JOS Lab1 Report

姓名：方嘉聪， 学号：2200017849

Challenge

1. ANSI转移代码 SGR基本格式 `\033[0;p1;p2;...;p_n m Text \033[0m`
2. 将原有的`cga_putc`改为`cga_putc_original`，设计一个新的`cag_putc(int c)`用以实现VGA的颜色显示。在我的代码中支持了形如 `\033[0;p1;...;pn m TEXT \033[0m`其中 `p1, ..., pn` 是颜色编码，即`3x`表示前景（文字），`4x`表示背景颜色。代码思路如下
3. 设置一个`attr`变量用以记录当前的颜色格式，使用一个 `vga_buf`存储已经读取的颜色编码，当读取到`m`或`;`时，调用函数`tranfer_buf_to_attr`将颜色编码转化为实际可用的二进制数值，嵌入在输入的字符中。具体的状态跳转见代码及注释。
   - ASCII text code: 0th - 7th bits
   - The foreground color: 8th - 11th bits
   - The background color: 12th - 14th bits
   - Blinking: 15th bit (Ignore it now.))
4. 下图为测试结果，测试代码在`kern/monitor.c/mon_backtrace()`中。事实上如果有输入的字符无意义，如`\033[a;31m...\033[0m\n` 目前实现的代码会按默认设置输出文字。没有做更多的鲁棒性测试，此外目前的实现是空格敏感的。

Part I: PC Bootstrap

1. Exercise 2:
   使用`si & x/10i`: BIOS主要将初始化各种硬件，使用`boot-loader`将Kernel从硬盘存储中读取出来，并将控制权移交给Kernel.

Part II: The Boot Loader

1. Exercise 3
   - At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
   - Answer: 阅读`boot/boot.S`中源码及注释，在

```
        movl    %cr0, %eax
        orl     $CR0_PE_ON, %eax
        movl    %eax, %cr0
```

中`movl %eax, %cr0`将`cr0`寄存器的PE为设置为1，切换到protected mode（即32位）`ljmp $PROT_MODE_CSEG, $protcseg`长跳转到32-bit code处。

- What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?
- Answer: 阅读反汇编`obj/boot/boot.asm`: 处理器处理的最后一个 boot loader 代码为

```
((void (*)(void)) (ELFHDR->e_entry))();
7d71:   ff 15 18 00 01 00       call   *0x10018
```

- *Where* is the first instruction of the kernel?
- Answer: 运行时`*0x10018`中存储的值为`0x10000c`对应Kernel执行的第一个指令`f010000c: movw $0x1234,0x472`.
- 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?
- Answer: 在`void bootmain(void)`中先将ELF Header读取到内存里`readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);`通过ELF header的信息判断需要读取的segment的位置和大小，即代码中的`readseg(ph->p_pa, ph->p_memsz, ph->p_offset);`进而决定加载kernel是需要读取多少扇区。

2. Exercise 4: 复习C语言指针. 注意一下`(int*) c = c + 1`和`c = (int *) ((char *) c + 1`的差别。以及`a+1//加一个元素的位置`，`&a + 1 //加上整个数组`.
3. Exercise 5: 修改`0x7c00` → `0x7c20`会出现一个整体的地址偏移。但BIOS会固定架在boot loader到`0x7c00`导致错误
4. Exercise 6: 刚进入`boot loader`时`0x100000`存储内容为空，当进入kernel时已经加载了ELF Header，`0x100000`内容发生修改。

```
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x1000b812      0x220f0011      0xc0200fd8
```

