JOS Lab1

Jen-tse Huang, 1500017836, hrz@pku.edu.cn

Exercise 1

Since I've learned the syntax of both AT&T and Intel, I just skip this exercise.

Exercise 2

- Load IDTR: `0xfd171: lidtw  %cs:0x6ab8`
- Load GDTR: `0xfd177: lgdtw  %cs:0x6a74`
- Set control register: `0xfd184: mov    %eax,%cr0`
- Read the first sector of the boot device into memory
- Note that every instruction does not belong to any function: e.g. `0x000efc67 in ?? ()`

Exercise 3

- Identify the exact assembly instructions that correspond to each of the statements in `readsect()`.

  Find that `00007c7c <readsect>:` in `boot.asm`, set breakpoint:

  ```
  b 0x7c7c
  ```

  and

  ```
  c
  ```

  Then use:

  ```
  x/50
  ```

  and finally get those instructions.

- Trace through the rest of `readsect()` and back out into `bootmain()`, and identify the begin and end of the `for` loop that reads the remaining sectors of the kernel from the disk.

  Find the `jmp` after the second `call 7cdc <readseg>` in `boot.asm`, then its the end of the `for` loop:

  ```
  7d69: eb e6                   jmp    7d51 <bootmain+0x3c>
  ```

  And the address it goes to is the begin of the `for` loop:

  ```
  7d51: 39 f3                   cmp    %esi,%ebx
  ```

- Find out what code will run when the loop is finished.

  ```
  7d6b: ff 15 18 00 01 00      call    *0x10018
  ```

  It calls the entry point from the ELF header.

- Q: At what point does the processor start executing 32-bit code?

  A: After this code below , the target architecture is assumed to be i386

  ```
  0x7c2d:      ljmp    $0x8,$0x7c32
  ```

- Q: What exactly causes the switch from 16- to 32-bit mode?

  A: `ljmp` sets CS and IP and changes their meanings

- Q: What is the *last* instruction of the boot loader executed?

  A: In `boot/main.c`:

  ```
  ((void (*)(void)) (ELFHDR->e_entry))();
  ```

- Q: What is the *first* instruction of the kernel it just loaded?

  A: In `obj/kern/kernel.asm`

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

- Q: *Where* is the first instruction of the kernel?

  A: We can find that: start address `0x0010000c`

  By:

  ```
  objdump -x obj/kern/kernel
  ```

- Q: 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?

  A: It loads ELFHeader first and get information from that. We can find that by the code below:

  ```
  readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
  ```

Exercise 4

Just remember that **when you add a pointer with a number, the address equals to (address += the size of the type of what this pointer points to)**

Exercise 5

I change the `-Ttext` parameter in `boot/Makefrag` to both `0x7A00` and `0x7CC0` (higher and lower address), the instruction:

```
[   0:7c2d] => 0x7c2d:  ljmp   0x8,0x7cf2
```

will go wrong under both case (with different address of IP).

Exercise 6

```
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
```

```
Breakpoint 2, 0x00007d6b in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
```

They are different because boot loader load the kernel code into memory, and they start from `0x100000`

Exercise 7

```
Breakpoint 1, 0x00100025 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
(gdb) x/8x 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000      0x00000000      0x00000000
0x00000000
0xf0100010 <entry+4>:   0x00000000      0x00000000      0x00000000      0x00000000
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
(gdb) x/8x 0xf0100000
0xf0100000 <_start+4026531828>: 0x1badb002      0x00000000      0xe4524ffe
0x7205c766
0xf0100010 <entry+4>:   0x34000004      0x0000b812      0x220f0011      0xc0200fd8
```

We can see that after this instruction, the memory in `0x00100000` and `0xf0100000` become same.

After commenting out `movl %eax, %cr0`, I met an error in:

```
0x10002a:    jmp    *%eax
```

and got an message:

```
qemu: fatal: Trying to execute code outside RAM or ROM at 0xf010002c
```

Exercise 8

In `printfmt.c`, find and change this:

```
// (unsigned) octal
case 'o':
    // Replace this with your code.
    num = getuint(&ap, lflag);
    base = 8;
    goto number;
```

Find "6828 decimal is 15254 octal!" in `qemu`, which means the changes are correct.

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

  A: `console.c` exports `cputchar()`, which is used by `printf.c` in `putch()`

- Q: 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;
  }
  ```

  A: `crt_pos >= CRT_SIZE` means the character is out of the range of the size of screen. So we need to roll up(move every row to the one above).

- Q: 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);
  ```

  - Q1: In the call to `cprintf()`, to what does `fmt` point? To what does `ap` point?

