OS lab1 解答过程

杨铭
5130379022

2015 年 9 月 28 日 - 2015 年 10 月 2 日

Contents

1 环境介绍

基本全部使用课程提供的环境，避免出现兼容性问题

虚拟机：Debian 6.0:jos-student.rar

PC emulator：QEMU Emulator 0.10.6

lab 源码：git

2 Part I PC Bootstrap

第一部分旨在了解 x86 指令，并通过 QEMU 模拟的 PC，模拟一个电脑从上电到 BIOS 的过程。整个过程中，PC 处于实模式中

练习 2

```
1  [f000:fff0] 0xffff0:  ljmp   $0xf000,%0xe05b
2  [f000:e05b] 0xfe05b:  xor    %ax,%ax
3  [f000:e05d] 0xfe05d:  out    %al,$0xd
4  [f000:e05f] 0xfe05f:  out    %al,$0xda
5  [f000:e061] 0xfe061:  mov    $0xc0,%al
6  [f000:e063] 0xfe063:  out    %al,$0xd6
7  [f000:e065] 0xfe065:  mov    $0x0,%al
8  [f000:e067] 0xfe067:  out    %al,$0xd4
9  [f000:e069] 0xfe069:  mov    $0xf,%al
10 [f000:e06b] 0xfe06b:  out    %al,$0x70
11 [f000:e06d] 0xfe06d:  in     $0x71,%al
12 [f000:e06f] 0xfe06f:  mov    %al,%bl
13 [f000:e071] 0xfe071:  mov    $0xf,%al
14 [f000:e073] 0xfe073:  out    %al,$0x70
15 [f000:e075] 0xfe075:  mov    $0x0,%al
16 [f000:e077] 0xfe077:  out    %al,$0x71
17 [f000:e079] 0xfe079:  mov    %bl,%al
18 [f000:e07b] 0xfe07b:  cmp    $0x0,%al
19 [f000:e07d] 0xfe07d:  je     0xfe0a7
20 [f000:e0a7] 0xfe0a7:  cli
21 [f000:e0a8] 0xfe0a8:  mov    $0xfffe,%ax
22 [f000:e0ab] 0xfe0ab:  mov    %ax,%sp
23 [f000:e0ad] 0xfe0ad:  xor    %ax,%ax
24 [f000:e0af] 0xfe0af:  mov    %ax,%ds
25 [f000:e0b1] 0xfe0b1:  mov    %ax,%ss
26 [f000:e0b3] 0xfe0b3:  mov    %bl,0x4b0
27 [f000:e0b7] 0xfe0b7:  cmp    $0xfe,%bl
28 [f000:e0ba] 0xfe0ba:  jne    0xfe0bf
29 [f000:e0bf] 0xfe0bf:  mov    %ax,%es
30 [f000:e0c1] 0xfe0c1:  mov    $0x80,%cx
31 [f000:e0c4] 0xfe0c4:  mov    $0x400,%di
32 [f000:e0c7] 0xfe0c7:  cld
33 [f000:e0c8] 0xfe0c8:  rep stos %ax,%es:(%di)
34 ......  次#128
35 [f000:e0ca] 0xfe0ca:  call   0xf17c1
36 [f000:17c1] 0xf17c1:  push   %bp
37 [f000:17c2] 0xf17c2:  mov    %sp,%bp
```

```
[f000:17c4] 0xf17c4:  mov    $0x194,%bx
[f000:17c7] 0xf17c7:  push   %bx
......
```

解答 首先在实模式下，跳转到内存的 0xfe05b 处开始执行 bios 的代码。将 AX 清零后，先对 0xd 和 0xda 端口输出，获取 DMA 权限，接着对一些基础 I/O 端口进行了初始化。随后关闭中断，初始化栈顶指针 SP、数据段 DS、栈堆段 SS、额外段 ES。然后将 CX 设置为 0x80，使得 rep 循环 128 次，为初始内存的连续 512 个字节填充

