

# JOS-LAB1 解题报告

BY stone

## 0.概述

- 本文档为上海交通大学软件学院 2015-fall-joslab1 解题报告，题目请点击 <http://ipads.se.sjtu.edu.cn/courses/os/> 查看。文档主要包括对本次 lab 的理解、问题的详细解答以及 exercises 的解题思路，相关代码请使用 `git clone git://github.com/stone-SJH/JOS-byStone.git` 指令获取。
- 本文档中，所有 exercise 题目以文本框形式描述，所有汇编代码以蓝色字体描述，所有 c 代码以红色字体描述，所有指令以绿色字体描述，正文以黑色字体描述。
- 对于后面几道与 coding 相关的 exercise，本文档只涉及其代码思路，具体代码及注释请查阅我的源代码，在源代码中所有修改相关处均有 `/* stone's solution for exerciseX */` 注释标明，可以使用 Ctrl+F 指令搜索 stone 或 exerciseX 即可。
- 文档涉及到的地址调用排序：0xffff0----->0x7c00----->0x10000c----->0x100000----->.....  
大致对应：BIOS Boot-Loader Kerner
- 鉴于个人水平所限，如果您对文档有任何问题或者异议请联系 [wy30123@163.com](mailto:wy30123@163.com) 石嘉昊。
- 未经本人授权严禁以各种形式复用或抄袭。

## 1.PC Bootstrap

Exercise 1. Read or at least carefully scan the entire PC Assembly Language book, except that you should skip all sections after 1.3.5 in chapter 1, which talk about features of the NASM assembler that do not apply directly to the GNU assembler. You may also skip chapters 5 and 6, and all sections under 7.2, which deal with processor and language features we won't use in JOS.

Also read the section "The Syntax" in Brennan's Guide to Inline Assembly to familiarize yourself with the most important features of GNU assembler syntax.

---

这部分主要就是看看资料，虽然很多。关于汇编方面的这本书，我大致浏览了一下，感觉只要认真做过 icslab 的应该对于汇编语言方面没有太大问题，所以就没仔细看，准备遇到问题的时候查阅解决吧。

## 1.1 simulation the x86

浏览一下，玩一玩 lab 环境，了解一下就行。

## 1.2 The PC's Physical Address Space

这部分介绍的物理地址空间结构和老师上课讲的内容大致一样，略过。

## 1.3 The ROM BIOS

BIOS 是开电源之后的第一个执行的代码 使用 make-gdb 开两个 terminal 跑一下 gdb 的使用在 icslab2 研究过，这边还算是比较简单，之后就是一些物理地址格式 (  $PA = 16 * \text{segment} + \text{offset}$  )。

Exercise2. 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 Jos reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

给的 Phil Storrs I/O Ports Description 挂掉了，导致从汇编获取的一些端口号并没有查到其真实含义。

下面是使用 si 单步调试获取的 BIOS IO 相关的汇编代码：

```
0xffff0    ljmp      0xf0000, 0xe05b
0xfe05b    xor       %ax, %ax
0xfe05d    out       %al, $0xd
0xfe05f    out       %al, $0xda
.....
```

---

```
0xfe063    out        %al, $0xd6
.....
0xfe067    out        %al, $0xd4
.....
0xfe06b    out        %al, $0x70
.....
0xfe073    out        %al, $0x70
.....
0xfe077    out        %al, $0x71
0xfe079    mov        %bl, %al
0xfe07b    cmp        $0x0, %al
0xfe07d    je         0xfe0a7
0xfe0a7    cli
```

可以看到 BIOS 主要是向一些端口(0xd,0xda,0xd6.....)中写入初始数值，可以猜测这些端口应该是一些

计算机设备例如网卡声卡等，最后跳转到 0xfe0a7 执行 cli 指令关闭中断。

而接下来，BIOS 从磁盘中读入 BOOT LOADER 并将控制权移交给它。

## 2.The Boot Loader

Exercise 3. Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that break point. 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 GNU disassembly in obj/boot/boot.asm and the 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.

