练习 1：

有关 Netwide assemble（NASM） 和 GNU assemble（GAS） 不同之处

NASM 和 GAS 之间最大的差异之一是语法。GAS 使用 AT&T 语法，这是一种相当老的语法，由 GAS 和一些老式汇编器使用；NASM 使用 Intel 语法，大多数汇编器都支持它，包括 TASM 和 MASM。(GAS 的现代版本支持 .intel_syntax 指令，因此允许在 GAS 中使用 Intel 语法。)

下面是从 GAS 手册总结出的一些主要差异：

* AT&T 和 Intel 语法采用相反的源和目标操作数次序。例如：
    - Intel: mov eax, 4
    - AT&T: movl $4, %eax
* 在 AT&T 语法中，中间操作数前面加 $；在 Intel 语法中，中间操作数不加前缀。例如：
    - Intel: push 4
    - AT&T: pushl $4
* 在 AT&T 语法中，寄存器操作数前面加 %。在 Intel 语法中，它们不加前缀。
* 在 AT&T 语法中，内存操作数的大小由操作码名称的最后一个字符决定。操作码后缀 b、w 和 l 分别指定字节（8 位）、字（16 位）和长（32 位）内存引用。Intel 语法通过在内存操作数(而不是操作码本身)前面加 byte ptr、word ptr 和 dword ptr 来指定大小。所以：
    - Intel: mov al, byte ptr foo
    - AT&T: movb foo, %al
* 在 AT&T 语法中，中间形式长跳转和调用是 lcall/ljmp $section, $offset; Intel 语法是 call/jmp far section:offset。在 AT&T 语法中，远返回指令是 lret $stack-adjust，而 Intel 使用 ret far stack-adjust。

在这两种汇编器中，寄存器的名称是一样的，但是因为寻址模式不同，使用它们的语法是不同的。另外，GAS 中的汇编器指令以 “.” 开头，但是在 NASM 中不是。

.text 部分是处理器开始执行代码的地方。global（或者 GAS 中的 .globl 或 .global）关键字用来让一个符号对链接器可见，可以供其他链接对象模块使用。在清单 1 的 NASM 部分中，global _start 让 _start 符号成为可见的标识符，这样链接器就知道跳转到程序中的什么地方并开始执行。与 NASM 一样，GAS 寻找这个 _start 标签作为程序的默认进入点。在 GAS 和 NASM 中标签都以冒号结尾。

GAS 使用 0x 前缀指定十六进制数字，NASM 使用 h 后缀。因为在 GAS 中中间操作数带 $ 前缀，所以 80 hex 是 $0x80。

int $0x80（或 NASM 中的 80h）用来向 Linux 请求一个服务。服务编码放在 EAX 寄存器中。EAX 中存储的值是 1（代表 Linux exit 系统调用），这请求程序退出。EBX 寄存器包含退出码（在这个示例中是 2），也就是返回给操作系统的一个数字。(可以在命令提示下输入 echo $? 来检查这个数字。)

最后讨论一下注释。GAS 支持 C 风格（/* */）、C++ 风格（//）和 shell 风格（#）的注释。NASM 支持以 “;” 字符开头的单行注释。

在内存变量声明中可以看到几点差异。NASM 分别使用 dd、dw 和 db 指令声明 32 位、16 位和 8 位数字，而 GAS 分别使用 .long、.int 和 .byte。

GAS 还有其他指令，比如 .ascii、.asciz 和 .string。在 GAS 中，像声明其他标签一样声明变量（使用冒号），但是在 NASM 中，只需在内存分配指令（dd、dw 等等）前面输入变量名，后面加上变量的值。

NASM 使用方括号间接引用一个内存位置指向的地址值：[var1]。GAS 使用圆括号间接引用同样的值：(var1)。本文后面讨论其他寻址模式的使用方法。

练习 2：使用 GDB 的 si 指令来跟踪 ROM BIOS 中的几个指令，并且分析这些指令是要做什么的。

```
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) si
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x65b4
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xfd3aa
0x0000e062 in ?? ()
(gdb) x/10i 0xfe062
   0xfe062:     jne    0xfd3aa
   0xfe066:     xor    %ax,%ax
   0xfe068:     mov    %ax,%ss
   0xfe06a:     mov    $0x7000,%esp
   0xfe070:     mov    $0xf431f,%edx
   0xfe076:     jmp    0xfd233
   0xfe079:     push   %ebp
   0xfe07b:     push   %edi
   0xfe07d:     push   %esi
   0xfe07f:     push   %ebx
(gdb)
```

可以看到 CS:IP 为 0xf000:0xfff0， 即执行 BIOS 的第一条指令，地址为 0xf000 * 0x10 + 0xfff0 = 0xffff0，此处指令内容为：

ljump 0xf000, 0xe05b

即跳转到 0xfe05b.

当 PC 机启动时，CPU 运行在实模式(real mode)下，而当进入操作系统内核后，将会运行在保护模式下(protected mode)。实模式是早期 CPU，比如 8088 处理器的工作模式，这类处理器由于只有 20 根地址线，所以它们只能访问 1MB 的内存空间。但是 CPU 也在不断的发展，之后的 80286/80386 已经具备 32 位地址总线，能够访问 4GB 内存空间，为了能够很好的管理这么大的内存空间，保护模式被研发出来。所以现代处理器都是工作在保护模式下的。但是为了实现向后兼容性，即原来运行在 8088 处理器上的软件仍旧能在现代处理器上运行，所以现代的 CPU 都是在启动时运行于实模式，启动完成后运行于保护模式。BIOS 就是 PC 刚启动时运行的软件，所以它必然工作在实模式。

第一条指令：`[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b`

下一条指令，把 0x0 这个立即数和$cs:0x6ac8 所代表的内存地址处的值比较，至于为什么这样比较，现在还不是很清楚。其中$cs:0x6ac8 就是我们刚刚介绍的在实模式下地址形成的格式，其中$cs 就代表 CS 段寄存器的值。

```
3. 0xfe062:   jne   0xfd2e1
```

jne 指令：如果 ZF 标志位为 0 的时候跳转，即上一条指令 cmpl 的结果不是 0 时跳转，也就是$cs:0x6ac8 地址处的值不是 0x0 时跳转。

```
4. 0xfe066:   xor   %dx, %dx
```

下一条指令地址是 0xfe066，可见上面的跳转指令并没有跳转。这条指令的功能是把 dx 寄存器清零。

```
5. 0xfe068:   mov   %dx %ss
6. 0xfe06a:   mov   $0x7000, %esp
7. 0xfe070:   mov   $0xf34d2, %edx
8. 0xfe076:   jmp   0xfd15c
9. 0xfd15c:   mov   %eax, %ecx
```

接下来的这些指令就是设置一些寄存器的值，具体含义现在不明白..

