OS Lab 1 Report

Author: Chengzu Ou (14302010037)

Description

Write a description about the procedure of the booting up. What difficulties did you meet during this lab and how you conquered them.

The first thing that a PC does when booting up is to execute instructions in BIOS. The BIOS is responsible for performing basic system initialization such as activating the video card and checking the amount of memory installed. After performing this initialization, the BIOS tries to load the operating system from some appropriate location. When the BIOS makes sure which disk the OS is in, it will load the first sector of the disk, a.k.a. *boot sector* into memory. Boot sector has a very important program called *boot loader* which is responsible for the loading of the entire operating system and some configuration work. After that, the operating system starts to run.

Thus, the procedure of booting up is basically BIOS -> boot loader -> operating system kernel.

At first I was trying to config the lab environment on my Ubuntu 16.04 Virtual Machine. But I keep getting errors when compiling `jos`. From the error message, I guess it must has something to do with the `gcc`. Since I did something changes to the `gcc` on this operating system to do the lab of another course, Computer Architecture, I figure that why bother trying to change the configuration back and forth when I can just simply use another machine to do the job. After all, it's all *virtual*. So I did this lab on my Ubuntu 18.04 Virtual Machine.

In exercise 3, I had some trouble understanding the relationship between `BIOS`, `boot/boot.s`, and `boot/main.c` at first. I can see that in `boot.s`, it switches from real mode to protected mode and in `main.c`, it loads the operating system kernel to the memory. Then I realize that `BIOS` is the first program to run when booting up the computer. After `BIOS` finished its job to set some hardware stuff up and load the first sector of the disk which stores the boot loader program, it calls `boot.s` to run boot loader. Boot loader comprise of `boot.s` and `main.c`.

In exercise 8, I first don't understand how `cprintf` could accept variable length of arguments. After searching online, I get that it is `va_start`, `va_arg`, and `va_end` that make all these things possible. But unfortunately, I could not see the exact operation of those functions(or macro to be precise). I took a bold guess and came up with the following answers.

In the last two exercises, it took me a lot of time figuring out the format of the output. I even went to see the grading script to get the desired output format.

Answers

Q1: At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

In file `boot.s`, cpu first works in real mode, which is 16-bit mode. After the instruction `ljmp $PROT_MODE_CSEG, $protcseg` in line 55, the processor switches to 32-bit mode.

The switch from 16- to 32-bit mode takes a lot of efforts. The part in `boot.s` that tries to enable line A20, including `seta20.1` and `seta20.2` is the preparation work of the switch.

The cause of the switch is also related to the instruction `lgdt gdtdesc` in line 48. This instruction load the value in `gdtdesc` into GDTR which is the register that stores some important information about GDT including the start address of GDT and the size of it.

After that, it sets the lowest bit of CR0, which is a control register, to 1. This indicates the start of protected mode.

Q2: What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

The last instruction that boot loader executed is the last statement of `bootmain()` in `boot.c` file, which is `((void (*)(void)) (ELFHDR->e_entry))();`. According to `boot.asm` file, this statement calls the function in address `*0x10018`, which is the start point of kernel program.

The first instruction of the kernel is in line 44 of `entry.s` file, `movw $0x1234,0x472`.

Q3: Where is the first instruction of the kernel?

According to `gdb`, we find that the first instruction of the kernel, which is `movw $0x1234,0x472` in `entry.s` is at address `0x10000c`.

Q4: 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?

The information of how many segments the OS has and how many sectors that each segment has is store in *program header table*, which in `boot.c` is the data structure `struct Proghdr`. In this table, every entry is correlated to a segment in OS and it contains the size of the segment, the start address of the sector, and the offset.

Program header table is stored in the header of ELF file, which starts at address `ELFHDR`.

Q5: Explain the interface between `printf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `printf.c`?

