# 操作系统实习报告

*曹胜操 - 1500012838*

## Lab 1: Booting a PC

### Part 1: PC Bootstr ap

> Exercise 1. 熟悉汇编语言，参考：
>
> https://pdos.csail.mit.edu/6.828/2016/reference.html
>
> http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

汇编语言我们已经在ICS等课程中接触过，在本门课程中我们同样需要大量阅读、编写汇编代码，因此我们必须对汇编语言有必要的熟悉。后面我们将遇到一些没有见过的汇编指令，可能需要查阅一些相关资料。

> Exercise 2. 使用GDB的`si`指令追踪ROM BIOS的一些指令，猜测它可能在做什么。不必弄清楚所有细节。参考：
>
> http://web.archive.org/web/20040404164813/members.iweb.net.au/~pstorr/pcbook/book2/book2.htm
>
> https://pdos.csail.mit.edu/6.828/2016/reference.html

GDB的部分追踪结果如下，查阅资料后将汇编代码意义注释于旁。

```
0xffff0:     ljmp   $0xf000,$0xe05b        跳转到一个较低的地址 有更多代码
0xfe05b:     cmpl   $0x0,%cs:0x6ac8        测试内存中某个值是否为零 此处为零
0xfe062:     jne    0xfd2e1
0xfe066:     xor    %dx,%dx                初始化一些寄存器
0xfe068:     mov    %dx,%ss
0xfe06a:     mov    $0x7000,%esp
0xfe070:     mov    $0xf34c2,%edx
0xfe076:     jmp    0xfd15c
0xfd15c:     mov    %eax,%ecx              此处为零
0xfd15f:     cli                          clear interrupt flag   阻止中断
0xfd160:     cld                          clear direction flag   串操作不再改变下标寄存器
0xfd161:     mov    $0x8f,%eax
0xfd167:     out    %al,$0x70              向0x70号端口(CMOS)写入值0x8f
0xfd169:     in     $0x71,%al              从0x71号端口(CMOS)读出值
0xfd16b:     in     $0x92,%al              对0x92号端口进行读写
0xfd16d:     or     $0x2,%al
0xfd16f:     out    %al,$0x92
0xfd171:     lidtw  %cs:0x6ab8             load interrupt descriptor table   读取中断向量表寄
存器
0xfd177:     lgdtw  %cs:0x6a74             load global descriptor table   读取全局描述符表寄存
器
0xfd17d:     mov    %cr0,%eax              将%cr0的最低位置一 表示保护模式
0xfd180:     or     $0x1,%eax
0xfd184:     mov    %eax,%cr0
... ...
```

总之，在BIOS启动时，它需要对诸多寄存器、描述符表、设备等进行初始化。之后它会搜索一个可启动的设备，读取其Boot loader，并将控制权转移过去。

## Part 2: The Boot Loader

> Exercise 3. 熟悉GDB指令。在地址`0x7c00`设置断点，追踪`boot/boot.S`中的代码，并且对比用`x/i`指令反汇编得到的代码、`boot/boot.S`中的源代码和`obj/boot/boot.asm`中的反汇编代码。
>
> 然后追踪`boot/main.c`中的`readsect()`，找到汇编指令和源代码之间的对应关系。回答以下问题：

- 处理器何时开始执行32位代码？什么导致了这一转变？

  从`movw $PROT_MODE_DSEG, %ax`开始，处理器执行的都是32位代码。其原因是前几条汇编指令通过设置全局描述符表寄存器`GDTR`和控制寄存器`cr0`，将实模式切换为保护模式，然后通过一个特殊的跳转`ljmp $PROT_MODE_CSEG, $protcseg`开始执行32位代码。

- Boot loader最后执行的一条指令是什么？Kernel最先执行的一条指令是什么？它在哪里？

  `0x7d6b: call *0x10018`：跳转到Kernel

  `0x10000c: movw $0x1234,0x472`：`kernel/entry.S`中的第一条指令

- Boot loader如何确定需要读取多少个扇区来获取整个Kernel？

  在`boot/main.c`中，读取了第一个扇区后，Boot loader从`ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff); eph = ph + ELFHDR->e_phnum;`可以发现ELF header中存储着需要读取多少扇区，即`ELFHDR->e_phnum`。

> Exercise 4. 熟悉C语言中的指针，参考*The C Programming Language (K&R)*。下载代码`pointers.c`并理解输出值来自何处。