这里要注意第 8 条指令，进行了绝对跳转。

```
10. 0xfd15f:   cli
```

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

```
11. 0xfd160:   cld
```

设置方向标识位为 0，表示后续的串操作比如 MOVS 操作，内存地址的变化方向，如果为 0 代表从低地址值变为高地址。

```
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。

```
15. 0xfd16b:   in   $0x92, %al
16. 0xfd16d:   or   $0x2, %al
17. 0xfd16f:   out   %al, $0x92
```

这三步操作又是在控制端口，此时被控制的端口号为 0x92，通过上面那个链接 http://bochs.sourceforge.net/techspec/PORTS.LST

我们可以查看到，它控制的是 PS/2 系统控制端口 A，而第 16，17 步的操作明显是在把这个端口的 1 号 bit 置为 1。这个端口的 bit1 的功能是

bit 1= 1 indicates A20 active

即 A20 位，即第 21 个地址线被使能，了解实模式和保护模式的同学肯定清楚，如果 A20 地址线被激活，那么系统工作在保护模式下。但是在之后的 boot loader 程序中，计算机首先要工作在实模式下啊。所以这里的这个操作，根据网上 http://kernelx.weebly.com/a20-address-line.html 所说应该是去测试可用内存空间。在 boot loader 之前，它肯定还会转换回实模式。

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

lidt 指令：加载中断向量表寄存器(IDTR)。这个指令会把从地址 0xf6ab8 起始的后面 6 个字节的数据读入到中断向量表寄存器(IDTR)中。中断是操作系统中非常重要的一部分，有了中断操作系统才能真正实现进程。每一种中断都有自己对应的中断处理程序，那么这个中断的处理程序的首地址就叫做这个中断的中断向量。中断向量表自然是存放所有中断向量的表了。关于中断向量表的介绍，大家可以戳这个链接 http://wiki.osdev.org/Interrupt_Descriptor_Table

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

把从 0xf6a74 为起始地址处的 6 个字节的值加载到全局描述符表格寄存器中 GDTR 中。这个表实现保护模式非常重要的一部分，我们在介绍 boot loader 时会具体介绍它。

```
20. 0xfd17d:  mov  %cr0, %eax
21. 0xfd180:  or  $0x1, %eax
22. 0xfd184:  mov  %eax, %cr0
```

计算机中包含 CR0~CR3 四个控制寄存器，用来控制和确定处理器的操作模式。其中这三个语句的操作明显是要把 CR0 寄存器的最低位(0bit)置 1。CR0 寄存器的 0bit 是 PE 位，启动保护位，当该位被置 1，代表开启了保护模式。但是这里出现了问题，我们刚刚说过 BIOS 是工作在实模式之下，后面的 boot loader 开始的时候也是工作在实模式下，所以这里把它切换为保护模式，显然是自相矛盾。所以只能推测它在检测是否机器能工作在保护模式下。

```
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
```

修改这些寄存器的值。这些寄存器都是段寄存器。

- CS（CodeSegment）和 IP（Instruction Pointer）寄存器一起用于确定下一条指令的地址。计算公式： physical address = 16 * segment + offset.
- PC 开始运行时，CS = 0xf000，IP = 0xfff0，对应物理地址为 0xffff0.第一条指令做了 jmp 操作，跳到物理地址为 0xfe05b 的位置。
  - CLI：Clear Interupt，禁止中断发生。STL：Set Interupt，允许中断发生。CLI 和 STI 是用来屏蔽中断和恢复中断用的，如设置栈基址 SS 和偏移地址 SP 时，需要 CLI，因为如果这两条指令被分开了，那么很有可能 SS 被修改了，但由于中断，而代码跳去其它地方执行了，SP 还没来得及修改，就有可能出错。
  - CLD：Clear Director。STD：Set Director。在字行块传送时使用的，它们决定了块传送的方向。CLD 使得传送方向从低地址到高地址，而 STD 则相反。
  - 汇编语言中，CPU 对外设的操作通过专门的端口读写指令来完成，读端口用 IN 指令，写端口用 OUT 指令。进一步理解“端口”的概念可以参考博客理解“统一编址与独立编址、I/O 端口与 I/O 内存”。
  - LIDT：加载中断描述符。LGDT：加载全局描述符。

练习 3

分别是/boot/boot.S 和/boot/main.c 文件。其中前者是一个汇编文件，后者是一个 C 语言文件。当 BIOS 运行完成之后，CPU 的控制权就会转移到 boot.S 文件上。所以我们首先看一下 boot.S 文件。

/boot/boot.S：

```
1 .globl start
2 start:
3   .code16                     # Assemble for 16-bit mode
4   cli                         # Disable interrupts
```

这几条指令就是 boot.S 最开始的几句，其中 cli 是 boot.S，也是 boot loader 的第一条指令。这条指令用于把所有的中断都关闭。因为在 BIOS 运行期间有可能打开了中断。此时 CPU 工作在实模式下。

```
5   cld                             # String operations increment
```

这条指令用于指定之后发生的串处理操作的指针移动方向。在这里现在对它大致了解就够了。

```
6   # Set up the important data segment registers (DS, ES, SS).
```

```
7   xorw    %ax,%ax             # Segment number zero
8   movw    %ax,%ds             # -> Data Segment
9   movw    %ax,%es             # -> Extra Segment
10  movw    %ax,%ss             # -> Stack Segment
```

这几条命令主要是在把三个段寄存器，ds，es，ss 全部清零，因为经历了 BIOS，操作系统不能保证这三个寄存器中存放的是什么数。所以这也是为后面进入保护模式做准备。

```
11  # Enable A20:
12  #   For backwards compatibility with the earliest PCs, physical
13  #   address line 20 is tied low, so that addresses higher than
14  #   1MB wrap around to zero by default.  This code undoes this.
15  seta20.1:
16  inb     $0x64,%al               # Wait for not busy
17  testb   $0x2,%al
18  jnz     seta20.1

