JOS-Lab-1 实验报告

熊伟伦
5120379076
azardf4yy@gmail.com

2014 年 9 月 26 日 - 10 月 6 日

目录

1 前言

这个 lab 包括文档我大概花了数十个小时（应该不超过 30 个小时）。

最坑的地方还是版本问题，我在 ubuntu 研究 qemu 的版本花了好几个小时，后来用 ubuntu 的 gcc 4.8 本来能跑练习 15 的 buffer overflow 的代码放在给的虚拟机里不能正常退出，发现是在 gcc 4.8 里的优化比 4.4 更强，跳过 overflow_me() 直接 call 了 start_overflow()，跳过了一层函数。ubuntu 虽然能用老版本的 gcc，但为了保险起见预防其他可能发生的问题，还是决定老老实实用给定的虚拟机跑，重新配置了 vim，vmware-tools 等工具。虽然开发环境的顺手很重要，但更重要的还是拿到分数。

2 环境配置

一开始的操作系统使用了 Ubuntu 14.04 64 位版本，QEMU 使用了 MIT 2009 年在 6.828 课程网站发布的打过补丁的 0.10.6 版本，即 ipads 课程网站所提供的虚拟机中使用的 QEMU，gcc 版本为 4.8.2，gdb 版本为 7.7。

做完之后放在给定的虚拟机里发现了问题，由于 gcc 版本不一致导致优化程度不一样，影响练习 15。保险起见，遂重新使用给定的虚拟机进行 lab。

该文档使用 XeLaTex 和 CTex 编写。

3 Lab1 架构描述

详细的每一部分在后面的练习描述中，这里大致概括下。

boot/文件夹为 boot loader 模块的源代码，conf/文件夹下为一些环境参数设置，inc/文件夹下包含了各个模块所需要的头文件，包括很多数据结构和宏，lib/文件夹下为一些辅助函数，主要和字符串输入输出有关，obj/下为生成的 asm 文件，kern/则为 kernel 模块的源代码。

首先，开机后程序在 0x000F0000 到 0x00100000 的位置会载入 BIOS，这一部分是主板通电后载入内存，是写死的。程序的初始 eip 为 0xffff0，在这个位置的指令是 jmp 0xFE05B，从这里开始会进行一系列的 IO 操作对各种硬件进行初始化，对这些 IO 口的分析在下面的练习中有写到。BIOS 还会将 boot 从硬盘的第一个扇区载入到内存的 0x7C00 到 0x7DFF 中，使用的是直接映射的技术，拷贝不需要通过 CPU 就能直接在内存和硬盘间进行，boot/sign.pl 会把 boot.S，main.c 编译而成的文件打包加上结尾符表示这 512 个 byte 是 boot。BIOS 最后的指令是跳转到 0x7C00 处，开始执行 boot。

boot 首先执行 boot.S 部分的代码，先做一些初始化设定包括禁止中断，打开 A20 地址线等，然后设置开启 32 位保护模式的一些寄存器，载入对应的 GDT。GDT 在 boot.S 的末尾进行了描述，偏移为 0，即虚拟地址就是物理地址，data 和 code 段的区

别在于限权上的区别，data 段可写，code 段可读可执行。GDT 载入完毕后会跳转到 32 位指令的保护模式中，.code32 指明了接下来的代码为 32 位编码。保护模式首先初始化各个寄存器，初始化完毕后跳转到 main.c 的代码中。

main.c 做的事首先是读取硬盘的第一个 page（一个 page 为 4096 byte），根据第一页中的 ELF 文件头的索引得到 ELF 各个部分的位置，获取内核的在硬盘中的全部位置，逐步将内核载入到内存中（这一循环读取载入的部分不详细说明，后面的练习中有分析），载入完毕后跳转到内核的 entry 入口标签。