3 Part II The Boot Loader

练习 3 通过 gdb 在 0x7c00 设置断点后的输出

```
[    0:7c00]:0x7c00:  cli
[    0:7c01]:0x7c01:  cld
[    0:7c02]:0x7c02:  xor  %ax,%ax
[    0:7c04]:0x7c04:  mov  %ax,%ds
[    0:7c06]:0x7c06:  mov  %ax,%es
[    0:7c08]:0x7c08:  mov  %ax,%ss
```

boot loaded 开始，关闭中断，设置 cld，初始化 ds,es,ss 指针
根据 boot.S 中的代码：

```
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
```

```
12    jnz     seta20.2
13
14    movb    $0xdf,%al   # 0xdf -> port 0x60
15    outb    %al,$0x60
16
17    lgdt    gdtdesc
18    movl    %cr0,%eax
19    orl     $CR0_PE_ON,%eax
20    movl    %eax,%cr0
21
22    ljmp    $PROT_MODE_CSEG, $protcseg
23
24    .code32
25  protcseg:
26  ......
27    move    $start,%esp
28    call  bootmain
29  ...
```

BIOS 程序从 boot.S 进入 main.c 中执行，并且加载了全局描述符表和设置了 CR0 的状态，切换进入了保护模式，调转至 32 位汇编代码模式

继续看 main.c 中的代码

```
1  ......
2  ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
3  eph = ph + ELFHDR->e_phnum;
4  for (; ph < eph; ph++)
5      readseg(ph->p_pa,ph->p_memsz,ph->p_offset);
6      // note: does not return!
7      ((void (*)(void))(ELFHDR->e_entry))();
```

在 main 中，通过 readseg 读取 ELF 文件头，然后循环读取所有的 program 到内存中

循环结束后，执行

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

进入内核

解答

1.

```
ljmp   $PRO_MODE_CSEG, $protcseg

  .code32
protcseg:
```

.code32 表示进入 32 位汇编模式

2. 通过分析，我们发现 main 中的 ELFHDR->e_entry() 执行后，进入内核程序, 通过观察 boot.asm 中调用这一函数所在的汇编代码

```
7d6d:     83 c4 0c              add $0xc,%esp
7d70:     39 f3                 cmp %esi,%ebx
7d72:     72 e8                 jb  7d5c <bootmain+0x40>
//((void(*)(void))(ELFHDR->e\_entry))();
7d74:     ff 15 18 00 01 00     call *0x10018
```

我们在 0x7d74 处设置断点: `b *0x7d74`，然后再执行一步后，使用 `x/i` 指令查看内核里的指令

```
          0x100015:     mov      $0x110000,%eax
          0x10001a:     mov      %eax,%cr3
          0x10001d:     mov      %cr0,%eax
          0x100020:     or       $0x80010001,%eax
          0x100025:     mov      %eax,%cr0
          ....
```

这里是内核初始化 CR3 也就是初始化页目录表基地址，然后重置 CR0 也就是控制页表的一些位

3. 通过读取 elf 文件中的 e_phnum 决定读多少个 sector

```
phe = ph + ELFHDR->e\_phnum;
for(; ph < phe; ph++)
    readseg(ph->p\_pa, ph->p\_p_memsz, ph->p\_offset);
```

下面是一个关于 C 的指针的小程序

```
1  void
2  f(void)
3  {
4      int a[4];
5      int *b = malloc(16);
6      int *c;
7      int i;
8
9      printf("1:a=%p,b=%p,c=%p\n", a,b,c);
10
11     c = a;
12     for(i = 0; i < 4; i++)
13         a[i] = 100 + i;
14     c[0] = 200;
15     printf("2:a[0]=%d,a[1]=%d,a[2]=%d,a[3]=%d\n",
16         a[0], a[1], a[2], a[3]);
17
18     c[1] = 300;
19     *(c + 2) = 301;
20     3[c] = 302;
21     printf("3:a[0]=%d,a[1]=%d,a[2]=%d,a[3]=%d\n",
22         a[0], a[1], a[2], a[3]);
23
24     c = c + 1;
25     *c = 400;
26     printf("4:a[0]=%d,a[1]=%d,a[2]=%d,a[3]=%d\n",
27         a[0], a[1], a[2], a[3]);
28
29     c = (int *)((char*) c + 1);
30     *c = 500;
31     printf("5:a[0]=%d,a[1]=%d,a[2]=%d,a[3]=%d\n"
32         a[0], a[1], a[2], a[3]);
33
34     b = (int *) a + 1;
35     c = (int *) ((char *) a + 1);
36     printf("6:a=%p,b=%p,c=%p\n", a, b, c);
37 }
```

