操作系统及实习（实验班）

JOS-Lab1

系统的启动和初始化
实习报告

姓名 佘俊峰
学号 1100012769
日期 2013.10.1

总体概述

这一部分主要是关于 JOS 如何启动和初始化的，所以主要研究了相关的 BIOS，bootloader 等技术。并且总体了解了 ELF 表，链接等知识，以及 kernel 怎么显示，stack 怎么在程序运行时变化，总体来说收获很大。

最重要的理解有三点。

1、 对于屏幕的输出

2、 栈的运行机制，栈，ebp，esp 等寄存器的在函数调用时的变化

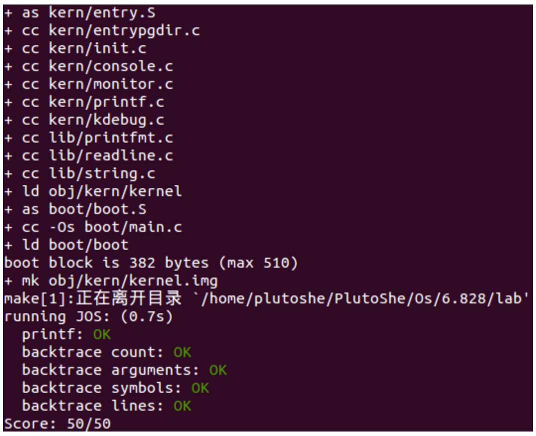
3、 对于系统如何启动有了一个深刻的理解

任务完成情况说明

第一周我大致完成了 qemu 的启动和环境的搭配

第二周我基本上完成了 lab 的所有 exercise 和 challenge

第三周由于上半周时间较紧，lab 的完成报告拖到了下半周才完成。



```
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]:正在离开目录 `/home/plutoshe/PlutoShe/Os/6.828/lab'
running JOS: (0.7s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
```

任务的具体解决

Deploy related environment

在启动 qemu 时，我并没有遇到太多的问题，最多只是在 configure 的时候，少了2个链接库，

sudo apt-get install ia32-libs lib32gcc1 lib32stdc++6

将之 sudo apt-get 随即就可以运行 qemu 模拟器了。使用 qemu 具体的情况如下图。

```
plutoshe@ubuntu:~/PlutoShe/Os/6.828/lab$ make qemu
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stackbacktrace:
ebp f010ff18 eip f0100087 args 00000000 00000000 00000000 00000000 f0100c4c
       kern/init.c:19: test_backtrace+47
ebp f010ff38 eip f0100069 args 00000000 00000001 f010ff78 00000000 f0100c4c
       kern/init.c:16: test_backtrace+29
ebp f010ff58 eip f0100069 args 00000001 00000002 f010ff98 00000000 f0100c4c
       kern/init.c:16: test_backtrace+29
ebp f010ff78 eip f0100069 args 00000002 00000003 f010ffb8 00000000 f0100c4c
       kern/init.c:16: test_backtrace+29
ebp f010ff98 eip f0100069 args 00000003 00000004 00000000 00000000 00000000
       kern/init.c:16: test_backtrace+29
ebp f010ffb8 eip f0100069 args 00000004 00000005 00000000 00010094 00010094
       kern/init.c:16: test_backtrace+29
ebp f010ffd8 eip f01000ea args 00000005 00001aac 00000644 00000000 00000000
       kern/init.c:43: i386_init+4d
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
       kern/entry.S:83: <unknown>+0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

```
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
setcolor - set the color for text
backtrace - Display information about ebp & eip
```

```
K> kerninfo
Special kernel symbols:
  _start                  0010000c (phys)
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101d45 (virt)  00101d45 (phys)
  edata  f0112320 (virt)  00112320 (phys)
  end    f0112964 (virt)  00112964 (phys)
Kernel executable memory footprint: 75KB
```

Exercise 1. Familiarize yourself with the assembly language materials available on the 6.828 reference page. You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.

We do recommend reading the section "The Syntax" in Brennan's Guide to Inline Assembly. It gives a good (and quite brief) description of the AT&T assembly syntax we'll be using with the GNU assembler in JOS.

基本的汇编语言在 ics 课上有所熟悉，所以阅读汇编代码只是需要理清思路即可。

之后在/inc 中读汇编代码的时候遇到了很多困难，通过查阅 PC Assembly Language Book 有相应的收获，而这本书在作者的官网上是有中文版的。

Exercise 2. Use GDB's `si` (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at Phil Storrs I/O Ports Description, as well as other materials on the 6.828 reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

启用 qemu 的 gdb 模式，然后在 gdb 下一步一步 si 进行观察，以下是相关代码，并附有解释。基本上只读到 A20总线之前的汇编代码位置。

