OSLab 1 Report

September 25, 2024

1 Overview

OSLab 1 consists of three main components: 1. PC Bootstrap, 2. Boot Loader, 3. Kernel

The primary objective of this lab is to familiarize students with the lab environment and essential tools. The boot loader process is quite straightforward: the boot sector is first loaded into memory, followed by the loading and execution of the kernel.

Most of the content covered in this lab overlaps with the ICS course, which simplifies the completion process and helps reinforce foundational concepts.

2 Exercises

2.1 Exercise 1

The first exercise aims to familiarize us with the lab environment. The only issue encountered was the need to install the correct version of GCC.

2.2 Exercise 2

The second exercise involves using GDB to trace into the ROM BIOS. Below are the first 10 instructions of the ROM BIOS:

1. **0xffff0: ljmp** $0x3630,$0xf000e05b
2. **0xfe05b: cmpw** $0x28,%cs:(%esi)
3. **0xfe062: jne** 0xd241d0ad
4. **0xfe066: xor** %edx,%edx
5. **0xfe068: mov** %edx,%ss
6. **0xfe06a: mov** $0x7000,%sp
7. **0xfe070: mov** $0xf39a,%dx
8. **0xfe076: jmp** 0x5575ec0d
9. **0xfec0b: cli**
10. **0xfec0c: cld**

These instructions are executed when the computer is powered on. The first instruction is a long jump to the reset vector, marking the entry point of the BIOS. The BIOS then initializes the hardware and loads the boot sector from the boot device. The last two instructions disable interrupts and clear the direction flag (I'm uncertain about the exact effect of the last two instructions).

2.3 Exercise 3

The third exercise requires answering questions about the boot loader.

2.3.1 Question 1

The code responsible for switching from real mode to protected mode is **0x7c00: ljmp** $0x8,$0x7c32, where the message *"The target architecture is set to 'i386'"* is printed.

2.3.2 Question 2

The final instruction of the boot loader is: $((void(*)(void))(ELFHDR->e\_entry))();$

The assembly representation is: **7d71: ff 15 18 00 01 00 call *0x10018**. The first instruction executed when the kernel is loaded is **movw** $0x1234,0x472 (this is part of the warm boot process).

2.3.3 Question 3

The first instruction of the kernel is located at **0x10000c**.

2.3.4 Question 4

Using the ELF file format, the boot loader loads the kernel into memory and jumps to the kernel's entry point.

2.4 Exercise 4

C programming has already been covered in previous courses.

2.5 Exercise 5

Changing the address can cause the boot loader to malfunction, possibly due to an incorrect address.

2.6 Exercise 6

Before the kernel is loaded, the first 8 words of memory at **0x00100000** are all zeros. After the kernel is loaded, the first 8 words at **0x00100000** are:

```
0x100000: 0x1badb002, 0x00000000, 0xe4524ffe, 0x7205c766
0x100010: 0x34000004, 0x2000b812, 0x220f0011, 0xc0200fd8
```

2.7 Exercise 7

The first incorrect instruction encountered is **jmp** *%eax.

2.8 Exercise 8

This exercise requires extending the **vprintfmt** function to support octal output. By replicating the code used for unsigned decimal output, we can easily implement the octal format. The process is straightforward by following the existing code structure.

One point of confusion was the need to add an extra \n before the test code to pass the test. This may be due to strict output matching requirements.

2.8.1 Question 1

The file **console.c** exports the **cputchar** function, which is used by **putch** in **printf.c**.

2.8.2 Question 2

The code is responsible for controlling the cursor position on the screen.

2.8.3 Question 3

The **fmt** pointer refers to the format string *"x %d, y %x, z %d"*.

2.8.4 Question 4

The output is *"Hello, world"*.

2.8.5 Question 5

The output displays the value stored at **12(%ebp)**.

2.9 Exercise 9

The stack pointer points to **bootstackup**.

2.10 Exercise 10

When **test_backtrace** is recursively called, three items are pushed onto the stack: **x-1**, the return address, and **%ebp**.

2.11 Exercise 11 & 12

These exercises involve implementing the backtrace function. Since we have already learned how the stack operates and what occurs during a function call in the ICS course, implementing the backtrace function was relatively straightforward.

Adding debugging information was achieved by mimicking the structure of the existing code.

2.12 Challenge

The challenge required adding support for colorful output. This was quite difficult, so I studied how to interpret ANSI escape sequences. I implemented a state machine to parse the escape sequences and output the corresponding colors, which I learned about through online resources.

I wrote a new **cga_put** function to interpret the escape sequences and print the corresponding colors. For normal characters, the original **cga_put** function was called.

Additionally, I implemented the **mon_set_color** function, which uses **cprintf** to output the corresponding color codes. While this function allows simultaneous changes to both the foreground and background colors, the output doesn't appear entirely smooth.

Finally, I registered the function in the **Command commands** structure and successfully tested the implementation.