函数的执行过程，而这一语句，就是 elfheader 中所指向的信息加载到 ph->p_pa 处。

在循环执行完成后，只执行了一条语句：

```
((void (*)(void)) (ELFHDR->e_entry))();
7d6b: ff 15 18 00 01 00     call   *0x10018
```

从 C 语言的语法分析，就是将 ELFHDR->e_entry 强制转化为 void (*)(void) 类型的函数指针然后执行这一函数指针，也就是说，将 CPU 的控制权从 bootloader 转移给内核。在 0x7d6b 处打上断点，进入内核后执行的第一条语句如1.9所示。也就是 0x10000c 处的 movw $1234, 0x472。

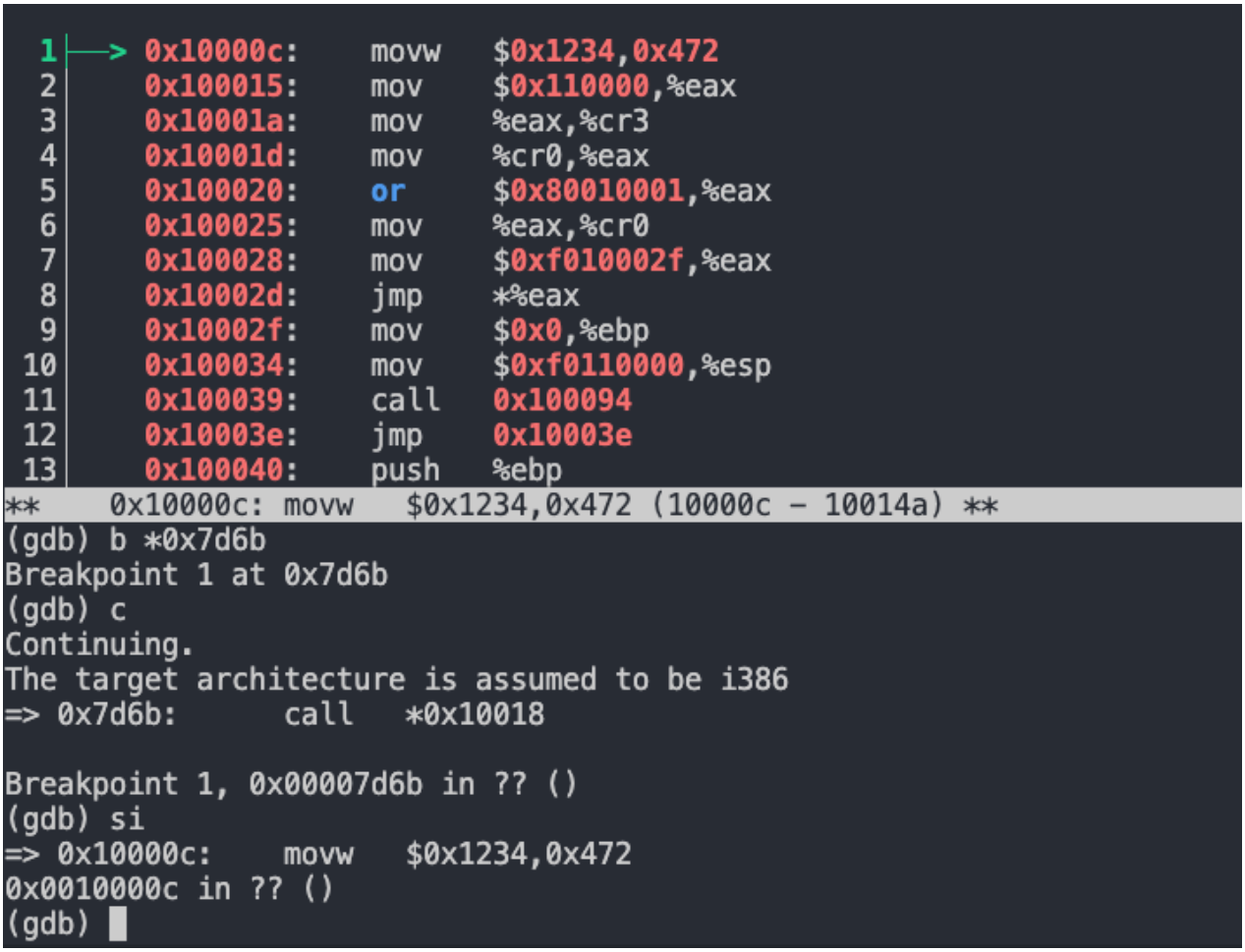

图 1.9: 进入内核后执行的第一条语句

至此，对于整个 bootloader 的分析与追踪已经完成。对于实验指导中提出的 4 个问题，其答案在 Exercise 3 的分

析与追踪的过程中已经给出：

1. 在什么时候处理器开始工作于 32 位模式? 是什么让 CPU 从 16 位模式切换到 32 位模式?

- 执行完 0x7c2d 处的 ljmp（对应 boot.S 中 ljmp $PROT_MODE_CSEG, $protcseg）语句后，处理器开始工作于 32 位模式。表面上，ljmp 语句将 CPU 切换为 32 位模式，实际上是因为此时 CPU 工作于保护模式下。

2. boot loader 中执行的最后一条语句是什么? 内核加载完成后执行的第一条语句又是什么?

- boot loader 执行的最后一条语句是 0x7d6b 处的 call *0x10018，对应 main.c 中的 `((void (*)(void)) (ELFHDR->e_entry))();` 这条语句执行完后，执行的第一条指令，也就是加载完内核后执行的第一条指令是 movw $0x1234, 0x472。

3. 内核的第一条指令在哪里?

- 内核的第一条指令在 0x1000c 处，对应的源码位于 kern/entry.S 中。

4. boot loader 是如何知道他要加载多少扇区才能吧整个内核加载进入内存的? 它是从哪里找到这些信息的?