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

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

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

这部分指令就是在准备把 CPU 的工作模式从实模式转换为保护模式。我们可以看到其中的指令包括 inb，outb 这样的 IO 端口命令。所以这些指令都是在对外部设备进行操作。

所以 16~18 号指令是在不断的检测 bit1。bit1 的值代表输入缓冲区是否满了，也就是说 CPU 传送给控制器的数据，控制器是否已经取走了，如果 CPU 想向控制器传送新的数据的话，必须先保证这一位为 0。所以这三条指令会一直等待这一位变为 0，才能继续向后运行。

当 0x64 端口准备好读入数据后，现在就可以写入数据了，所以 19~20 这两条指令是把 0xd1 这条数据写入到 0x64 端口中。当向 0x64 端口写入数据时，则代表向键盘控制器 804x 发送指令。这个指令将会被送给 0x60 端口。

```
27   # Switch from real to protected mode, using a bootstrap GDT
28   # and segment translation that makes virtual addresses
```

```
29   # identical to their physical addresses, so that the
30   # effective memory map does not change during the switch.
31   lgdt    gdtdesc
32   movl    %cr0, %eax
33   orl     $CR0_PE_ON, %eax
34   movl    %eax, %cr0
```

首先 31 号指令 lgdt gdtdesc，是把 gdtdesc 这个标识符的值送入全局映射描述符表寄存器 GDTR 中。这个 GDT 表是处理器工作于保护模式下一个非常重要的表。具体可以参照我们的 Appendix 1 关于实模式和保护模式的介绍。至于这条指令的功能就是把关于 GDT 表的一些重要信息存放到 CPU 的 GDTR 寄存器中，其中包括 GDT 表的内存起始地址，以及 GDT 表的长度。这个寄存器由 48 位组成，其中低 16 位表示该表长度，高 32 位表该表在内存中的起始地址。所以 gdtdesc 是一个标识符，标识着一个内存地址。从这个内存地址开始之后的 6 个字节中存放着 GDT 表的长度和起始地址。我们可以在这个文件的末尾看到 gdtdesc，如下：

```
 1 # Bootstrap GDT
 2 .p2align 2                                # force 4 byte alignment
 3 gdt:
 4   SEG_NULL                                # null seg
 5   SEG(STA_X|STA_R, 0x0, 0xffffffff)       # code seg
 6   SEG(STA_W, 0x0, 0xffffffff)             # data seg
 7
 8 gdtdesc:
 9   .word   0x17                            # sizeof(gdt) - 1
10   .long   gdt                             # address gdt
```

其中第 3 行的 gdt 是一个标识符，标识从这里开始就是 GDT 表了。可见这个 GDT 表中包括三个表项(4,5,6 行)，分别代表三个段，null seg，code seg，data seg。由于 xv6 其实并没有使用分段机制，也就是说数据和代码都是写在一起的，所以数据段和代码段的起始地址都是 0x0，大小都是 0xffffffff=4GB。

在第 4~6 行是调用 SEG()子程序来构造 GDT 表项的。这个子函数定义在 mmu.h 中，形式如下：

```
  #define SEG(type,base,lim)                                  \
                    .word (((lim) >> 12) & 0xffff), ((base) &
0xffff);     \
                    .byte (((base) >> 16) & 0xff), (0x90 | (type)),
\
                    (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) &
0xff)
```

可见函数需要 3 个参数，一是 type 即这个段的访问权限，二是 base，这个段的起始地址，三是 lim，即这个段的大小界限。gdt 表中的每一个表项的结构如图所示：

```
struc gdt_entry_struct

        limit_low:   resb 2
        base_low:    resb 2
        base_middle: resb 1
        access:      resb 1
        granularity: resb 1
        base_high:   resb 1

endstruc
```

每个表项一共 8 字节，其中 limit_low 就是 limit 的低 16 位。base_low 就是 base 的低 16 位，依次类推，所以我们就可以理解 SEG 函数为什么要那么写（其实还是有很多不理解的。。）。

然后在 gdtdesc 处就要存放这个 GDT 表的信息了，其中 0x17 是这个表的大小-1 = 0x17 = 23，至于为什么不直接存表的大小 24，根据查询是官方规定的。紧接着就是这个表的起始地址 gdt。

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

再回到刚才那里，当加载完 GDT 表的信息到 GDTR 寄存器之后。紧跟着 3 个操作，32~34 指令。这几步操作明显是在修改 CR0 寄存器的内容。CR0 寄存器还有 CR1~CR3 寄存器都是 80x86 的控制寄存器。其中$CR0_PE 的值定义于"mmu.h"文件中，为 0x00000001。可见上面的操作是把 CR0 寄存器的 bit0 置 1，CR0 寄存器的 bit0 是保护模式启动位，把这一位值 1 代表保护模式启动。

```
35    ljmp    $PROT_MODE_CSEG, $protcseg
```

这只是一个简单的跳转指令，这条指令的目的在于把当前的运行模式切换成 32 位地址模式

```
protcseg:
```

```
  # Set up the protected-mode data segment registers
36  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
37  movw    %ax, %ds                # -> DS: Data Segment
38  movw    %ax, %es                # -> ES: Extra Segment
39  movw    %ax, %fs                # -> FS
40  movw    %ax, %gs                # -> GS
41  movw    %ax, %ss                # -> SS: Stack Segment
```

修改这些寄存器的值。这些寄存器都是段寄存器

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

接下来的指令就是要设置当前的 esp 寄存器的值，然后准备正式跳转到 main.c 文件中的 bootmain 函数处。我们接下来分析一下这个函数的每一条指令：

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

这里面调用了一个函数 readseg，这个函数在 bootmain 之后被定义了：

void readseg(uchar *pa, uint count, uint offset);

它的功能从注释上来理解应该是，把距离内核起始地址 offset 个偏移量存储单元作为起始，将它和它之后的 count 字节的数据读出送入以 pa 为起始地址的内存物理地址处。

所以这条指令是把内核的第一个页(4MB = 4096 = SECTSIZE*8 = 512*8)的内容读取的内存地址 ELFHDR(0x10000)处。其实完成这些后相当于把操作系统映像文件的 elf 头部读取出来放入内存中。

