

Operating System MIT 6.828 JOS lab1 报告

——陈灏 (1100012776)

Part 1. PC Bootstrap

1.2 Simulating the x86

出现过的问题：

1、编译 jos 缺少库：

因为我使用的机器是 64 位的机器，因此在对 jos 编译的时候报错提示：

lib/printfmt.c:42: 对‘__udivdi3’未定义的引用

lib/printfmt.c:50: 对‘__umoddi3’未定义的引用

通过查找资料发现缺少 32 位 gcc 库，因此安装 gcc-multilib, ia32-lib, lib32gcc1 lib32stdc++6 即可。

2、gdb 使用 qemu 联合调试，不能自动找到.gdbinit 加载 kernel。

在 gdb 中输入 source .gdbinit 即可。

1.3 The PC's Physical Address Space

Q: 该部分提到如今 x86 体系下 32-bit 机在可以支持超过 4GB 的 *physical RAM*, 这是如何做到的？

A: 通过查资料发现 Physical Address Extension 是 x86 处理器的一个新功能，一般增加了 4 位地址线以使得可访问的物理内存达到 64GB，但是实际上虚拟地址仍然是 32 位，因此对于单一进程来能够访问的地址空间依然为 4GB。

Exercise 2:

当计算机启动时，计算机会自动将 CS:IP 置为 f000:fff0，即指向地址 0xffff0 处，执行 BIOS，该位置下是一条跳转指令，跳转至 BIOS 更前一段的位置进行一系列初始化操作，从主引导扇区 (Master Boot Record) 中并读入 boot loader，将控制权转交给 boot loader。

Part 2: The Boot Loader

Exercise 3:

Q1: At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

A1: 在 boot/boot.s, line 48 ~ 51 使计算机从实模式变为保护模式. 在 boot/boot.S 的 line 57 .code32 表示开始采用 assemble for 32-bit mode

```
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    gdt_desc
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
```

Q1.5: `ljmp $PROT_MODE_CSEG, $protcseg` 为什么在 `.code32` 前?

A1.5: 在保护模式在, 程序通过段选择子+程序偏移来定位一个数据, 通过段选择子来查找全局/局部描述表的位置来得到真正的段地址, 以此需要将 CS 引向正确的 GDT index。GDT (全局描述符表) 中 d/b 位控制使用 code segment 为 16bit 或者 32bit code。而该位置下默认为 0, 使用 16 位地址以及 16 位或 8 位操作数。刚开始从实模式切换至保护模式, PC 还没有被更新, 因此依旧默认 d/b 位为 0。因此该指令只能执行在 16bit 编码下。

Q2: *What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?*

A2: boot/main.c line 60: `(void (*)(void)) (ELFHDR->e_entry)();` 这个为 boot loader 最后一条执行的指令。

通过 gdb 我们发现, kernel 执行的第一条指令是: `0x10000c: movew $0x1234, 0x472`。即为 kernel/entry.S 中的 line 44。

Q3: *Where is the first instruction of the kernel?*

A3: `0x10000c: movew $0x1234, 0x472`

Q4: *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?*

A4: boot/main.c 就是 boot loader 将 kernel 读入进内存。从代码中可以知道 `ELFHDR->e_phnum, p_memsz` 里面的信息决定了需要读多少。通过资料可知: `ELFHDR->e_phnum` 指的是程序头部个数, `p_memsz` 标识了该段在内存中的长度。

Exercise 4:

注意:

对于指针操作, 一个容易出错的地方就是在于: `int a[2];`

1、`(a + 1)` 指向的是 `a[1]` 的地址。当然 `a++`, `++a` 也是指向 `a[1]` 的地址。

2、`(unsigned)((&a[1]) - (&a[0])) = 1`。

此处的意思在于指针的移位操作依赖于指针的类型。

Q: `pointers.c` 中的输出 line 5 似乎出现了问题的原因是什么?