- 这些信息位于操作系统的 Program Header Table 中，这个表存放在 ELF Header 中，通过读取这些信息就知道要读取多少扇区才能把整个内核送入内存。

1.2.1 Loading the Kernel

进一步讨论 boot loader 的 C 语言部分之前先回顾一些有关 C 语言的基础知识。

> Exercise 4
>
> 阅读并学习关于 C 语言指针的知识（最好的参考书籍是 K&R）。
>
> 阅读 K&R 5.1 章到 5.5 章的内容，下载 pointer.c，然后编译运行它。确保你理解了所有打印出来的值是从哪里来的，尤其是第 1 行和第 6 行的指针以及第二行到第 4 行的值，以及为什么第 5 行的值看题来像是程序崩溃了。

由于这部分涉及 C 语言不是本课程主要学习内容，因此请读者下来自行阅读和练习。但务必要搞懂这个程序，因为学习计算机最好的方法就是实践。

为了为了能够理解 boot/main.c，首先需要知道 ELF 文件的结构。ELF 文件由 3 部分组成：带有加载信息的文件头，程序段表以及程序段。每一个段都是一块连续的代码或者数据。它们在运行时首先要被加载到内存中，而 boot loader 的任务就是要把它们加载到内存中。对于 jOS 而言，我们对 3 个段非常感兴趣：

- .text 段：存放程序的可执行代码
- .rodata 段，存放所有的只读数据，如字符串常量
- .data 段，存放所有被初始化后的数据段，比如有初值的全局变量。

通过 objdump 命令可以得到有关这些段的信息。运行 `objdump -h obj/kern/kernel` 命令后得到如下结果：

```
obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001841  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       0000075c  f0101860  00101860  00002860  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         000038c5  f0101fbc  00101fbc  00002fbc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00001901  f0105881  00105881  00006881  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000648  f0112300  00112300  00013300  2**5
```

```
                  CONTENTS, ALLOC, LOAD, DATA
  6 .comment      00000035  00000000  00000000  00013948  2**0
                  CONTENTS, READONLY
```

可以看出，其中不仅仅包含.text, .rodata 以及.data 段，还包含一些其他的段信息。在这些信息中对于每一个段都有一个 VMA 以及一个 LMA，VMA 就是这个段的逻辑地址，而 LMA 则为这个段被加载到内存后的物理地址。

通过 `objdump -x obj/kern/kernl` 命令，我们可以看到 ELF 中哪些部分将被加载到内存，以及被加载到内存中的哪个地址，其输出如下：

```
obj/kern/kernel:     file format elf32-i386
obj/kern/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 0x00007182 memsz 0x00007182 flags r-x
    LOAD off    0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
         filesz 0x0000a948 memsz 0x0000a948 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx
...
```

Program Header 中列出的就是所有被加载到内存中的段的信息，也就是 Program Headers Table 中的表项，所有而需要被加载到内存的段都被标记为 LOAD。BIOS 会把 boot sector 加载到 0x7c00 处，这是 boot sector 的加载地址，也是其链接地址。我们可以通过 boot/Makefrag 中的-Ttext 修改链接地址。

> Exercise 5
>
> 再次追踪进入 boot loader 然后尝试辨认出哪些语句会在 boot loader 的链接地址变化后 “坏掉”。在 boot/Makefrag 中修改链接地址为一个错误的地址，然后运行 make clean。用 make 重新编译整个 lab，并追踪进 boot loader 看发生了什么。最后别忘了把链接地址改回来并运行 make clean。

回答：BIOS 仍然会把 boot loader加载到 0x7c00 处，因此在 0x7c00 处打上打断点并调试。执行到断点后，可以看到，前几句程序依然是正常执行。接下来单步执行，在执行到 ljmp 指令时，本应该直接执行 ljmp 的下一条指令地址，但是这里由于链接地址的错误直接跳转错误，导致了错误，程序不能够继续运行。

对于内核而言，其加载地址和链接地址是不同的。这和 boot loader 不一样。bootloader 将内核加载在低地址处，但是内核运行在高地址处。通过 objdump -f obj/kern/kernel 可以看到 ELF 头部中的 e_entry 字段。这个字段存放的就是可执行程序入口的链接地址。这一命令输出如下：

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

> Exercise 6
>
> 首先重新编译并启动 qemu/gdb，然后在 0x7c00 处打上断点。运行到断点处，使用 x/8x 0x00100000命令检查 0x00100000 处的 8 个字。然后在引导加载程序进入内核时再次检查。 他们为什么不同？ 第二个断点有什么?

Exercise 6 告诉我们，从 boot loader 进入内核后执行的第一条指令在 0x10000c 处，因此在 0x10000c处打上断点。运行到这一断点并重新使用 x/8x 0x00100000 检查，其结果如图 1.10 所示

```
  1 ---> 0x10000c:    movw   $0x1234,0x472
  2      0x100015:    mov    $0x110000,%eax
  3      0x10001a:    mov    %eax,%cr3
  4      0x10001d:    mov    %cr0,%eax
  5      0x100020:    or     $0x80010001,%eax
  6      0x100025:    mov    %eax,%cr0
  7      0x100028:    mov    $0xf010002f,%eax
  8      0x10002d:    jmp    *%eax
  9      0x10002f:    mov    $0x0,%ebp
 10      0x100034:    mov    $0xf0110000,%esp
 11      0x100039:    call   0x100094
 12      0x10003e:    jmp    0x10003e
**    0x10000c: movw   $0x1234,0x472 (10000c - 10014a) **
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 1, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c
766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200
fd8
```

图 1.10: 从 boot loader 进入内核后 0x00100000 处的内容

观察 obj/kern/kernel.asm 的前几条指令，其指令内容如下：

```
.globl entry
entry:
  movw  $0x1234,0x472     # warm boot
f0100000: 02 b0 ad 1b 00 00     add    0x1bad(%eax),%dh
f0100006: 00 00                 add    %al,(%eax)
f0100008: fe 4f 52              decb   0x52(%edi)
f010000b: e4                    .byte 0xe4

f010000c <entry>:
f010000c: 66 c7 05 72 04 00 00  movw   $0x1234,0x472
f0100013: 34 12
```

可以看出，gdb 中 0x00100000 处的内容与此处的内容一致。此时可以回答 Exercise 6 提出的两个问题：此处内容是因为 boot loader 将内核加载到了 0x00100000 处，这 8 个字的内容就是内核的前若干条指令。

1.3 The Kernel

1.3.1 Using virtual memory to work around position dependence

在使用 boot loader 时，链接地址和加载地址是一样的，也就是说虚拟地址和物理地址是一样的。但是进入到内核程序后这两种地址就不再相同了。内核的虚拟地址会被链接到一个非常高的地址空间，对于 jOS 而言是 0xf0100000。剩下的较低部分的空间给其他的用户程序使用。

但是由于许多机器没有 0xf0100000 的物理内存，因此在运行的时候需要把虚拟地址映射到物理地址 0x00100000 处运行。此时加载的物理地址仅仅高于 BIOS ROM。此时 PC 仅最少需要 1MB 物理内存。

