Report for lab1, Yichen Mao

Environment Configuration

```
Hardware Environment:
Memory:          4GB
Processor:       Intel® Core™ i5 CPU @ 2.00GHz × 2
Graphics:        Intel® Ivybridge Mobile
OS Type:         64 bit
Disk:            20GB

Software Environment:
OS:              Ubuntu 18.04 LTS(x86_64)
Gcc:             Gcc 5.4.0
Make:            GNU Make 4.1
Gdb:             GNU gdb 7.11.1
```

Test Compiler Toolchain

```
$ objdump -i   # the 5th line say elf32-i386
$ gcc -m32 -print-libgcc-file-name
/usr/lib/gcc/x86_64-linux-gnu/5/32/libgcc.a
```

QEMU Emulator

```
 # Clone the IAP 6.828 QEMU git repository
 $ git clone https://github.com/geofft/qemu.git -b 6.828-1.7.0
 $ cd qemu
 $ ./configure --disable-kvm --target-list="i386-softmmu x86_64-softmmu"
 $ make
 $ sudo make install
```

PC Bootstrap

Simulating the x86

```
houmin@cosmos:~/lab$ make
+ 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 390 bytes (max 510)
+ mk obj/kern/kernel.img
```

After compiling, we now have our boot loader(obj/boot/boot) and out kernel(obj/kern/kernel), So where is the disk?
Actually the `kernel.img` is the disk image, which is acting as the virtual disk here. From kern/Makefrag we can see that
both our boot loader and kernel have been written to the image(using the `dd` command).

Now we can running the QEMU like running a real PC.

```
houmin@cosmos:~/lab$ make qemu
```

```
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log
WARNING: Image format was not specified for 'obj/kern/kernel.img' and probing guessed
raw.
         Automatically detecting the format is dangerous for raw images, write
operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
6828 decimal is XXX 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
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>
```

This Complete The Lab.

Part 1 : PC Bootstrap

Exercise 1.

Task : 学习和熟悉了解汇编语言。

Exercise 2.

Task : Use GDB's si command to trace into the ROM BIOS and guess what it might be doing.

```
Instructions
01 [f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
02 [f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6ac8
03 [f000:e062]    0xfe062: jne    0xfd2e1
04 [f000:e066]    0xfe066: xor    %dx,%dx
05 [f000:e068]    0xfe068: mov    %dx,%ss
06 [f000:e06a]    0xfe06a: mov    $0x7000,%esp
07 [f000:e070]    0xfe070: mov    $0xf34c2,%edx
```

```
08 [f000:e076]    0xfe076: jmp    0xfd15c
09 [f000:d15c]    0xfd15c: mov    %eax,%ecx
10 [f000:d15f]    0xfd15f: cli
11 [f000:d160]    0xfd160: cld
12 [f000:d161]    0xfd161: mov    $0x8f,%eax
13 [f000:d167]    0xfd167: out    %al,$0x70
14 [f000:d169]    0xfd169: in     $0x71,%al
15 [f000:d16b]    0xfd16b: in     $0x92,%al
16 [f000:d16d]    0xfd16d: or     $0x2,%al
17 [f000:d16f]    0xfd16f: out    %al,$0x92
18 [f000:d171]    0xfd171: lidtw  %cs:0x6ab8
19 [f000:d177]    0xfd177: lgdtw  %cs:0x6a74
20 [f000:d17d]    0xfd17d: mov    %cr0,%eax
21 [f000:d180]    0xfd180: or     $0x1,%eax
22 [f000:d184]    0xfd184: mov    %eax,%cr0
23 [f000:d187]    0xfd187: ljmpl  $0x8,$0xfd18f
message:The target architecture is assumed to be i386
24 => 0xfd18f:     mov    $0x10,%eax
25 => 0xfd194:     mov    %eax,%ds
......
```

```
Explain
01 跳转到地址 [0xfe05b]
02~03 比较地址 [cs:0x6ac8] 4 字节是否为 0 ，此时寄存器 cs 值为 0xf000, 可以计算出地址为 [0xf6ac8] ,观察到该地址处 4 字节为0，因此不执行跳转语句。
04~08 将寄存器 dx 和 ss 清空，将栈顶 (%esp) 设置为 0x7000，将寄存器 edx 设置为 0xf34c2，跳转到地址 [0xfd15c]。
09 将寄存器 eax 值赋值到寄存器 ecx。(寄存器 eax 和 ecx 初始值都为 0，不清楚发生什么)
10 cli 禁止中断发生。
11 cld 将方向寄存器清空。
12~14 将寄存器 eax 赋值为 0x8f，其中第 7 位为 1 ，将 NMI disabled，第 0~6 位为 0x0f ，执行完 out %al,$0x70 后，寄存器 eax为 0，查表 [https://bochs.sourceforge.io/techspec/PORTS.LST]，可得 normal execution of POST(Power-on self-test)。
15~17 查表得到第二位 indicates A20 active。将其置为 1，即激活A20。
18 加载 idt
19 加载 gdt
20~22 将寄存器 cr0 最低位置 1 ，即开启保护模式
23 跳转到地址 [0xfd18f]
```

