

实验报告

lab1

姓名：李彤

学号：14307130132

注意事项：

- 1.请在每个 **exercise** 之后简要叙述实验原理，详细描述实验过程。
- 2.请将你认为的关键步骤附上必要的截图。
- 3.有需要写代码的实验，必须配有代码、注释以及对代码功能的说明。
- 4.你还可以列举包括但不限于以下方面：实验过程中碰到的问题、你是如何解决的、实验之后你还留有
哪些疑问和感想。
- 5.请在截止日期前将代码和报告上传到 **ftp** 的指定目录下，文件名为 **os_lab1_学号.zip**，该压缩文件中应包
含实验报告和代码，其中实验报告格式为 **pdf**，置于压缩文件的根目录。
- 6.如果实验附有 **question**，请在对应 **exercise** 后作答，这是实验报告评分的重要部分。
- 7.**Challenge** 为加分选做题。每个 **lab** 可能有多个 **challenge**，我们会根据完成情况以及难度适当加分，这部
分的实验过程描述应该比 **exercise** 更加详细。

Exercise 1. Familiarize yourself with the assembly language materials available on [the 6.828 reference page](#). You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.

We do recommend reading the section "The Syntax" in [Brennan's Guide to Inline Assembly](#). It gives a good (and quite brief) description of the AT&T assembly syntax we'll be using with the GNU assembler in JOS.

Exercise 2. Use GDB's **si** (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at [Phil Storrs I/O Ports Description](#), as well as other materials on the [6.828 reference materials page](#). No need to figure out all the details - just the general idea of what the BIOS is doing first.

0xffff0 :	ljmp \$0xf000,\$0xe05b	设置 cs 段选择符为 0xf000, eip 为 0xe05b, 跳转到 0xf000<<4+0xe05b=0xfe05b
0xfe05b:	cmpl \$0x0,%cs:0x6ac8	比较 0 和地址 cs<<4+0x6ac8 处的值, 不相等则跳转到 0xfd2e1
0xfe066:	xor %dx,%dx	将 dx 寄存器清空 (同时说明上一条指令没有跳转)
0xfe068:	mov %dx,%ss	给 ss 段选择符赋值为 0
0xfe06a:	mov \$0x7000,%esp	
0xfe070:	mov \$0xf34c2,%edx	
0xfe076:	jmp 0xfd15c	跳转到 0xfd15c
0xfd15c:	mov %eax,%ecx	
0xfd15f:	cli	将 IF 标志置为 0, 关闭中断
0xfd160:	cld	将 DF 标志位置为 0
0xfd161:	mov \$0x8f,%eax	先将 eax 置为 0x8f, 下一步送端口 0x70
0xfd167:	out %al,\$0x70	将 al 寄存器值送端口 0x70, 查阅资料 0x70 端口 CMOS RAM index register port, 资料中说 bit7=1 则 NMI disabled, 关闭了不可屏蔽中断, 其余 7 位为 index
0xfd169:	in \$0x71,%al	将端口 0x71 的值装入 al 寄存器
0xfd16b:	in \$0x92,%al	将端口 0x92 的值装入 al 寄存器 (似乎覆盖了上条语句?)
0xfd16d:	or \$0x2,%al	
0xfd16f:	out %al,\$0x92	等价与将端口 0x92 的值 or 了 2, 查阅资料端口 0x92 对应的是 PS2/system control port, 将 bit1 设为 1 表示 A20 激活
0xfd171:	lidtw %cs:0x6ab8	将地址为 cs<<4+0x6ab8 处的 6 个 byte 装入 IDTR, 分别为 base 和 limit

0xfd177: lgdtw %cs:0x6a74 将地址为 cs<<4+0x6a74 处的 6 个 byte 装入 GDTR, 分别为 base 和 limit

.....

由上述指令, 可推测 BIOS 主要在为一些设备做初始化, 同时设置全局描述符表和中断描述符表等。

Exercise 3. Take a look at the [lab tools guide](#), especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

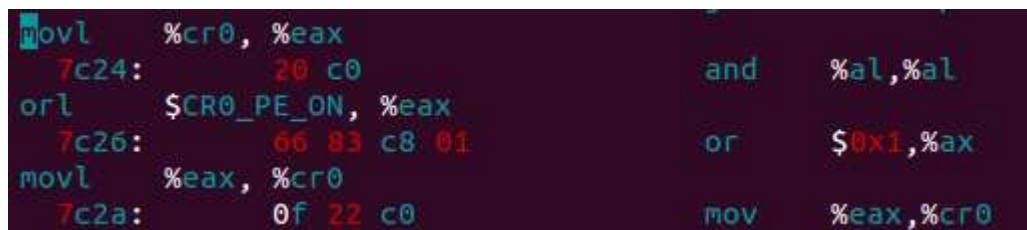
Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `boot/boot.S`, using the source code and the disassembly file `obj/boot/boot.asm` to keep track of where you are. Also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.