代码的主要部分：

```
void
f(void)
{
    int a[4];
    int *b = malloc(16);
    int *c;
    int i;

    printf("1: a = %p, b = %p, c = %p\n" , a, b, c);

    c = a;
    for (i = 0; i < 4; i++)
    a[i] = 100 + i;
    c[0] = 200;
    printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n" ,
       a[0], a[1], a[2], a[3]);

    c[1] = 300;
    *(c + 2) = 301;
    3[c] = 302;
    printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n" ,
       a[0], a[1], a[2], a[3]);

    c = c + 1;
    *c = 400;
    printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n" ,
       a[0], a[1], a[2], a[3]);

    c = (int *) ((char *) c + 1);
    *c = 500;
    printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n" ,
       a[0], a[1], a[2], a[3]);

    b = (int *) a + 1;
    c = (int *) ((char *) a + 1);
    printf("6: a = %p, b = %p, c = %p\n" , a, b, c);
}
```

程序的输出结果：

```
1: a = 0x7ffc9b635b60, b = 0x1875010, c = (nil)
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7ffc9b635b60, b = 0x7ffc9b635b64, c = 0x7ffc9b635b61
```

`a` `b` `c`是三个`int*`类型的指针。

第一行输出了它们的值，即一个地址，`c`没有赋初值，所以可能是`(nil)`。

之后将`a`指向的数组内容进行修改，又通过`c`来修改了第一个元素，所以可以看到第二行中`a[0]`的不同。

第三、四行中展示了通过指针找到其指向处之后某些元素的几种方法：`c[1]` `*(c + 2)` `3[c]` `*(++c)`，这些都是合法的。

第五行中，`c`原本指向`a[1]`，但`c = (int *) ((char *) c + 1)`使之指向了`a[1]`与`a[2]`之间的位置。因此修改`*c`会导致两者的异常改变。

第六行解释了不同指针类型造成的上述差异。由于`int`占4字节，`char`占1字节，可以看到`b`比`a`大4，而`c`只比`a`大`1`。

> Exercise 5. 再次追踪Boot loader的前几条指令。找到链接地址不正确时，出错的第一条指令。修改`boot/Makefrag`中的链接地址，观察会发生什么。

在`boot/Makefrag`中，将`-Ttext 0x7c00`改为`-Ttext 0x7000`，重新编译运行。可以发现在运行时Boot loader的起始地址仍为`0x7c00`。向下继续运行，会发现到`ljmp $PROT_MODE_CSEG, $protcseg`这一指令时出现了错误。它试图跳转到`$0x7032`而不是原本应该的`$0x7c32`处，这一错误使得Boot loader无法再继续运行下去。

> Exercise 6. GDB可以查看内存内容。重启，分别检查BIOS进入Boot loader时和进入Kernel时`0x00100000`处8个字的内存内容。它们为什么不一样？第二次时那里有什么内容？

进入Boot loader时：

```
(gdb) x/8wx 0x00100000
0x100000:   0x00000000  0x00000000  0x00000000  0x00000000
0x100010:   0x00000000  0x00000000  0x00000000  0x00000000
```

进入Kernel时：

```
(gdb) x/8wx 0x00100000
0x100000:   0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010:   0x34000004  0x0000b812  0x220f0011  0xc0200fd8
```

Boot loader会将Kernel读取到从`0x00100000`开始的内存里，第二次所看到的内存内容正是Kernel开始的一部分，即：

```
f0100000:   02 b0 ad 1b 00 00       add    0x1bad(%eax),%dh
f0100006:   00 00                   add    %al,(%eax)
f0100008:   fe 4f 52                decb   0x52(%edi)
f010000b:   e4                      .byte 0xe4
... ...
```

注意小端法造成的显示差异。

## Part 3: The Kernel

> Exercise 7. 使用QEMU和GDB追踪进入JOS kernel，在指令`movl %eax, %cr0`处停下。在执行这条指令前后，检查`0x00100000`和`0xf0100000`处的内存内容。
>
> 新的地址映射建立后，如果映射不正确，出错的第一条指令会是哪条？注释掉`movl %eax, %cr0`来检查。

改变`%cr0`之前：

```
(gdb) x/8wx 0x00100000
0x100000:    0x1badb002   0x00000000   0xe4524ffe   0x7205c766
0x100010:    0x34000004   0x0000b812   0x220f0011   0xc0200fd8
(gdb) x/8wx 0xf0100000
0xf0100000 <_start+4026531828>:  0x00000000   0x00000000   0x00000000   0x00000000
0xf0100010 <entry+4>:    0x00000000   0x00000000   0x00000000   0x00000000
```

改变`%cr0`之后：

```
(gdb) x/8wx 0x00100000
0x100000:    0x1badb002   0x00000000   0xe4524ffe   0x7205c766
0x100010:    0x34000004   0x0000b812   0x220f0011   0xc0200fd8
(gdb) x/8wx 0xf0100000
0xf0100000 <_start+4026531828>:  0x1badb002   0x00000000   0xe4524ffe   0x7205c766
0xf0100010 <entry+4>:    0x34000004   0x0000b812   0x220f0011   0xc0200fd8
```