A: 有了上面注意的理解, 我们发现 `pointers.c` 中有这样一句话:

`(int* c; int a[4]; 并且执行下面语句之前 c = &a[1];)`

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

`*c = 500;`

因此 `(char *)c + 1` 指向的并不是 `&a[2]`, 而是在指向比 `&a[1]` 恰好大 1 的地址, 因此对该处赋值就会造成不一样的效果。

Exercise 5:

我将 boot/Makefrag 中链接地址由 `0x7C00` 改为 `0x7C08`, 编译正常, 运行在

`ljmp $0x8, $0x7c3a` 处出现 Triple fault. 即 boot/boot.S 中的 line 55。出现原因分析: 由于链接修改了跳转地址, 使得跳转至错误的地址, 导致 Triple fault。

```

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    gdt_desc
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

```

Exercise 6:

```

(gdb) c
Continuing.
[ 0:7c00] => 0x7c00: cli
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
0x100000:  0x00000000  0x00000000  0x00000000  0x00000000
0x100010:  0x00000000  0x00000000  0x00000000  0x00000000
(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  0x6000b812  0x220f0011  0xc0200fd8

```

由于通过分析 main.c 我们可以知道，boot loader 将 kernel 从硬盘中读入到内存中，之后转入执行，而读入到内存中的地址由 kernel 的目标文件指定的 load address 决定，恰好就由 0x00100000 开始。因此在第二个断点处该地址下即为 kernel 的代码。

Part 3: The Kernel

Exercise 7:

通过 gdb 调试，从 JOS kernel 执行到 `movl %eax, %cr0` 之前 0xf0100000 中的内容均为 0xffffffff 0xffffffff ...。当执行完 `movl %eax, %cr0` 之后，0xf0100000 开始的内容和 0x00100000 开始的内容完全一致。通过分析程序我们可以发现，这条语句实际上打开了页寻址的机制。

通过 entry.S 中的注释可以知道其将[0, 4MB)与[KERNBASE, KERNBASE+4MB)的虚拟地址映射到了[0, 4MB)的物理地址上。而 KERNBASE 恰好为 0xf0000000 因此当页寻址打开后，将看到两段虚拟内存映射的内容相同。

注释掉之后 movl %eax, %cr0 后：

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

在 jmp *%eax 后就产生了 Triple fault，因为没有开启页寻址机制，而又不存在 0xf010002c 的高地址，因此就崩溃了。

Exercise 8:

%o 的方式和%d 的方式几乎类似，只是 base = 8 即可。

```
// (unsigned) octal
case 'o':
    // (MIT 6.828, lab1, Ex.8) my code :
    num = getint(&ap, lflag);
    if ((long long) num < 0) {
        putch('-', putdat);
        num = -(long long) num;
    }
    base = 8;
    goto number;
```

Q1: Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

A1: console.c 和硬件打交道，处理与显示端口的 I/O。printf.c 中的 cprintf 为程序使用的接口，putch(int ch, int *cnt)用 console.c 中的 cputchar(ch)进行输出。

Q2: Explain the following from console.c:

A2: 即当输入的列数内容超过屏幕能够显示的列数的时候，将最上面一行删去，整体向上平移一行，这样使得能够存放下新的一行信息。

Q3: In the call to cprintf(), to what does fmt point? To what does ap point? List (in order of execution) each call to cons_putc, va_arg, and vcprintf. For cons_putc, list its argument as well. For va_arg, list what ap points to before and after the call. For vcprintf list the values of its two arguments.

A3: fmt point to 0xf0101b64, which is address of string "x %d, y %d, z %d\n".
ap point to stack that is cprintf arguments.

By GDB, I get the lists:

vcprintf(fmt, ap)	fmt="x %d, y %d, z %d\n", ap = 0xf0115fd4 (= &x)
cons_putc	c=120
cons_putc	c=32
va_arg	ap=0xf0115fd4 (= &x) → 0xf0115fd8 (= &y)
cons_putc	c=49
cons_putc	c=44
cons_putc	c=32
cons_putc	c=121
cons_putc	c=32
va_arg	ap=0xf0115fd8 (= &y) → 0xf0115fdc (= &z)
cons_putc	c=51
cons_putc	c=44
cons_putc	c=32
cons_putc	c=122
cons_putc	c=32
va_arg	ap=0xf0115fdc (= &z) → 0xf0115fe0
cons_putc	c=52
cons_putc	c=10

Q4: What is the output

A4: 输出"He110 World", 原因在于 57616 的 16 进制就为 e110, 而 &i 地址下的值为 0x72 0x6c 0x64 0x00, 转化为字符串即为"rld\0"。当然这取决于 x86 为