首先执行以下命令：

```
>(gdb) b* 0x7c00
>(gdb) c
Continue.
Breakpoint in 0x7c00.
>(gdb) x/10i 0x7c00
```

可以发现输出与 obj/boot/boot.asm 文件内容相同。

---

接下来找到 boot/main.c 的载入点，使用 `objdump -f kernel` 指令可以获取 main 函数的载入点为

```
0x10000c      movw      $0x1234, 0x472
```

在 kern/ 目录下进行查找，发现此汇编指令位于 kern/entry.S 文件 Line 44 处。

之后设置断点 0x10000c，开始继续跟踪。

```
>(gdb) b* 0x10000c
```

```
>(gdb) c
```

到达 0x10000c 位置之后继续单步调试，可以得到：

```
# Load the physical address of entry_pgdir into cr3.  entry_pgdir
```

```
# is defined in entrypgdir.c.
```

```
movl    $(RELOC(entry_pgdir)), %eax
```

```
movl    %eax, %cr3
```

```
# Turn on paging.
```

```
movl    %cr0, %eax
```

```
orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
```

```
movl    %eax, %cr0
```

```
# Now paging is enabled, but we're still running at a low EIP
```

```
# (why is this okay?).  Jump up above KERNBASE before entering
```

```
# C code.
```

```
mov $relocated, %eax
```

```
jmp *%eax
```

同 kern/entry.S Line48 位置的代码，粗略阅读一下，可以看出这段代码是在做几件事：

- 用新分配的一个 entry\_pgdir 放入 cr3 作为页表
- 修改 cr0, enable paging。
- 跳到 relocated 位置，后面的代码是在设定栈相关内容，这个在第四部分 Stack 内详解。

了解了这部分的全过程之后，exercise3 后面的 3 个问题自然迎刃而解。

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

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

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?

Q1.

在/boot/boot.S Line48 位置处：

```
lgdt    gdt_desc
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
.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
```

Q2.

Boot loader 最后一条指令：

```
((void (*)(void)) (ELFHDR->e_entry))();/* 跳转到 entry()即 entry.S*/
```

Kernel 第一条指令：

```
0x10000c      movw    $0x1234, 0x472
```

Q3.

Boot loader 获取要读入的整个 kernel 的 sector 信息相关代码在 boot/main.c 中：

```
// 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);
```

使用的命令是 readseg()函数，这个函数在 console.c 中实现，是较为底层的函数。

可以看到先读入一个完整的 ELFHeader,之后根据 ELFheader 中的信息来知道需要读取的 section 位置和数量等信息，然后——读入。

## 2.1 Loading the Kernel

**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](#)).

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, though not as strongly recommended. [A tutorial by Ted Jensen](#) that cites K&R heavily is available in the course readings.

*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.

这部分是关于指针的阅读和练习。我的 c 指针一直用的不太好，但是这本书也实在没耐心看完，依旧是浏览了一下目录之后准备当工具书用。

做了一下练习(pointers.c)，大部分还是比较好理解的，说说收获吧：

- $3[c] = c[3]$
- $(int*)((char* c) + 1)$ 这个直觉感觉是取后面一个 int ,但是实际上是+1 只移动了 8byte( 1 个 char ),