Trace into `bootmain()` in `boot/main.c`, and then into `readsect()`. Identify the exact assembly instructions that correspond to each of the statements in `readsect()`. Trace through the rest of `readsect()` and back out into `bootmain()`, and identify the begin and end of the `for` loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

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

执行完指令 0x7c2d: `ljmp $0x8, $0x7c32` 后开始执行 32 位代码。(其中 0x8 为 CS 段选择符的值对应 index=1, 根据此到 GDT 中寻找对应段描述符, 发现 base 为 0, 加上段内偏移量 0x7c32, 下一条指令从 0x7c32 开始执行)

将 `cr0` 寄存器的 `protected-enable` 标志位置为 1 完成了从 16 位模式到 32 位模式的切换。(具体通过下列指令完成)



```
movl    %cr0, %eax
7c24:    20 c0                and    %al, %al
orl     $CR0_PE_ON, %eax
7c26:    66 83 c8 01          or     $0x1, %ax
movl    %eax, %cr0
7c2a:    0f 22 c0             mov    %eax, %cr0
```

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

boot loader 执行的最后的一条指令:

```
((void (*)(void)) (ELFHDR->e_entry))();  
7d61:      ff 15 18 00 01 00      call    *0x10018
```

kernel 执行的第一条指令:

```
(gdb) si  
> 0x10000c:      movw    $0x1234,0x472
```

Where is the first instruction of the kernel?

```
(gdb) si  
> 0x10000c:      movw    $0x1234,0x472
```

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?

在 ELF 文件中有一部分为程序头表, 每个表项对应可执行文件的一个段, 每个表项中储存了该段在文件中的偏移位置, 该段的大小。boot loader 根据这些信息决定要装入多少个扇区到内存。

ELF 头文件中有程序头表在文件中的偏移量及每个表项所占空间, 根据这些信息可以找到每个段对应的表项, 表项中存储了 boot loader 装入 kernel 所需要的该段在文件中的偏移量等信息。

Exercise 4. Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an [Amazon Link](#)) or find one of [MIT's 7 copies](#).

Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for [pointers.c](#), run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in lines 1 and 6 come from, how all the values in lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.

There are other references on pointers in C (e.g., [A tutorial by Ted Jensen](#) that cites K&R heavily), though not as strongly recommended.

Warning: Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

```
1: a = 0x7fff0918b130, b = 0x1b8e010, c = 0x40074d
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7fff0918b130, b = 0x7fff0918b134, c = 0x7fff0918b131
```

第一行输出的三个为数组 `a` 的地址，`malloc` 函数返回的地址（存在指针变量 `b` 中），指针变量 `c` 存储的值（而不是指针 `c` 本身的地址，所以和 `a` 不连续）

第二行结果解释：for 循环将 `a` 数组分别设置为 100,101,102,103,而将 `a` 的值赋给 `c` 后，`c[0]=200` 等价与 `a[0]=200`,即得第二行结果。

第三行结果解释：`c[1]=300` 等价于 `a[1]=300`,`*(c+2)=301` 等价于 `c[2]=301` 等价于 `a[2]=301`,
`3[c]=302` 等价于 `c[3]=302` 等价于 `a[3]=302`,即得第三行结果。

第四行结果解释：`c=c+1` 后指针变量 `c` 存储的为 `a[1]` 的地址，所以 `*c=400` 等级于 `a[1]=400`，即得第四行结果。

第五行结果解释：将 `c` 强制类型转换后+“1”则 `c` 的值确实只加了 1,再强制类型转换为 `int *` 后则作为整形指针指向的是 `a[1]` 的三个字节和 `a[2]` 的一个字节构成的“整数”，所以赋值出现问题。