    A: `fmt` points to `"x %d, y %x, z %d\n"`, and `ap` points to `x, y, z`

  - Q2: List (in order of execution) each call to `cons_putc`, `va_arg`, and `vcprintf`.

    For `cons_putc`, list its argument as well.

    ```
    void
    cputchar(int c)
    ```

```
{
    cons_putc(c);
}

static void
cga_putc(int c)
{
    ...
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
    ...
    }
```

For `va_arg`, list what `ap` points to before and after the call.

After the call, the address that `ap` points to will plus the length of the type, which is the second argument of `va_arg`

```
static unsigned long long
getuint(va_list *ap, int lflag)
{
    ...
        return va_arg(*ap, unsigned long long);
    ...
        return va_arg(*ap, unsigned long);
    ...
        return va_arg(*ap, unsigned int);
}

static long long
getint(va_list *ap, int lflag)
{
    ...
        return va_arg(*ap, long long);
    ...
        return va_arg(*ap, long);
    ...
        return va_arg(*ap, int);
}

void
vprintfmt(void (*putch)(int, void*), void *putdat, const char *fmt, va_list
ap)
{
    ...
            precision = va_arg(ap, int);
    ...
            putch(va_arg(ap, int), putdat);
    ...
            err = va_arg(ap, int);
```

```
    ...
            if ((p = va_arg(ap, char *)) == NULL)
    ...
                (uintptr_t) va_arg(ap, void *);
```

For `vcprintf` list the values of its two arguments.

```
int
vcprintf(const char *fmt, va_list ap)
{
    int cnt = 0;

    vprintfmt((void*)putch, &cnt, fmt, ap);
    return cnt;
}

int
cprintf(const char *fmt, ...)
{
    ...
    cnt = vcprintf(fmt, ap);
    ...
}
```

- Q: Run the following code.

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

  - Q1: What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise.

    A: In `kern/init.c`, add the code in this function:

    ```
    void
    i386_init(void)
    ```

    Finally I got the output:

    ```
    He110 World
    ```

    `%x` will treat `57616` as hex, which will be `e110`

    `%s` will treat i as a string, and because of the little-endian storage, it will be `0x72`, `0x6c`, `0x64`, `0x00` in ASCII, which means 'r', 'l', 'd' and '\0'.

  - Q2: 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?

A: I will set `i = 0x726c6400` and will not change `57616`. Because it does not change the type of `57616`, the four bytes will be read together.

- Q: In the following code, what is going to be printed after `y=`? Why does this happen?

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

  A: Add the code in the same place as the question before. And I got this output:

  ```
  x=3 y=1600
  ```

  The `1600` comes from the next four bytes after `3` in the memory

- Q: 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?

  A: We need to change the interface of `cprintf` from `const char *fmt, ...` to `..., const char *fmt`, and the `...` must be the inverse of `%` in `*fmt`

Exercise 9

- Q: Determine where the kernel initializes its stack, and exactly where in memory its stack is located.

  A: In `kernel.asm`, we can find that:

  ```
  f0100034:	bc 00 00 11 f0       	mov    $0xf0110000,%esp
  ```

  At `f0100034`, kernel initializes its stack. And the stack address is `0xf0110000`

- Q: How does the kernel reserve space for its stack?

  A: In `inc/memlayout.h`:

  ```
  // Kernel stack.
  #define KSTACKTOP	KERNBASE
  #define KSTKSIZE	(8*PGSIZE)   		// size of a kernel stack
  #define KSTKGAP		(8*PGSIZE)   		// size of a kernel stack guard
  ```

  We can find the definition of `KSTKSIZE`. And in `inc/mmu.h`:

  ```
  #define PGSIZE		4096		// bytes mapped by a page
  ```

  We can find the definition of `PGSIZE`

- Q: At which "end" of this reserved area is the stack pointer initialized to point to?

  A: The lowest end.

Exercise 10

In `kernel.asm`, we can find `test_backtrace` at `0xf0100040`. Then its first several instructions:

```
f0100040:   55                      push   %ebp
f0100041:   89 e5                   mov    %esp,%ebp
f0100043:   53                      push   %ebx
f0100044:   83 ec 0c                sub    $0xc,%esp
f0100047:   8b 5d 08                mov    0x8(%ebp),%ebx
```

It saves `%ebp` in stack and saves `%esp` in `%ebp`, then push `%ebx`.