也就是前一个 int 的 8-31 位+后一个 int 的 0-7 位组合在一起的一个 int，所以输出的数值也比较奇怪。

另外，这个 lab 后面要做的那些类型转换才真的晕。做完之后才发现这些根本不是事儿。

关于 ELF，在 inc/elf.h 中有详细定义。

下面是使用 objdump 指令获取的 all sections：

### ELF all sections：

Sections:					
Idx	Name	Size	VMA	LMA	File off Algn
0	.text	00001b25	f0100000	00100000	00001000 2**4
			CONTENTS, ALLOC, LOAD, READONLY, CODE		
1	.rodata	00000780	f0101b40	00101b40	00002b40 2**5
			CONTENTS, ALLOC, LOAD, READONLY, DATA		
2	.stab	00003c0d	f01022c0	001022c0	000032c0 2**2
			CONTENTS, ALLOC, LOAD, READONLY, DATA		
3	.stabstr	00001a9d	f0105ecd	00105ecd	00006ecd 2**0
			CONTENTS, ALLOC, LOAD, READONLY, DATA		
4	.data	0000a300	f0108000	00108000	00009000 2**12
			CONTENTS, ALLOC, LOAD, DATA		
5	.bss	00000660	f0112300	00112300	00013300 2**5
			ALLOC		
6	.comment	0000001c	00000000	00000000	00013300 2**0
			CONTENTS, READONLY		

**Exercise 5.** 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.)

使用指令：

---

```
>(gdb)x/8x 0x00100000
```

设定断点位置为 0x7c00, 0x10000c

```
>(gdb) b* 0x7c00
```

```
>(gdb) b* 0x10000c
```

可以发现在 0x10000c 之前均为 0

在 0x10000c 之后为

```
0x100000: 0x1badb002 0x00000000 0xe4523ffe 0x7205c760
```

```
0x100010: 0x34000004 0x0000b812 0x220f0011 0x c0200fd8
```

在这个基础上，使用指令

```
>(gdb) x/8i 0x100000
```

得到

```
0x100000      add 0x1bad(%eax), %dh
```

```
0x100006      add %al, (%ead)
```

```
.....
```

对照 main.c 不难发现，这是在读取 elfheader 以及 sections 内容。

而在 0x10000c 前后，指令 x/8x 读取结果不同的原因应当是，0x100000 之后的代码属于内核代码，

而内核代码在 boot loader 之前并未载入，所以在 boot loader 之前读不到，而之后能读出数值。

## 2.2 Link vs. Load Address

在 BIOS 阶段，没有 Link 和 Load 之分，二者相同，而在进入 Boot loader 之后，地址有了 Link Address

和 Load Address 之分，我的理解是 Link Address = virtual address, Load Address = physical addresss。

Exercise 6. Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in boot/Makefrag to something wrong, run make clean, recompile the lab with make, and trace into the boot loader again to see what happens. Don't forget to change the link address back and make clean afterwards!

修改 boot/Makefrag 中的 0x7c00->0x7c0a，之后重新 make。



---

观察 boot/boot.asm 文件的变化：

- 原本的 0x7c00 cli -> 0x7c0c cli , 之后所有指令顺下移
- 在 cli 指令前增加了两条 nop 指令 ( 这个原因应该这是由于位对齐 )

进入 gdb 模式进行调试:

```
>(gdb) b* 0x7c00  
>(gdb) c
```

之后发现两个问题：

- 从 0x7c02 cli 开始而不是从 boot.asm 中的 0x7c0c cli 开始。
- 增加的两条 nop 指令还在，正因如此才从 0x7c02 cli 而非 0x7c00 cli 开始。

尝试对这个现象进行解释：明显修改 boot/Makefrag 中的地址是 link address。Binary 文件不随 link address 改变而改变，但是因此导致的指令改变会导致 Binary 改变(两条 nop 指令)，而 jmp 等跳转指令全部是使用 link address，这一点体现在改变之后几乎所有的 jmp 全乱了。

## 3. The Kernel

### 3.1 Using segmentation to work around position dependence

Exercise 7. Use QEMU and GDB to trace into the JOS kernel and find where the new virtual-to-physical mapping takes effect. Then examine the Global Descriptor Table (GDT) that the code uses to achieve this effect, and make sure you understand what's going on.

What is the first instruction after the new mapping is established that would fail to work properly if the old mapping were still in place? Comment out or otherwise intentionally break the segmentation setup code in kern/entry.S, trace into it, and see if you were right.