读取完这个内核的 elf 头部信息后，需要对这个 elf 头部信息进行验证，并且也需要通过它获取一些重要信息。

```
2 if (ELFHDR->e_magic != ELF_MAGIC)
3         goto bad;
```

elf 头部信息的 magic 字段是整个头部信息的开端。并且如果这个文件是格式是 ELF 格式的话，文件的 elf->magic 域应该是=ELF_MAGIC 的，所以这条语句就是判断这个输入文件是否是合法的 elf 可执行文件。

```
4 ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
```

我们知道头部中一定包含 Program Header Table。这个表格存放着程序中所有段的信息。通过这个表我们才能找到要执行的代码段，数据段等等。所以我们要先获得这个表。

这条指令就可以完成这一点，首先 elf 是表头起址，而 phoff 字段代表 Program Header Table 距离表头的偏移量。所以 ph 可以被指定为 Program Header Table 表头。

```
5 eph = ph + ELFHDR->e_phnum;
```

由于 phnum 中存放的是 Program Header Table 表中表项的个数，即段的个数。所以这步操作是吧 eph 指向该表末尾。

```
6 for (; ph < eph; ph++)
    // p_pa is the load address of this segment (as well
    // as the physical address)
7     readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

这个 for 循环就是在加载所有的段到内存中。ph->paddr 根据参考文献中的说法指的是这个段在内存中的物理地址。ph->off 字段指的是这一段的开头相对于这个 elf 文件的开头的偏移量。ph->filesz 字段指的是这个段在 elf 文件中的大小。ph->memsz 则指的是这个段被实际装入内存后的大小。通常来说 memsz 一定大于等于 filesz，因为段在文件中时许多未定义的变量并没有分配空间给它们。

所以这个循环就是在把操作系统内核的各个段从外存读入内存中。

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

e_entry 字段指向的是这个文件的执行入口地址。所以这里相当于开始运行这个文件。也就是内核文件。 自此就把控制权从 boot loader 转交给了操作系统的内核。

下面是在 terminal 上用 gdb 对 boot.s 程序跟踪，首先要设置一个断点，设置到 boot.S 程序开始运行的地方，因为我们之前已经介绍过，BIOS 会把 boot sector 复制到 0x7c00 地址处，所以 boot.S 的起始运行地址就是 0x7c00.

所以我们在 gdb 窗口中输入 b *0x7c00，然后再输入 c，表示继续运行到断点处，在这里我们输入

```
x/30i 0x7c00
```

把存放在 0x7c00 以及之后 30 字节的内存里面的指令反汇编出来，我们可以拿它直接和 boot.S 以及在 obj/boot/boot.asm 进行比较，如下：

```
(gdb) x/30i 0x7c00
=> 0x7c00:      cli
   0x7c01:      cld
   0x7c02:      xor    %ax,%ax
   0x7c04:      mov    %ax,%ds
   0x7c06:      mov    %ax,%es
   0x7c08:      mov    %ax,%ss
   0x7c0a:      in     $0x64,%al
   0x7c0c:      test   $0x2,%al
   0x7c0e:      jne    0x7c0a
   0x7c10:      mov    $0xd1,%al
   0x7c12:      out    %al,$0x64
   0x7c14:      in     $0x64,%al
   0x7c16:      test   $0x2,%al
   0x7c18:      jne    0x7c14
   0x7c1a:      mov    $0xdf,%al
   0x7c1c:      out    %al,$0x60
   0x7c1e:      lgdtw  0x7c64
   0x7c23:      mov    %cr0,%eax
```

首先是 obj/boot/boot.asm

```
00007c00 <start>:
.set CR0_PE_ON,      0x1         # protected mode enable flag

.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
    7c00:       fa                      cli
  cld                         # String operations increment
    7c01:       fc                      cld

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
    7c02:       31 c0                   xor    %eax,%eax
  movw    %ax,%ds             # -> Data Segment
    7c04:       8e d8                   mov    %eax,%ds
  movw    %ax,%es             # -> Extra Segment
    7c06:       8e c0                   mov    %eax,%es
  movw    %ax,%ss             # -> Stack Segment
    7c08:       8e d0                   mov    %eax,%ss

00007c0a <seta20.1>:
  # Enable A20:
```

然后是 boot.S

```
.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

  # 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
```

可见这三者在指令上没有区别，只不过在源代码中，我们指定了很多标识符比如 set20.1，.start，这些标识符在被汇编成机器代码后都会被转换成真实物理地址。比如 set20.1 就被转换为 0x7c0a，那么在 obj/boot/boot.asm 中还把这种对应关系列出来了，但是在真实执行时，即第一种情况中，就看不到 set20.1 标识符了，完全是真实物理地址。我观察到的差异有： `boot.S` 的指令含有表示长度的 b,w,l 等后缀，而 `boot.asm` 和 GDB 没有；同样一条指令，`boot.S` 和 GDB 是操作 ax 寄存器，而 `boot.asm` 却是操作%eax。

第二部分，对 bootmain 函数语句分析，
由前边讨论知道 bootloader 在 7c00 处开始加载，阅读 boot.asm 可以知道在 7c45 进入 bootmain 函数，用 x/i 命令到该地址，结果如下：

```
   0x7c3e:      mov    %ax,%ss
   0x7c40:      mov    $0x7c00,%sp
   0x7c43:      add    %al,(%bx,%si)
   0x7c45:      call   0x7d08
   0x7c48:      add    %al,(%bx,%si)
   0x7c4a:      jmp    0x7c4a
   0x7c4c:      add    %al,(%bx,%si)
```

读 boot.asm 文件知道 readrect 函数存在 7c7c 地址处，并在 readseg 函数里 diaoyo 能够，所以在 bootmain 中第一次调用 readseg 时首次调用 readsect，如下图：

```
bootmain(void)
{
    7d0a:       55                      push   %ebp
    7d0b:       89 e5                   mov    %esp,%ebp
    7d0d:       56                      push   %esi
    7d0e:       53                      push   %ebx
        struct Proghdr *ph, *eph;

        // read 1st page off disk
        readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
    7d0f:       6a 00                   push   $0x0
    7d11:       68 00 10 00 00          push   $0x1000
```

在 terminal 里反汇编出 0x7c7c 之后的 20 条指令，结果如下：