4-7 行是变量初始化，a 为栈上一个数组的指针，b 为堆中长度为 16 个字节的指针，c 为选控制值，而 i 为未赋值的临时变量。第一句 printf 打出了 3 个指针的地址

1: a = 0037FC24, b = 00748D88, c = 0074932D

下面指针 c 指向数组 a，给 a 前四项赋值 100,101,102,103，然后给 c 的第一项赋值 200，注意这里 c 的第一项即使 a 的第一项，所以 a[0]=200，第二句 printf 打出

2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103

然后用 3 种不同的方式给 c 也就是 a 的后三项赋值，注意到 3[c] 等价于 c[3]，所以第三句 printf 打出

3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302

接下来将 c 向前挪一个位置，指向 a[1]，然后将其改为 400：

4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302

这里比较蛋疼，将 c 强制转换为 char 类型的指针然后向前挪一个位置，实际上是一个字节，由于是 bigendian，实际上 c 现在指向 a[1] 的低 2 位：
划线位置即为 c 所指向的位置，改动后变为

a[0]	a[1]	a[2]	a[3]
C3 00 00 00	90 <u>01</u> 00 00	2d 01 00 00	2e 01 00 00

所以打印结果为

a[0]	a[1]	a[2]	a[3]
C3 00 00 00	90 <u>f4 01 00</u>	<u>00</u> 01 00 00	2e 01 00 00

5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302

最后是指针的移动，没有什么好说的

5: a = 0037FC24, b = 0037Fc28, a[2] = 256, c = 0037FC25

练习 5 在 boot loader 运行结束前，内核代码还没有被装载进内存，所以在第一个断点打印是没有数据的

练习 6 这个练习让我们把 Makefrag 中 Ttext 的地址改掉，观察重新 make 后的结果。可以想象，Ttext 应该就是代码段地址的表示，如果与生成的二进制文件的地址不一致，则会导致与相对位置有关的指令出现错误

```
$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7c00 -o $@.out $^
```

将 0x7c00 改成 0x7000 后重新 make，查看 boot.asm 中的指令发现

```
cli
  7000:   fa      cli
cld
  7001:   fc      cld
```

对照上面的代码可以知道在修改 Ttext 之前，cli 的地址应为 7c00

4 Part III Kernel

4.1 Using segmentation to work around position dependence

终于要详细地近距离观察 JOS 这个操作系统了（终于要写代码了）

操作系统的内核通常在内存的高地址，例如：0xf0100000, 为了给其他程序留下足够的空间

练习 7 跟 part2 类似，我们在 `0x7d74` 设置断点，然后使用 `si` 指令向下执行发现

```
0x10000c:    movw    $0x1234,0x472
0x100015:    mov     $0x110000,%eax
0x10001a:    mov     %eax,%cr3
0x10001d:    mov     %cr0,%eax
0x100020:    or      $0x80010001,%eax
0x100025:    mov     %eax,%cr0
0x100028:    mov     $0xf010002f,%eax
0x10002d:    jmp     *%eax
0xf010002f:  mov     $0x0,%ebp
......
```

最后一行可以看到新的地址映射已经起作用，指令从 0x10002d 跳转至 0xf010002f，进入 entry.S 查看

```
  movl   %(RELOC(entry_pgdir)), %eax
  movl   %eax, %cr3
  # Turn on pagging.
  movl   %cr0, %eax
  orl    $(CR0_PE|CR0_PG|CR0_WP),%eax
  movl   %eax, %cr0

  # Now paging is enabled, but were still running at a low EIP
  # (why is this okay?). Jump up above KERNBASE before entering
  # C code.
  mov    $relocated, %eax
  jmp    *%eax
relocated:
....
  movl   $0x0,%ebp      #nuke frame pointer
```

正是 gdb 打出的这几行汇编代码。

4.2 Formatted Printing to the Console

练习 8 实现八进制输出

在 printfmt.c 中的 vprintfmt() 函数中可以看到一堆 case，代表着不同的输出格式，找到

```
case '0':
    // Replace this with your code.
    // display a number in octal form and the
    // form should begin with '0'
    putch('X',putdat);
    putch('X',putdat);
    putch('x',putdat);
    break;
```