内核首先执行的是 entry.S 文件中的指令，entry.S 首先将寻址模式从 32 位段模式进入 32 位段页模式，该模式的很多宏在 inc/文件夹下定义，包括虚拟地址映射到物理地址，偏移量为 0xF0000000。完成这些操作后，初始化栈空间，最后跳转到 i386_init 函数中。

i386_init 函数定义在 kern/init.c 文件中，这里做的事情就是整个程序表现在终端上给我们看到的形式，也就是整个系统最终停留的地方。大部分的练习的函数调用都在这里。先进行一系列的 cprintf 和 test_backtrace 的调用测试练习完成的情况。cprintf 主要测试格式化输出中的练习，包括 8 进制输出，空余格输出，计数等功能的测试。测试完格式化输出后，系统会调用 test_backtrace 函数，这个函数调用了我们需要写的 backtrace 函数进行函数的递归回溯查看，通过栈的结构一层一层显示回最上层的函数，其中还需要根据 ELF 的符号表进行一些 debug 的信息显示。完成这些步骤后该函数会调用 overflow_me 函数，我们填充这个函数覆盖掉它返回 caller 的 eip 跳转到 do_overflow 完成 buffer overflow 的练习。

最后完成这些指令后，调用 monitor 函数循环读取用户的指令，就像一个 linux 的 terminal 一样，monitor 函数由 monitor.c，monitor.h 和其他头文件定义，我们还需要完成一个 time 指令，通过 Intel x86 架构的 rdtsc 指令计算完成一个 terminal 的操作所需要的 CPU 周期。至此，这个 lab 的流程大致走完。

这就是执行 make qemu 的过程中会发生的一系列事情，也就是 lab1 的大致框架流程，详细的 cprintf 函数，monitor 等函数的扩展就不在架构描述中说明了，在后面的练习的解题分析过程中会详细描述。

4 第一部分: PC Bootstrap

> 练习 1：回忆起了 ICS 中用过的汇编指令，了解了下 80386 的段寄存器的演变历史以及在 80386 中段寄存器的使用方式，实模式和保护模式的寻址方式的差异等。先略读了这一部分，再后面的过程中碰到问题需要详细了解汇编指令再回头看。

整个 Bootstrap 过程大致是启动后加载 BIOS，早期的 BIOS 写在主板的 ROM 中被加载到内存的 0xf0000 到 0xfffff 的部分。初始的段寄存器 CS=0xf000，指令指针寄存器低 16 位 IP=0xfff0，根据 386 的实模式寻址，最终地址为 CS«4+IP=0xffff0。所

以从内存地址 0xffff0 开始执行第一条汇编指令，同样采用实模式的寻址方式 ljmp 到 0xfe05b 开始执行 BIOS 的一系列指令。

4.1 /boot/文件夹分析

看过/boot/文件夹下的文件后，得到结论：boot.S 是首先执行的 boot loader 的指令，主要完成初始化切换到保护模式的一些指令，最终 call bootmain 进入 main.c 编译的 boot loader。main.c 负责跳转到内核代码。而 sign.pl 文件的作用是将 boot.S 和 main.c 编译后的内容的空余部分填充空字符并且最结尾添加 0x55aa 表示这段 512 个 byte 的代码是 boot loader。

4.2 BIOS 执行分析

以下代码为 BIOS 启动后单步调试接下来执行的一系列指令，之后很长一部分指令没有 I\O 操作，故只截取该段进行分析。

```
1  0xffff0:      ljmp   $0xf000,$0xe05b
2  0xfe05b:      xor    %ax,%ax
3  0xfe05d:      out    %al,$0xd
4  0xfe05f:      out    %al,$0xda
5  0xfe061:      mov    $0xc0,%al
6  0xfe063:      out    %al,$0xd6
7  0xfe065:      mov    $0x0,%al
8  0xfe067:      out    %al,$0xd4
9  0xfe069:      mov    $0xf,%al
10 0xfe06b:      out    %al,$0x70
11 0xfe06d:      in     $0x71,%al
12 0xfe06f:      mov    %al,%bl
13 0xfe071:      mov    $0xf,%al
14 0xfe073:      out    %al,$0x70
15 0xfe075:      mov    $0x0,%al
16 0xfe077:      out    %al,$0x71
```