在这个实验中，采用分页管理的方法来实现映射，但是不是使用的通常的分页管理器，而是自己写了一个 `lab/kern/entrypgdir.c` 进行映射。因此其功能很简单，只能够把 0xf0000000~0xf0400000 以及0x00000000~0x00400000 映射到 0x0000000~0x00400000 的范围内。不再这两个虚拟地址范围内的地址都会引起硬件异常。

> Exercise 7
>
> 使用 QEMU 和 GDB 跟踪进入 JOS 的内核，并停在 movl %eax, %cr0 处，检查 0x00100000 处的内容以及 0xf0100000 处的内容。现在用 stepi 单步执行这个命令然后再检查 0x00100000 处的内容和 0xf0100000 处的内容，并确保你弄懂了刚才发生了什么。如果不能正常建立映射，在新映射建立后最先不能够正常工作的指令是哪一条？注释掉 kern/entry.S 中的 movl %eax, %cr0 指令，然后看看你是否是正确的。

回答：启动 qemu 和 gdb，由于 Exercise 7 已知程序的入口是 0x10000c，因此首先在 0x10000c 处打上断点并运行到 0x10000c 处，然后使用 x/8x 检查 0x00100000 和 0xf100000 处的内容，内容如图 1.11

```
  1 ──> 0x10000c:    movw   $0x1234,0x472
  2     0x100015:    mov    $0x110000,%eax
  3     0x10001a:    mov    %eax,%cr3
  4     0x10001d:    mov    %cr0,%eax
  5     0x100020:    or     $0x80010001,%eax
  6     0x100025:    mov    %eax,%cr0
  7     0x100028:    mov    $0xf010002f,%eax
  8     0x10002d:    jmp    *%eax
  9     0x10002f:    mov    $0x0,%ebp
 10     0x100034:    mov    $0xf0110000,%esp
 11     0x100039:    call   0x100094
 12     0x10003e:    jmp    0x10003e
**    0x10000c: movw   $0x1234,0x472 (10000c - 10014a) **
Breakpoint 1, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c
766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200
fd8
(gdb) x/8x 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000      0x00000000      0x00000
000     0x00000000
0xf0100010 <entry+4>:   0x00000000      0x00000000      0x00000000    0
x00000000
(gdb)
```

图 1.11：执行 move %eax,%cr0 前的两处内存中的内容

由 Exercise 6 已知，0x00100000 处的内容就是内核的前几条指令，而从图1.11中可知，0xf0100000 处的内容全为 0。继续执行一步 stepi 以后，在次运行两个 x/8x 指令，这时显示的内容如1.12所示。

```
  1     0x10000c:    movw   $0x1234,0x472
  2     0x100015:    mov    $0x110000,%eax
  3     0x10001a:    mov    %eax,%cr3
  4     0x10001d:    mov    %cr0,%eax
  5     0x100020:    or     $0x80010001,%eax
  6     0x100025:    mov    %eax,%cr0
  7 ──> 0x100028:    mov    $0xf010002f,%eax
  8     0x10002d:    jmp    *%eax
  9     0x10002f:    mov    $0x0,%ebp
 10     0x100034:    mov    $0xf0110000,%esp
 11     0x100039:    call   0x100094
 12     0x10003e:    jmp    0x10003e
**    0x10000c: movw   $0x1234,0x472 (10000c - 10014a) **
0x00100028 in ?? ()
(gdb) x/8x 0xf0100000
0xf0100000 <_start+4026531828>: 0x1badb002      0x00000000      0xe4524
ffe     0x7205c766
0xf0100010 <entry+4>:   0x34000004      0x0000b812      0x220f0011    0
xc0200fd8
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c
766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200
fd8
```

图 1.12: 执行 movl %eax, %cr0 后的两处内存中的内容

可以看出，在执行了 movl %eax, %cr0 指令以后，两处的内容完全相同。这说明了此时已经成功启用了页表并完成了地址映射。

下面注释掉 movl %eax, %cr0，make clean 之后重新编译运行。逐步使用 stepi 之后在一个 jmp 指令后出现了问题。0x10002a 处的 jmp 指令需要跳转到 0xf010002c 处，但是没有分页管理不会进行虚拟地址映射到物理地址的转化，因此访问超出内存限制后程序崩溃。

1.3.2 Formatted Printing to the Console

终于到了实操写代码的环节了，也是正式需要我们动手的环节，接下来阅读 kern/printf.c，lib/printfmt.c 和 kern/console.c 的代码，分析他们之间的关系。

> Exercise 8
>
> 有一小部分用于打印八进制数（“%o” 格式）的代码被遗漏了，找出并补全这部分程序。

Exercise 8 实验过程回答：

要进行这个练习首先要分析 kern/printf.c，kern/console.c 以及 lib/printfmt.c 之间的关系。初步观察 3 个文件，发现 kern/printf.c 中的 cprintf, vprintf 函数调用了 lib/printfmt.c 中的 vprintfmt 函数。kern/printf.c 中的 putch 程序和 lib/printfmt 程序调用了 kern/console.c 中的 cputchar 程序。进