```
(gdb) x/20i 0x7c7c
   0x7c7c:      push   %bp
   0x7c7d:      mov    %sp,%bp
   0x7c7f:      push   %di
   0x7c80:      push   %bx
   0x7c81:      mov    0xc(%di),%bx
   0x7c84:      call   0x7c68
   0x7c87:      (bad)
   0x7c88:      (bad)
   0x7c89:      mov    $0x1f2,%dx
   0x7c8c:      add    %al,(%bx,%si)
   0x7c8e:      mov    $0x1,%al
   0x7c90:      out    %al,(%dx)
   0x7c91:      movzbw %bl,%ax
   0x7c94:      mov    $0xf3,%dl
   0x7c96:      out    %al,(%dx)
   0x7c97:      movzbw %bh,%ax
   0x7c9a:      mov    $0xf4,%dl
   0x7c9c:      out    %al,(%dx)
   0x7c9d:      mov    %bx,%ax
```

与 boot.asm 内比较可以发现一致，并在 0x7c84 处调用 0x7c68 处的函数，即调用 waitdisk 函数等待磁盘空闲响应，回到 bootmain 函数，接着读取磁盘剩余内核部分，也就是从第一次调用 readsect 之后再次调用 readsect 为读取内核其余部分的 for 循环的开始，读 boot.asm 文件可以知道从 7d61 开始循环，在此处设置断点并 continue 知道走完 bootloader，并在 QEMU 打印信息。

```
(gdb) b *0x7d61
Breakpoint 2 at 0x7d61
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d61:      call   *0x10018

Breakpoint 2, 0x00007d61 in ?? ()
(gdb)
```

由此知道在 0x7d61 后进入内核，查看内核第一条命令的物理地址如下

```
Breakpoint 2, 0x00007d61 in ?? ()
(gdb) x/2i 0x10018
   0x10018:     or      $0x0,%al
   0x1001a:     adc     %al,(%eax)
(gdb) x /1x 0x10018
0x10018:        0x0010000c
(gdb)
```

查看命令结果如下

```
(gdb) disassemble 0x10000c,0x10001c
Dump of assembler code from 0x10000c to 0x10001c:
   0x0010000c:  movw   $0x1234,0x472
   0x00100015:  mov    $0x110000,%eax
   0x0010001a:  mov    %eax,%cr3
End of assembler dump.
(gdb)
```

二、回答问题

1. 问：处理器从哪里开始执行 32 位代码？是什么导致了 16 位代码到 32 位代码的切换？ 答：
   - 处理器应该是从 `boot.S` 文件中的 `.code32` 伪指令开始执行 32 位代码。补充：ljmp 语句使得处理器从 real mode 切换到 protected mode，地址长度从 16 位变为 32 位。
   - 处理器由 16 位代码到 32 位代码的切换，主要是通过设置 cr0 寄存器的 PE 位（是否开启保护模式）和 PG 位（启用分段式还是分页式）来触发的。
2. 而以下指令完成了从实模式到保护模式的转换。cr0 寄存器的 0 位置 1。

```
[   0:7c23] => 0x7c23:  mov    %cr0,%eax
0x00007c23 in ?? ()

[   0:7c26] => 0x7c26:  or     $0x1,%eax
0x00007c26 in ?? ()

[   0:7c2a] => 0x7c2a:  mov    %eax,%cr0
0x00007c2a in ?? ()
```

12. 在 boot.c 对应以下代码：

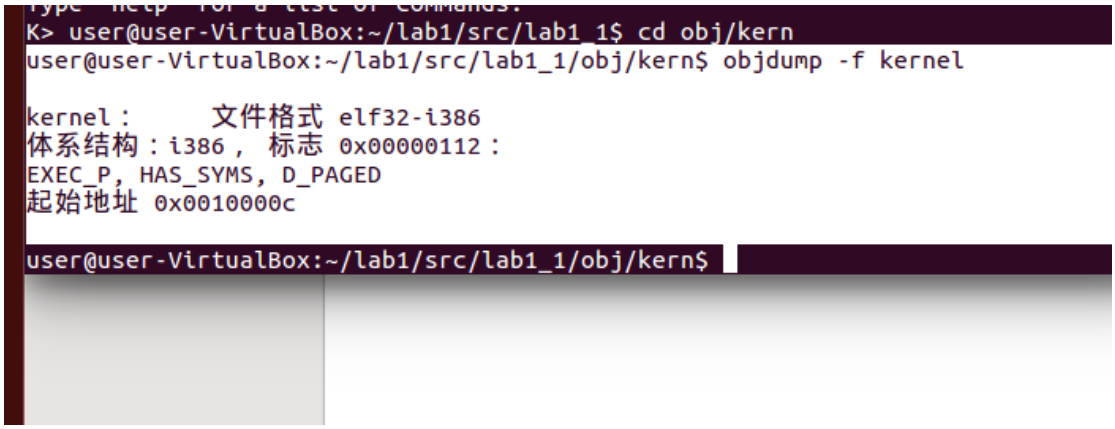
```
lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE_ON, %eax
  movl    %eax, %cr0

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

21.
22. .code32 # Assemble for 32-bit mode
23. 阅读可知 CR0 寄存器 PE 位设置为 1，则开启了 32 位保护模式，0 则是实模式
24.

25. 问：boot loader 执行的最后一条指令是什么？boot loader 加载内核后，内核的第一条指令是什么？ 答：

- boot loader 的最后一条指令是 `7d6b: ff 15 18 00 01 00 call *0x10018`
- 内核的第一条指令是 `0x10000c: movw $0x1234,0x472`