Use `b` and `c` and `i r` to check the `%esp`:

```
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=5) at kern/init.c:13
13      {
(gdb) i r
...
esp            0xf010ffdc       0xf010ffdc
...
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=4) at kern/init.c:13
13      {
(gdb) i r
...
esp            0xf010ffbc       0xf010ffbc
...
```

We can find that the value in `%esp` changes `0x20`, so there are 8 32-bit values in the stack.

Exercise 11

First, hook this new function into the kernel monitor's command list:

```
static struct Command commands[] = {
    ...
    { "mon_backtrace", "Display information of the stack", mon_backtrace },
    ...
};
```

In any function, `0x0(%ebp)` and `0x4(%ebp)` are the same. `0x0(%ebp)` is the `%ebp` of last function. `0x4(%ebp)` is the return address of this function. So we can use `0x0(%ebp)` iteratively.

```
int
```

```c
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    cprintf("Stack backtrace:\n");
    uint32_t ebp = read_ebp(), eip;
    while (ebp != 0)
    {
        eip = *((uint32_t *)ebp + 1);
        cprintf("  ebp %08x  eip %08x  args", ebp, eip);
        uint32_t *args = (uint32_t *)ebp + 2;
        for (int i = 0; i < 5; i ++)
        {
            cprintf(" %08x", args[i]);
        }
        cprintf("\n");
        ebp = *((uint32_t *)ebp);
    }
    return 0;
}
```

- Q: Why does the return instruction pointer typically point to the instruction after the `call` instruction?

  A: Because when it returns, the next instruction locates next to the `call`.

- Q: Why can't the backtrace code detect how many arguments there actually are? How could this limitation be fixed?

  A: It does not know how many arguments that last function pushed in stack. I think maybe we can add an argument in the function whose value equals to the number of arguments.

- Q: What's the way to tell in which function to stop by studying `kern/entry.S`?

  A: In `kern/entry.S`, there is an instruction:

  ```
  movl    $0x0,%ebp           # nuke frame pointer
  ```

  So we can stop when we find that `%ebp` equals to zero.

Exercise 12

In `kern/kdebug.c`, to complete function `debuginfo_eip(uintptr_t addr, struct Eipdebuginfo *info)`, we need to read `inc/stab.h`. And then we find the stabs type used for line numbers `N_SLINE`. By reading the function `stab_binsearch`, we can finally fill in the missing code:

```c
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
if(lline <= rline) {
    info->eip_line = stabs[lline].n_desc;
} else {
    info->eip_line = -1;
}
```

Finally, add this code into `mon_backtrace` in `kern/monitor.c`:

```
        struct Eipdebuginfo eip_info;
        debuginfo_eip(eip, &eip_info);
        cprintf("         %s:%d: %.*s+%d\n",
                eip_info.eip_file, eip_info.eip_line,
                eip_info.eip_fn_namelen, eip_info.eip_fn_name,
                eip - eip_info.eip_fn_addr);
```

- Q: In `debuginfo_eip`, where do `__STAB_*` come from?

  A: In `kern/kernel.ld`:

  ```
      /* Include debugging information in kernel memory */
      .stab : {
          PROVIDE(__STAB_BEGIN__ = .);
          *(.stab);
          PROVIDE(__STAB_END__ = .);
          BYTE(0)     /* Force the linker to allocate space
                     for this section */
      }

      .stabstr : {
          PROVIDE(__STABSTR_BEGIN__ = .);
          *(.stabstr);
          PROVIDE(__STABSTR_END__ = .);
          BYTE(0)     /* Force the linker to allocate space
                     for this section */
      }
  ```

  Output of `objdump -h obj/kern/kernel`:

  ```
  obj/kern/kernel:     file format elf32-i386

  Sections:
  Idx Name          Size      VMA       LMA       File off  Algn
    0 .text         00001921  f0100000  00100000  00001000  2**4
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
    1 .rodata       000007e8  f0101940  00101940  00002940  2**5
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
    2 .stab         00003a8d  f0102128  00102128  00003128  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
    3 .stabstr      0000194a  f0105bb5  00105bb5  00006bb5  2**0
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
    4 .data         0000a300  f0108000  00108000  00009000  2**12
                    CONTENTS, ALLOC, LOAD, DATA
    5 .bss          0000064c  f0112300  00112300  00013300  2**5
                    CONTENTS, ALLOC, LOAD, DATA

    6 .comment      00000035  00000000  00000000  0001394c  2**0
  ```

```
                  CONTENTS, READONLY
```

Output of `objdump -G obj/kern/kernel` is too long to show, and we can find many STAB information.

After `gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c` and in `init.S`, there are many symbols of variables and codes.