```
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b  Long Jump to 0xfe05b
[f000:e05b] 0xfe05b:    jmp    0xfc85e
[f000:c85e] 0xfc85e:    mov    %cr0,%eax
[f000:c861] 0xfc861:    and    $0x9fffffff,%eax
[f000:c867] 0xfc867:    mov    %eax,%cr0   设置 CR0寄存器，CD 和 NW 设置为0，
[f000:c86a] 0xfc86a:    cli    屏蔽中断
[f000:c86b] 0xfc86b:    cld    操作方向标志位 DF
[f000:c86c] 0xfc86c:    mov    $0x8f,%eax
[f000:c872] 0xfc872:    out    %al,$0x70  输出位址8fh 至70H（CMOS）
[f000:c874] 0xfc874:    in     $0x71,%al  从 Port 71H （CMOS）读取停机状态
[f000:c876] 0xfc876:    cmp    $0x0,%al  //并判断是否为0
[f000:c878] 0xfc878:    jne    0xfc88d
```

这里我们可以看到启动后执行的第一条指令是在内存 0x000FFFF0 处，然后进行了一个长跳转，根据之前对 bios 的理解，BIOS 在内存中的上限是 0x00100000,于是在 0x000FFFF0 处执行第一条指令的话必然要跳转这样才会有更多的 BIOS 指令可以执行。

这一步是我一开始最痛苦的一步，首先我没有学过微机原理，对于段式存储以及相应的实模式和保护模式不是很清楚，并且一开始我并没有去查阅相关的资料，而是直接做了他的 exercise，这就让我花费了很多不必要的时间在这上面。本身自己一开始对于 boot 没有一个大体的概念，使得自己一开始就做这个 exercise 感到一头雾水。这让我清楚了工欲善其事必先利其器，在没有一个方向的时候，去盲目努力只是白费力气

> **Exercise 3.** Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.
>
> Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `boot/boot.S`, using the source code and the disassembly file `obj/boot/boot.asm` to keep track of where you are. Also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.
>
> Trace into `bootmain()` in `boot/main.c`, and then into `readsect()`. Identify the exact assembly instructions that correspond to each of the statements in `readsect()`. 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 out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

这个 exercise 主要是让我们清楚 bootloader 的启动，具体作用。

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

在 bootloader 启动之后，在读完他的 gdt 表之后，在以下语句中将 cr0寄存器更改，使得系统从实模式变为了保护模式

lgdt   gdtdesc

movl   %cr0, %eax

orl $CR0_PE_ON, %eax

movl   %eax, %cr0

ljmp $PROT_MODE_CSEG, $protcseg

Q: What is the last instruction of the boot loader executed

最后一条在 bootloader 中的语句是

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

查询了相关的 boot.asm，最后一条语句为

```
    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
    7d63:    ff 15 18 00 01 00        call    *0x10018
```

Q: and what is the first instruction of the kernel it just loaded?

f010000c: 66 c7 05 72 04 00 00   movw $0x1234,0x472

Q: Where is the first instruction of the kernel?

```
The target architecture is assumed to be i386
=> 0x7d63:      call   *0x10018

Breakpoint 1, 0x00007d63 in ?? ()
(gdb) x/1h 0x10018
0x10018:        0x000c
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
```

在0x10000c 处

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?

Bootloader 可以从 elf 文件的文件头中读取相关的信息。查询相关的 main.c 的代码就可以看到哦啊一下的相关信息，Elf->phoff：指定了第一个 section 的位置。Elf->phnum：指定了 section 的个数。

```
ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);

eph = ph + ELFHDR->e_phnum;

for (; ph < eph; ph++)

  readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

Exercise 4. Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an Amazon Link) or find one of MIT's 7 copies.

Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for pointers.c, run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in lines 1 and 6 come from, how all the values in lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.

There are other references on pointers in C, though not as strongly recommended. A tutorial by Ted Jensen that cites K&R heavily is available in the course readings.