Part 2 : The Boot Loader

Exercise 3.

Task :

1. Set a breakpoint at address 0x7c00 and compare the original boot loader source code with both the disassembly in and GDB.
2. Trace into function **bootmain()** and **readsect()**, identify the exact assembly instructions that correspond to each of the statements in readsect().
3. Identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk in function **bootmain()**.
4. Find out what code will run when the loop is finished, set a breakpoint there.

Work:

1. 比较 boot.asm 和 boot.S 和 GDB 反汇编的结果可以发现

```
boot.S:   xorw %ax,%ax
boot.asm: xor %eax,%eax
GDB:      xor %ax,%ax

在 xor，mov，in，test，out，or 等指令中，boot.S 通过在指令之后加上 w，b，l 等表示操作的位长，而 boot.asm 通过在寄存器来体现操作的位长。
```

```
boot.S:   movl $start,%esp
boot.asm: mov  $0x7c00,%esp
GDB:      mov  $0x7c00,%esp

boot.S 中一些标志在 boot.asm 和 GDB 反汇编中用对应地址代替。
```

2. 跟踪进入函数 bootmain() 和 readsect() ，并判断函数 readsect() 中每条指令对应汇编。

```
进入函数 bootmain() 0x7c45: call 0x7d0b
进入函数 readsect() 0x7cf1: call 0x7c81
```

```
函数 readsect() 源代码
function readsect()
void
readsect(void *dst, uint32_t offset)
{
  // wait for disk to be ready
  waitdisk();

  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
```

```
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // wait for disk to be ready
  waitdisk();

  // read a sector
  insl(0x1F0, dst, SECTSIZE/4);
}
```

进入函数 readsect() 时进行的一些寄存器操作。

```
=> 0x7c81:  push   %ebp
=> 0x7c82:  mov    %esp,%ebp
=> 0x7c84:  push   %edi
=> 0x7c85:  mov    0xc(%ebp),%edi

waitdisk();
=> 0x7c88:  call   0x7c6c

outb(0x1F2, 1);
=> 0x7c8d:  mov    $0x1f2,%edx
=> 0x7c92:  mov    $0x1,%al
=> 0x7c94:  out    %al,(%dx)

outb(0x1F3, offset);
=> 0x7c95:  mov    $0xf3,%dl
=> 0x7c97:  mov    %edi,%eax
=> 0x7c99:  out    %al,(%dx)

outb(0x1F4, offset >> 8);
=> 0x7c9a:  mov    %edi,%eax
=> 0x7c9a:  mov    %edi,%eax
=> 0x7c9f:  mov    $0xf4,%dl
=> 0x7ca1:  out    %al,(%dx)

outb(0x1F5, offset >> 16);
=> 0x7ca2:  mov    %edi,%eax
=> 0x7ca4:  shr    $0x10,%eax
=> 0x7ca7:  mov    $0xf5,%dl
=> 0x7ca9:  out    %al,(%dx)

outb(0x1F6, (offset >> 24) | 0xE0);
=> 0x7caa:  shr    $0x18,%edi
=> 0x7caa:  shr    $0x18,%edi
=> 0x7caf:  or     $0xffffffe0,%eax
=> 0x7cb2:  mov    $0xf6,%dl
x7cb4:  out    %al,(%dx)

outb(0x1F7, 0x20);
```

```
=> 0x7cb5:  mov    $0xf7,%dl
=> 0x7cb7:  mov    $0x20,%al
=> 0x7cb9:  out    %al,(%dx)

waitdisk();
=> 0x7cba:  call   0x7c6c

insl(0x1F0, dst, SECTSIZE/4);
=> 0x7cbf:  mov    0x8(%ebp),%edi
=> 0x7cc2:  mov    $0x80,%ecx
=> 0x7cc7:  mov    $0x1f0,%edx
=> 0x7ccc:  cld
=> 0x7ccd:  repnz insl (%dx),%es:(%edi)

=> 0x7ccf:  pop    %edi
=> 0x7cd0:  pop    %ebp
=> 0x7cd1:  ret
```

```
0x7d4b: pushl  0x4(%ebx)
0x7d4e: pushl  0x14(%ebx)
0x7d51: pushl  0xc(%ebx)
0x7d54: call   0x7cd2
0x7d59: add    $0x20,%ebx
0x7d5c: add    $0xc,%esp
0x7d5f: cmp    %esi,%ebx
0x7d61: jb     0x7d4b
```

```
循环结束后程序会执行下列语句
7d63: ff 15 18 00 01 00     call   *0x10018
```