在这里实现八进制的输出即可

```
case '0':
    putch('0',putdat);
    num = getuint(&ap, lflag);
```

```
4     base = 8;
5     goto number;
```

可以看到已经可以正常输出八进制数了

```
6828 decimal is 015254 octal!
```

练习 9

看了 grade.sh 才知道这题的意思：让 printf 能够正确处理'+' 这个修饰符，加了'+' 的输出，当打出的数是正整数 (jos 现在还不支持浮点数) 时，前面带上加号，如果是负整数则即使有'+' 的修饰也不带'+'，而是'-'

先找到 printnum 这个函数 1. 仔细观察 printf.c 中的函数，写上对'+' 的处理：

```
1  if(num >= base) {
2      printnum(putch, putdat, num / base, base, width - 1, padc);
3  } /*my code
4      else if(padc == '+'){
5      putch(padc, putdat);// handle with '+'
6    */
7  } else {
8      while (--width > 0)
9          putch(padc, putdat);
10 }
```

然后在函数 vprintfmt 中处理输出为负数时的情况

```
1  case '+':
2      padc = '+'
3      goto reswitch;
4  ......
5  case 'd':
6      num = getint(&ap, lflag); // num is signed
7      if ((long long) num < 0) { // num < 0
8          putch('-',putdat); // already have minus sign
9          num = -(long long) num;
10         /*mycode
```

```
11          if(padc == '+'){
12              padc = ' ';
13          }
14          */
15      }
16      base = 10;
17      goto number;
```

屏幕输出

show me the sign: +1024, -1024

练习 9 完成

```
1   static void
2   putch(int ch, int *cnt)
3   {
4       cputchar(ch);
5       (*nct)++;
6   }
7
8   int
9   vcprintf(const char *fmt, va_list ap)
10  {
11      int cnt = 0;
12
13      vprintfmt((void*)putch, &cnt, fmt, ap);
14      return cnt;
15  }
16
17  int
18  cprintf(const char *fmt, ...)
19  {
20      va_list ap;
21      int cnt;
22
23      va_start(ap,fmt);
24      cnt = vcprintf(fmt,ap);
25      va_end(ap);
26
```

```
27      return cnt;
28  }
```

在 init.c 中调用了 cprintf，而 cprintf 调用了 vcprintf，并传了一个函数参数 putch，putch 调用了 console 的 cputchar。关注 cprintf，参数使用了可变的参数列表

```
1  cprintf(const char *fmt, ...)
```

并且用 ap 记录这些参数，传递给 vcprintf

```
1  va_start(ap,fmt); // 以固定参数的地址为起点确定变参的内存起始地址
2  cnt = vcprintf(fmt,ap);
3  va_end(ap);
```

而在 console.c 中，函数使用底层接口 out 通过显示 I/O 端口直接向屏幕输出字符

2. 解释 console.c 中下列代码的意义

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

这里网站上的代码和 git 上的源码不一致，第三行的 memcpy 源码为 memmove, 去网上查了这两个函数除了在处理内存发生重叠时，memcpy 不保证结果正确，而 memmove 保证结果正确以外，没有其他区别，这里应该为 memmove

首先分析代码片段中各个变量的意思，在 console.h 中我们可以看到这些宏定义

```
1  #define CRT_ROWS  25
2  #define CRT_COLS  80
3  #define CRT_SIZE  (CRT_ROW * CRT_COLS)
```

crt_pos 表示输出在控制台的光标在一行中的位置，而 CRT_ROWS 和 CRT_COLS 分别代表控制台一页的行数和列数，CRT_SIZE 是控制台一页的大小。这段代码是判断当前光标位置是否超过一页，即输出是否需要翻页。如果超过则将最上面一行删掉，剩下的内容整体向上移动一行，然后清空最后一行，使光标的位置位于最后一行的开始。

3. 通过上面对 cprintf 等系列函数的分写，容易知道 fmt 是” x %d, y %x, z %d”,ap 指的是”x,y,x”
在 init.c 中加入题目所给代码跑一边 make qemu 发现：
x 1, y 3, z 4 在 kernel.asm 找到加入代码对应的汇编代码 (省略了机器码)

```
f010026e   movl   $0x4,0xc(%esp)
f0100276   movl   $0x3,0x8(%esp)
f010027e   movl   $0x1,0x4(%esp)
f0100286   movl   $0xf0101c0d,(%esp)
f010028d   call   f0100a6b <cprintf>
```