一步使用静态分析工具进行分析，cprintf 的调用关系图（call graph）如图1.13所示。

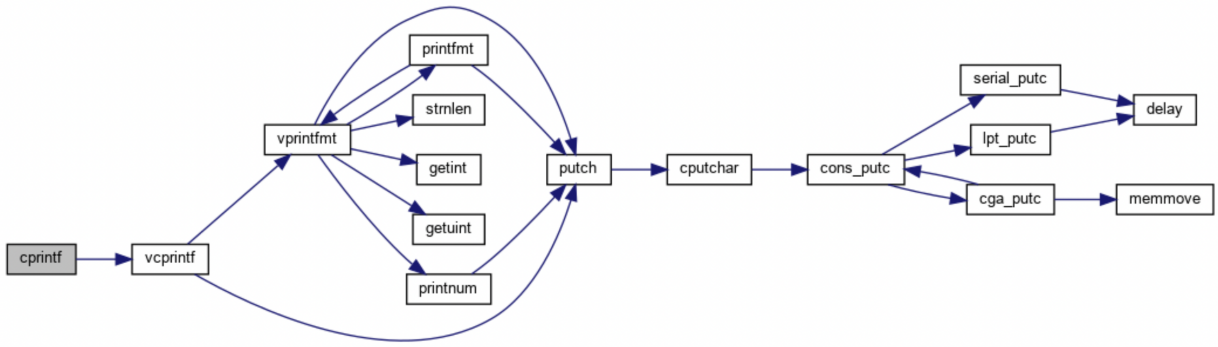

图 1.13: cprintf 的调用关系图

首先可以看到，这些函数中最接近底层的是 cons_putc，通过它的注释可以看到 cons_putc 的作用是输出一个字符到控制台。而 cons_putc 是通过调用 serial_putc，lpt_putc 以及 cga_putc 实现的。serial_putc 的代码及有关常量定义如下：

```
#define COM1    0x3F8
#define COM_LSR   5 // In:  Line Status Register
#define   COM_LSR_DATA  0x01  //   Data available
#define   COM_LSR_TXRDY 0x20  //   Transmit buffer avail

static void
serial_putc(int c)
{
  int i;

  for (i = 0;
```

```
        !(inb(COM1 + COM_LSR) & COM_LSR_TXRDY) && i < 12800;
        i++)
      delay();

  outb(COM1 + COM_TX, c);
}
```

通过查询端口功能，并分析代码，发现代码`!(inb(COM1 + COM_LSR)& COM_LSR_TXRDY)`是用于检测 03FD 的 bit5 是否为 1，而这一位用于判断发送数据缓冲区寄存器是否为空。outb 则是向 03F8 发送数据 c 而 03F8 被写入时功能为发送数据到串口。因此 serial_putc 的作用是向串口发送一个字符。

通过同样的方法，可以判断处 lpt_putc 的作用是向并口发送这个字符。最后，cga_putc 的作用是向 cga 设备，也就是计算机的显示器发送一个字符。cga_putc 对于需要输出的字符进行判断，如果是如'\t'、'\b' 等特殊字符则移动光标的位置或者执行相应动作，否则直接将字符显示输出在屏幕上。switch 后面的 if 用于判断缓冲区显示的内容不能够超过显示器大小 CRT_SIZE。

向上分析，cputchar 直接调用 cons_putc，而 putchar 直接调用 cputchar，并将计数器加一计数。vprintfmt 则调用 cputc 进行打印输出。printfmt 则是帮助 vprintfmt 进行递归调用的一个帮助函数。观察 vprintfmt，发现其由 4 个参数：

1. void (*putch)(int, void*) 是一个输出一个字符的函数指针。其第二个地址是字符输出位置的地址的地址，由于需要实现将值输出到这个地址后地址 +1，因此这个函数指针的第二个参数不是地址而是地址的地址。
2. void *putdat 是这个字符要存放的内存地址指针
3. const char* fmt 是输入的格式化字符串
4. va_list ap 是格式化字符串的可变长参数。

这个 vprintfmt 函数就是需要被修改的函数，将输出八进制的部分修改为如下代码即可实现 8 进制输出。

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

修改完成以后重新编译，即在命令行中执行 make clean && make && make qemu 以后，qemu 启动后会出现一行说明修改成功的输出:6828 decimal is 15254 octal!，如图1.14所示

```
***
*** Now run 'make gdb'.
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::25000 -D qemu.log  -S
6828 decimal is 15254 octal!
```

图 1.14: 重新编译并启动 qemu 后的部分输出

接下来对于实验指导中 Exercise 8 之后的问题，给出解答。

1. 解释 printf.c 和 console.c 之间的接口，尤其是 console.c 导出了那个函数？这个函数是如何在 printf.c 中被使用的?

- 在 Exercise 8 中已经给出分析部分。console.c 中除了静态函数之外所有的函数都导出并为外部提供服务了。printf.c 调用 console.c 中的 cputchar 函数进行显示输出。

2. 解释如下 console.c 中的代码：

```
if (crt_pos >= CRT_SIZE) {
    int i;

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

- 这部分代码是为了当输出超过显示屏的大小的时候将显示屏向上滚动一行，由于显示屏大小为 80 × 25，因此将 1~24 行复制并放在 0~23 行，然后将最后一行清空。mmove 的作用是滚动，而 for 循环的作用是清空最后一行。

3. 单步执行如下代码:

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

(a) 在调用 cprintf 的过程中，fmt 指向那里？ap 指向哪里?

(b) 按照执行的顺序列出对于 cons_putc, va_arg, 以及 vcprintf 的调用。对于 cons_putc，列出它的参数。对于 va_arg，列出调用前后 ap 指向哪里。对于 vcprintf 列出它的两个参数的值。

- 在 kern/monitor.c 中的 mon_backtrace 中添加这两条语句并打开 gcc 进行追踪，发现 fmt 指向格式化字符串”x %d, y%x, z%n”，而 ap 指向参数 x, y, z。如图1.15（由于在 cprintf 中不能正常看到 fmt 以及 ap 的值因此实际取值是在 vcprintf 中进行的。

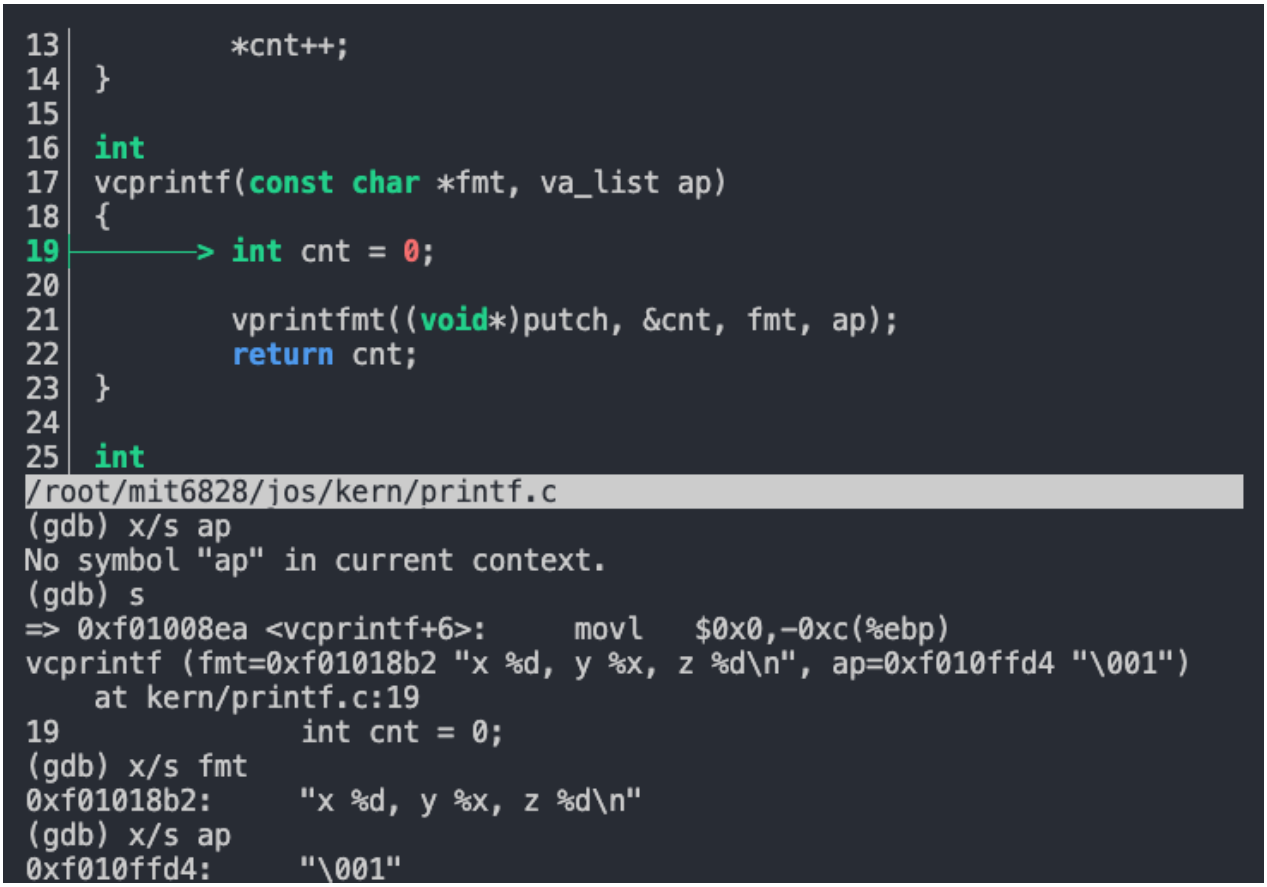

图 1.15: fmt 以及 ap 的取值

- 对于这三个函数的调用，getint 使用 va_arg 根据不同的参数类型从参数列表中取出下一个参数。第一次对于 va_arg 进行调用前 ap 为 1,3,4，调用后变为 3,4。

4. 运行如下代码：

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s\n", 57616, &i)
```