从阅读中获取重要信息：

PA: 0x00000000 -----> 0xffffffff

对应

---

VA: 0xf0000000 -----> 0xffffffff

那么查找何处开始使用 va 的关键就是查找第一个 0xfXXXXXXXXX。

使用 gdb，进入内核：

```
>(gdb) b* 0x10000c
>(gdb) c
[>(gdb) si]
```

直至：

```
0x0010002d  jmp     *%eax
0xf010002f  mov     $0x0, %ebp
```

进入 kern/entry.S 查找，相关内容在 Line73，.relocated 前一段。

而相关代码在前面已经研读过，那么重复的不再赘述，这里重点关注与页切换相关的 cr3 指令，发现将一个 entrypgdir 写入了 cr3，之后追踪 entrypgdir 找到 kern/entrypgdir.c，发现与其实现相关的一个重要宏 **KERNBASE 0xf0000000**，明显这就是 va<->pa 互相切换的关键。

Exercise 8. 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. Remember the octal number should begin with '0'.

Exercise 9. You need also to add support for the "+" flag, which forces to precede the result with a plus or minus sign (+ or -) even for positive numbers.

/\*终于开始写代码了\*/

这两个 exercise 都是比较简单的代码，与 printf 指令相关的。根据给的提示 console.c printf.c printfmt.c 三个文件，粗略阅读以下不难发现其中的 cprintf 指令是嵌套调用 vprintf()和 vprintfmt()，找到最底层的 printfmt.c 中的 vprintfmt() 函数进行修改即可。

Ex9 比较需要注意的一点是 "+"flag 是可以和其他 flag 并列的，也就是比"o""d"等 flag 高一级，可以像"%+d"这样使用，所以写法应当模仿"-flag 而非"o"flag 的写法，即需要递归而不是直接读其后面的一个数字。

/\*然而天真的我现在才发现这个 exercise 的重点还是恶心的问题\*/

---

Q1.

console.c export 的 cputchar() 作为一个底层 ( disk ) 操作接口在 printf.c 中的 putch() 方法中被调用。

Q2.

这段最开始没看懂 , 参考了 xv6 源码的相关部分之后大致弄清了。这段代码主要完成了滚屏的任务 , 具体操作就是当光标位置 ( crt\_pos ) 大于整个显示 SIZE 时 , 将全屏的内容复制均上滚一行 ( 最上面一行被覆盖 ) , 然后将最后一行清空 , 光标位置放置在最后一行的开始。

Q3.

以下均在 init.c 中添加给定指令并重新编译获取结果 , 就不再强调了。

查看 kernel.asm :

```
0xf0100137      movl      $0x4, 0xc(%esp)
0xf0100156      call      f0100b11<cprintf>
....
```

结合 cprintf() 相关代码 , 不难推测 fmt 是 format pointer 的缩写 , 即保存格式字符串 ; 而 ap 则是保存 List<不定输入> 的首指针 ; va\_arg 则是一个将 ap 指针向后推移一位的作用从而在汇编形成了递归向后循环的情况。

Q4.

输出 : He110 World

原因 : 在 little endian 环境中 ,

57616 = (e110)16 , 用 %x flag 输出则是 e110。

而 "00" "64" "6c" "72" 则对应 "r" "l" "d" "\0"。

Q5.

输出 : x=3 y=16323828

原因 : 正常打印出 x 之后 , 调用 va\_arg 使 ap 指针后移 , 而并没有给定 y 则本次后移到了内存某未知领域 ( 雾 ) , 就打印出了一些奇怪的东西。

Q6.

在 inc/stdarg.h 中找到 va\_arg 宏相关内容 , 如果 gcc 修改了其压栈顺序方式 , 那么只需要修改 va\_start, va\_end, va\_arg 三个宏的内容即可 ( va\_start 与 va\_end 调换 , va\_arg 改变方向即 + 换成 - )。

Exercise 10. Enhance the `cprintf` function to allow it print with the `%n` specifier, you can consult the `%n` specifier specification of the C99 `printf` function for your reference by typing "man 3 printf" on the console. In this lab, we will use the `char *` type argument instead of the C99 `int *` argument, that is, "the number of characters written so far is stored into the signed char type integer indicated by the `char *` pointer argument. No argument is converted." You must deal with some special cases properly, because we are in kernel, such as when the argument is a NULL pointer, or when the char integer pointed by the argument has been overflowed. Find and fill in this code fragment.