练习 2：首先吐槽下查看 I\O port 的连接地址需要翻墙，万恶的 GFW。
汇编指令 out %al, $port 向 $port 端口写入一个 8 位的数据，一系列操作首先输出输出%al 的值给 0xd 和 0xdaIO 口，分别与 DMA 进行通信并且获取 DMA 的控制权，随后向 0xd6 和 0xd4IO 口进行输出，可能是对 DMA 进行一些初始化设置，开启不经过 CPU 的直接存储器访问，可能会从网卡、声卡、显卡、硬盘等设备读入数据进入内存中，进行设备的初始化操作等等。之后还会对 0x70 和 0x71 口进行数据 IO 设置不可被中断，开启时钟操作。

5 第二部分: The Boot Loader

传统 PC 从 Disk 载入 Boot Loader，对于 Disk，BIOS 将其第一个 sector 共 512 byte（即 boot.S，main.c 以及 sign.pl 生成的 512 byte 的数据）载入进内存的 0x7c00 到 0x7dff 中，并且跳转到 0x7c00 开始执行载入的 Boot Loader。

当然，现代的 BIOS 也可以从 CD-ROM 以及 USB 启动，就如同我们装系统时进入的 BIOS 选择启动设备顺序一样，不同的设备的载入数据长度也不一样。

5.1 切换到保护模式

```
1  .globl start
2  start:
3    .code16                     # Assemble for 16-bit mode
4    cli                         # Disable interrupts
5    cld                         # String operations increment
6
7    # Set up the important data segment registers (DS, ES, SS).
8    xorw    %ax,%ax             # Segment number zero
9    movw    %ax,%ds             # -> Data Segment
10   movw    %ax,%es             # -> Extra Segment
11   movw    %ax,%ss             # -> Stack Segment
```

上述汇编代码是进入 Boot Loader 首先执行的指令,cli 和 cld 对 CPU 进行一些控制设置，然后将寄存器清零。

```
1  # Enable A20:
2    #   For backwards compatibility with the earliest PCs, physical
3    #   address line 20 is tied low, so that addresses higher than
4    #   1MB wrap around to zero by default.  This code undoes this.
5  seta20.1:
6    inb     $0x64,%al               # Wait for not busy
7    testb   $0x2,%al
8    jnz     seta20.1
9
10   movb    $0xd1,%al               # 0xd1 -> port 0x64
11   outb    %al,$0x64
12
13 seta20.2:
14   inb     $0x64,%al               # Wait for not busy
15   testb   $0x2,%al
16   jnz     seta20.2
```

```
17
18    movb    $0xdf,%al               # 0xdf -> port 0x60
19    outb    %al,$0x60
```

这一部分内容经过我上网查看了解，大概是为了兼容 8086/8088 的寻址模式，需要打开 A20 地址线，才能访问高位内存，属于为了兼容而遗留的代码。由此联想到 Intel 现在的指令集依然兼容很多年前的，导致指令集非常庞大，电路设计需要额外的一部分兼容老版本的指令，导致设计越来越复杂。

练习 3:

(1)

```
1    lgdt    gdtdesc
2    movl    %cr0, %eax
3    orl     $CR0_PE_ON, %eax
4    movl    %eax, %cr0
5
6    # Jump to next instruction, but in 32-bit code segment.
7    # Switches processor into 32-bit mode.
8    ljmp    $PROT_MODE_CSEG, $protcseg
9
10   .code32                     # Assemble for 32-bit mode
11 protcseg:
```

上述代码的第 1 到 4 行将 cr0 寄存器的最后一位设置为 1 开启了保护模式，并且跳转到 32 位模式的汇编代码中，正式从实模式进入保护模式。

(2) 短短的 main.c 生成的汇编真是很长。

执行的最后一条语句为进入内核入口:

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

对应汇编为:

```
1 7d61: ff 15 18 00 01 00           call   *0x10018
```

上一条 call 的是对应地址所保存的值，在 gdb 中输入

```
(gdb) p/x(*0x10018)
$S1 = 0x10000c
```

果然下一条指令，即 kern 的第一条指令为:

```
1 0x10000c:     movw    $0x1234, 0x472
```

随后找到这条指令存在于/kern/entry.S 中的第 44 行

```
1 .global entry
2 entry:
3         movw    $0x1234, 0x472          #warm boot
```

(3) 参考下面的代码（来自 main.c 的 bootmain 函数中），Boot Loader 首先读取 disk 的第一个 page，得到 ELF 头，随后验证得到的数据是否是 ELF 头，然后读取第一个 program 头的地址和 program 的总数，通过 readseg 从第一个 program 开始循环读取所有的 program 到内存中，readseg 会调用 readsect 并且根据读取的数据长度选择对应的 sector 进行读取。readsect 函数十分底层，直接根据 sector 偏移调用 IO 口从 disk 读取数据。

```
1  // read 1st page off disk
2  readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
3
4  // is this a valid ELF?
5  if (ELFHDR->e_magic != ELF_MAGIC)
6          goto bad;
7
8  // load each program segment (ignores ph flags)
9  ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
10 eph = ph + ELFHDR->e_phnum;
11 for (; ph < eph; ph++)
12         // p_pa is the load address of this segment (as well
13         // as the physical address)
14         readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

5.2 载入内核

练习 4：K&R 的那本 C Programming Language 的影印版我大一下学期就买了，排版巨烂，还不如电子版。

pointers.c 大部分理解难度不大，有几个比较有意思的地方。

```
1 3[c] = 302;
```

我第一次看见还可以这样写。

```
1 c = (int *) ((char *) c + 1);
2 *c = 500;
```

这两行代码是将 c 先视作 char 指针，因此加 1 只移动 8 位而不是 32 位，所以此时再视作 int 指针，c 指向的 int 为 a[1] 的第 9 到第 32 位以及 a[2] 的低 8 位，修改 c 指向的值会同时覆盖 a[1] 跟 a[2]。

练习 5：从之前得到，BIOS 跳转到 0x7c00 开始执行 Boot Loader，Boot Loader 载入完内核后跳转到 0x10000c 开始执行内核指令，所以在这 2 个点设置断点。

从下面的输出可以看出，在 Boot Loader 执行之前从 0x100000 开始的 32 个 byte 都是 0，而 Boot Loader 执行完之后有数据了。答案很明显，因为 0x100000 属于内核代码，在 Boot Loader 之前还没有将内核代码载入到内存中，在 Breakpoint2 跳转到内核执行，此时内核代码已经载入，所以 0x100000 有数据。

```
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
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 0x100000
0x100000:  0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010:  0x34000004  0x0000b812  0x220f0011  0xc0200fd8
```

5.3 链接地址 vs 载入地址

练习 6：根据练习上下的说明介绍，大致是说生成的 Binary 的地址和指令的关系如果和 link 阶段设置的全局变量值不一致，会导致一些依赖相对位置的指令出错。

先不改变/boot/Makefrag，make 之后查看/obj/boot/boot.asm，下面截取第一条指令和跳转到 32 位保护模式的指令：

```
start:
  .code16
  cli
    7c00:       fa      cli
```

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

```
7c2d:    ea 32 7c 08 00 66 b8   ljmp     $0xb866,$0x87c32
```

然后，我将/boot/Makefrag 中的 0x7c00 改为 0x7c04，目的是使得 BIOS 执行完毕后跳转到 0x7c04，make 后查看/obj/boot/boot.asm，同样的两条指令：