`console.c` is used exports function on how to show a character on console and some operation on I/O ports. Function `cputchar` is a high level console I/O used by `readline` and `cprintf`. It calls `cons_put`, which output a character to the console. There are 3 sub programs in `cons_putc`: `serial_putc`, `lpt_putc`, and `cga_putc`. `serial_putc` is used to output a character to serial port. `lpt_putc` is used to output a character to parallel port. And `cga_putc` is to show character on console.

`printf.c` defines some functions used for format printing such as `printf` and `sprintf`. Functions in this file calls functions exported by `printfmt.c` and eventually `console.c` to put characters on console, such as `cputchar` function.

Q6: Explain the following from `console.c`:

```
if (crt_pos >= CRT_SIZE) {
    int i;
    memmove(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;
}
```

`crt_buf` is a character buffer, in which stores characters to be put on console. `crt_pos` is the position of the last character on console.

When `crt_pos` is larger or equal to `CRT_SIZE` (which is `80*25`), for that we've already know that the range of `crt_pos` is from `0` to `80*25-1`, it means the content shown on console has exceeded on page. Thus we need to scroll down a line.

We use `memcpy` to copy characters from line `1` to line `79` in `crt_buf` to line `0` to line `78`. And use a `for` loop to set the last line, line `79`, to empty characters. At last, we update `crt_pos`.

Q7: Trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

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

In the call to `cprintf()`, `fmt` points to the format string to be shown. In this case, it's `"x %d, y %x, z %d\n"`.

The type of `ap` is `va_list`. It is used to handle variable number of arguments. It points to the set of all input arguments.

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.

The first function to be called is `vcprint`. It is called by `cprint`. The first argument, `fmt`, is the formatted string `"x %d, y %x, z %d\n"`, and the second argument, `ap` is the list of other arguments passed to `cprint`, in this case, `x, y, z`.

The second function to be called is `cons_putc`. It is called by function `cputchar`, which is called by `putch`, which is called by `printfmt` after the calling of `getint`. The argument of `cons_putc` is just a single `c`, which in this case is `"x"` because it's the first character to be printed on console.

The last function to be called is `va_arg`. It is called by function `getint` in `vprintfmt` which is called by `vcprint`. The first time that function `va_arg` is called, it's called to handle `"x %d"`. Thus, before calling, `ap` points to `x, y, z`, which is `1, 3, 4`. After calling, `x` is removed from the list, leaving only `y, z`, which is `3, 4`, in the list that `ap` points to.

Q8: 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 at in the step-by-step manner of the previous exercise.

The output is `"He110, World"`.

When calling `cprint` to print the string, it first read through the string and print every "normal" characters, by which I mean the character that is not start with `%`. Thus it will first print `H` just as usual.

Then, it comes to `%x`. This mean print the first argument after the string as hex. The first argument is `57616`. The corresponding hex is `e110`. Thus it prints `He110`.

The next "abnormal" character is `%s`, which asks the second argument to be printed as a string. Unfortunately, the second argument is `&i`, which is the address of the variable `i`, which is an unsigned integer. So we should just print `i` as a string. For that x86 is little-endian, so the `int` type variable `i` will be stored as `0x72, 0x6c, 0x64, 0x00` within memory, which is `'r', 'l', 'd', '\0'` according to ASCII table.

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?

If x86 is big-endian, `i` should be `0x726c6400`. We do not need to modify `57616` because it has nothing to do with little-endian or big-endian.

Q9: In the following code, what is going to be printed after `'y='`? (note: the answer is not a specific value.) Why does this happen?

```
cprintf("x=%d y=%d", 3);
```

The output is `x=3 y=-267380580`.

It is because we does not specify the argument of `y`, so the output value will be uncertain.

Q10: 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?

As we can see, `cprintf` use `va_start` to calculate the start address of `ap` and `va_arg` to get the current argument in `ap` and move the point to the next one.

If GCC changed its calling convention so that it pushed arguments on the stack in declaration order, then we should change the behavior of `va_start` and `va_arg` so that they would **subtract** the address rather than **add** the address to get the next argument.