可见，设置状态寄存器后，可以正确地将高地址映射到低地址上。如果不这么做，稍后的`jmp *%eax`就会试图跳转到一个非常大的、没有被正确映射的地址去，导致错误。

> Exercise 8. 找到并补全输出八进制数`"%o"`的代码段。回答以下问题：

这段代码在`lib/printfmt.c`中，参考其它格式即可补全：

```
// (unsigned) octal
case 'o':
    num = getuint(&ap, lflag);
    base = 8;
    goto number;
```

- 解释`printf.c`和`console.c`之间的接口。更具体地，`console.c`提供了什么函数？它又是怎么被`printf.c`使用的？

  在`printf.c`的`putch()`函数中调用了`console.c`中的`cputchar()`函数。它负责在控制台输出一个字符。
- 解释`console.c`中的下列代码：

```
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;
}
```

当控制台上的字符总数已经超限时，需要将原有字符向上移动一行，将新的一行全部设为空白字符，然后调整当前字符应输出的位置。

- 追踪以下代码：

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

  - 调用`cprintf()`时，`fmt`和`ap`分别指向什么？

    `fmt`指向格式串`"x %d, y %x, z %d\n"`。`ap`则指向参数列表。

  - 按顺序列出每次对`cons_putc` `va_arg` `vcprintf`的调用及其参数。

    `vcprintf()`：`fmt`指向`"x %d, y %x, z %d\n"`，`ap`指向第一个参数`x`，这里的值为`1`。

    `cons_putc()`：`c`为字符`'x'`。

    `cons_putc()`：`c`为字符`' '`。

    `va_arg()`：之前`ap`指向第一个参数`x`，这里的值为`1`；。之后`ap`指向第二个参数`y`，这里的值为`3`。

    `cons_putc()`：`c`为字符`'1'`。

    以此类推，直到输出完整个字符串。

- 运行以下代码：

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

输出是什么？解释这个输出。如果x86使用大端法，应该如何修改`i`达到同样的输出效果？`57616`这个数需要改吗？

输出`He110 World`。`57616`用十六进制表示就是`e110`，而如果把`i`解释为字符串就是`"rld"`。对于大端法，应把`i`改为`0x726c6400`，而`57616`不需要修改。

- 在以下代码中，`y=`后会输出什么？为什么会这样？

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

输出是`x=3 y=-267380484`。由于没有指定参数，实际输出的是内存中接下来位置中的一个未知内容。

- 这里GCC改变了调用传统，是按照声明顺序压入参数的，即最右边的参数最后被压栈。如果按照相反的顺序，应该怎么修改`cprintf`或者其接口？

应该修改`va_start` `va_arg` `va_end`等可变参数相关函数。

> Challenge 修改控制台，使之能够打印不同的颜色。

在`kern/console.c`中，我们可以发现打印一个字符时调用了`cga_putc()`函数，这里可以控制VGA输出的颜色。

参数`c`的第一个字节为输出的字符的ASCII码，而第二个字节可以指定字符的前景色和背景色。在默认情况下，这里被设置为`0x07`，即黑底白字。我们可以把它改为`0x71`，使之输出白底蓝字。当然，我们还可以进一步修改输出函数的接口，使之有更加丰富的功能。

> Exercise 9. 确定Kernel在哪里初始化了栈，以及栈存储在内存的何处。Kernel如何预留这部分空间？栈指针指向它的哪一端？

在`kern/entry.S`中有以下代码：

```
movl    $0x0,%ebp
movl    $(bootstacktop),%esp
```

将`%ebp`置零，`%esp`设为`$(bootstacktop)`，完成了对栈的初始化。观察`obj/kern/kernel.asm`可以发现实际的值为`$0xf0110000`。在`kern/entry.S`末尾定义了`bootstack`和`bootstacktop`，其间有`KSTKSIZE`字节的空间。参考`inc/memlayout.h`和`inc/mmu.h`可以发现`KSTKSIZE`大小为32K，即栈的大小为32KB。`%esp`一开始指向它的高地址端，栈是向下生长的。

> Exercise 10. 找到函数`test_backtrace`的地址，设置断点并检查每次被调用时发生了什么。每个递归调用层会压入多少个32位字？这些数据是什么？

第一次调用时，`%esp`的值为`0xf010ffdc`。第二次为`0xf010ffbc`，第三次为`0xf010ff9c`，以此类推。因此，每层递归会压入8个32位字。第四次调用刚开始时，查看内存结果为：

```
0xf010ff7c:  0xf0100068   0x00000002   0x00000003   0xf010ffb8
0xf010ff8c:  0x00000000   0xf010094b   0x00000004   0xf010ffb8

0xf010ff9c:  0xf0100068   0x00000003   0x00000004   0x00000000
0xf010ffac:  0x00000000   0x00000000   0x00000005   0xf010ffd8

0xf010ffbc:  0xf0100068   0x00000004   0x00000005   0x00000000
0xf010ffcc:  0x00010094   0x00010094   0x00010094   0xf010fff8

0xf010ffdc:  0xf01000d4   0x00000005   0x00001aac   0x00000644
0xf010ffec:  0x00000000   0x00000000   0x00000000   0x00000000
```