Warning: Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

主要是理解指针的概念，通过以前的积累基本上对于指针的概念有一个很好的了解，只是看到了这么一种运用方式 3[c]可以直接被视为 c+3

Exercise 5. Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in `boot/Makefrag` to something wrong, run `make clean`, recompile the lab with `make`, and trace into the boot loader again to see what happens. Don't forget to change the link address back and `make clean` again afterward!

我修改了 在 boot/Makefrag 的 Ttext 为 0x8c00。

在 make clean 和 make 之后在 ljmp $PROT_MODE_CSEG, $protcseg 这条语句这里发生了错误，该处的跳转为 ljmp $0x8, $0x8c32（link address 为$0x8c32）。

由于 BIOS 将 boot loader 加载.gdt 相关数据到0x7C00处，因此 ljmp 跳转到的指令的 load address 为$0x7c32，由于 link address 和 load address 不一致，因此导致出错。

Exercise 6. We can examine memory using GDB's `x` command. The GDB manual has full details, but for now, it is enough to know that the command `x/Nx ADDR` prints *N* words of memory at *ADDR*. (Note that both 'x's in the command are lowercase.) *Warning*: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).

Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

没加载内核之前，这片内存是空的，全都是0。执行 bootloader 加载内核这片才被附上值。

```
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8h 0x100000
0x100000:       0x0000  0x0000  0x0000  0x0000  0x0000  0x0000  0x0000  0x0000
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8h 0x100000
0x100000:       0xb002  0x1bad  0x0000  0x0000  0x4ffe  0xe452  0xc766  0x7205
(gdb) 
```

```
(gdb) x/8i 0x100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0x66,%al
   0x10000d:    movl   $0xb81234,0x472
   0x100017:    add    %dl,(%ecx)
   0x100019:    add    %cl,(%edi)
   0x10001b:    and    %al,%bl
```

Exercise 7. Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the `stepi` GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.

What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the `movl %eax, %cr0` in `kern/entry.S`, trace into it, and see if you were right.

查询 kernel.asm，发现 entry.s 的位置为0x10025

在 0x10025处设置断点，之后通过比较0x10000和0xf010000，发现之后读入了一行。

```
Breakpoint 1 at 0x100025
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x100025:    mov    %eax,%cr0

Breakpoint 1, 0x00100025 in ?? ()
(gdb) x/8h 0x100000
0x100000:       0xb002  0x1bad  0x0000  0x0000  0x4ffe  0xe452  0xc766  0x7205
(gdb) x/8h 0xf0100000
0xf0100000:     0xffff  0xffff  0xffff  0xffff  0xffff  0xffff  0xffff  0xffff
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/8h 0x100000
0x100000:       0xb002  0x1bad  0x0000  0x0000  0x4ffe  0xe452  0xc766  0x7205
(gdb) x/8h 0xf0100000
0xf0100000:     0xb002  0x1bad  0x0000  0x0000  0x4ffe  0xe452  0xc766  0x7205
```

GDT 表引入不正确，会在下图 mov $0xf010002f,%eax 所示的地方会发生第一次 fail，

```
    # Now paging is enabled, but we're still running at a low EIP
    # (why is this okay?).  Jump up above KERNBASE before entering
    # C code.
    mov $relocated, %eax
f0100028:   b8 2f 00 10 f0          mov    $0xf010002f,%eax
    jmp *%eax
f010002d:   ff e0                   jmp    *%eax
```

```
    # Load the physical address of entry_pgdir into cr3.  entry_pgdir
    # is defined in entrypgdir.c.
    movl    $(RELOC(entry_pgdir)), %eax
    movl    %eax, %cr3
    # Turn on paging.
    movl    %cr0, %eax
    orl $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0

    # Now paging is enabled, but we're still running at a low EIP
    # (why is this okay?).  Jump up above KERNBASE before entering
    # C code.
    mov $relocated, %eax
    jmp *%eax
```

Exercise 8. We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

具体的代码修改在 printfmt.c 处，在 case 语句处进行如下的修改

```
        case 'o':
            // Replace this with your code.
            //putch('0', putdat);
            //putch('X', putdat);
            //putch('X', putdat);
            //break;
            num = getuint(&ap, lflag);
            base = 8;
            goto number;
```

following exercise is just run into qemu to see the result

Be able to answer the following questions:

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

printf.c 中的 putch(int, int *)调用了 console.c 中的 cputchar(int)。

Printf.c 在调用 printfmt.c 中的 vprintfmt 会使用这个函数。

2、Explain the following from console.c:

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

屏幕写满(CRT_SIZE)后，删除屏幕第一行文本，其他文本均向上移动一行

3、For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

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?