输出是什么？按照上一题的方法逐步解释输出是怎么来的。输出依赖于 x86 系统是小端这一事实。如果 x86 是大端的，当 i 为何值时才会有相同的输出？你需要将 57616 更改为一个不同的值吗?

- 与上一题类似，重新编译并运行，最后得到的运行结果为 “He110, World”，如图1.16所示。

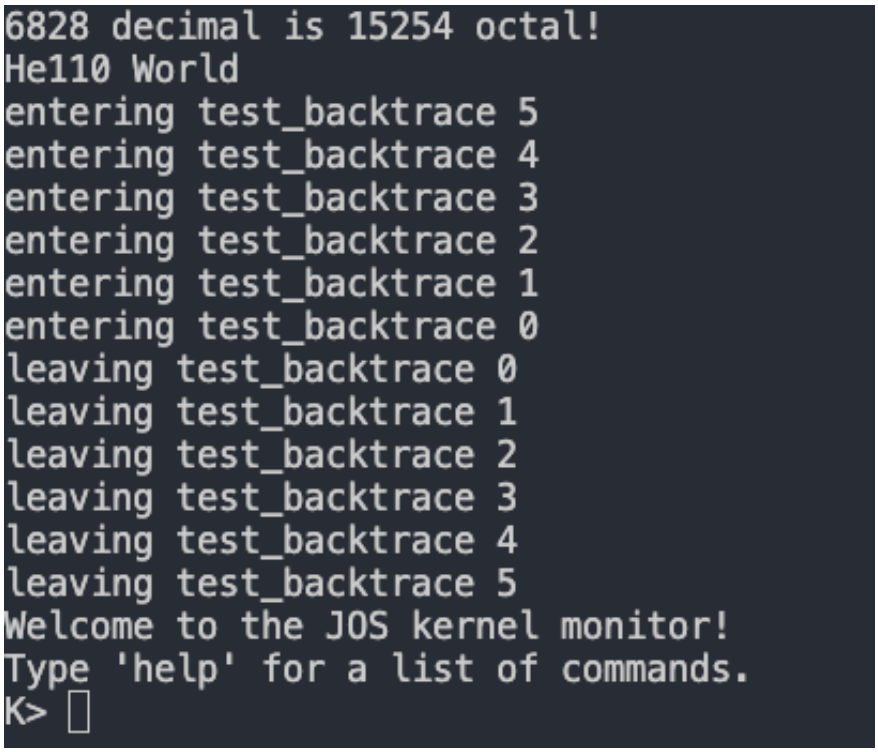

图 1.16: 重新编译后的运行结果

输出这个值的原因是：第一个%x 要求按照 16 进制输出第一个参数，而第一个参数的值为 57616，对应的 16 进制为 e110，因此前半部分输出为 He110，而第二个格式化字符%s 要求将 &i 作为一个字符串输出，此时将 i 进行拆分。由于 x86 是小端模式，因此地址由低向高增长时字节也由 LSB 向 MSB 增长。因此将 0x00646c72 拆分后，变为 0x72('r'), 0x6c('l'), 0x54('d'), 0x00('\0')。于是后半部分输出 World。当 i 是大端时，需要能够将 i 改为 0x726c6400 才能有相同的输出，但是不需要改变 57616 这个值。

5. 在接下来的代码中，'y=' 之后打印的是什么？（注意：答案不是一个特定值。）这为什么会发生?

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

- 同样的，将这个代码输入 monitor.c 后重新编译运行，输出的结果为 "x=3 y=-267292872"， x 后面输出的为 3，因为第一个参数为 3，而 y 没有指定参数，因此将输出一个不确定的值，实际上，是从栈中 3 这个参数的后方多取了一个数作为参数。

6. 让我们假设 GCC 改变了函数调用转换，并将参数入栈顺序更改为声明顺序，也就是说最后一个参数最后入栈。你需要如何更改 cprintf 或它的接口才能使传递任意参数仍然可行?