加一个空检查和溢出检查就可以了（此处溢出检查设定 256 即 0xff）。

Exercise 11. Modify function `printnum()` in `lib/printfmt.c` file to support pattern of the form `"%-"` when outputs number. Padding spaces on the right if `cprintf`'s parameter includes pattern of the form `"%-"`. For example, when call function:

```
cprintf("test:[%-5d]", 3)
, the output result should be
      "test:[3    ]"
(4 spaces after '3'). Before modify printnum(), make sure you know what happened in
function vprintfmt().
```

这边已经有写好的 `padc` 作为是否存在“-”flag 的检查和存储长度的 `width`，而且已经有其他逻辑完备的 `printnum()` 函数，那么只需要在已有的逻辑上添加对 `padc` 的检查即可，对于空格的处理也大致与其他符号的处理相同，只是把 `while (--width > 0) putchar(padc, putdat);` 中的 `padc` 替换成 ' '（空格）即可完成。

## 3.2 The Stack

Exercise 12. 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?

上面对 `entry.S` 的分析中发现了 Line73 中涉及到栈初始化。

```
Movl    $0x0, %ebp
Movl    $(bootstacktop), %esp
```

可以看出将 esp 寄存器初始化为 bootstacktop，查找这个值的定义，在 Line98 处找到，根据注释可以理解栈的位置在 ELF .data section 中，bootstacktop 是在栈定义完后定义的，即是栈的最高位 ( 0xefc0000 )。

Exercise 13. 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. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

在obj/kern/kernel.asm 中查找到 test\_backtrace 递归的相关代码：

f01000e4:	55	push	%ebp	4
f01000e5:	89 e5	mov	%esp,%ebp	
f01000e7:	53	push	%ebx	4
f01000e8:	83 ec 14	sub	\$0x14,%esp	20
f01000eb:	8b 5d 08	mov	0x8(%ebp),%ebx	
		cprintf("entering test_backtrace %d\n", x);		
f01000ee:	89 5c 24 04	mov	%ebx,0x4(%esp)	
f01000f2:	c7 04 24 f2 1f 10 f0	movl	\$0xf0101ff2,(%esp)	
f01000f9:	e8 b1 0b 00 00	call	f0100caf <cprintf>	
		if (x > 0)		
f01000fe:	85 db	test	%ebx,%ebx	
f0100100:	7e 0d	jle	f010010f <test_backtrace+0x2b>	
		test_backtrace(x-1);		
f0100102:	8d 43 ff	lea	-0x1(%ebx),%eax	
f0100105:	89 04 24	mov	%eax,(%esp)	
f0100108:	e8 d7 ff ff ff	call	f01000e4 <test_backtrace>	4

我使用绿色底纹标注的 4 条指令是每次递归压栈的条目，其中最后一条是在递归调用的时候压%ebp 寄存器内容，其他都是很显而易懂的。那么一共为 4+4+20+4 = 32bytes。

Exercise 14. 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.

在 kern/monitor.c 中使用提示的 read\_ebp()函数获取 ebp 指针之后，通过对栈结构的了解，ebp[1]即是 eip 指针，其后则是传入对象。注意 ebp eip 所需的是地址而 args 所需的是值，不要被乱七八糟的类型转换搞晕就好，另外注意格式。

此处可以在 kern/monitor.c 的 commands[]中添加 backtrace 指令进行自行测试。

#### Exercise15.

.....  
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:

.....