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. Forvcprintf list the values of its two arguments.

在 cprintf()中, fmt 指向的是格式字符串,在上例中即"x %d, y %x, z %d \n",而 ap 指向的是不定参数表的第一个参数地址,在上例中即 x。
fmp 指向字符串”x %d, y %x, z%d\n”的地址。
ap 指向函数参数 x 的地址（该地址位于进程栈的8(%ebp)）。

vprintfmt()解析 fmt，并调用 va_arg 从 stack 中读取参数。

1.调用 x 前, va_arg 指向(uint32_t *)ebp+3，调用后 va_arg 指向(uint32_t *)ebp+4。
cons_putc(49)

2.调用 y 前, va_arg 指向(uint32_t *)ebp+4，调用后 va_arg 指向(uint32_t *)ebp+5。
cons_putc(51)

3.调用 z 前, va_arg 指向(uint32_t *)ebp+5，调用后 va_arg 指向(uint32_t *)ebp+6。
cons_putc(52)

4、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. Here's an ASCII table that maps bytes to characters.

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?

He110 World。其中 e110是57616的16进制表示。

*0x00646c72*需要修改为*0x726c6400*。

57616不需要修改。

5、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);
```

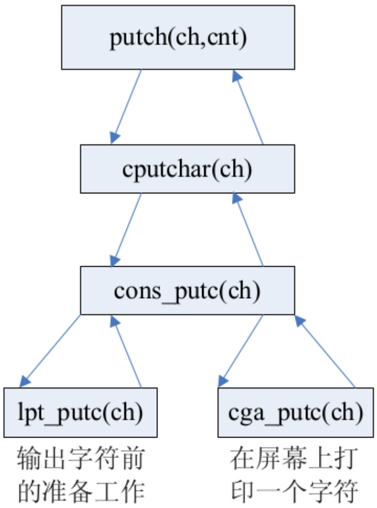
y 打印出来的是调用 cprintf 前的 stack top 所指向内存的4个字节。

6、 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?

可以在参数表最后压入参数的个数

Challenge Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret [ANSI escape sequences] embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on [the 6.828 reference page] and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

首先 challenge 需要了解具体的 print 过程，可以如图所示

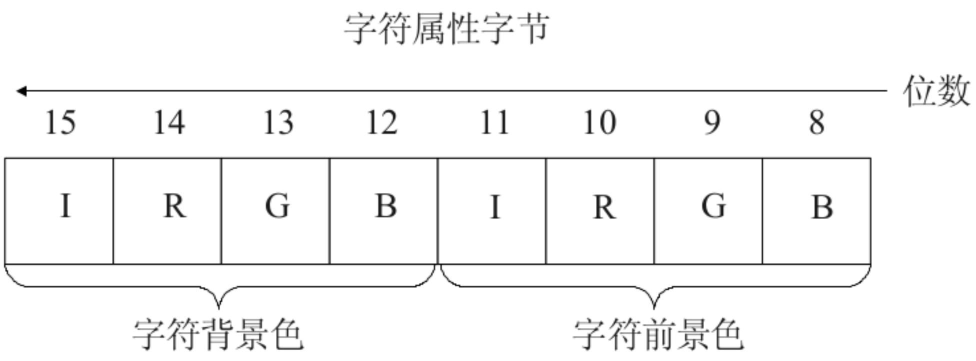


具体运用到屏幕输出的工作的在 console.c 中。

而对应的颜色的处理，在 cga_putc 函数中，如图

```
static void
cga_putc(int c)
{
    // if no attribute given, then use black on white
    c = c + attribute_color;
    //if (!(c & ~0xFF))
    //  c |= 0x0700;
```

而对应的我们需要修改的就是对应的01串，关于颜色的描述，如下图所示



首先我想到的办法是无函数缺省参数，如同 void a(int b = c)这样的形式，可以在我没有定义时，缺省的为默认值，之后发现 c 语言并没有这样的语法，
而之后想到同名函数重载同样没有这样的方法，而从外部引入变量会破化该程序的封闭性，所以我定义了对应的 attribute_color 这样的一个 int 值，从内部函数来修改他来保护他的封闭行，我在 console.c 中定义了对应的变量和函数

