Lab 1: Booting a PC

notes and answers

Kangwei Ling

February 11, 2017

Contents

1 Part 1: PC Bootstrap **1**

2 Part 2: The Boot Loader **1**

2.1 Loading the Kernel 2

3 Part 3: The Kernel **2**

3.1 Using virtual memory to work around position dependence 2

3.2 Formatted Printing to the Console 3

3.3 The Stack 6

1 Part 1: PC Bootstrap

2 Part 2: The Boot Loader

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

  The instruction `ljmp $PROT_MODE_CSEG, $protcseg` does the job. After this instruction, the processor start executing 32-bit code. This jump clears the processor pre-fetch queue (with 16bit code) start executing code specified with 32-bit code.

- 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 of the boot loader executed is

```
call *0x10018
```

The first instruction of the kernel just loads is

```
movw   $0x1234,0x472
```

- Where is the first instruction of the kernel?

  `0x1000c`

- 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 number of program segments is in the elf header. As sector 1 which contains the elf header is loaded at first, we can get the information. And for each program segment, the program header contains enough information about the offset, total size of the segment.

2.1 Loading the Kernel

- **Exercise 5**

  `lgdt gdtdesc` would go wrong if the link address of the bootloader is changed. And the program actually stops when executing the long jump instruction `ljmp $PROT_MODE_CSEG, $protcseg`, because this long jump's address are totally wrong which is based on the link address, different from the real loaded address.

  absolute jump and access the global data engender errors due to the incorrection of the address.

- **Exercise 6**

  At the point the BIOS enters the boot loader, the 8 words are all 0. At the point the boot loader enters the kernel, the 8 words are filled with code of the kernel, which is just loaded from disk.

3 Part 3: The Kernel

3.1 Using virtual memory to work around position dependence

- **Exercise 7**

After `movl %eax, %cr0`, the content resides at 0x00100000 and at 0xf0100000 is identical, the mapping works.

The following instruction will fail if the mapping weren't in place

```
mov     $relocated, %eax
jmp     *%eax    # fail to work properly
```

The address of $relocated is not been mapped, thus jumping to the place where kernel doesn't show up.

3.2 Formatted Printing to the Console

1. The **console.c** file exports `cputchar` for use in **printf.c**. In **printf.c**, this `cputchar` is used in `putch` function, which call `cputchar` and update a count variable. Again, `putch` is passed to `vprintfmt` in **printfmt.c** in a higher level abstraction `vcprintf`. And finally to print something, we call the `cprintf` function to utilize the variable-length argument list on `vcprintf`. The actual call of `putch` is in `printfmt.c`.

   The **printf.c** file acts as a bridge between low level communication with hardware (**console.c**) and high level format print utility (**printfmt.c**).

2. The following code from console.c:

```
// What is the purpose of this?
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;
}
```

   runs under the occasion that the screen has been filled (rows x cols, buffer full), new space is needed to continue write. Thus this code copy the data after the first row to override previous buffer, which remove the first line. Graphically, the text console is moved up 1 line.

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

   - In the call to cprintf(), `fmt` points to the string "`x %d, y %x, z %d\n`". After `va_start`, `ap` is pointing to the argument `x` on stack.
   - The calling sequence is listed below:

```
cprintf("x %d, y %x, z %d\n", x, y, z)
|
vcprintf("x %d, y %x, z %d\n", ap)
|
vprintfmt((void*)putch, &cnt, fmt, ap)
|
---- cons_putc('x')
---- cons_putc(' ')
---- va_arg(*ap, int)     // before: [x,y,z] , after: [y,z] (on stack)
  -- cons_putc('1')
---- cons_putc(',')
---- cons_putc(' ')
---- cons_putc('y')
---- cons_putc(' ')
---- va_arg(*ap, int)     // before: [y,z] , after: [z]
  -- cons_putc('3')
---- cons_putc(',')
---- cons_putc(' ')
---- cons_putc('z')
---- cons_putc(' ')
---- va_arg(*ap, int)     // before: [z] , after:
  -- cons_putc('4')
```

1. For the following code:

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

it will print `HE110 World`.

The hexadecimal form of 57616 is `E110`. The ascii code for 'r' is `0x72`, 'l' `0x6c`, 'd' `0x64`. Interpret the integer `0x00646c72` as a string, the result would be 0x72, 0x6c, 0x64, \0 in little endian.

If big-endian is used, reverse i by bytes (i = 0x726c6400). 57616 stays unchanged.

1. for

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

Random error will occur. Or the value after 'y=' is a value from stack that doesn't belong the frame of current function.

1. If GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last, then we need to change the implementation for `va_arg`.

2. **Challenge**

   In `cga_putc`, the higher 8 bits is used to embed print attributes: background color and foreground color. I just need to put correct number into these bits.

   The 16 color palette is used

   Full CGA 16-color palette

0	black #000000	8	gray #555555
1	blue #0000AA	9	light blue #5555FF
2	green #00AA00	10	light green #55FF55
3	cyan #00AAAA	11	light cyan #55FFFF
4	red #AA0000	12	light red #FF5555
5	magenta #AA00AA	13	light magenta #FF55FF
6	brown #AA5500	14	yellow #FFFF55
7	light gray #AAAAAA	15	white (high intensity) #FFFFFF

   For convenience, in **stdio.h**

```
// colors
enum COLOR { c_black = 0, c_blue, c_green, c_cyan, c_red, c_magenta, c_brown,
       c_lgray, c_gray, c_lblue, c_lgreen, c_lcyan, c_lred, c_lmagenta,
       c_yellow, c_white};
```

The color variables are defined in **printf.c**, where the color is embedded into **putch**'s ch.

```
enum COLOR foreground_color = c_lgray;
enum COLOR background_color = c_black;
static void
putch(int ch, int *cnt)
{
  ch &= 0xff;
  ch |= background_color << 12;
  ch |= foreground_color << 8;
  cputchar(ch);
  *cnt++;
}
```

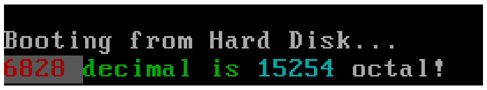
To change the color, I use `%F` for foreground color and `%B` for background color, these changes are made in **printfmt.c**

```
// forground color change
case 'F':
  foreground_color = getcolor(&ap);
  break;
// background color change
case 'B':
  background_color = getcolor(&ap);
  break;
```

As an example

```
cprintf("%F%B6828 %F%Bdecimal is %F%o %Foctal!\n", c_red, c_gray,
        c_green, c_black, c_cyan, 6828, c_lgray);
```

will have the following effect



```
Booting from Hard Disk...
6828 decimal is 15254 octal!
```

3.3 The Stack

- **Exercise 9**

The kernel initialize its stack in **entry.S**, the stack is reserved as a chunk of global data space, the stack pointer is initialized to the higher address end.

```
.data
###################################################################
# boot stack
###################################################################
  .p2align  PGSHIFT    # force page alignment
  .globl    bootstack
bootstack:
  .space    KSTKSIZE
  .globl    bootstacktop
bootstacktop:
```

- **Exercise 10**

  Set a breakpoint at `0xf0100040`.

  Between each call to `test_backtrace`, the `%esp` differs by 32 bytes. The contents are: return address,saved %ebp, saved %ebx, reserved local space(16 bytes), argument x.