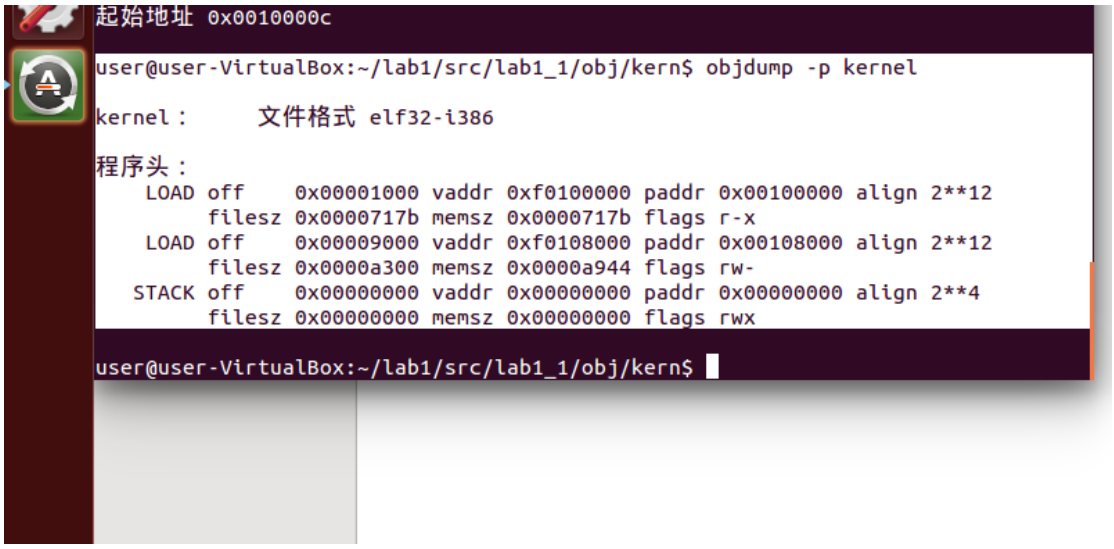
可以通过 objdump 命令查看：



26. 问：内核的第一条指令的地址在哪里？
答： 根据 gdb 调试结果，内核的第一条指令的地址为 0x10000c.

问：boot loader 怎么知道为了从磁盘中读取整个内核的内容需要加载多少扇区？它从哪里获得这个信息？

根据对 main.c 的分析，显然是通过 ELF 文件头获取所有 program header table，在 ELF 头文件里，变量 e_phoff 记录文件第一个 program header table 相对文件开始的偏移量，e_phnum 记录了每个 program header table 的数量。 program header table 记录了三个重要信息用以描述段 (segment)：p_pa (物理内存地址)，p_memsz (所占内存大小)，p_offset (相对文件的偏移地址)。根据这三个信息，对每个段，从 p_offset 开始，读取 p_memsz 个 byte 的内容（需要根据扇区(sector)大小对齐），放入 p_pa 开始的内存中。通过 objdump 命令可以查看到：



问题一：1. Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

Printf.c 为顶层输出的文文件，实现c 语言言的printf 函数；

Console .c 主要封装了了将字符打印到屏幕上的函数；

Printfmt.c 实现了了printf 的参数的输出以及输出格式；

三者的关系为：printf 通过调用用printfmt.c 的函数vprintfmt 实现输出串串的格式转换以及参数的输出。通过将console.c 的函数cputch 再封装为函数putch 将字符打印到屏幕上。

2. Explain the following from console.c:

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

crt_buf:这是一个字符数组缓冲区，里面存放着要显示到屏幕上的字符

crt_pos:这个表示当前最后一个字符显示在屏幕上的位置，在介绍这个变量前我们还要知道一些知识，这是我在网上自己查询的。

早期的计算机如果想显示信息给用户只能通过文字模式，比如当你现在打开电脑时，进入桌面之前，所有的信息都是通过文字显示在屏幕上的。那么这种模式就叫做文字模式，那么这个 console.c 源程序中考虑的就是一种非常常见的文字模式，80x25 文字模式，即整个屏幕上允许显示最多 25 行字符，每行最多显示 80 个字符。所以一共代表了 80x25 个位置。当我们要显示某

个特定字符到屏幕某个位置上面时，我们必须要指定显示的位置，和显示字符给屏幕驱动器 cga。

而在 console.c 文件中，子程序 cga_putc(int c)就是完成这项功能，把字符 c 显示到屏幕当前显示的下一个位置。比如当前屏幕中已经显示了三行数据(0 号行，1 号行，2 号行)，并且第三行已经显示了 40 个字符，此时执行 cga_putc(0x65)，那么就会把 0x65 对应的字符'A'显示到 2 号行第 41 个字符处。所以 cga_putc 需要两个变量，crt_buf，这个一个字符数组指针，该字符数组就是当前显示在屏幕上的所有字符。crt_pos 则表示下一个要显示的字符存放在数组中的位置，其实通过这个值也可以推导出它显示在屏幕上的位置。比如 crt_pos = 85，那么它就应该显示在第 2 行（即 1 号行），第 6 字符（5 号字符）处。所以 crt_pos 的取值范围应该是从 0~(80*25-1)。

上面题目中要分析的这段代码位于 cga_putc 中，cga_putc 的分为三部分，第一部分是根据字符值 int c 来判断到底要显示成什么样子。而第二部分就是上述代码。第三部分则是把你决定要显示的字符显示到屏幕的指定位置上。咱们具体分析第二部分，

当 crt_pos >= CRT_SIZE，其中 CRT_SIZE = 80*25，由于我们知道 crt_pos 取值范围是 0~(80*25-1)，那么这个条件如果成立则说明现在在屏幕上输出的内容已经超过了一页。所以此时要把页面向上滚动一行，即把原来的 1~79 号行放到现在的 0~78 行上，然后把 79 号行换成一行空格（当然并非完全都是空格，0 号字符上要显示你输入的字符 int c）。所以 memcpy 操作就是把 crt_buf 字符数组中 1~79 号行的内容复制到 0~78 号行的位置上。而紧接着的 for 循环则是把最后一行，79 号行都变成空格。最后还要修改一下 crt_pos 的值。

：联系代码上下文，可以理解这段代码的作用。首先，CRT(cathode ray tube)是阴极射线显示器。根据 console.h 文件中的定义，CRT_COLS 是显示器每行的字长（1 个字占 2 字节），取值为 80；CRT_ROWS 是显示器的行数，取值为 25；而`#define CRT_SIZE (CRT_ROWS * CRT_COLS)`是显示器屏幕能够容纳的字数，即 2000。当 crt_pos 大于等于 CRT_SIZE 时，说明显示器屏幕已写满，因此将屏幕的内容上移一行，即将第 2 行至最后 1 行（也就是第 25 行）的内容覆盖第 1 行至倒数第 2 行（也就是第 24 行）。接下来，将最后 1 行的内容用黑色的空格塞满。将空格字符、0x0700 进行或操作的目的是让空格的颜色为黑色。最后更新 crt_pos 的值。总结：这段代码的作用是当屏幕写满内容时将其上移 1 行，并将最后一行用黑色空格塞满。