```
static uint16_t attribute_color = 0x0700;
void set_attribute_color(uint16_t back, uint16_t fore);
void cons_init(void);
int cons_getc(void);
```

而于此同时需要在 monitor.c 中定义匹配的命令行和对应的函数，通过阅读 monitor.c 程序，发现 NCOMMANDS 由 command 命令地址长度计算。

```
static struct Command commands[] = {
    { "help", "Display this list of commands", mon_help },
    { "kerninfo", "Display information about the kernel", mon_kerninfo },
    { "setcolor", "set the color for text", mon_setcolor },
    { "backtrace", "Display information about ebp & eip", mon_backtrace}
};
#define NCOMMANDS (sizeof(commands)/sizeof(commands[0]))
```

所以 command 命令行的函数类型必须跟 command[0]一致，一开始我就犯下了直接将函数定义为 void，从而导致了如下错误

kern/monitor.c:27:2: 错误： 从不兼容的指针类型初始化 [-Werror]

kern/monitor.c:27:2: 错误： (在‘commands[2].func’的初始化附近) [-Werror]

之后需要在 monitor.c 中补充对应的颜色对比值，此处参照了张弛的报告上的颜色

```
#define CMDBUF_SIZE 80
#define COLOR_WHT 7;
#define COLOR_BLK 0;
#define COLOR_BLU 1;
#define COLOR_GRN 2;
#define COLOR_RED 4;
#define COLOR_GRY 8;
#define COLOR_YLW 15;
#define COLOR_ORG 12;
#define COLOR_PUR 6;
#define COLOR_CYN 11;
```

而且需要分出背景和字体颜色，所以我在 setcolor 中写入了两个判断语句，对对应的颜色进行对比，具体的代码如下

```c
int mon_setcolor(int argc, char **argv, struct Trapframe *tf) {
    //argv
    uint16_t ch_color1, ch_color;
    if(strcmp(argv[2],"blk")==0)
            ch_color1=COLOR_BLK
    else if(strcmp(argv[2],"wht")==0)
            ch_color1=COLOR_WHT
    else if(strcmp(argv[2],"blu")==0)
            ch_color1=COLOR_BLU
    else if(strcmp(argv[2],"grn")==0)
            ch_color1=COLOR_GRN
    else if(strcmp(argv[2],"red")==0)
            ch_color1=COLOR_RED
    else if(strcmp(argv[2],"gry")==0)
            ch_color1=COLOR_GRY
    else if(strcmp(argv[2],"ylw")==0)
            ch_color1=COLOR_YLW
    else if(strcmp(argv[2],"org")==0)
            ch_color1=COLOR_ORG
    else if(strcmp(argv[2],"pur")==0)
            ch_color1=COLOR_PUR
    else if(strcmp(argv[2],"cyn")==0)
            ch_color1=COLOR_CYN
    else ch_color1=COLOR_WHT;
    if(strcmp(argv[1],"blk")==0)
            ch_color=COLOR_BLK
    else if(strcmp(argv[1],"wht")==0)
            ch_color=COLOR_WHT
    else if(strcmp(argv[1],"blu")==0)
            ch_color=COLOR_BLU
    else if(strcmp(argv[1],"grn")==0)
            ch_color=COLOR_GRN
    else if(strcmp(argv[1],"red")==0)
            ch_color=COLOR_RED
    else if(strcmp(argv[1],"gry")==0)
            ch_color=COLOR_GRY
    else if(strcmp(argv[1],"ylw")==0)
            ch_color=COLOR_YLW
    else if(strcmp(argv[1],"org")==0)
            ch_color=COLOR_ORG
    else if(strcmp(argv[1],"pur")==0)
            ch_color=COLOR_PUR
    else if(strcmp(argv[1],"cyn")==0)
            ch_color=COLOR_CYN
    else ch_color=COLOR_WHT;
    set_attribute_color((uint64_t) ch_color, (uint64_t) ch_color1);
    cprintf("console back-color :  %d \n        fore-color :  %d\n", ch_color, ch_color1);
    return 0;
```

实际的效果让我比较满意