Part III: The Kernel

1. Exercise 7: 在执行`movl %eax, %cr0`后`0x00100000`和`0xf0100000`中存储的内容相同，内存映射已建立。注释`mol %eax, %cr0`后程序无法建立正确的虚拟内存映射，在执行`jmp *%eax`时内存引用出现错误，造成panic。
2. Exercise 8: 模仿十六进制输出即可实现`%o`八进制输出。注意到grade程序是按照每一行进行匹配，所以需要在`kern/init.c/void i386_init`中添加一个换行符`cprintf("\n6828 decimal is %o octal!\n", 6828);`
    1. `console.c`中提供了输入/出字符的功能，主要处理I/O各种接口。`console.c`提供了high level API `cputchar/getchar/iscons`，其中`printf.c/putch`调用了`cputchar`用以在终端上输出字符。
    2. 这段代码当输入字符位置`crt_pos`超过页面的大小`CRT_SIZE`时，将现有的字符向上移动一行(调用`memmove`)，将最后一行清空后写入字符。
    3. - `fmt`指向的地址存储字符串`"x %d, y %x, z %d\n"`，`ap`指向的就是`x,y,z`的值 1,3,4.
        - 调用顺序为

```
vcprintf("x %d, y %x, z %d\n", va_list{1, 3, 4})
cons_putc('x')
cons_putc(' ')
va_arg: ap: va_list{1, 3, 4} -> va_list{y, z}
cons_putc('1')
cons_putc(',')
cons_putc(' ')
cons_putc('y')
cons_putc(' ')
```

```
va_arg: ap: va_list{3, 4} -> va_list{4}
cons_putc('3')
cons_putc(',')
cons_putc(' ')
cons_putc('z')
cons_putc(' ')
va_arg: ap: va_list{4} -> va_list{}
cons_putc('4')
cons_putc('\n')
```

4. He110 World （517616 → `0xe110` → He110； `0x00646c72` 小端法对应ASCII为orld 故总的为 He110 World
5. `y=` 后面会输出一个随机值（我的机器是4027587896）取决于`ap`向的地址之后存储的是什么内容(输出`12(%ebp)`处的值)。
6. 假设GCC改变了默认行为， `cprintf(..., int n, const char* fmt)` 先计算有多少个参数，作为`cprintf`的一个变量传入。

3. **Exercise 9:** 查询`kern/entry.S`, kernel在`.space KSTKSIZE`处为栈分配空间并初始化，栈顶指针初始化的值为`bootstacktop`(在我的反汇编中为`0xf0110000`)
4. **Exercise 10:** 每次递归调用`test_backtrace(x-1)`会将`%ebp`寄存器存储的值、调用函数的返回地址、参数`x-1...`（如下图，依次为从左到右）

```
(gdb) x/16x $esp
0xf010ffb8:     0xf010ffd8      0xf0100076      0x00000004
```

(事实上在正式执行`test_backtrace(x-1)`前还会将`%esi,%ebx`压入栈中)

5. **Exercise 11:** 通过分析`obj/kern/kernel.asm`，注意到`%ebp`指向地址存储的值为上一层的`%ebp`。使用`read_ebp()`读取`%ebp`，依次读取出`%eip`及各个参数

```
cprintf("ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n",
                                ebp,
                                *((uint32_t *)(ebp + 4)),
                                *((uint32_t *)(ebp + 8)),
                                *((uint32_t *)(ebp + 12)),
                                *((uint32_t *)(ebp + 16)),
                                *((uint32_t *)(ebp + 20)),
                                *((uint32_t *)(ebp + 24)));
ebp = *((uint32_t *)(ebp));
```

一个比较坑的bug是把`ebp`写成了`*((uint32_t *)ebp)`导致测试程序里`test_backtrace_count`一直过不了 :(

6. **Exercise 12:** 链接器为`.stab/.stabstr`分配对应的`segment`，通过`objdump -G obj/kern/kernel`可以查看`Symbol Table`。模仿`kdebug.c`中的逻辑补充函数即可，在`stab.n_desc`中存储着描述符（可能包含行数），此外注意`cprintf`的格式即可。

Result

`make grade`测试结果如下，成功通过。

```
make[1]: Leaving directory '/home/ubuntu/6.828/lab'
running JOS: (0.6s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
```