Little-endian。

如果是 Big-endian, 则 $i = 0x726c6400, 57616$ 不需要改变。

Q5: *What is going on when lack "y"*

A5: ap 在输出完 x 后, 会将 ap 的地址移向下一个输出数的地址, 而这个地址是存放在栈上的, ap 通过取栈中数来取输出数的地址。如果 cprintf 上缺少参数, 则 ap 会取出相应位置栈上信息, 于是会输出该位置作为地址下的值。

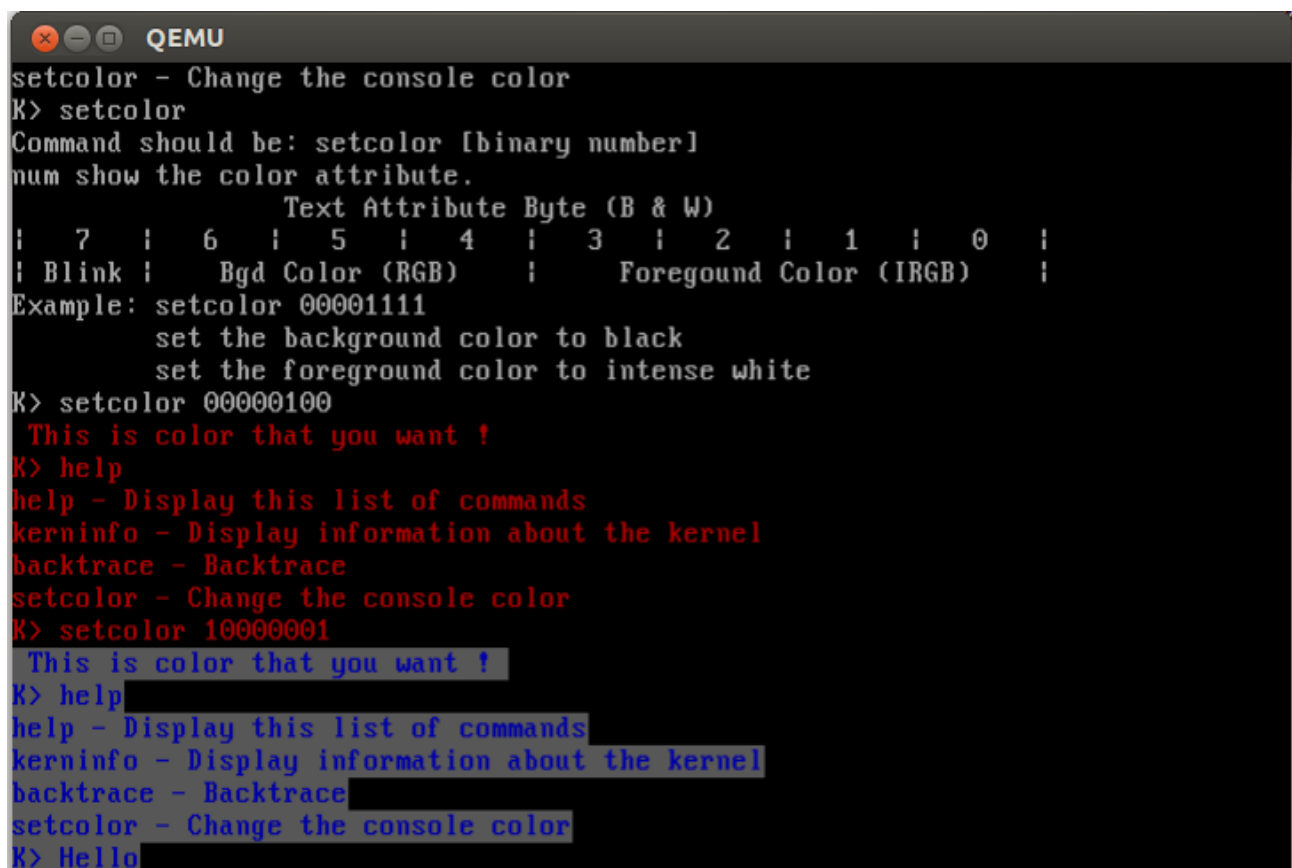
Q6: *Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?*

A6: 若改变压栈的顺序, 则应该首先让 ap 指向最后一个参数, 并且输出了一个数据后, 应该将 ap 进行自减而不是像原来那样自加, 使得指向下一个参数。

Challenge:

在 cons_putc 为 output a character to the console, 因此观察里面所用到的函数, 在 cga_putc(c) 中, 注释给了提示: // if no attribute given, then use black on white. 通过试验, 发现更改高 8 位确实可以更改颜色。

通过查阅资料, 得知设置颜色的方法, 为高 8 位中的前 4 位控制 Bgd color, 后四位控制 foreground color(text color). 因此我在 monitor.c 中增加了修改颜色的命令, 在 monitor.c 中设置了一个变量 int user_setcolor 表示目前设置的颜色, 并在 monitor.c 中使用了 extern int user_setcolor 来在输出的时候进行设置颜色。具体颜色设置方案即在 cga_putc 函数中第一句加上 $c |= (user_setcolor << 8)$ 即可。具体演示如下:



```
QEMU
setcolor - Change the console color
K> setcolor
Command should be: setcolor [binary number]
num show the color attribute.

      Text Attribute Byte (B & W)
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Blink | Bgd Color (RGB) | Foreground Color (IRGB) |
Example: setcolor 00001111
        set the background color to black
        set the foreground color to intense white
K> setcolor 00000100
This is color that you want !
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
backtrace - Backtrace
setcolor - Change the console color
K> setcolor 10000001
This is color that you want !
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
backtrace - Backtrace
setcolor - Change the console color
K> Hello
```

Exercise 9:

从 kern/entry.S 中可以看出：

```
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
84
85
86 .data
87 #####
88 # boot stack
89 #####
90 .p2align    PGSHIFT    # force page alignment
91 .globl      bootstack
92 bootstack:
93 .space      KSTKSIZE
94 .globl      bootstacktop
95 bootstacktop:
96
```

entry.S 通过定义数据段一个 bookstack 留了 KSTKSIZE 大小的空间，而 bootstacktop 恰好就处在栈的底部（栈的最高地址处），于是变通过 movl 指令使得 %esp = bootstacktop 即初始化了 %esp，指向栈的底部。

通过 kernel.asm 中可以清楚地发现 bootstacktop = 0xf0116000，即 kernel 栈开始的位置。

Exercise 10:

在 0xf0100040 处设置了断点，分别取了刚进入 test_backtrace 时,x=5,4,3 的时候寄存器状态：

b *0xf0100040	x = 5	x = 4	x = 3
ebp	0xf0115ff8	0xf0115fc8	0xf0115fa8
esp	0xf0115fcc	0xf0115fac	0xf0115f8c

通过 kernel.asm 中代码以及观察验证，每曾递归需要\$0x20 个字节，也就是 8 个 32-bit words。

通过 kernel.asm 的分析，对于每层递归使用的栈中的大致情景如下：（假设 x > 1）

return address
old ebp
old ebx
x (cprintf args)
0xf01019a0 (cprintf args)
...
...
x-1 (next level args)

Exercise 11:

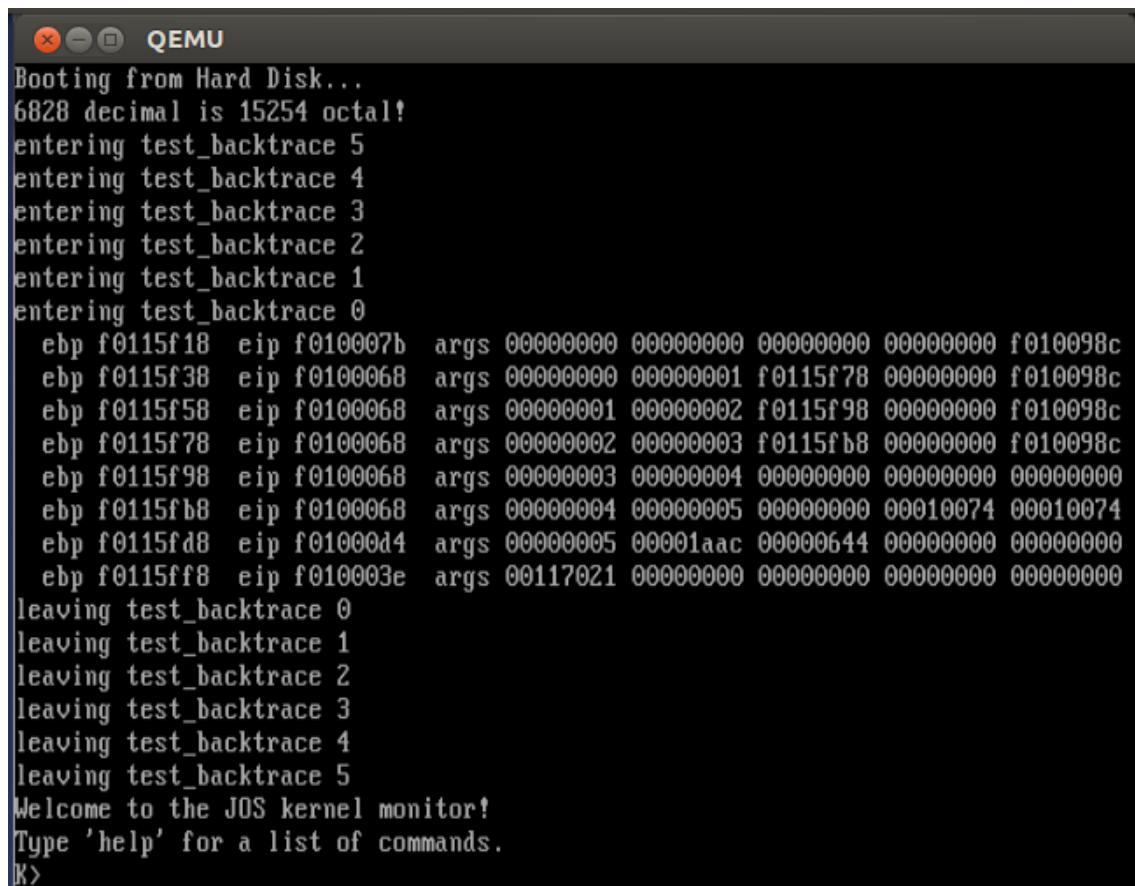
栈的结构:

arg_n
...
arg_1
arg_0
Return address
old ebp (new ebp point two this address)
...

因此通过函数获取当前的 ebp 值, 则*(ebp)就为更低一层递归的 old ebp 的地址。因此不断 while 利用 ebp 来得到栈信息即可。而结束条件为 ebp = 0, 这个可以在 entry.S 中 call i386_init 前, 将 0 赋给了 %ebp。
因此, 代码如下:

```
60
61 int
62 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
63 {
64     uint32_t* ebp = (uint32_t*)read_ebp();
65     uint32_t eip;
66
67     // in entry.S show the top ebp = 0
68     for (; ebp != 0; ebp = (uint32_t*)(*ebp)) {
69         eip = *(ebp + 1);
70         // arg[i] = *(ebp + 2 + i);
71         cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n", ebp, eip, *(ebp+2), *(ebp+3), *(ebp+4), *(ebp+5), *(ebp+6));
72     }
73     return 0;
74 }
75
76
```

结果如下:



```
QEMU
Booting from Hard 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
  ebp f0115f18  eip f010007b  args 00000000 00000000 00000000 00000000 f010098c
  ebp f0115f38  eip f0100068  args 00000000 00000001 f0115f78 00000000 f010098c
  ebp f0115f58  eip f0100068  args 00000001 00000002 f0115f98 00000000 f010098c
  ebp f0115f78  eip f0100068  args 00000002 00000003 f0115fb8 00000000 f010098c
  ebp f0115f98  eip f0100068  args 00000003 00000004 00000000 00000000 00000000
  ebp f0115fb8  eip f0100068  args 00000004 00000005 00000000 00010074 00010074
  ebp f0115fd8  eip f01000d4  args 00000005 00001aac 00000644 00000000 00000000
  ebp f0115ff8  eip f010003e  args 00117021 00000000 00000000 00000000 00000000
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>
```


Exercise 12:

在 kernel.ld 中有这样一段代码：

```
23 }
24
25 /* Include debugging information in kernel memory */
26 .stab : {
27     PROVIDE(__STAB_BEGIN__ = .);
28     *(.stab);
29     PROVIDE(__STAB_END__ = .);
30     BYTE(0) /* Force the linker to allocate space
31             for this section */
32 }
33
34 .stabstr : {
35     PROVIDE(__STABSTR_BEGIN__ = .);
36     *(.stabstr);
37     PROVIDE(__STABSTR_END__ = .);
38     BYTE(0) /* Force the linker to allocate space
39             for this section */
38.2-8
```

通过查找资料，PROVIDE: The PROVIDE keyword may be used to define a symbol, such as `etext', only if it is referenced but not defined. The syntax is PROVIDE(symbol = expression).