```
K> setcolor red blu
console back-color :  4
        fore-color :  1
K> setcolor grn blk
console back-color :  2
        fore-color :  0
K> setcolor blk org
console back-color :  0
        fore-color :  12
K> setcolor pur wht
console back-color :  6
        fore-color :  7
```

Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

在 entry.s 中有相关的初始化代码

```
relocated:

	# Clear the frame pointer register (EBP)
	# so that once we get into debugging C code,
	# stack backtraces will be terminated properly.
	movl	$0x0,%ebp			# nuke frame pointer

	# Set the stack pointer
	movl	$(bootstacktop),%esp

	# now to C code
	call	i386_init

	# Should never get here, but in case we do, just spin.
spin:	jmp spin


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

可以从代码中知道程序是通过.space 保留了栈的空间。

之后通过查询 kernel.asm，

```
	# Set the stack pointer
	movl	$(bootstacktop),%esp
f0100034:	bc 00 00 11 f0		mov	$0xf0110000,%esp
```

找到在 entry 的反编译中，对于栈的初始化 esp 为0xf0110000

Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the `test_backtrace` function in `obj/kern/kernel.asm`, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words?

Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the tools page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

如下图在两次 test_backtrace 函数调用时，esp 差了32字节，

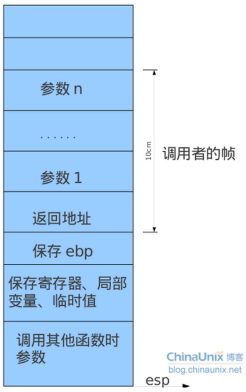
```
Breakpoint 1, test_backtrace (x=5) at kern/init.c:13
13	{
(gdb) i r
eax            0x1	1
ecx            0x3f8	1016
edx            0x3f8	1016
ebx            0x10094	65684
esp            0xf0110fdc	0xf0110fdc
ebp            0xf0110ff8	0xf0110ff8
esi            0x10094	65684
edi            0x0	0
eip            0xf0100040	0xf0100040 <test_backtrace>
eflags         0x86	[ PF SF ]
cs             0x8	8
ss             0x10	16
ds             0x10	16
es             0x10	16
fs             0x10	16
gs             0x10	16
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>:	push   %ebp
```

```
Breakpoint 1, test_backtrace (x=4) at kern/init.c:13
13	{
(gdb) i r
eax            0x4	4
ecx            0x3d4	980
edx            0x3d5	981
ebx            0x5	5
esp            0xf0110fbc	0xf0110fbc
ebp            0xf0110fd8	0xf0110fd8
esi            0x10094	65684
edi            0x0	0
eip            0xf0100040	0xf0100040 <test_backtrace>
eflags         0x6	[ PF ]
cs             0x8	8
ss             0x10	16
ds             0x10	16
es             0x10	16
fs             0x10	16
gs             0x10	16
(gdb)
```

通过查询对应的汇编代码，发现为返回地址，ebp，ebx，给调用的 test_backtrace 的 x 参数，以及剩下的20个字节为临时变量留下空间。

Exercise 11. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run `make grade` to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

已知具体的 ebp，eip，参数关系如图所示，



所以可由完成的代码如下

```
uint32_t *eip, *ebp;
ebp = (uint32_t*) read_ebp();
eip = (uint32_t*) ebp[1];
cprintf("Stackbacktrace:\n");
while (ebp!=0) {
    cprintf("ebp %08x eip %08x args %08x %08x %08x %08x %08x\n",ebp, eip, ebp[2] ,ebp[3], ebp[4], ebp[5] ,ebp[6]);
    ebp=(uint32_t*)ebp[0];
    eip=(uint32_t*)ebp[1];

}
```

Exercise 12. Modify your stack backtrace function to display, for each `eip`, the function name, source file name, and line number corresponding to that `eip`.

In `debuginfo_eip`, where do `__STAB_*` come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- look in the file `kern/kernel.ld` for `__STAB_*`
- run `i386-jos-elf-objdump -h obj/kern/kernel`
- run `i386-jos-elf-objdump -G obj/kern/kernel`
- run `i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c`, and look at init.s.
- see if the bootloader loads the symbol table in memory as part of loading the kernel binary

Complete the implementation of `debuginfo_eip` by inserting the call to `stab_binsearch` to find the line number for an address.