```
start:
  .code16
  cli
    7c04:        fa        cli
```

```
ljmp    $PROT_MODE_CSEG, $protcseg
  7c31:    ea 36 7c 08 00 66 b8   ljmp     $0xb866,$0x87c36
```

可知该文件根据 link 生成，地址是相对生成的地址，但是 break *0x7c00 后发现单步调试的指令依然符合原来的，说明生成的 Binary 文件不变，但当执行到 ljmp 跳转保护模式时, 如下：

```
[    0:7c2d] => 0x7c2d:   ljmp    $0x8,$0x7c36
```

可知与相对地址无关的 Binary 在修改 link 的参数后不变，但是地址相关的操作例如 ljmp 则会受到影响，因此 jmp 之后指令会乱掉，随后继续 si 竟然进入了 0xfe05b，与预期的比肯定不对了。值得一提的是，前面的 seta20 段代码也有 jmp，但是由于条件跳转没有执行，所以指令没有乱掉。

6 第三部分: The Kernel

6.1 使用分段操作位置相关

根据提示，我用 gdb 执行到 kern 的第一步指令地址为 0x10000c（load address)，而/obj/kern/kernal.asm(link address) 对应的指令地址为 0xf010000c，这也许就是指导中说的不一致。根据指导，0xf010000c 为虚拟地址，对应的物理地址为 0x0010000c。此时 kern 还没有切换到虚拟地址。

练习 7：继续上面所说，进入 kern 后继续单步调试，可以看见在 0x0010002d 执行完后新的虚拟物理地址对应起效了，下一个指令的地址在 0xf010002f。

```
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
(gdb) p/x(*0x10018)
$1 = 0x10000c
(gdb) si
=> 0x100015:    mov    $0x110000,%eax
```

```
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb)
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb)
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb)
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb)
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb)
=> 0x10002d:    jmp    *%eax
0x0010002d in ?? ()
(gdb)
=> 0xf010002f <relocated>:      mov    $0x0,%ebp
relocated () at kern/entry.S:73
73              movl    $0x0,%ebp               # nuke frame pointer
```

查看对应的/kern/entry.S，对应的是这么几句代码（我把注释去掉了）：

```
1  movw    $0x1234,0x472                   # warm boot
2  movl    $(RELOC(entry_pgdir)), %eax
3  movl    %eax, %cr3
4  movl    %cr0, %eax
5  orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
6  movl    %eax, %cr0
7  mov     $relocated, %eax
8  jmp     *%eax
9
10 relocated:
11 movl    $0x0,%ebp                       # nuke frame pointer
```

可见在上述代码的第 2 到 6 行（即/kern/entry.S 中的 56 到 61 行），读取 entry_pddir 更新 GDT。

总而言之经过这几行命令读取了新的 GDT，追查 entry_pgdir，它又定义在/kern/entrypgdir.c 中。

```
__attribute__((__aligned__(PGSIZE)))
pde_t entry_pgdir[NPDENTRIES] = {
        // Map VA's [0, 4MB) to PA's [0, 4MB)
        [0]
                = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,
        // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
        [KERNBASE>>PDXSHIFT]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P + PTE_W
};
```

可见新的 GDT 将输入的虚拟地址进行 -KERNBASE 运算，其中 PTE_P 和 PTE_W 定义在/inc/mmu.h 中，是 page 操作的 flag，不详细去查看。KERNBASE 定义在/inc/memlayout.h 中：

```
#define KERNBASE  0xF0000000
```

-KERNBASE 操作就把最高的 f 去掉，因此得到的虚拟地址和物理地址的映射。

为了验证玩坏虚拟地址映射后哪一步会出问题，我将/kern/entry.S 的加载 GDT 的 56 到 61 行注释掉，如下所示，加载到 jmp 指令时根据 link 载入到%eax 的值为 $0xf010001c，跳转到 relocated 标签，但由于没有进行虚拟地址物理地址的映射，出现了：

```
0xf010001c <relocated>: (bad)
```

即下方完整一段 gdb 指令显示出的第 17 行。因此这条 jmp 指令是没有更新 GDT 后第一条出错的指令。