因此__STAB_*的定义就在 kernel.ld 上完成了：__STAB_BEGIN__ __STAB_END__指向.stab 的开头和结尾，同理__STABSTR_BEGIN__

为了实现 backtrace，只需要在 kdebug.c 中将 eip_line 的寻找补充完整。这个利用 stab_binsearch 即可。

具体代码如下：

```
172
173 // Search within [lline, rline] for the line number stab.
174 // If found, set info->eip_line to the right line number.
175 // If not found, return -1.
176 //
177 // Hint:
178 // There's a particular stabs type used for line numbers.
179 // Look at the STABS documentation and <inc/stab.h> to find
180 // which one.
181 // Your code here.
182 lfun = lline;
183 rfun = rline;
184 stab_binsearch(stabs, &lfun, &rfun, N_SLINE, addr);
185 if (lfun <= rfun) {
186     // stab[lfun] points to right SLINE entry
187     info->eip_line = stabs[lfun].n_desc;
188     lline = lfun;
189     rline = rfun;
190 } else {
191     // not found
192     return -1;
193 }
194
```

并且将 monitor.c 修改为：

```

60
61 int
62 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
63 {
64     uint32_t* ebp = (uint32_t*)read_ebp();
65     uint32_t eip;
66     struct Eipdebuginfo eip_debug_info;
67
68     // in entry.S show the top ebp = 0
69     for (; ebp != 0; ebp = (uint32_t*)(*ebp)) {
70         eip = *(ebp + 1);
71         // arg[i] = *(ebp + 2 + i);
72         cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n", ebp, eip, *(ebp+2), *(ebp+3), *(ebp+4), *(ebp+5), *(ebp+6));
73
74         debuginfo_eip(*(ebp + 1), &eip_debug_info);
75         cprintf("    %s:%d: ", eip_debug_info.eip_file, eip_debug_info.eip_line);
76         cprintf("%.s", eip_debug_info.eip_fn_namelen, eip_debug_info.eip_fn_name);
77         cprintf("+%u\n", (unsigned int)(eip - eip_debug_info.eip_fn_addr));
78     }
79     return 0;
80 }
81

```

结果如下：

```

QEMU
kern/init.c:16: test_backtrace+40
ebp f0115f98 eip f0100068 args 00000003 00000004 00000000 00000000 00000000
kern/init.c:16: test_backtrace+40
ebp f0115fb8 eip f0100068 args 00000004 00000005 00000000 00010074 00010074
kern/init.c:16: test_backtrace+40
ebp f0115fd8 eip f01000d4 args 00000005 00001aac 00000644 00000000 00000000
kern/init.c:48: i386_init+64
ebp f0115ff8 eip f010003e args 00117021 00000000 00000000 00000000 00000000
kern/entry.S:83: <unknown>+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> backtrace
ebp f0115f68 eip f01009a2 args 00000001 f0115f80 00000000 f0115fc8 f0118540
kern/monitor.c:163: monitor+246
ebp f0115fd8 eip f01000e1 args 00000000 00001aac 00000644 00000000 00000000
kern/init.c:52: i386_init+77
ebp f0115ff8 eip f010003e args 00117021 00000000 00000000 00000000 00000000
kern/entry.S:83: <unknown>+0
K> _

```

总结：

至此JOS Lab1 就完结了，估算时间：配置系统花费了大概 4 个小时，Lab 时间大概 10 个小时，报告大概用时 6 小时，总共用时大约为 20 小时。

通过 JOS Lab1，最大的收获就是清楚的知道了一个操作系统是如何跑起来的，通过代码和查资料，对于一个操作系统的启动有了更加细致的认识。

通过观察分析代码以及助教给我们的讲解，我了解到了即使是再细小的细节，再短的代码里面都蕴含着操作系统者的智慧，以及各种计算机技术。

最后感谢助教，同学的各种帮助，以及张驰师兄的报告参考。