结合汇编代码中的以下行：

```
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
f010004a:   53                      push   %ebx
f010004b:   68 e0 18 10 f0          push   $0xf01018e0
f0100050:   e8 2f 09 00 00          call   f0100984 <cprintf>
f0100055:   83 c4 10                add    $0x10,%esp
f0100058:   85 db                   test   %ebx,%ebx
f010005a:   7e 11                   jle    f010006d <test_backtrace+0x2d>
f010005c:   83 ec 0c                sub    $0xc,%esp
f010005f:   8d 43 ff                lea    -0x1(%ebx),%eax
f0100062:   50                      push   %eax
f0100063:   e8 d8 ff ff ff          call   f0100040 <test_backtrace>
f0100068:   83 c4 10                add    $0x10,%esp
f010006b:   eb 11                   jmp    f010007e <test_backtrace+0x3e>
f010006d:   83 ec 04                sub    $0x4,%esp
f0100070:   6a 00                   push   $0x0
f0100072:   6a 00                   push   $0x0
f0100074:   6a 00                   push   $0x0
f0100076:   e8 0a 07 00 00          call   f0100785 <mon_backtrace>
f010007b:   83 c4 10                add    $0x10,%esp
f010007e:   83 ec 08                sub    $0x8,%esp
f0100081:   53                      push   %ebx
f0100082:   68 fc 18 10 f0          push   $0xf01018fc
f0100087:   e8 f8 08 00 00          call   f0100984 <cprintf>
f010008c:   83 c4 10                add    $0x10,%esp
f010008f:   8b 5d fc                mov    -0x4(%ebp),%ebx
f0100092:   c9                      leave
f0100093:   c3                      ret
```

可以推断每层栈帧中，地址最低处存储着下一层的返回地址，最高处存储着上一层的`%ebp`，中间则是一些参数、局部变量等。

> Exercise 11. 按照格式要求实现Backtrace函数。

函数`mon_backtrace()`实现如下：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    cprintf("Stack backtrace:\n");
    uint32_t* ebp = (uint32_t*)read_ebp();
    while (ebp)
    {
        cprintf("  ebp %08x  eip %08x  args", ebp, ebp[1]);
        for (int i = 2; i < 7; ++i) cprintf(" %08x", ebp[i]);
        cprintf("\n");
        ebp = (uint32_t*)*ebp;
    }
    return 0;
}
```

寄存器`%ebp`保存着栈帧底的地址。而在栈帧底，又储存着上一层调用的栈帧的`%ebp`，因此`ebp = (uint32_t*)*ebp`可以实现对栈帧的层层追踪。其余只要注意格式要求即可。

> Exercise 12. 修改Backtrace函数，使之能够对每个`eip`显示源文件名、函数名、行号等信息。
>
> 补全函数`debuginfo_eip`中关于查找行号的部分。
>
> 添加`backtrace`命令，并扩展你的`mon_backtrace`函数。

在`kern/kdebug.c`中，参考其它部分的查找，可以补全对于行号的查找：

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

在`kern/monitor.c`中，添加一行内容即可加入命令：

```
static struct Command commands[] = {
    { "help", "Display this list of commands", mon_help },
    { "kerninfo", "Display information about the kernel", mon_kerninfo },
    { "backtrace", "Display backtrace", mon_backtrace },
};
```

最后在`kern/monitor.c`中扩展函数`mon_backtrace()`：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    cprintf("Stack backtrace:\n");
    struct Eipdebuginfo info;
    uint32_t* ebp = (uint32_t*)read_ebp();
    while (ebp)
    {
        cprintf("  ebp %08x  eip %08x  args", ebp, ebp[1]);
        for (int i = 2; i < 7; ++i) cprintf(" %08x", ebp[i]);
        debuginfo_eip(ebp[1], &info);
        cprintf("\n         %s:%d: %.*s+%d\n", info.eip_file,
                info.eip_line, info.eip_fn_namelen, info.eip_fn_name,
                ebp[1] - info.eip_fn_addr);
        ebp = (uint32_t*)*ebp;
    }
    return 0;
}
```

注意`Eipdebuginfo`结构内容的使用方法以及格式要求即可。

## Reference

- https://docs.oracle.com/cd/19455-01/806-3773/6jct9o0af/index.html
- http://www.felixcloutier.com/x86/
- http://web.archive.org/web/20040404164813/members.iweb.net.au/~pstorr/pcbook/book2/book2.htm
- http://www.cnblogs.com/fatsheep9146/category/769143.html
- http://blog.csdn.net/scnu20142005027/article/details/51264186E