赋值前 `a[1]`,`a[2]` 在内存中的值：（little endian）

0000 1001 1000 0000 0000000000000000 | 1011 0100 1000 00000000000000000000

赋值后：

0000 1001 0010 1111 1000 000000000000 | 0000 0000 1000 00000000000000000000

`a[1]`,`a[2]` 分别为 `0x1f490=128144`,`0x100=256`,符合实际结果。

第六行结果解释：

对一个指针变量+“1”则该变量实际的值会加上指针变量对应的类型长度。所以 `b=a+4`,`c=a+1`。其中 `4=sizeof(int)`,`1=sizeof(char)`。

Exercise 6. We can examine memory using GDB's `x` command. The [GDB manual](#) has full details, but for now, it is enough to know that the command `x/Nx ADDR` prints N words of memory at `ADDR`. (Note that both 'x's in the command are lowercase.) *Warning:* The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in `xorw`, which stands for word, means 2 bytes).

Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at `0x00100000` at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

```
(gdb) x/8x 0x00100000
0x100000:      0x00000000      0x00000000      0x00000000      0x00000000
0x100010:      0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:      movw    $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x00100000
0x100000:      0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:      0x34000004      0x0000b812      0x220f0011      0xc0200fd8
```

`0x100000` 为 kernel 的装入位置，当 boot loader 刚开始运行时，kernel 还没有被装入，所以全部为 0，当 kernel 刚开始运行时，kernel 已经装入内存，所以 `0x100000` 开始对应的为 kernel 的二进制文件内容，与 `obj/kern/kernel` 反汇编得到的文件符合。

Exercise 7. Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at `0x00100000` and at `0xf0100000`. Now, single step over that instruction using the **stepi** GDB command. Again, examine memory at `0x00100000` and at `0xf0100000`. Make sure you understand what just happened. What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the `movl %eax, %cr0` in `kern/entry.S`, trace into it, and see if you were right.

在 `movl %eax, %cr0` 前:

```
(gdb) x/4x 0x100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000 0x00000000 0x00000000 0x00000000
```

在 `movl %eax, %cr0` 后

```
(gdb) x/4x 0xf0100000
0xf0100000 <_start+4026531828>: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
(gdb) x/4x 0x100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
```

`mov %eax %cr0` 将调整后的值赋给 `cr0`，开启了 `cr0` 寄存器中的 `page` 标志位，启动了分页机制。将 `0xf0100000` 开始的虚拟内存映射到物理内存 `0x00100000`。所以开启分页后 `0xf0100000` 开始的虚拟内存中内容与 `0x00100000` 开始的物理内存中相同。

将 `mov %eax %cr0` 注释掉后，执行地址为 `0xf010002c` 的指令出错，因为没有开启分页机制，导致虚拟内存 `0xf010002c` 没有对应的物理内存。

```
(gdb) si
=> 0xf010002c <relocated>: add %al, (%eax)
relocated () at kern/entry.S:74
74      movl $0x0, %ebp # nuke frame pointer
```


Exercise 8

代码如图:

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

Question1

console.c 向高层提供了 cputchar, getchar, iscons 三个函数, printf.c 中 putchar 函数中使用了 console.c 提供的 cputchar 函数。

Question 2

```
#define CRT_ROWS      25
#define CRT_COLS      80
#define CRT_SIZE      (CRT_ROWS * CRT_COLS)
```

由 console.h 中得, CRT_ROWS 是行数, CRT_COLS 是列数, 即每行的字符数, 则 CRT_SIZE 为整个屏幕上最多的字符数, 所以 if 语句是在判断当前已打印的字符是否已经超过屏幕上限, 而 memmove 函数是将当前屏幕上的字符整体向上平移一行, 最初的第一行被覆盖, 同时原先的最后一行会出现 2 次, 所以用一个 for 循环将重复的一行清零, 最终将位置指向平移后的最后一行的末尾。

Question 3

fmt 是一个字符指针, 指向字符串“x %d, y %x, z %d\n”,
使用 gdb 查看 fmt, 同时查询 fmt 指向地址的内容,

```
cprintf (fmt=0xf0101db8 "x %d, y %x, z %d\n")
(gdb) x/s 0xf0101db8
0xf0101db8:      "x %d, y %x, z %d\n"
```

符合。

ap 指向 cprintf 的其余参数, x, y, z。

调用顺序:

vcprintf(f0101db8, f010ff74)