Questions :

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

答：执行完语句 [0x7c2d: ljmp $0x8,$0x7c32] 后，给出提示信息:[The target architecture is assumed to be i386]，此时系统开始执行 32-bit 代码。转换原因是 [0x7c23: mov %cr0,%eax]，[0x7c26: or $0x1,%eax]，[0x7c2a: mov %eax,%cr0] 三条指令将 %cr0 的最低位从 0 修改为 1，而 %cr0 最低位表示是否处于 protected 模式，也就是系统从 real 模式转变为了 protected 模式。

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

答：boot load 执行的最后一条指令为：

```
((void (*)(void)) (ELFHDR->e_entry))();
0x7d63: ff 15 18 00 01 00   call   *0x10018
```

kernel 执行的第一条指令为：

```
    0x10000c: movw   $0x1234,0x472
```

3. Where is the first instruction of the kernel?

```
kernel 执行的第一套指令地址为 0x10000c,源码位于 kern/entry.S 的第44 行。
```

4. 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?

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

上述代码先将内核中前 512*8 字节的内容填充到 ELFHDR 开头的地方，其中前若干个字节内容构成结构体。

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

ph 为 第一个段对应的结构体的地址，ELFHDR->e_phnum 记录段的数量。
对于每一个段，通过 ph->p_pa 和 ph->p_memsz 可以计算出段加载的首地址和尾地址，由此可以计算出每个段所需要扇区，所有段的扇区之和即所需要的扇区数量。

其中，ELFHDR结构体位于内核偏移量为0处。而每个段的结构体可由ELFHDR计算得到。

Exercise 4.

Task :熟悉了解指针。

Exercise 5.

Task : 修改 boot/Makefrag 中链接地址观察会发生什么。

Work:

```
程序会在执行下述语句中出错
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x1c32
```

Exercise 6.

Task :思考刚进入 boot loader 和 从 boot loader 进入kernel 时，内存地址 0x00100000 内容变化。

```
刚进入 boot loader 时，内存 0x00100000 处都是 0
而从 boot loader 进入 kernel 是，内存 0x00100000 处存在内容，且正好是 kernel 的二进制代码，这是因为 boot loader 把 kernel 加载到了内存 0x00100000 处。
```

Part 3: The Kernel

Exercise 7.

Task :

1. 观察语句`movl %eax, %cr0`执行前后内存0x00100000和0xf0100000处变化。
2. 注释 `movl %eax, %cr0` 观察会在那个语句出错。

Work:

1.

```
执行前:
0x100000: 0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010: 0x34000004  0x7000b812  0x220f0011  0xc0200fd8
0x100000 处为 kernel 二进制代码。

0xf0100000 <_start+4026531828>: 0x00000000  0x00000000  0x00000000  0x00000000
0xf0100010 <entry+4>: 0x00000000  0x00000000  0x00000000  0x00000000
0xf0100000 处全为 0。

执行后:
0x100000: 0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010: 0x34000004  0x7000b812  0x220f0011  0xc0200fd8
0x100000 处仍为 kernel 二进制代码。

0xf0100000 <_start+4026531828>: 0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0xf0100010 <entry+4>: 0x34000004  0x7000b812  0x220f0011  0xc0200fd8
0xf0100000 处变为 kernel 二进制代码。
```

2.

```
修改代码后，执行语句
=> 0xf010002c <relocated>:  add    %al,(%eax)
qemu报错：fatal：Trying to execute code outside RAM or ROM at 0xf010002c
```

Exercise 8.