进入 gdb 调试状态在 f010026e 设置断点，然后用 si 逐步执行看以看到 cprintf 到 vcprintf 到 vprintfmt 到 putch 到 cputcha 到 cons_putc

4. 运行一段有趣的 print 代码

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

输出: He110 World
把十进制的 57616 按 16 进制输出，$57616_{10} = e110_{16}$
这样就和 H 组成了 Hello(实际上是 He 妖妖灵)
后面将 i 当做一个字符串输出，查看 i 在内存空间的数据结构

72 6c 64 00

对照 Ascii 表发现就是 rld，所以拼成了 World

5.print 的 ap 少一个参数，会输出什么？
根据上面的分析，控制第二个以后参数的结构体 ap 在输出时，每次从里面拿一个参数出来根据类型返回给调用者，然后 ap 向后移动一个参数。所以如果少一个参数，则 ap 会移动到结构体外，发生内存泄露。而从这个位置读取的数据也将是随机的。

6. 假如 GCC 在改变参数压栈的顺序，导致最后的参数最后入栈，我们只需要将 ap 指向原来结构体的尾端，然后向前遍历即可。

练习 10 实现%n 的功能，查了一下，%n 之前所有的字符计数，赋值给传入的参数，注意到 putdat

```
int
vprintf(const char *fmt, va_list ap)
{
    int cnt = 0;

    vprintfmt((void*)putch, &cnt, fmt, ap);
    return cnt;
}
```

这里 colorbox[rgb]0.9,0.9,0.9vprintf 的第二个参数 cnt 就是后面的 putdat，也就是后面输出时字符的计数
受此启发，实现代码如下：

```
case 'n':
    char *ip = va_arg(ap, char *);
    char *ps = putdat;
    *ip = putdat;
```

貌似没有错误检查..... 果然跑的时候出现了错误！但是报了个奇怪的错误：
Could not open 'dev kqemu' - QEMU acceleration layer not activated
然后打出了几行在 chunm1: 27 chunm2: 30 这里停下进入了死循环
去 init.c 里看了一下，果然是空指针的锅

```
printf("%n\n",NULL);
```

在下面还有一个 printf 相比是针对数据越界的 bug 的测试了

```
memset(ntest, 0xd, sizeof(ntest) - 1);
printf("%s%n", ntest, &chnum1);
```

修复了 BUG 以后

```
case 'n':
    char *ps = putdat;
    if(*int *)putdat > 0x80){
        char *ip = va_arg(ap, char*);
```

```
        *ip = *ps;
        printfmt(putch, putdat, "%s", overflow_error);
    }
    else {
    char *ip = va_arg(ap, char*);
    if(ip == NULL) {
        printfmt(putch, putdat, "%s", error_null);
    }
    else {
        *ip = putdat;
    }
    break;
```

本练习完成

练习 11 要求当 padc 是'-' 时，在输出右边用空格补上

观察 printnum，这个函数用递归将数字和 padc 打出来，因为要在输出的右边输出，所以只需判断 padc，然后在递归的最后一次打印相应数量的空格即可。

```
static int rflag = 0;
static int rwidth = 0;
printnum(void (*putch)(int void*), void *putdat,
        unsigned long long num, unsigned base,
        int width, int padc)
{
    if (padc == '-' && rwidth == 0) {
        rflag = 1;
        rwidth = width;
    }

    if (num >= base) {
        printnum(putch,putdat,num/base,base,width-1,padc);
    } else if(padc == '+') {
        putch(padc, putdat);
    } else if(padc != '-') {
        while (--width > 0)
                putch(padc,putdat);
    }

```

```
21      putch("0123456789abcdef"[num%base],putdat);
22      if(padc == '-' && rflag == 1) {
23          while (--rwidth > 0)
24              putch(padc,putdat);
25          rflag = 0;
26      }
27  }
```

