OS Lab1 实验报告

（郑懿 1611361）

一、 实验准备

在本实验一开始，本人原本打算使用最新的 Ubuntu19.04，但是发现 gcc 版本过新，又无法降级，尝试了多种方法，都无法找到对应版本的 gcc。就改为较为旧的 Ubuntu14.04。虽然 gcc 版本依旧过高，只能暂且进行尝试，如果以后有需求在更改。

二、 实验过程

练习 1:

简单了解了一下 x86 汇编语言，并没有太大的困难。

练习 2:

实验所用的 qemu，只是简单的模拟了一下 BIOS 加载引导程序到内存中，然后启动系统。BIOS 首先执行了一个跳转指令：ljmp $0xf000,$0xe05b，因为 0xFFFF0 到 0x100000 只有 16 字节，因此选跳转到第一个地方执行。通过上网查阅资料可以知道，BIOS 如何判断从哪里加载 Boot Loader。BIOS 将所检查的磁盘的第一个扇区载入内存，放在 0x0000:0x7c00 处，如果该扇区最后两个字节是” 55 AA” ，那么就是一个引导扇区，否则继续检查下一个磁盘驱动器。

练习 3:

Q1:处理器什么时候开始执行 32 位代码？如何完成的从 16 位到 32 位模式的切换？

在阅读 boot/boot.S 文件中，可以看到一行注释：

```
.set CR0_PE_ON,      0x1         # protected mode enable flag
```

既然将 CRO_PE_ON 设置为 flag，于是寻找，这个 flag，可以发现 boot,S 有以下代码：

```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to their physical addresses, so that the
# effective memory map does not change during the switch.
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
```

注释里也写了是从这里开始进入保护模式的，再通过 gdb 逐步查看，发现该段代码出现在 0:7c26，所以实模式转化为保护模式是从 0:7c26 开始的。

Q2:引导加载程序 Boot Loader 执行的最后一个指令是什么，加载的内核的第一个指令是什么？

在阅读 main.ck 恶意看到，如果一切正常的话，最后一句是：

```
        // call the entry point from the ELF header
        // note: does not return!
        ((void (*)(void)) (ELFHDR->e_entry))();
```

随后继续阅读 boot.asm 中，了解编译后的文件可知：

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

因为引导系统加载内核，所以内核的第一条代码应该就是引导的下一行代码，因为在此处，call 了 0x10018，所以在 0x10018 寻找加载内核的第一条指令，
0x10000c:　　movw　　$0x1234,0x472

Q3:内核的第一条指令在哪里？
因为上一个问题已经找到了 0x10000c，所以第一条指令就是 0x10000c 所对应的指令。

Q4:引导加载程序如何决定为了从磁盘获取整个内核必须读取多少扇区？在哪里可以找到这些信息？，
因为涉及了 ELF 相关知识，在上网了解过相关知识后，阅读 main.c 时候发现，一句代码：

```
        // load each program segment (ignores ph flags)
        ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
```

注释告诉我们，就是这句代码分配了每个程序的扇区，所以主要是在 main 函数里制定了寻找的区域和读取的扇区的数量。

练习 4:
由于已经学过相关知识，就大概了解了一下。

练习 5:
根据指示，将 0x7c00 修改为 0x8c00，随后用反汇编查看，发现.Text 的 VMA LMA 都更改了地址。但随后调试的时候，由于连接器按照 0x8c00 的链接器起始位置对这些常量进行替换，但是 BIOS 还是把 bootloader 读到了 0x7c00 位置的，所以就产生冲突，在 ljmp 指令就报错了。是因为 BIOS 在出厂的时候就设置好了，把磁盘第一个扇区加载到了 0x7c00。

练习 6:
在代码进入到 bootloader 之前，可以发现 0x10000 都是 0，之后加载到了内核，在此查看就发现 0x10000 有了内容，原因是内核被加载进来，所以会有了存储。同时也说明了，每次计算机都会自动清空 0x10000 的空间。

练习 7:
通过 gdb 显示，在分页极致之后，两个地址只想了相同的地址：

```
(gdb) b *0x100025
Breakpoint 1 at 0x100025
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x100025:    mov    %eax,%cr0

Breakpoint 1, 0x00100025 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
(gdb) x/8x 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000      0x00000000      0x00000000      0
x00000000
0xf0100010 <entry+4>:   0x00000000      0x00000000      0x00000000      0x000000
00
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
(gdb) x/8x 0xf0100000
0xf0100000 <_start+4026531828>: 0x1badb002      0x00000000      0xe4524ffe      0
x7205c766
0xf0100010 <entry+4>:   0x34000004      0x0000b812      0x220f0011      0xc0200f
d8
(gdb)
```

而在我们注释掉了 movl %eax, %cr0，由于无法开启分页机制，所以导致程序崩溃。

练习 8:

对于打印八进制数字，只需要模仿十进制即可：

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

Q1:

通过阅读代码可以知道，console.c 是负责提供 cputchar 函数给 printf.c 中的 printf 函数调用，printf 中的函数负责分类解析各种各样的字符进行输出。

Q2:

这段代码通过阅读可以知道，是每次碰到每行写满，进行换行，将前面的行循环上移，再将光标达到屏幕最左端。

Q3:

cprintf 函数中，fmt 指向的是字符串，也就是 “x %d, y%x, z%d，ap 指的是参数的第一个，也就是 x。随后每次 ap 都会每次想下移动多需要的类型位置。

Q4:

输出结果是 HE110 Word，&i 对应的序列从低位开始计算的 72 6c 64 00 正好是字符串 rld。

如果变更高子节的，则只需要 i 改为 0x726c6400，57616 不需要更改。

Q5:

输出结果：

```
Worldx=3 y=-267321412
```

每次打印出来变量的值都是根据 va_arg 从 ap 指针不断向后取值，所以每次都会得到一个不固定的值，但在一段时间内是固定的。

Q6:

如果更改调用约定，压栈顺序正好相反，所以原来是加上去的地址，现在是减去。

Challenge：

首先，在 cga_putc 函数里找到了一句注释：

```
static void
cga_putc(int c)
{
        // if no attribute given, then use black on white
        if (!(c & ~0xFF))
                c |= 0x0700;
```

但并没有在该函数里找到如何更改颜色，在上网查阅资料后发现，应该如同汇编一样，高八位是颜色，所以在 cprintf 处更改，得出以下效果：

练习 9:

在 kernel 中有几句代码：

```
        movl    $0x0,%ebp

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

由注释，我们可以知道，是对%ebp 和%esp 进行初始化，将%esp 设置为栈顶部，往低地址伸长。

练习 10:

在 kern.asm 中有多 test_backtrace，参数从 5 减到了 1，每一次递归调用，%ebp 减去 0x14，分配给栈。

练习 11:

通过阅读 entry.S，我们可以知道%ebp 是通过指向老%ebp，所以类似于链表的方式，就可以一调用整个链。由于每次都会清楚%ebp，所以当他为空时候，停止调用，可以用一个 while 循环来停止，代码如下：

```
        int i;
        uint32_t eip;
        uint32_t* ebp = (uint32_t *)read_ebp();

        while (ebp) {
                eip = *(ebp + 1);
                cprintf("ebp %x eip %x args", ebp, eip);
                uint32_t *args = ebp + 2;
                for (i = 0; i < 5; i++) {
                        uint32_t argi = args[i];
                        cprintf(" %08x ", argi);
                }
                cprintf("\n");

                ebp = (uint32_t *) *ebp;
        }
        return 0;
```

回溯成功，所以方案是正确的。

练习 12:

通过阅读 kebug.c 里的几个函数，可以发现 stab_binsearch 是利用地址符号进行二分搜索，所以补全 debuginfo_eip 函数，存储下具体行号，代码如下：

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

随后，在十一题基础上，调用这个函数，并按要求输出，代码如下：

```
// Your code here.
int i;
uint32_t eip;
uint32_t* ebp = (uint32_t *)read_ebp();

while (ebp) {
        eip = *(ebp + 1);
        cprintf("ebp %x eip %x args", ebp, eip);
        uint32_t *args = ebp + 2;
        for (i = 0; i < 5; i++) {
                uint32_t argi = args[i];
                cprintf(" %08x ", argi);
        }
        cprintf("\n");
        struct Eipdebuginfo debug_info;
        debuginfo_eip(eip, &debug_info);
        cprintf("\t%s:%d: %.*s+%d\n",
                debug_info.eip_file,
                debug_info.eip_line,
                debug_info.eip_fn_namelen,
                debug_info.eip_fn_name,
                eip - debug_info.eip_fn_addr);
        ebp = (uint32_t *) *ebp;
}
```

就完成了要求。

三、 实验结果

使用 qemu，测试回溯函数结果：

```
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 f010ff18 eip f0100087 args 00000000  00000000  00000000  00000000  f01009b6
            kern/init.c:19: test_backtrace+71
ebp f010ff38 eip f0100069 args 00000000  00000001  f010ff78  00000000  f01009b6
            kern/init.c:16: test_backtrace+41
ebp f010ff58 eip f0100069 args 00000001  00000002  f010ff98  00000000  f01009b6
            kern/init.c:16: test_backtrace+41
ebp f010ff78 eip f0100069 args 00000002  00000003  f010ffb8  00000000  f01009b6
            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!
```

通过 make grade，测试最后结果：

```
running JOS: (0.9s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
```

痛殴这次试验，加深了汇编以及硬件相关一些知识，也为操作系统的学习奠定了基础。