(f0101db8 为字符串指针, 指向字符串“x %d, y %x, z %d\n”, f010ff74 为 x 的地址, 内容为 1)

cons_putc('x')

cons_putc(' ')

va_arg (before f010ff74(x=1) after f010ff78(y=3))

cons_putc('1')

cons_putc(',')

```

cons_putc(' ')
cons_putc('y')
cons_putc(' ')
va_arg (before f010ff78(y=3), after f010ff7c(z=4))
cons_putc('3')
cons_putc(',')
cons_putc(' ')
cons_putc('z')
cons_putc(' ')
va_arg (before f010ff7c(z=4) after f010ff80)
cons_putc('4')
cons_putc('\n')

```

Question 4

He110 World

结果如图，解释：

57616 等于十六进制下的 e110，%x 以十六进制输出，所以有 He110，%s 以字符串形式输出给定地址中的字符串，传入的参数为 i 的地址，i=0x00646c72,每个字节对应一个字符，分别为\0,d,l,r,而 x-86 为小端，所以输出为 Wo 后面跟 rld。

如果是大端，则将 i 设置为 0x726c6400，57616 不用更改。

Question 5

结果：

x=3 y=-267380196

解释：

由于字符串中有 2 个"%d"，所以会打印两个数字，打印完给定的 3 后会打印在栈中位于 3 上方的一个数字，是一个未知的数字。

Question 6

需要修改 stdarg.h 中的宏，将

```

#define va_start(ap,v) ( ap = (va_list)&v + _INTSIZEOF(v) )
#define va_arg(ap,t) ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
#define va_end(ap) ( ap = (va_list)0 )

```

改为：

```

#define va_start(ap,v) ( ap = (va_list)&v - _INTSIZEOF(v) )
#define va_arg(ap,t) ( *(t *)((ap -= _INTSIZEOF(t)) + _INTSIZEOF(t)) )
#define va_end(ap) ( ap = (va_list)0 )

```

Challenge

```

if (!(c & ~0xFF))
    c |= 0x0700;
{
    if (((c&0xFF)<='z'&&(c&0xFF)>='a')||((c&0xFF)<='Z'&&(c&0xFF)>='A'))
        c|=0x0900;
    if (((c&0xFF)>='0'&&(c&0xFF)<='9'))
        c|=0x0200;
}

```

在 console.c 中，将所有字母设为蓝色，所有数字设为绿色，代码如下：
效果图：

```
ebp f010ff18 eip f0100087 args 00000000 00000000 00000000 00000000 f01009c5
  kern/init.c:19: test_backtrace+71
ebp f010ff38 eip f0100069 args 00000000 00000001 f010ff78 00000000 f01009c5
  kern/init.c:16: test_backtrace+41
ebp f010ff58 eip f0100069 args 00000001 00000002 f010ff98 00000000 f01009c5
  kern/init.c:16: test_backtrace+41
ebp f010ff78 eip f0100069 args 00000002 00000003 f010ffb8 00000000 f01009c5
  kern/init.c:16: test_backtrace+41
ebp f010ff98 eip f0100069 args 00000003 00000004 00000000 00000000 00000000
  kern/init.c:16: test_backtrace+41
ebp f010ffb8 eip f0100069 args 00000004 00000005 00000000 00010094 00010094
  kern/init.c:16: test_backtrace+41
ebp f010ffd8 eip f01000ea args 00000005 00001aac 00000644 00000000 00000000
  kern/init.c:43: i386_init+77
ebp f010fff8 eip f010003e args 00111021 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.
{>
```

Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

```
    movl    $0x0,%ebp                # nuke frame pointer
f010002f:    bd 00 00 00 00                mov    $0x0,%ebp

    # Set the stack pointer
    movl    $(bootstacktop),%esp
f0100034:    bc 00 00 11 f0                mov    $0xf0110000,%esp
```

地址 f010002f 和地址 f0100034 的指令初始化了栈，其中将 ebp 初始设为 0,以便回溯栈帧链时可以及时停止，esp 的初值 0xf0110000 为栈顶地址。

```
bootstack:
    .space   KSTKSIZE
    .globl   bootstacktop
bootstacktop:
```

在 entry.S 中找到上述代码，为栈预留了空间，其中 KSTKSIZE 在 memlayout.h 中定义， $8 \times \text{PGSIZE} = 8 \times 4096 = 32\text{KB}$ ，即栈的大小。

```
// Kernel stack.
#define KSTACKTOP    KERNBASE
#define KSTKSIZE     (8*PGSIZE)
#define KSTKGAP      (8*PGSIZE)
```

栈在内存中是由高地址向低地址拓展，则栈在内存中应占据 0xf0110000 开始及其下的 32KB 空间。ESP (stack pointer) 初始时指向为栈保留的空间的高地址处。

Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the `test_backtrace` function in `obj/kern/kernel.asm`, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words?

Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the [tools](#) page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

每次 `test_backtrace` 函数被调用，会先将原先的 `ebp` 压入栈中，并将 `esp` 赋给 `ebp`，成为当前函数的栈帧的栈顶，保存被调用者保存寄存器，然后执行函数的主体部分。

每次 `test_backtrace` 递归调用自己时，有 8 个 32bit-words 被压入栈中，分别为上个函数的栈帧的 `ebp`，被调用者保存寄存器 `ebx` 的旧值，5 个 32-bit words 的临时空间（用于存放该函数调用函数时的参数或临时变量等），以及该函数调用函数的返回时的 `eip`。（1+1+5+1）

Exercise 11. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run **make grade** to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like. If you use `read_ebp()`, note that GCC may generate "optimized" code that calls `read_ebp()` *before* `mon_backtrace()`'s function prologue, which results in an incomplete stack trace (the stack frame of the most recent function call is missing). While we have tried to disable optimizations that cause this reordering, you may want to examine the assembly of `mon_backtrace()` and make sure the call to `read_ebp()` is happening after the function prologue.

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    cprintf("Stack backtrace:\n");
    uint32_t ebp=read_ebp();
    struct Eipdebuginfo info;
    while (ebp!=0)
    {
        uint32_t eip=((uint32_t *))(ebp+4);
        cprintf("  ebp %08x eip %08x args ",ebp,eip); //内存中地址为ebp+4的空间存储着该函数的返回地址，通常也在调用该函数的函数内部
        uint32_t * ebpt=((uint32_t *))(ebp);
        cprintf("%08x %08x %08x %08x\n",*(ebpt+2),*(ebpt+3),*(ebpt+4),*(ebpt+5));
        //本次函数调用的参数分别存储在地址为ebp+8,ebp+12,ebp+16.....的内存中，实际参数个数不一定为5,由于ebpt为指向32位整数的指针
        //所以每往上寻找一个参数只需加1 (sizeof(uint32_t)=4)
        debuginfo_eip(eip,&info);
        //通过已有函数接口，传入eip，查询是哪个函数调用了当前函数及其详细信息
        cprintf("    %s:%d: %.s+%d\n",info.eip_file,info.eip_line,info.eip_fn_name,info.eip_fn_name,info.eip_fn_name,info.eip_fn_addr);
        ebp=*(ebpt);
    }
    return 0;
}
```

代码及注释如图。

Exercise 12. Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.

In `debuginfo_eip`, where do `__STAB_*` come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- look in the file `kern/kernel.ld` for `STAB *`
- run `i386-jos-elf-objdump -h obj/kern/kernel`
- run `i386-jos-elf-objdump -G obj/kern/kernel`
- run `i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c`, and look at `init.s`.
- see if the bootloader loads the symbol table in memory as part of loading the kernel binary

Complete the implementation of `debuginfo_eip` by inserting the call to `stab_binsearch` to find the line number for an address.

Add a backtrace command to the kernel monitor, and extend your implementation of `mon_backtrace` to call `debuginfo_eip` and print a line for each stack frame of the form:

`K> backtrace`

Stack backtrace:

```
ebp f010ff78 eip f01008ae args 00000001 f010ff8c 00000000 f0110580 00000000
    kern/monitor.c:143: monitor+106
ebp f010ffd8 eip f0100193 args 00000000 00001aac 00000660 00000000 00000000
    kern/init.c:49: i386_init+59
ebp f010fff8 eip f010003d args 00000000 00000000 0000ffff 10cf9a00 0000ffff
    kern/entry.S:70: <unknown>+0
```

`K>`

Each line gives the file name and line within that file of the stack frame's eip, followed by the name of the function and the offset of the eip from the first instruction of the function (e.g., `monitor+106` means the return eip is 106 bytes past the beginning of `monitor`).

Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

Tip: `printf` format strings provide an easy, albeit obscure, way to print non-null-terminated strings like those in STABS tables. `printf("%.s", length, string)` prints at most `length` characters of `string`. Take a look at the `printf` man page to find out why this works.

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

仿照框架代码其提示，搜索`[lline,rline]`区间，若找到，返回对应的行号，否则返回-1。