```
=> 0x7d61:      call   *0x10018

Breakpoint 1, 0x00007d61 in ?? ()
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
(gdb)
=> 0x100015:    mov    $0xf010001c,%eax
0x00100015 in ?? ()
(gdb)
=> 0x10001a:    jmp    *%eax
0x0010001a in ?? ()
```

```
(gdb) p/x($eax)
$1 = 0xf010001c
(gdb) si
=> 0xf010001c <relocated>:      (bad)
relocated () at kern/entry.S:73
73              movl    $0x0,%ebp               # nuke frame pointer
(gdb)
```

6.2 格式化打印到控制台

练习 8：首先实现八进制输出，可以看到正常的输出会出现这么一段。

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

追踪程序运行，大致从 kernel 的 entry 进入 i386_init，递归地调用了 cprintf,vcprintf，vprintfmt，putch，ccutchar，cons_putchar，serial_putc，lpt_putc，cga_putc，再在最后 3 个函数中调用 IO 口，最终显示在屏幕上。

修改输出为 8 进制，找到对应的代码在/lib/printfmt.c 的 vprintfmt 的 case'o'，只需要模仿 10 进制和 16 进制，将代码修改为：

```
case 'o':
        // Replace this with your code.
        // display a number in octal form and the form should begin with '0'

        /* origin code
        putch('X', putdat);
        putch('X', putdat);
        putch('X', putdat);
        break;
        */

        // solution for exercise-8
        putch('0', putdat);
        num = getuint(&ap, lflag);
        base = 8;
        goto number;
```

屏幕输出：

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

(1) 根据前面的分析我们知道，/lib/printfmt.c 和/kern/console.c 的交互是 printfmt.c 调用 console.c 的 cputchar 函数。属于从高层调用底端显示等层。

```
static void
putch(int ch, int *cnt)
{
        cputchar(ch);
    (*cnt)++;
}
```

```
void
cputchar(int c)
{
        cons_putc(c);
}
```

(2)

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

可以看见上面的代码后面就是调用 IO 口的代码，函数名叫 cga_putc，可推出是直接调用显卡等 IO 设备进行字符串输出的代码。根据代码推测，大概用于检测屏幕一行是否输出满了，如果满了就出现一个空行，当前位置移动到空行行首。

(3) 这个练习好麻烦，我已经要抓狂了。冷静了一下，我在/kern/init.c 增加了如下的第 2,3 行代码：

```
cprintf("6828 decimal is %o octal!%n\n%n", 6828, &chnum1, &chnum2);
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

make 后找到对应的/obj/kern/kernel.asm 中的代码：

```
        int x = 1, y = 3, z = 4;
        cprintf("x %d, y %x, z %d\n", x, y, z);
f0100131:       c7 44 24 0c 04 00 00    movl   $0x4,0xc(%esp)
f0100138:       00
```

```
5  f0100139:       c7 44 24 08 03 00 00    movl   $0x3,0x8(%esp)
6  f0100140:       00
7  f0100141:       c7 44 24 04 01 00 00    movl   $0x1,0x4(%esp)
8  f0100148:       00
9  f0100149:       c7 04 24 37 1b 10 f0    movl   $0xf0101b37,(%esp)
10 f0100150:       e8 1c 09 00 00          call   f0100a71 <cprintf>
```

在 f0100150 设了个断点开始逐步调试，开始回答问题：

根据/kern/printf.c，结合平时写代码的经验，很显然 fmt 指向传入的字符串，ap 指向传入的额外的参数。

进入 cprintf 函数，我单步调试了好久，打印寄存器里的值得到 ap 的值和 fmt 的值，验证了上面的说法，call vcprintf 之前先将参数压栈再 call，然后我还要进入/lib/printfmt.c 的 vprintf 函数查看调用，这个函数就是开始修改 8 进制输出的那个函数，巨长，而且生成的汇编简直长的不能忍。看了 va_arg 函数的参数和实现，这个函数就是把一个参数的指针算上第二个参数类型将指针往后移动，即读取下一个 cprintf 的额外参数。这题长哭了。