Add a `backtrace` command to the kernel monitor, and extend your implementation of `mon_backtrace` to call `debuginfo_eip` and print a line for each stack frame of the form:

```
K> backtrace
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143: monitor+106
  ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000 00000000
         kern/init.c:49: i386_init+59
  ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
         kern/entry.S:70: <unknown>+0
K>
```

Each line gives the file name and line within that file of the stack frame's `eip`, followed by the name of the function and the offset of the `eip` from the first instruction of the function (e.g., `monitor+106` means the return `eip` is 106 bytes past the beginning of `monitor`).

Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

Tip: printf format strings provide an easy, albeit obscure, 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.

You may find that some functions are missing from the backtrace. For example, you will probably see a call to `monitor()` but not to `runcmd()`. This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the `-O2` from `GNUMakefile`, the backtraces may make more sense (but your kernel will run more slowly).

对于 stab.h 的描述理解了它在 elf 表中的作用，他是符号表部分,这一部分的功能是程序报错时可以提供错误信息

```
// Entries in the STABS table are formatted as follows.
struct Stab {
    uint32_t n_strx;    // index into string table of name
    uint8_t n_type;         // type of symbol
    uint8_t n_other;        // misc info (usually empty)
    uint16_t n_desc;        // description field
    uintptr_t n_value;  // value of symbol
};
```

在 kdebug.c 中发现对于 stab 表的查找是个二分查找，需要给予他左断点和右端点以及对应的地址，所以对应上面的信息，给出了如下的代码，

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

我们需要的是他的描述信息，所以对应的是 n_desc

在对应的 mon_backtrace 进行对应的输出即可

```c
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
// Your code here.
//  cprintf("%08x", read_ebp());
    uint32_t *eip, *ebp;
    ebp = (uint32_t*) read_ebp();
    eip = (uint32_t*) ebp[1];
    cprintf("Stackbacktrace:\n");
    while (ebp!=0) {
        cprintf("ebp %08x eip %08x args %08x %08x %08x %08x %08x\n",ebp, eip, ebp[2] ,ebp[3], ebp[4], ebp[5] ,ebp[6]);
        struct Eipdebuginfo temp_debuginfo;
        debuginfo_eip((uintptr_t) eip, &temp_debuginfo);
        cprintf("       %s:%d: ", temp_debuginfo.eip_file, temp_debuginfo.eip_line);
        uint32_t i = 0;// = temp_debuginfo.eip_fn_namelen;
        while  (i < temp_debuginfo.eip_fn_namelen){
            cprintf("%c", temp_debuginfo.eip_fn_name[i]);
            i++;
        }
        int p = (int)eip;
        int q = (int)temp_debuginfo.eip_fn_addr;
        cprintf("+%x\n", p - q);
        ebp=(uint32_t*)ebp[0];
        eip=(uint32_t*)ebp[1];
    }

//  cprintf("%d", read_esp());
    return 0;
}
```

上述两个 exercise 的具体效果如下

```
plutoshe@ubuntu:~/PlutoShe/Os/6.828/lab$ make qemu
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stackbacktrace:
ebp f010ff18 eip f0100087 args 00000000 00000000 00000000 00000000 f0100c4c
       kern/init.c:19: test_backtrace+47
ebp f010ff38 eip f0100069 args 00000000 00000001 f010ff78 00000000 f0100c4c
       kern/init.c:16: test_backtrace+29
ebp f010ff58 eip f0100069 args 00000001 00000002 f010ff98 00000000 f0100c4c
       kern/init.c:16: test_backtrace+29
ebp f010ff78 eip f0100069 args 00000002 00000003 f010ffb8 00000000 f0100c4c
       kern/init.c:16: test_backtrace+29
ebp f010ff98 eip f0100069 args 00000003 00000004 00000000 00000000 00000000
       kern/init.c:16: test_backtrace+29
ebp f010ffb8 eip f0100069 args 00000004 00000005 00000000 00010094 00010094
       kern/init.c:16: test_backtrace+29
ebp f010ffd8 eip f01000ea args 00000005 00001aac 00000644 00000000 00000000
       kern/init.c:43: i386_init+4d
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
       kern/entry.S:83: <unknown>+0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```