Lab1过程记录

配置lab环境

直接下载sjtu提供的虚拟机，解压缩后在vmware中打开

```
git clone -b lab1 http://ipads.se.sjtu.edu.cn:1312/lab/jos-2019
cd jos-2019-spring
make && make grade
```

由于sjtu提供的虚拟机已经将环境安装好了所以编译顺利成功并且得分为零分，可以开始慢慢面向测试用例编程了

了解PC物理地址布局

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
```

```
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

其中最重要的是从0xF0000到0xFFFFF的占了64KB的BIOS，BIOS负责执行基本系统初始化，例如激活视频卡和检查安装的内存量。执行此初始化后，BIOS从某些适当的位置（如软盘，硬盘，CD-ROM或网络）加载操作系统，并将机器的控制权交给操作系统。

使用QEMU gdb

打开两个Terminal，在第一个Terminal中输入make qemu-gdb，然后在第二个Terminal中输入make gdb，看到如下输出

```
For help, type "help".
Type "apropos word" to search for commands related to "word".
+ target remote localhost:26000
warning: No executable has been specified and target does not su
determining executable automatically.  Try using the "file" comm
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
warning: A handler for the OS ABI "GNU/Linux" is not built into
of GDB.  Attempting to continue with the default i8086 settings.

(gdb)
```

可以看出PC先执行0x000ffff0物理地址的位置,这是在ROM BIOS 顶部64KB的地方，然后通过ljmp 跳到了0xfe05b(我也不知道为什么，通过si命令看到的)
然后使用gdb的si命令跟踪BIOS都做了些什么

当BIOS运行时，它会设置一个中断描述符表并初始化各种设备，如VGA显示器,初始化PCI总线和BIOS知道的所有重要设备后，它会搜索可引导设备，如

软盘，硬盘驱动器或CD-ROM。最终，当它找到可引导磁盘时，BIOS从磁盘读取引导加载程序并将控制权转移给它。

Boot Loader

当BIOS找到可引导的软盘或硬盘时，它将512字节的引导扇区加载到物理地址0x7c00到0x7dff的内存中，然后使用jmp指令将CS：IP设置为0000：7c00，将控制权传递给引导装载机。

在这个lab中将使用传统的硬盘启动机制，这意味着我们的启动加载程序必须适合512字节。引导加载程序由一个汇编语言源文件boot/boot.S和一个C源文件boot/main.c以及一个反编译出的汇编文件obj/boot/boot.asm组成

boot loader进行了模式切换,即从real mode切换成了32-bit protected mode，于此同时它直接从磁盘读取了kernel

从16位切换到32位的具体时间和代码如下

```
# 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
# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcse
```

通过gdb定位boot loader 和 kernel的交界处的两条命令分别是

```
call   *0x10018
movw   $0x1234,0x472
```

在boot/main.c的bootmain 函数里 `ELFHDR->e_phnum` 从而知道加载多少个扇区

printf底层实现

通过代码完成对应exercise

%o输出八进制以0引导，实现类似于case 'u' ， 将base变为8同时先putch ('0')即可

有符号的实现，先通过case '+' 来判断是否需要实现有符号的输出，然后在case 'd'中判断传入数字是否大于0，如果是则输出'+'

%-的实现，通过printnumhelp实现一个类似printnum的函数，通过递归的方式打印，并且先答应数字再答应padding的空格，在printnum中通过判定来绝对是否调用该函数

%n的实现，通过va_arg来读取接收字符串当前长度的变量，如果变量为null则报错，然后通过putdat来判断当前字符串长度，如果超过了254（255就overflow了），则报错并将变量设为-1，不然就将变量设为当前字符串长度

Stack

Stack 中的 栈帧 如下图所示

```
+------------------+
|                  |
|      函数参数     |
|                  |
+------------------+
|                  |
|                  |
|      返回地址     |
|                  |
|                  |
+------------------+
|      旧的ebp      |
+------------------+  <- 新的ebp
|                  |
|      本地变量     |
+------------------+  <- esp
|                  |
+------------------+
|                  |
|                  |
|                  |
+------------------+
```

通过read_ebp()函数得到当前ebp内所存的地址，也即当前旧的ebp的地址的地址，然后通过循环调用，以及上述x86栈帧结构得到了想要的数据，并通过特定的格式打印出来

通过debuginfo这个数据结构得到具体的信息，并在kdebug.c中通过在debuginfo_eip中调用stab_binsearch来得到对应的行所在的位置

通过printf中%n的使用将地址强行写入到返回的栈中，需要注意的是，因为要求正常返回，所以在返回地址中用do_overflow的地址来覆盖，并在下一行中用原来的正常地址覆盖，从而保证在do_overflow函数返回以后仍然可以进入正常的流程继续执行