作业 1：

> **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.**

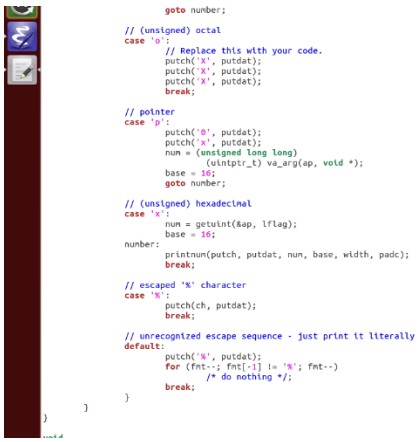
因为，每次进入jos时，总会输出：

```
6828 decimal is XXX octal!
```

所以，在kern/init.c中找到对应的代码：

```
cprintf("6828 decimal is %o octal!\n", 6828);
```

在lib/console.c中的代码



```
                goto number;

        // (unsigned) octal
        case 'o':
                // Replace this with your code.
                putch('X', putdat);
                putch('X', putdat);
                putch('X', putdat);
                break;

        // pointer
        case 'p':
                putch('0', putdat);
                putch('x', putdat);
                num = (unsigned long long)
                        (uintptr_t) va_arg(ap, void *);
                base = 16;
                goto number;

        // (unsigned) hexadecimal
        case 'x':
                num = getuint(&ap, lflag);
                base = 16;
        number:
                printnum(putch, putdat, num, base, width, padc);
                break;

        // escaped '%' character
        case '%':
                putch(ch, putdat);
                break;

        // unrecognized escape sequence - just print it literally
        default:
                putch('%', putdat);
                for (fmt--; fmt[-1] != '%'; fmt--)
                        /* do nothing */;
                break;
        }
    }
}
```

其中，将printfmt中的下列代码修改

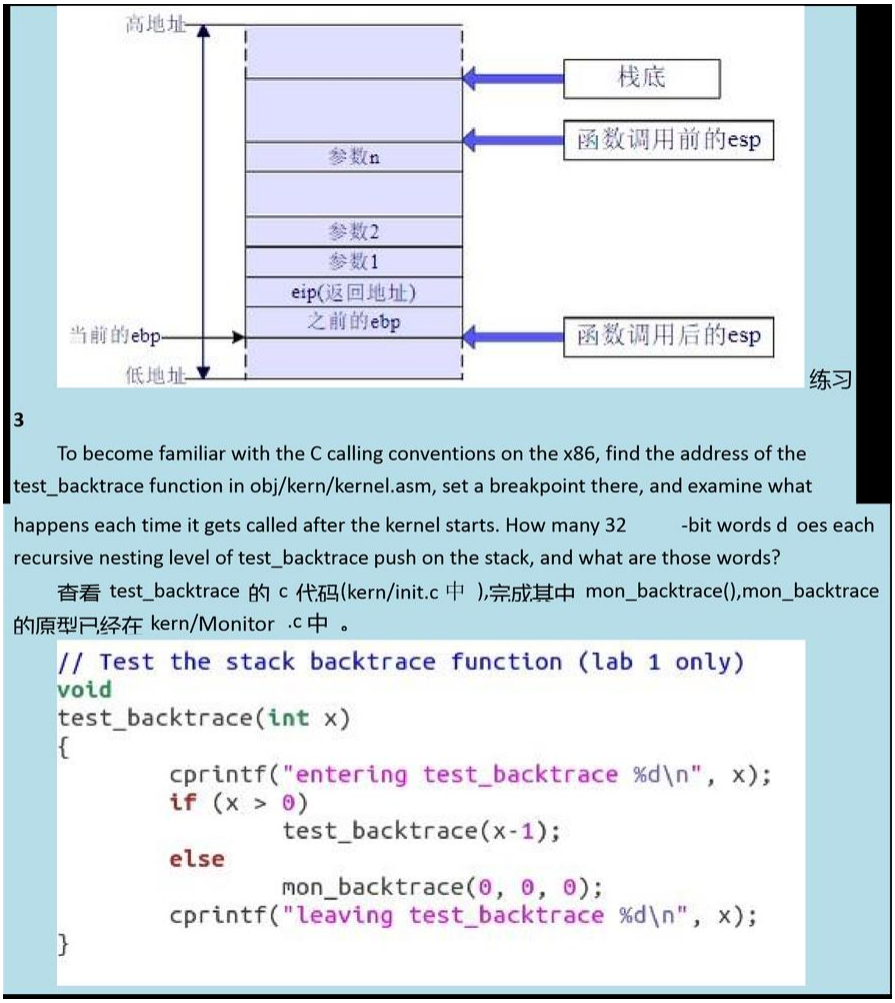
```
case 'o':
        // Replace this with your code.
        putch('X', putdat);
        putch('X', putdat);
        putch('X', putdat);
        break;
```

将它替换成8进制代码：

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

通过Vprintfmt()，识别%o时候，利用getunit获得输入数字，base改为8，跳转到number，将printnum处理成不同进制进行输出。

思考 2：



函数调用过程：（调用函数 和 被调函数）

阅读kern/init.c，在i386_init函数中找到：

```
test_backtrace(5);
```

这是一个递归调用：

```
void test_backtrace(int x)
 {
cprintf("entering test_backtrace %d\n", x);
if (x > 0) test_backtrace(x-1);
else mon_backtrace(0, 0, 0);
cprintf("leaving test_backtrace %d\n", x);
}
```