- 将变长参数置于格式化字符串的前方即可，即将 cprintf 的接口变为 `int cprintf(..., const char *fmt)`。

1.3.3 Tha Stack

这一份将重新编写一个 kernel monitor 程序用于记录堆栈的变化，该变化是由一系列被保存到堆栈的 IP 寄存器的值组成的。

> Exercise 9
>
> 判断操作系统的内核从哪一条指令开始初始化它的堆栈空间，以及这个堆栈空间在内存的哪个地方？内核是如何给堆栈保留一块内存空间的？堆栈指针是指向这个区域的哪一端的?

回答：

在进入内核之前，整个 boot loader 是没有对%esp 和%ebp 的内容进行修改的，因此这其中没有初始化堆栈空间的语句。而在 entry.S 中，发现其最后几条指令如下：

```
relocated:

  # 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

  # now to C code
  call  i386_init
```

通过最后一条指令跳转到 C 函数中，以及前两条语句的注释可以判断，movl $0x0,%ebp 以及 movl $(bootstacktop),%esp 就是用于初始化堆栈空间的语句。这两条语句中，第一条将 0 赋值给%ebp，第二条将 $(bootstacktop) 赋值给%esp，此时需要进入 gdb 进行调试。当执行到这两条指令时，gdb 显示如图1.17所示。

```
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 pointe
75
76          # Set the stack pointer
77  ------> movl    $(bootstacktop),%esp
78
79          # now to C code
80          call    i386_init
81
82          # Should never get here, but in case we do, just spin.
83  spin:   jmp     spin
/root/mit6828/jos/kern/entry.S
Continuing.
The target architecture is assumed to be i386
=> 0xf010002f <relocated>:      mov    $0x0,%ebp

Breakpoint 1, relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame po
inter
(gdb) si
=> 0xf0100034 <relocated+5>:    mov    $0xf0110000,%esp
relocated () at kern/entry.S:77
77              movl    $(bootstacktop),%esp
```

图 1.17: gdb 显示 $(bootstacktop) 输出

从图中可以发现，movl $(bootstacktop),%esp 在运行时为 mov $0xf0110000,%esp。也就是说栈顶的地址为 0xf0110000。而根据 entry.S 最后的标志可以发现，在栈顶前分配了 KSTKSIZE 这么多的空间。KSTKSIZE 在 inc/memlayout.h 中定义。其大小为 (8*PGSIZE)，也就是 32KB，因此栈的地址空间为 0xf0108000~0xf0110000。

综上所述，对于 Exercise 9 提出的问题进行回答:

1. 内核从哪一条指令开始初始化它的堆栈空间?

- 从 movl $0x0,%ebp 以及 movl $(bootstacktop),%esp 开始初始化堆栈空间。

2. 这个堆栈空间在内存的哪个地方?

- 虚拟地址 0xf0108000~0xf0110000，对应的物理地址 0x00108000~0x00110000。

3. 内核是如何给堆栈保留一块内存空间的?

- 通过在 entry.S 中的数据段声明一块大小为 32KB 的空间作为堆栈使用。

4. 堆栈指针是指向这个区域的哪一端的?

- 由于栈是向下增长的，故堆栈指指向最高地址。最高地址就是 bootstacktop 的值 0xf0110000。

X86 堆栈指针寄存器%esp 指向堆栈中正在被使用的部分的最低地址。更低的地址空间还没有被利用。在 32 位模式中，每一次对堆栈的操作都是以 32bit 为单位的，因此%esp 是 4 字节对齐的。%ebp 寄存器是记录每一个程序的栈帧的相关信息的寄存器，每一个程序运行时都会分配给一个栈帧，用于存放临时变量，传递参数等。

> Exercise 10
>
> 找到 obj/kern/kernel.asm 中 test_backtrace 子程序的地址，设置断点并讨论启动内核后这个程序被调用时发生了什么。对于递归的调用，多少 32 位字被压入栈中？这些字分别是什么?

`Exercise 10 实验过程`

在 obj/kern/kernel.asm 中，test_backtrace 所对应的子程序在开始时对应如下几条指令:

```
f0100040: 55                    push   %ebp
f0100041: 89 e5                 mov    %esp,%ebp
f0100043: 53                    push   %ebx
f0100044: 83 ec 0c              sub    $0xc,%esp
```

这 5 条指令用于保存父程序的栈信息，并为当前子程序分配新的栈帧。在 i386_init 中运行了子程序 test_backtrace(5)。在调用这一子程序之前，%esp = 0xf010ffe0, %ebp = 0xf010fff8，如图1.18所示。

```
34          cons_init();
35
36          cprintf("6828 decimal is %o octal!\n", 6828);
37
38          // Test the stack backtrace function (lab 1 only)
39  ------> test_backtrace(5);
40
41          // Drop into the kernel monitor.
42          while (1)
43                  monitor(NULL);
44  }
/root/mit6828/jos/kern/init.c
=> 0xf01000b6 <i386_init+34>:   add    $0x8,%esp
36              cprintf("6828 decimal is %o octal!\n", 6828);
(gdb) n
=> 0xf01000c8 <i386_init+52>:   movl   $0x5,(%esp)
39              test_backtrace(5);
(gdb) p $esp
$1 = (void *) 0xf010ffe0
(gdb) p $ebp
$2 = (void *) 0xf010fff8
```

图 1.18: 调用 test_backtrace(5) 前的栈寄存器信息

在调用过程中，首先 call 指令将 i386_init 的返回地址压入栈中，此时 esp 变为 0xf010ffdc，此时进入 test_backtrace(5) 子程序。然后开始执行上述的 5 个指令：push %ebp 用于保存%ebp 寄存器的值，执行完成后%esp 变为 0xf010ffd8；然后 mov %esp, %ebp 把%ebp 的值更新为%esp 的值，也就是 0xf010ffd8；然后 push %ebt 保护寄存器%ebx 的值，执行完成后 esp 变为 0xf010ffd4；最后 sub $0xc,%esp 将%esp 减去 0xc，%esp 变为 0xf010ffc0，这是用于分配空间存储临时变量。因此，在执行完上述指令后，%esp 和%ebp 分别变为 0xf010ffc0 和 0xf010ffd8。而这就是 test_backtrace(5) 执行时栈寄存器的变化范围。由于递归调用，像这样的变化还需要执行 test_backtrace(4)~test_backtrace(0) 5 次。

也就是说每次递归调用压入 32 字节，也就是 8 个 32 位字。在调用到 test_backtrace(0) 时，%esp 将为 0xf010ff20 而%ebp 将为 0xf010ff38。实际的调试过程也验证了这一点。如图 1.19 所示：

```
10  // Test the stack backtrace function (lab 1 only)
11  void
12  test_backtrace(int x)
13  {
14  ------> cprintf("entering test_backtrace %d\n", x);
15          if (x > 0)
16                  test_backtrace(x-1);
17          else
/root/mit6828/jos/kern/init.c
(gdb) p $esp
$12 = (void *) 0xf010ff20
(gdb) p $ebp
$13 = (void *) 0xf010ff38
```