首先根据给出的格式确定需要的是什么：

```
kern/monitor.c:143: monitor+106
```

确定需要的是 function name, source file name, line number, offset

使用指令：

```
Objdump -G obj/kern/kernel | grep SO #文件名
```

```
Objdump -G obj/kern/kernel | grep FUN #函数名
```

可以查看 boot 过程中的调用函数及其文件。

再继续看提示可以找到 /kern/kdebug.h，仔细阅读可以发现 Eipdebuginfo 中的成员可以满足我们这项工作的需求，即：

```
"Info.eip_file:info.eip_line: info.eip_fn_name+(eip-info.eip_fn_addr)"
```

那么继续看其实现 /kern/kdebug.c，发现只有一个成员函数需要自己写（eip\_line），模仿一下其他成员的写法，再参考 stab\_binsearch()+stab.h 即可完成。

现在所需要的变量都获得了 ( 在实例化的 Eipdebuginfo 中 ), 回到 exercise14 的位置按照格式打印出来即可。

Exercise 16. Recall the buffer overflow attack in ICS Lab. Modify your start\_overflow function to use a technique similar to the buffer overflow to invoke the do\_overflow function. You must use the above cprintf function with the %n specifier you augmented in "Exercise 9" to do this job, or else you won't get the points of this exercise, and the do\_overflow function should return normally.

/\*最开始没看懂这题要干嘛, 看到了之后感觉好深井冰, 要自己炸自己的 stack。。。\*/

与在 icslab3 做过的类似, 这个甚至更简单, 因为有内核控制权可以轻松获取栈的信息, 直接把 overflow\_me 压栈的 return address 替换成 do\_overflow 的地址就行了, 另外由于要正常返回所以要保护 do\_overflow 的 return address 不要被刷掉就行。只是这个题要用 printf 写而不能直接进行地址替换, 那么就用垃圾字符串( 用'\0'来控制长度比较简单 )往上顶就行了, 需要注意的就是在 exercise9 的时候写过 %n flag 本身是有 overflow 情况的, 注意不要被自己搬的石头砸了自己的脚。

Exercise 17. There is a "time" command in Linux. The command counts the program's running time. In this exercise, you need to implement a rather easy "time" command. The output of the "time" is the running time (in clocks cycles) of the command. The usage of this command is like this: "time [command]".

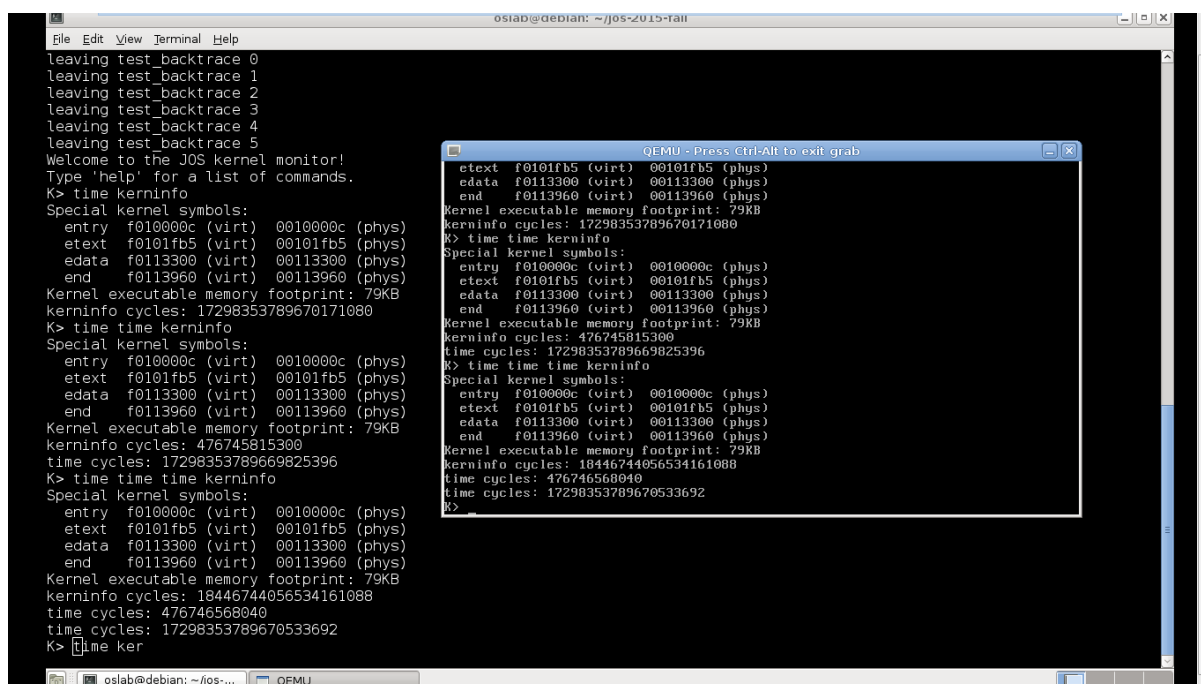
```
K> time kerninfo
Special kernel symbols:
_start f010000c (virt) 0010000c (phys)
etext f0101a75 (virt) 00101a75 (phys)
edata f010f320 (virt) 0010f320 (phys)
end f010f980 (virt) 0010f980 (phys)
Kernel executable memory footprint: 63KB
kerninfo cycles: 23199409
```