然后我把自己增加的代码删掉了。

(4) 我又在老地方改了代码，如下第 2,3 行：

```
1 cprintf("6828␣decimal␣is␣%o␣octal!%n\n%n", 6828, &chnum1, &chnum2);
2 unsigned int i = 0x00646c72;
3 cprintf("H%x␣Wo%s", 57616, &i);
```

输出（我把后面的 pading 去掉了，因为没有换行，另外 110 是数字不是字母）：

```
He110 World
```

接下来进行分析，十进制的 57616 就是 0xe110，0x00646c72 用小端法表示就是 0x72，0x6c，0x64，0x00，对应于 ASCII 就是"rld\0"

假设是大端法表示的话，i 需要倒过来，即 0x726c6400，而对于 57616，既然是大端法写入的时候是大端法表示的数字，读出也就是大端法表示，存储方式改变，写入和读取不变，所以不改变数字依然输出 e110。

然后我又把修改的代码删掉重新 make 了。

(5) 这道题比较简单并且重复前面的了我就不写代码验证了，可以预测到，x 正确输出 3，而 y 会输出一个莫名其妙的东西，因为根据第 3 问的分析可知 ap 会继续向后移动到一块没有修改任何数值的未知用途的区域。

(6) 既然题目说 GCC 改变了参数压栈的顺序，那么对应的我们只要将 cprintf 中的额外参数的传入顺序逆过来就好了，前面分析过这个内容通过 ap 指针控制，可以将 ap 指针和 va_arg 反过来操作。

练习 9：现在发现练习 8,9 都可以跑 grade，练习 8 我看输出是对的但是跑不出来后来发现要把 jos.out 删掉再跑，坑了一段时间，因此接下来的练习都和 grade 有关，可以做完了就测，目前是 20 分。

```
vprintfmt(void (*putch)(int, void*),
                void *putdat, const char *fmt, va_list ap)
```

再次梳理下 vprintfmt 参数的意义，putch 是传入的底层调用的函数，putdat 是传入的字符串读取到的位置，是一个在外部的函数申明的引用，fmt 就是字符串的开头的指针，ap 就是额外参数的对象。接下来开始实现功能。

```
1  char* input_pos = putdat;
2  char* extra_para = va_arg(ap, char*);
3  if (extra_para == NULL) {
4          cprintf("%s", null_error);
5  }
6  else if ((*input_pos) & 0x80){          // if > 127
7          cprintf("%s", overflow_error);
8          *extra_para = 0xff;             // -1
9  }
10 else {
11         *extra_para = *input_pos;
12 }
```

这一段代码也在我提交的源代码中，input_pos 读取 putdat，extra_para 获取需要写入的值的指针，然后判断是否是空指针，是否大于 sign char 的最大值 127，由于 C 语言不好在 if 中做强制转换，sign char 与 int 做大小比较有点怪怪的，而且范围是不允许在 -1 到 -128 之间，所以用一个优雅的位运算就能解决这个判断问题，我看了 grade 脚本在 overflow 的情况下返回 -1 给参数，所以将 0xff 给 sign char，最终没有问题的话就把 input_pos 指向的值传给参数就好了。目前得分 30 分。

练习 10：

```
1  static int padding_space = 0;
2  static int padding_max_width = 0;
3  static int one_number_flag = 0;
4  static void
5  printnum(void (*putch)(int, void*), void *putdat,
```

```
6           unsigned long long num, unsigned base, int width, int padc)
7 {
8     if (width > padding_max_width)
9         padding_max_width = width;
10    if (num >= base) {
11        if (one_number_flag == 0)
12            one_number_flag = 2;
13        printnum(putch, putdat, num / base, base, width - 1, padc);
14    } else {
15        if (one_number_flag == 0)
16            one_number_flag = 1;
17        while (--width > 0)
18            padding_space++;
19    }
20    putch("0123456789abcdef"[num % base], putdat);
21    if (width == padding_max_width || one_number_flag == 1){
22        while(padding_space-- > 0)
23            putch(' ', putdat);
24        padding_space = 0;
25        padding_max_width = 0;
26        one_number_flag = 0;
27    }
28 }
```