4.3 Stack

练习 12 还记得内核从读取 elf 文件头以后进入了 entry 这个函数中，在 entry.S 中找到

```
1   # stack backtraces will be terminated properly
2   movl  $0x0,%ebp           #nuke frame pointer
3
4   # Set the stack pointer
5   movl  $(bootstacktop),%esp
6
7   # now to C code
8   call  i386_init
```

这里设置了栈指针，位置为.data 段，然后跳转到 C 代码 init.c 中初始化栈

```
1   memset(edata, 0, end-edata);
```

练习 13 在 kernel.asm 中找到 test_backtrace 对应的汇编代码

```
1   f01000e4:  push  %ebp
2   f01000e5:  mov   %esp,%ebp
3   f01000e7:  push  %ebp
4   f01000e8:  sub   $0x14,%esp
5   f01000eb:  mov   0x8(%ebp),%ebx
6   f01000ee:  mov   %ebx,0x4(%esp)
7   f01000f2:  movl  $f0101cf2,(%esp)
8   f01000f9:  call  f0100a4b <cprintf>
9   f01000fe:  test  %ebx,%ebx
10  f0100100:  jle   f010010f<test_backtrace+0x2b>
11  f0100102:  lea   -0x1(%ebx),%eax
```

```
f0100105:  mov   %eax,(%esp)
f0100108:  call  f01000e4 <test_backtrace>
f010010d:  jmp   f010012b <test_backtrace+0x47>
...
```

两个 push，一次 sub $0x14，一次 call，所以一共压栈 4+4+4+20=32byte

练习 14 完善 bcaktrace 的输出和 grade 里一致，看到 test_backtrace 调用了 mon_backtrace，对照 grade.sh 中的 grep，找到这个函数码几行 cprintf。这里感觉到之前做的那个关于 C 的指针的练习派上用场了。

```
int
mon_backtrace(int argc, char **argcv, struct Trapframe *tf)
{
    uint32_t ebp = read_ebp();
    uint32_t eip = read_eip();
    cprintf("Stack␣backtrace:\n");
    while(ebp != 0x0){
        eip = *((uint32_t *)ebp+1);
        uint32_t tmp = ebp;
        cprintf("␣eip␣%08x␣␣ebp␣*08x␣␣args␣%08x
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%08x␣%08x␣%08x␣%08x\n",
        eip,ebp,*((uint32_t *)tmp+2),*((uint32_t *)tmp+3),
        *((uint32_t *)tmp+4),*((uint32_t *)tmp+5),
        *((uint32_t *)tmp+6));
        ebp = *((uint32_t *)ebp); // next stack
    }
    overflow_me();
    cprintf("Backtrace␣success\n");
}
```

练习 15 在刚刚完成栈的信息打印的同时，额外打印一些信息。
在 kdebug.h 和 kdebug.h 中找到相关结构体 按照 grade 中的要求打打印即可，注意在 kdebug.c 中完成为 eip_line 的搜索

```
stab_binsearch(stabs, &lline, &rline, N\_SLINE, addr);
info->eip\_line = lline? -1:stabs[lline].n\_desc;
```

const char *eip_file	EIP 的源文件
int eip_line	EIP 的源代码的行数
const char *eip_fn_name	调用 EIP 的函数
int eip_fn_namelen	函数名的长度
uintptr_t eip_fn_addr	函数地址
int eip_fn_narg	函数参数的个数

练习 16 利用 buffer overflow 的方法调用一个函数，在反汇编代码中可以看到

```
void
overflow_me(void)
{
......
f01007bb:   call f0100798<start_overflow>
f01007c0:   leave
f01007c1:   ret
```

由于我们可以改写的是 start_overflow 这个函数，所以需要将这个地址改为 do_overflow 的入口
再查看 do_overflow 发现

```
void
do_overflow(void)
{
f01007d6:   push %ebp
f01007d7:   mov  %esp,%ebp
f01007d9:   sub  $0x18,%esp
f01007dc:   movl $0xf0102271,(%esp):
f01007e3 call f0100c17 <cprintf>
...
```

利用之前实现的 cprintf 的 n 参数功能，将数组 str 的长度作为变量，赋值给 pret_addr 即可
注意要保证程序正常返回，正确覆盖返回地址的地址应该是 0xf01007d9 而不是 0xf01007d6，即 read_pretaddr() 返回值加 3

练习 17 完成一个计时的功能，调用 read_tsc() 函数即可