Task: 观察填补代码。

```
将 printfmt.c 中 函数 vprintmt 第 209~211 修改为以下代码：
num = getuint(&ap, lflag);
base = 8;
goto number;
```

Question :

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 调用函数 cputchar 传递出输出的字符，console.c 具体实现的函数 cputchar，将要输出的字符打印到屏幕上。
   ```

2. Explain the following from `console.c`。

   ```
   1      if (crt_pos >= CRT_SIZE) {
   2              int i;
   3              memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) *
   sizeof(uint16_t));
   4              for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
   5                      crt_buf[i] = 0x0700 | ' ';
   6              crt_pos -= CRT_COLS;
   7      }

   CRT_SIZE = 25*80  表示终端可显示的字符数
   CRT_COLS = 80     表示终端一行可显示的字符数
   crt_pos           表示当前终端显示的字符数
   若当前显示字符数大于等于可显示的字符数，那么 memmove 函数就是将后 24 行的内容复制到前 24 行，然后 for 循环将最后一行清空，由于少了一行，因此最后当前显示的字符数 -= CRT_COLS
   ```

3. Trace the execution of the following code step-by-step:

   ```
   fmt 指针指向的是 "x %d, y %x, z %d\n" 的地址。
   ap va_list 是可变参数列表，xyz对应的值存在栈 0xf0116f04 开始的 12 个字节中。
   ```

```
vcprintf (fmt = 0xf0101ade "x %d, y %x, z %d\n", ap=0xf0116f04 "\001")
cons_putc (c = 0xf0101adf " ")
cons_putc (c = 0xf0101ae0 "%")
va_arg(*ap, int) ap:0xf0116f04 "0x01" -> 0xf0116f08 "0x03"                //
0xf0100f95
cons_putc (c = 32)
cons_putc (c = 0xf0101ae3 " ")
cons_putc (c = 0xf0101ae4 "y")
cons_putc (c = 0xf0101ae5 " ")
cons_putc (c = 0xf0101ae6 "%")
va_arg(*ap, unsigned int) ap:0xf0116ebc "0x08" -> 3
//0xf0100cc0
cons_putc (c = 32)
cons_putc (c = 0xf0101ae9 " ")
cons_putc (c = 0xf0101aea "z")
cons_putc (c = 0xf0101aeb " ")
cons_putc (c = 0xf0101aec "%")
va_arg(*ap, int) ap:0xf0116f0c "0x04" -> 0xf0116f10 "0xf01008c0"
```

4. 运行程序，解释结果。

```
代码
  unsigned int i = 0x00646c72;
  cprintf("H%x Wo%s", 57616, &i);
输出结果为： He110 World
解释：第一个输出为 %x,即要求按照 16 进制输出，57616(10) = e110(16)，第二个把 i 看作字符串输出，i 从高位到低位每个 bit 分别对应字符 NUL，'d'，'l'，'r'，由于是小端法，因此高位对应低地址，因此输出为 'rld'。如果是大端法，那么 i 应该设置为 0x726c6400。
```

5. 解释下列代码。

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

```
输出结果为 x=3 y=-267292872
如第三问回答中提到，在调用 cprintf 函数过程中会将输出值的参数存放在栈中，然后再将栈的首地址作为参数传到内层函数。该代码中，由于y每个给出具体值，因此未修改对应栈元素，输出过程中，访问到栈中元素，由于栈中元素随机生成，因此最终会输出一个不可控的值。
```

6. 解决GCC后面的参数后入栈的问题。

```
修改 cprintf 内部实现，扫描字符串 fmt，可以计算出参数的个数以及每个参数的类型，以此将栈空间的参数翻转。
```

Challenge

Task: 能够以不同的颜色在终端输出文字。