K>

Here, 23199409 is the running time of the program in cycles. As JOS has no support for time system, we could use CPU time stamp counter to measure the time.

Hint: You can refer to instruction "rdtsc" in Intel Mannual for measuring time stamp. ("rdtsc" may not be very accurate in virtual machine environment. But it's not a problem in this exercise.)

最后一个练习是关于一条新指令的，比较简单，使用 `read_tsc()` 这个底层函数来计数，另外实现一下命令嵌套(用递归)即可，这个练习好像在 `make grade` 中没有占分，我贴一下自己的运行结果如下：



```
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> time kerninfo
Special kernel symbols:
  entry f010000c (virt) 0010000c (phys)
  etext f0101fb5 (virt) 00101fb5 (phys)
  edata f0113300 (virt) 00113300 (phys)
  end f0113960 (virt) 00113960 (phys)
Kernel executable memory footprint: 79KB
kerninfo cycles: 17298353789670171080
K> time time kerninfo
Special kernel symbols:
  entry f010000c (virt) 0010000c (phys)
  etext f0101fb5 (virt) 00101fb5 (phys)
  edata f0113300 (virt) 00113300 (phys)
  end f0113960 (virt) 00113960 (phys)
Kernel executable memory footprint: 79KB
kerninfo cycles: 476745815300
time cycles: 17298353789669825396
K> time time time kerninfo
Special kernel symbols:
  entry f010000c (virt) 0010000c (phys)
  etext f0101fb5 (virt) 00101fb5 (phys)
  edata f0113300 (virt) 00113300 (phys)
  end f0113960 (virt) 00113960 (phys)
Kernel executable memory footprint: 79KB
kerninfo cycles: 18446744056534161088
time cycles: 476746568040
time cycles: 17298353789670533692
K> time ker
```

## 4.得分

我在自己的虚拟机（网站提供的镜像）上 `make grade` 跑分为 90/90，截图如下：



```
make[1]: Leaving directory `/home/oslab/jos-2015-fall'
make[1]: Entering directory `/home/oslab/jos-2015-fall'
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 399 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory `/home/oslab/jos-2015-fall'
sh ./grade-lab1.sh
make[1]: Entering directory `/home/oslab/jos-2015-fall'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/oslab/jos-2015-fall'
Printf: OK (36.0s)
OK (36.0s)
OK (36.0s)
OK (36.0s)
Backtrace: Count OK, Args OK, Symbols OK (36.0s)
OK (36.0s)
Score: 90/90
oslab@debian:~/jos-2015-fall$
```

如果 ta 在测试的时候出现问题也请练习我 [wy30123@163.com](mailto:wy30123@163.com) 。谢谢。