调用之前，调用者将参数从右到左压进栈，然后调用call，call会将eip即返回地

址入栈，同时转到被调函数；在被调函数预处理中，将调用函数的ebp入栈，然后将

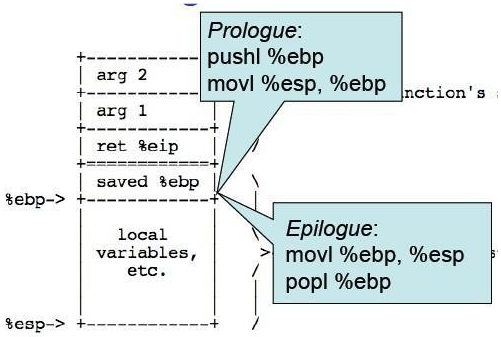
当前esp赋给ebp，再 将ebx入栈保护寄存器状态

在test_backtrace调用时，会压入4个32的字，分别是参数，返回地址，调用函数

的 ebp，ebx

所以，每次调用一共 4+4+4+20=32bytes，8 个双字入栈。

因为，在内存中栈底是固定的，栈顶是变化的，JOS中的函数调用后堆栈的结构：

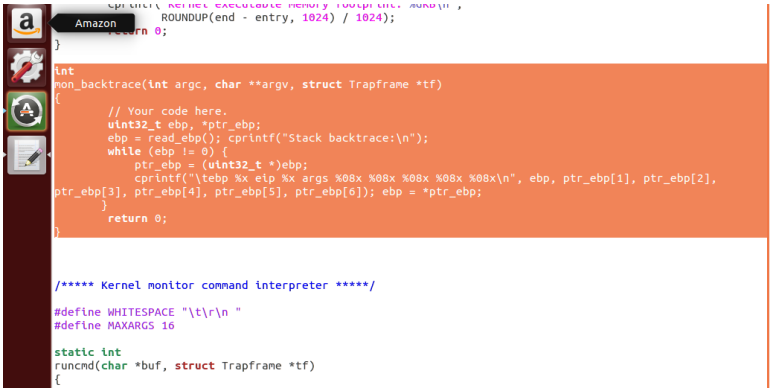


esp的含义是“这个地址以下的空间是未被使用的堆栈控件”，ebp的含义是“这个地址以下至esp的空间是属于目前所执行函数的堆栈空间。

通过阅读汇编代码可以发现，一个函数在调用之前，其调用者会将参数压栈，也就是压入arg2 和arg1，然后调用call，call的动作会把ret%eip压栈，同时转到函数体执行，在函数体执行的开头有一段预处理代码，即图中的prologue，会将ebp寄存器(call指令不改变ebp的值，此时的ebp还是上一个函数的）内容压栈，然后将当前esp赋值给ebp，随后进行现场保存的工作，存储在local variables空间里，值得注意的是，在预处理时会一下申请足够的空间，包括保存现场所需空间，局部变量所需空间，调用其它函数所压入变量的空间，意即图中arg1,arg2是属于上一个函数的local variables空间，故backtrace不能准确的判断出函数所传参数个数而统一要求打印出5个参数。

因此，通过ebp不断寻找上层的ebp，直到回溯所有的函数，在entry.S中可以看到，在调用i386_init之前，将ebp置0了，因此当ebp为0的时候就是函数返回的时候。

代码如下：



调试结果：

```
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
文件 t_backtrace 2
t_backtrace 1
entering test_backtrace 0
Stack_backtrace:
 ebp:0xf010ff18 eip:0xf0100087 args:0x00000000 0x00000000 0x00000000 0x00000000
0xf01009c2
            kern/init.c:19 test_backtrace+71
 ebp:0xf010ff38 eip:0xf0100069 args:0x00000000 0x00000001 0xf010ff78 0x00000000
0xf01009c2
            kern/init.c:16 test_backtrace+41
 ebp:0xf010ff58 eip:0xf0100069 args:0x00000001 0x00000002 0xf010ff98 0x00000000
0xf01009c2
            kern/init.c:16 test_backtrace+41
 ebp:0xf010ff78 eip:0xf0100069 args:0x00000002 0x00000003 0xf010ffb8 0x00000000
0xf01009c2
            kern/init.c:16 test_backtrace+41
 ebp:0xf010ff98 eip:0xf0100069 args:0x00000003 0x00000004 0x00000000 0x00000000
0x00000000
            kern/init.c:16 test_backtrace+41
 ebp:0xf010ffb8 eip:0xf0100069 args:0x00000004 0x00000005 0x00000000 0x00010094
0x00010094
            kern/init.c:16 test_backtrace+41
 ebp:0xf010ffd8 eip:0xf01000ea args:0x00000005 0x00001aac 0x00000644 0x00000000
0x00000000
            kern/init.c:43 i386_init+77
 ebp:0xf010fff8 eip:0xf010003e args:0x00111021 0x00000000 0x00000000 0x00000000
0x00000000
            kern/entry.S:83 <unknown>+0
leaving test_backtrace 0
```

挑战作业：

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:

阅读 kern/kdebug.c，并通过二分法查找到 stab 表确定的函数，进行添加：

```
        // file.  Search the whole file for the line number.
        info->eip_fn_addr = addr;
        lline = lfile;
        rline = rfile;
    }
    // Ignore stuff after the colon.
    info->eip_fn_namelen = strfind(info->eip_fn_name, ':') - info->eip_fn_name;


    // 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 {
        return -1;
    }


    // Search backwards from the line number for the relevant filename
    // stab.
    // We can't just use the "lfile" stab because inlined functions
```

在 kern/monitor.c 添加命令：

```
        // return -1 to force monitor to exit
        int (*func)(int argc, char** argv, struct Trapframe* tf);
};

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

};
#define NCOMMANDS (sizeof(commands)/sizeof(commands[0]))

/***** Implementations of basic kernel monitor commands *****/

int
mon_help(int argc, char **argv, struct Trapframe *tf)
{
        int i;

        for (i = 0; i < NCOMMANDS; i++)
                cprintf("%s - %s\n", commands[i].name, commands[i].desc);
        return 0;
}

int
mon_kerninfo(int argc, char **argv, struct Trapframe *tf)
{
        extern char _start[], entry[], etext[], edata[], end[];
```

以及更改 mon_backtrace 函数：

```
                ROUNDUP(end - entry, 1024) / 1024);
        return 0;
}

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

}



/***** Kernel monitor command interpreter *****/

#define WHITESPACE "\t\r\n "
#define MAXARGS 16

static int
```

C ▾ 制表符宽度：8 ▾ 行 78，列 2 插入

实际就是在这里调用 debuginfo_eip 这个函数。传入的 arg1 实际就是返回地址，即 ebp + 4 地址中所存的内容。另外就是注意一个输出方法：

printf("%.*s", length, string)

作用就是输出 string 的最多 length 个字符。较长的函数名因此可以较美观地显示。