上述代码在我提交的/lib/printfmt.c 中可以找到，我去掉了注释。其中我额外声明了 3 个全局变量，padding_space 用于记录需要空的个数，padding_max_width 作为一个比较的 flag 让我能比较方便的判断出什么时候输出最后一个字符，然后可以输出空格，用这个判定会遗漏不超过 base 的数，所以再用 one_number_flag 判断是否是一位的数进行修补。

输出：

```
pading space in the right to number 22: 22      .
```

然后我还修改 init.c 并测试了几个不同情况都符合预期，目前得分 40。

6.3 栈

练习 11：

```
1 movl    $0x0,%ebp                         # nuke frame pointer
2
```

```
3 # Set the stack pointer
4 movl    $(bootstacktop),%esp
```

以上这部分代码初始化了栈，它将 0 赋给 ebp 寄存器，将下面代码中标记的 bootstack 赋给 esp 寄存器，即地址最高的栈低。栈存储在 ELF 的.data 段。

```
1     .p2align    PGSHIFT         # force page alignment
2     .globl      bootstack
3 bootstack:
4     .space      KSTKSIZE
5     .globl      bootstacktop
6 bootstacktop:
```

练习 12：查看/obj/kern/kernel.asm 的 76 行开始直到调用 92 行的递归 call，一共压栈了 4+4+20+4=32 个 byte。

这个时候有人告诉我不用写练习的步骤，然后我仔细看了下课程网站的说明，确实不用写，接下来的几个练习我就简单的说明下我写的过程。

练习 13：主要规范下输出的格式，符合 shell 测试脚本。

练习 14：这个练习大部分时间在进行源代码和资料的阅读。查阅了一些 ELF 的资料，ELF 文件中会有 symbol table 保留进行 debug 用，这大概就是 debug 和 release 编译的区别了。因此根据一步步提示，并且查看其他的源代码进行模仿，完成这个练习。函数主要的过程是先查找对应的文件名，然后查找对应的函数名，最后找到行号，一步一步来。使用提供的二分查找的函数十分方便就能完成练习。之后我查看发现 eip 永远是在这个函数调用指令的后一个指令，存储规则就是这样，影响不大。

练习 15：花费时间比较长，而且 gcc4.8 和 4.4 的优化程度不一样导致我后来在提供的虚拟机上重写了这个练习，不过方法差不多。大致是将 stack_overflow 中保存的 ret 用的 eip 改到 do_overflow 函数中，要注意的是需要跳过 do_overflow 函数开头的栈整理的两行代码，然后 do_overflow 函数 ret 直接就会到 mon_backtrace 函数。

相当于原本是 start_overflow 应该 ret 到 overflow_me 函数再 ret 到 mon_backtrace 函数，通过注入将 overflow_me 函数替换成了 do_overflow 函数。

比较蛋疼的一点是练习要求使用 cprint 和%n 完成，因此我将原本可以直接传递的 do_overflow 地址通过%n 来完成，个人觉得这完全是一步没有意义的画蛇添足的要求。

练习 16：相对来说比前面几个跟栈有关的容易，查了查 rdtsc 的资料很快就完成了这个练习。主要在 monitor.c 中加了 mon_time 函数，另外还要再 monitor.h 中对函数进行注册。

另外我还对嵌套，多个参数进行了测试。发现嵌套和 linux 的几乎一模一样，time time kerninfo 这种，每次 time 会将 argc-=1，argv+=1，这样来去掉 time 指令递归地执行后面的。

在我的虚拟机中，time time，time help 这种都是 6 位数的样子，time kerninfo 是 7 位数，time backtrace 是 8 位数，完全符合规律。

至此，所有练习完成。鼓掌撒花。