```
1. 可知在函数 cga_putc 中，参数 c 的第 8~11 位表示输出字的颜色，12~15 位表示背景的颜色。
2. 通过 ANSI Codes 中 ESC[Ps;...;Psm 实现修改文字颜色的目的，即在 cprintf 的字符串中插入该语句实现。
3. 在函数 cga_putc 进行字符处理，解析处修改颜色的参数，并将最近一次修改存放在变量 Color 中。
4. 修改语句 if (!(c & ~0xFF))c |= 0x0700; => if (!(c & ~0xFF))c |= Color; 即可
详细程序可见 程序 console.c 中 函数 cga_putc，调用示例可见程序 monitor.c 中函数 mon_backtrace。
```

（效果如上）

Exercise 9.

Task: 观察代码 kern/entry.S 和 kern/init.c ， 了解初始化栈过程。

Question :

1. 内核什么时候初始化栈以及栈存放在内存那个地方。

```
=> 0xf010002f <relocated>:  mov    $0x0,%ebp
=> 0xf0100034 <relocated+5>:  mov    $0xf0117000,%esp
上述指令完成对栈的初始化。

KSTKSIZE = 8*PGSIZE = 8 * 4096
栈顶地址 = 0xf0117000
故栈存放在内存 [0xf010f000,0xf0117000]处。
```

2. 内核是如何为栈保留空间以及栈指针最初指向哪一端。

```
在 entry.S 的数据段申明了栈空间
bootstack:
  .space    KSTKSIZE
  .globl    bootstacktop

栈指针最初指向地址高的一端。
```

Exercise 10.

Task：分析函数 test_backtrace。

```
函数 test_backtrace 地址为 0xf0100040
每次传入 8 个 32-bit 到栈中，以其中一次调用为例解释每个内容的含义。
x = 1
0xf0116f40: 0x00000000  0x00000001  0xf0116f78  0x00000000
0xf0116f50: 0xf0100898  0x00000002  0xf0116f78  0xf0100069
第 8 个是函数调用的返回地址，表示当前函数结束后跳转到对应的指令的地址。
第 7 个是 push $ebp 指令时寄存器 ebp 的值，表示当前栈帧的栈底。
第 6 个是 push $ebx 指令时寄存器 ebx 的值，除了第一次调用该函数外，其他调用时寄存器都被指令 mov 0x8(%ebp),%ebx 修改为上一次的 x。
第 3~5 参数都是在函数 cprintf 中修改其中第 3 个是栈信息，第 5 个是函数返回地址，指向函数 putch(和当前函数无关)
第 2 个是由指令 mov 0x8(%ebp),%ebx 和 mov %ebx,0x4(%esp)修改为当前 x 的值。
第 1 个是调用函数的参数，也就是下一次 x 的值。
```

Exercise 11.

Task：实现函数 backtrace。

```
代码如下
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
  cprintf("Stack backtrace:\n");
  uint32_t eip;
  for (uint32_t ebp = read_ebp(); ebp; ebp = *((uint32_t *) ebp)){
    eip = *((uint32_t *) ebp + 1);
    cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n",
    ebp,eip,*((uint32_t *) ebp + 2),
            *((uint32_t *) ebp + 3),
        *((uint32_t *) ebp + 4),
        *((uint32_t *) ebp + 5),
        *((uint32_t *) ebp + 6));
  }
  return 0;
```

```
}
```
通过 Exercise 10 中的观察我们可以知道，ebp 就是其中的第 7 个参数， eip 就是其中的第 8 个参数，再往上几个参数是前一层栈帧中保留的下一次函数调用的参数，因此恰好为本次调用的参数，依次输出。

Exercise 12.

Task： 继续完善程序

首先完善查找 eip_line 的程序，使用已给出的二分查找即可。

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

完善 mon_backtrace 函数，通过 debuginfo_eip 函数由 eip 获得函数信息，并输出。

```
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
  cprintf("Stack backtrace:");
  uint32_t eip;
  struct Eipdebuginfo info;
  for (uint32_t ebp = read_ebp(); ebp; ebp = *((uint32_t *) ebp)){
    eip = *((uint32_t *) ebp + 1);
    debuginfo_eip(eip, &info);
    cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n         %s:%d:
%.*s+%d\n",
    ebp,eip,*((uint32_t *) ebp + 2),
            *((uint32_t *) ebp + 3),
        *((uint32_t *) ebp + 4),
        *((uint32_t *) ebp + 5),
        *((uint32_t *) ebp + 6),
        info.eip_file,
        info.eip_line,
        info.eip_fn_namelen,
        info.eip_fn_name,
        eip - info.eip_fn_addr);
  }
  return 0;
}
```

修改 command，增加

```
{ "backtrace", "Display information about the stack backtrace", mon_backtrace}
```