图 1.19: 执行到 test_backtrace(0) 时的栈寄存器值

上述练习已经有足够多的信息以供实现一个 stack backtrace 程序了，将之称为 mon_backtrace()，其原型已经定义在 kern/monitor.c 中。除了实现之外，好需要将其 hook 到 kernel monitor 命令列表中，使其能够直接被用户调用。backtrace 函数应该显示一个如下的调用帧关系：

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

每一行都包含 ebp, eip 和 args，也就是有关的寄存器以及参数。而第一行的 Stack backtrace 表明现在执行的是 mon_backtrace 子程序。其后依次外推直到最外层。

> Exercise 11
>
> 实现上述 backtrace 程序，并使用相同的格式进行输出。使用 make grade 进行验证。

`Exercise 11 实验过程`

根据程序调用时栈的关系来实现显示正在执行的程序的栈的信息，栈的结构如图1.20所示。

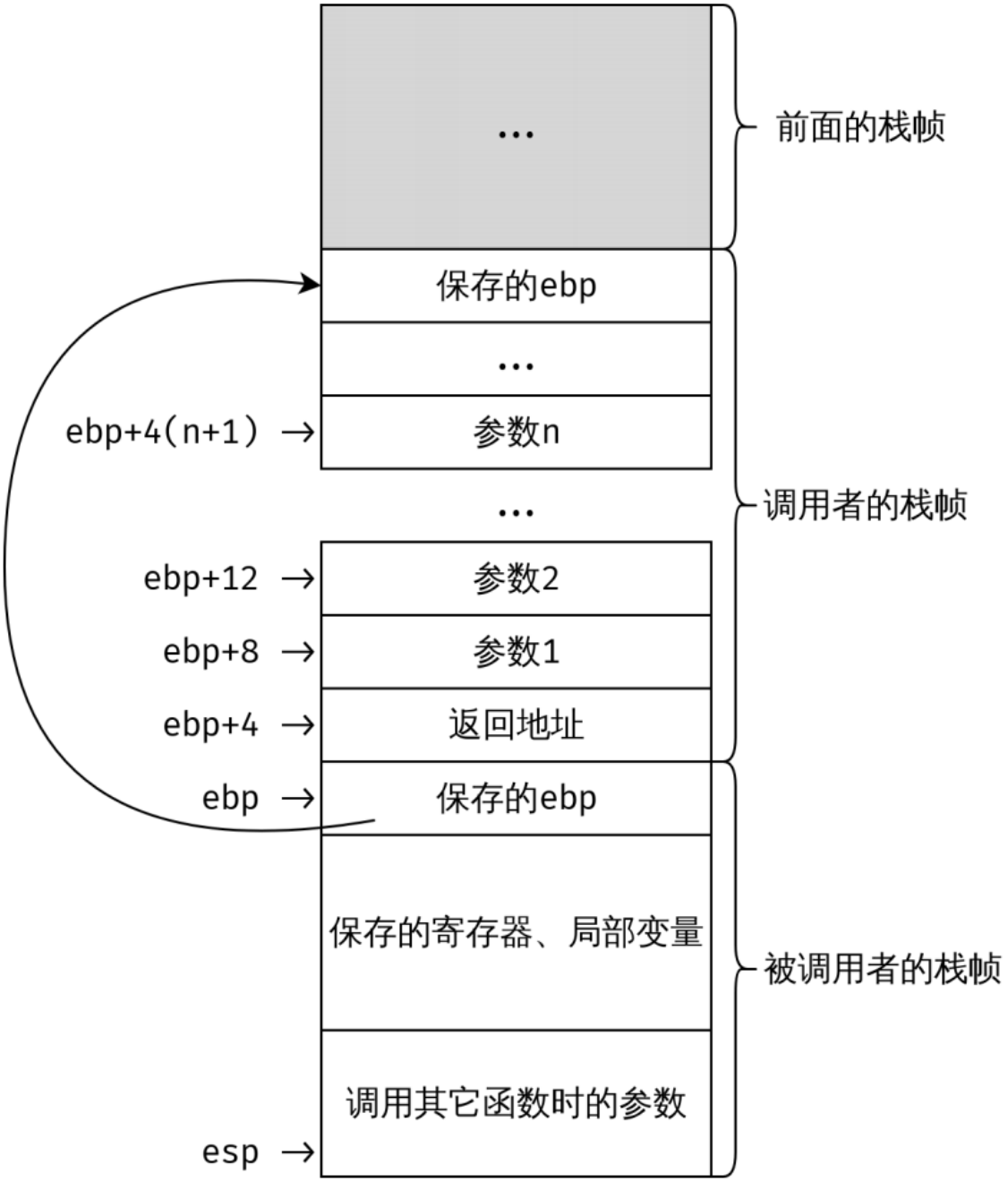

图 1.20: 栈帧结构

可以看出，保存的%ebp 是连接当前栈帧和上一个（调用者的）栈帧的关键，它使各个栈帧通过类似于链表的结构连接起来。最终，在`monitor.c`函数`mon_backtrace`中填写的代码如下：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
  // Your code here.
  uint32_t ebp, eip;

  for(ebp = read_ebp(); ebp != 0; ebp = *((uint32_t *) ebp)){
    eip = *((uint32_t *)ebp + 1);
    cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n", ebp,
eip,
```

```
10      *((uint32_t *) ebp + 2), *((uint32_t *) ebp + 3), *((uint32_t *) ebp +
    4),
11      *((uint32_t *) ebp + 5), *((uint32_t *) ebp + 6));
12    }
13    return 0;
14  }
```

使用 make grade 进行测试。从结果中可以看出，backtrace count 以及 backtrace arguments 已经通过测试，余下的 backtrace arguments 和 backtrace lines 需要在 `Exercise 12` 中进一步补全。

> Exercise 12
>
> 继续修改 backtrace 函数，令其对于每个 eip 能显示函数名，源文件以及对应的行数。

`Exercise 12 实验过程`

stab 表是用于存储源文件以及编译后文件对应关系的一张表。使用 objdump -G obj/kern/kernel 命令，查看 stab 表中的内容，例如部分输出如下：