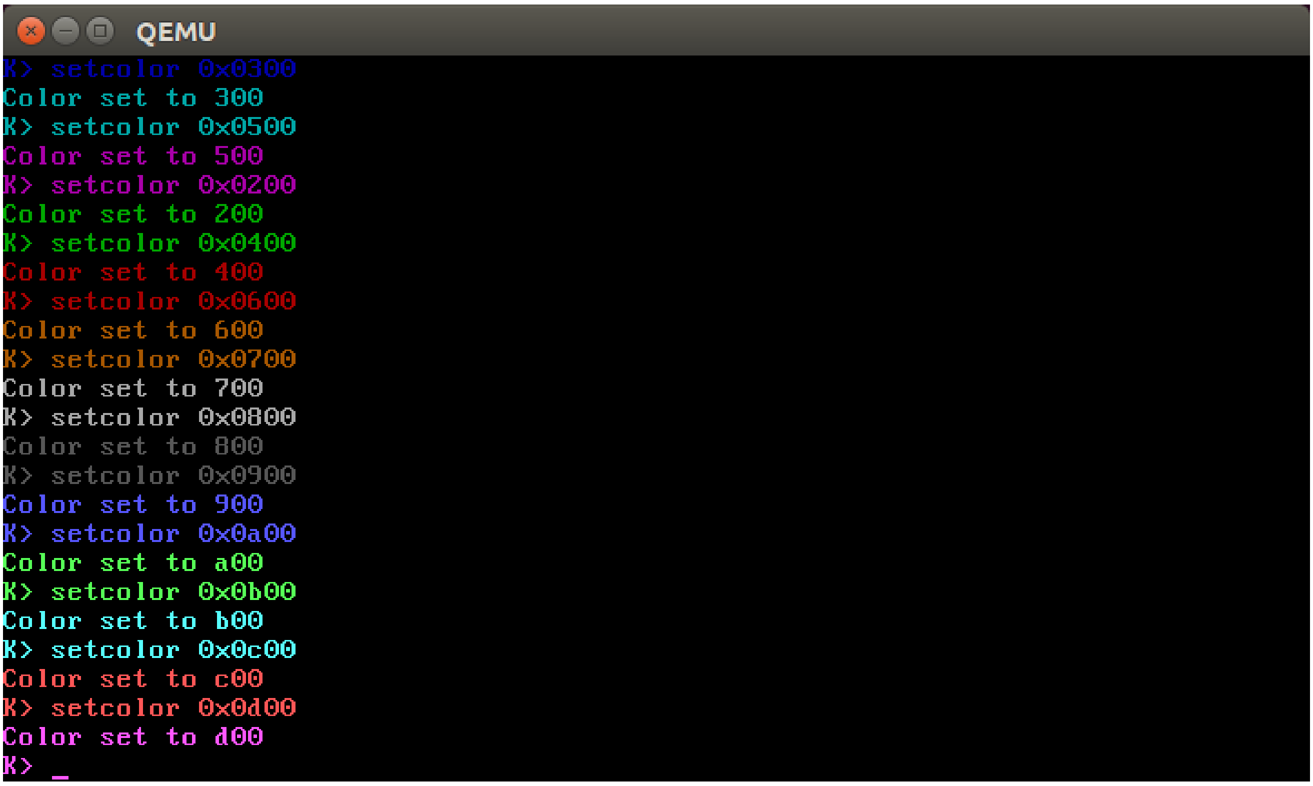
The bootloader loads the symbol table in memory as part of loading the kernel binary.

Q: `printf` format strings provide a way to print non-null-terminated strings like those in STABS tables. `printf("%.*s", length, string)` prints at most `length` characters of `string`. Take a look at the `printf` man page to find out why this works.

A: `.*` represents the precision of a number, which means length of a string. the `*` is determined by the `int` number in the `fmt`

Challenge 1

In this challenge, I implemented a command `setcolor` in the `monitor.c`, which has effect like this:



I set a variable in `console.h` named `COLOR_`, it will be initialized in `init.c` with the value `0x0700`.

And then I changed all the `0x0700`s in function `cga_putc(int c)` in `console.c` to the variable `COLOR_`.

Finally by modifying `monitor.h` and `monitor.c`, I added the command. There is the function that will be called by this command:

```
int
mon_setcolor(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 2)
    {
        cprintf("Usage: setcolor [int]\n");
        return 0;
    }
    COLOR_ = (int)strtol(argv[1], NULL, 0);
    COLOR_  &= ~0x11;
    cprintf("Color set to %x\n", COLOR_);
    return 0;
}
```

Score

```
running JOS: (1.3s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
```