LAB1

516030910391 徐天强

Part 1: PC Bootstrap

Exercise 1: 阅读

Exercise 2: use GDB' si

Part 2: The Boot Loader

Exercise 3:

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

ljmp $PORT_MODE_CSEG, $protcseg

在这个点 gdb 显示 the target architecture is assumed to be i386，通过这个 longjump，重新设置了 cs 和 eip 寄存器，使得之后在 32 位模式下寻址时，通过去 GDT 中查找对应的段地址。

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

Void (*)(void)) (ELFHDR->e_entry))()

Movw $0x1234, 0x472

3. 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?

Elf 的 header 提供了相关信息

Exercise 4: 阅读

Exercise 5:

1. 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.)

Push %ebx

Movw $0x1234, 0x472

因为在刚刚进入 bootloader 的时候，kernel 的代码还没有进入内存，而运行到 kernel 的代码时，bootloader 已经把 kernel 代码放入内存了

Exercise 6: 一些 magic number 发生了改变

Part 3: The Kernel

Exercise 7:

1. 设置 cr0，将 PE 位置上的时候
2. jmp *%eax

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'.

借鉴 16 进制 case 的写法，将 base 设置为 8 即可

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.

新增一个 plusflag，在识别到'%+'模式的时候就把 plusflag 置上，然后在之后判断，如果该 number 非负，就在它之前 print'+'即可

1. Explain the interface between printf.c and console.c. Specifically, what function does console.cexport? How is this function used by printf.c?

printf.c 调用 console.c 提供的函数。Console.c export cputchar

cputchar 往输出流中放一个字符

2. Explain the following from console.c:

将屏幕上的字符上移一行，然后新的一行中的字符全是空格

3. For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

Trace the execution of the following code step-by-step:

3.1 In the call to cprintf(), to what does fmt point? To what does ap point?

```
fmt = "x %d, y %x, z %d\n"
```

ap = int *[3] = {&x, &y, &z}

3.2 List (in order of execution) each call to cons_putc, va_arg, and vcprintf. For cons_putc, list its argument as well. For va_arg, list what ap points to before and after the call. For vcprintf list the values of its two arguments.

vcprintf -> cons_putc-> va_arg 调用关系：vcprintf 调用 cons_putc, cons_putc 调用 va_arg

4. Run the following code.

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

What is the output? Explain how this output is arrived out in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

output : He110 World

cprintf->vcprintf->vprintfmt-> va_arg

首先处理常规字符 H 然后通过%x 识别当前的 ap 指向的 int 需要按照 base=16 来输出，调用 printnum -> putch -> cputchar-> cons_putc

之后再处理常规字符 Wo, 之后识别到%s， 通过调用 va_arg 将 i 识别成 char *,并返回给指针 p，再一个一个调用 putch

4.1 The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

0x726c6400 不需要

5. In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

在栈上相邻的值

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

可以在将 cprintf 修改为接受两个参数，第二个参数接受一个指针数组

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.

使用该函数提供好的变量-putdat，该变量提供 printf 输出字符数的计数，因此，在识别'%n'模式之后，通过 va_arg，获得对应参数，然后将此参数 cast 为 char*之后，即可将 putdat 存入该参数所指的空间中。除此之外，需要对于 null 指针进行判断，又因为是 signed char，所以溢出判断为 127.

Exercise 11.

Modify the function printnum() in lib/printfmt.c to support "%-" when printing numbers. With the directives starting with "%-", the printed number should be left adjusted. (i.e., paddings are on the right side.) For example, the following function call:

```
cprintf("test:[%-5d]", 3)
```

, should give a result as
"test:[3]"
(4 spaces after '3'). Before modifying printnum(), make sure you know what happened in function vprintffmt().

利用已有的 printnum()函数，它实现了在输出数字左边输出空格的逻辑。为了不破坏已有的性质，通过 if-else 语句进行判断，如果 padding 的字符为‘-’即进入右边 padding 的逻辑（即我实现的逻辑）。使用 for 循环添加 padding 字符即可。

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?

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

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?

每次减小 0x20 即 32 即 32 个 bytes

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.

我们学习了 calling convention，了解到 eip 与函数参数与 ebp 的位置关系，这就不难通过取值得到 eip args 的值，通过*ebp 可以获得上一个 frame 的 ebp，并且学习源码之后发现当 ebp == 0 时即可停止 trace。通过 cprintf 将这些 print 出来即可。

题目还要求需要把这个函数添加到 commands 数组里，仿照 help 和 kerninfo 就可以完成。

Exercise 15. Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.

在 stab.h 中发现了这一行：#define N_SLINE 0x44// text segment line number

即可用它作为 stab_binsearch 的参数，找到具体的行数

查询 stab 数据结构可知，其中的 n_value 即为当前 stab 保存的值，将它放入 eip_line 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 10" to do this job, or else you

won't get the points of this exercise, and the do_overflow function should return normally.

这里比较 tricky 的地方就是需要 do_overflow 正确返回，因此需要将原有的 retaddr 上移 4 个 bytes，然后再将当前的 retaddr 改成 do_overflow

Exercise 17. There is a "time" command in Linux. The command counts the program's running time.

通过在网上查找的 rdtsc 的使用方法，获得开始执行时间和结束执行时间，做差即可获得执行时间。

这里的第二个参数，即为要执行的 command，我们通过遍历 monitor.c 中的 Command 数组，获得对应 command 的函数指针，传入参数个数与参数指针，调用该函数等待返回即可。