```
1   339    SOL    0      0      f01005a6 2985   ./inc/x86.h
2   340    SLINE  0      66     00000053 0
3   341    SOL    0      0      f01005ae 2970   kern/console.c
4   342    SLINE  0      152    0000005b 0
5   343    SOL    0      0      f01005b1 2985   ./inc/x86.h
6   344    SLINE  0      16     0000005e 0
7   345    SOL    0      0      f01005b4 2970   kern/console.c
8   346    SLINE  0      152    00000061 0
9   347    SOL    0      0      f01005ba 2985   ./inc/x86.h
10  348    SLINE  0      66     00000067 0
11  349    SLINE  0      16     0000006f 0
12  350    SOL    0      0      f01005c5 2970   kern/console.c
13  351    SLINE  0      156    00000072 0
14  352    SLINE  0      157    00000078 0
15  353    SOL    0      0      f01005d6 2985   ./inc/x86.h
16  354    SLINE  0      66     00000083 0
17  355    SLINE  0      16     000000d4 0
18  356    SOL    0      0      f010062f 2970   kern/console.c
19  357    SLINE  0      99     000000dc 0
20  358    SOL    0      0      f0100638 2985   ./inc/x86.h
21  359    SLINE  0      16     000000e5 0
22  360    SOL    0      0      f010063e 2970   kern/console.c
```

前五列对应的就是 stab，第 7 列是 stabstr，分别对应着 kdebug.c 中的 **STAB_BEGIN**以及 **STABSTR_BEGIN**。

通过 stab_binsearch 对于 stab 进行检索。通过注释可以知道 stab_binsearch 接受 5 个参数：

- const struct Stab *stabs：stab 表的起始位置。
- int *region_left：查找下界。

- int *region_right：查找上界。
- int type：查找的表项类型，对应图1.27的第二列。
- uintptr_t addr 要查找的值，对应图1.27的第五列。

通过这个函数对于 stab 表进行查找，可知在 debuginfo_eip 中需要补全的部分为：

```
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
info->eip_line = lline > rline ? -1 : stabs[rline].n_desc;
```

将 debuginfo_eip 补全以后，需要对于 kern/monitor.c 中的 mon_backtrace 进行扩展，使其调用这一函数以获得相应的调试信息。扩展后的 `mon_backtrace` 如下：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
  // Your code here.
  uint32_t ebp, eip;
  struct Eipdebuginfo info;
  for(ebp = read_ebp(); ebp != 0; ebp = *((uint32_t *) ebp)){
    eip = *((uint32_t *)ebp + 1);
    debuginfo_eip(eip, &info);
    cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n
%s:%d: %.*s+%d\n", ebp, eip,
    *((uint32_t *) ebp + 2), *((uint32_t *) ebp + 3), *((uint32_t *) ebp +
4),
    *((uint32_t *) ebp + 5), *((uint32_t *) ebp + 6), info.eip_file,
info.eip_line,
    info.eip_fn_namelen, info.eip_fn_name, eip - info.eip_fn_addr);
  }
  return 0;
}
```

然后重新编译，运行 make grade 的结果如图 1.21 所示，可以看到已经通过全部的测试。

```
+ 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 390 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/root/mit6828/jos'
running JOS: (1.2s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
```

图 1.21: 最后重新编译后运行 make grade 结果

最后，需要将 backtrace 命令整合进入终端中，这时只需将 `kern/monitor.c` 中的 commands 更改为如下几行即可。

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

更改后重新编译并进入 qemu 进行测试，可以看到，输入 help 后能够显示 backtrace 命令，并且输入 backtrace 后能够正确的输出结果了，如图1.22 所示

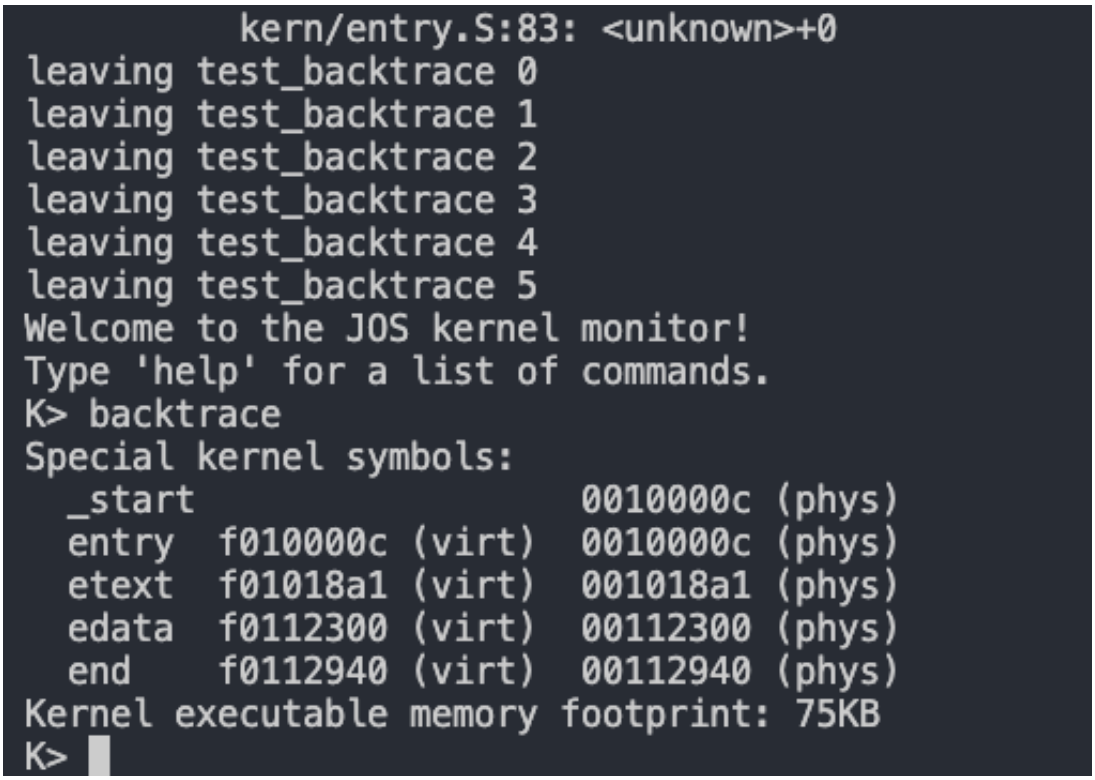

图 1.22: 将 backtrace 整合到终端中