

## Lab1 : 系统软件启动过程

解振达 PB14210109

Nov 28, 2017

---

### 实验目的：

操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件-**bootloader** 来完成这些工作。为此，我们需要完成一个能够切换到 **x86** 的保护模式并显示字符的 **bootloader**，为启动操作系统 **ucore** 做准备。**lab1** 提供了一个非常小的 **bootloader** 和 **ucore OS**，整个 **bootloader** 执行代码小于 512 个字节，这样才能放到硬盘的主引导扇区中。通过分析和实现这个 **bootloader** 和 **ucore OS**，可以了解到：

#### > 计算机原理

- CPU 的编址与寻址：基于分段机制的内存管理
- CPU 的中断机制
- 外设：串口/并口/CGA，时钟，硬盘

#### > Bootloader 软件

- 编译运行 **bootloader** 的过程
- 调试 **bootloader** 的方法
- PC 启动 **bootloader** 的过程
- ELF 执行文件的格式和加载
- 外设访问：读硬盘，在 CGA 上显示字符串

#### > ucore OS 软件

- 编译运行 **ucore OS** 的过程
  - **ucore OS** 的启动过程
  - 调试 **ucore OS** 的方法
  - 函数调用关系：在汇编级了解函数调用栈的结构和处理过程
  - 中断管理：与软件相关的中断处理
  - 外设管理：时钟
- 

### 实验内容与结果：

**lab1** 中包含一个 **bootloader** 和一个 **OS**。这个 **bootloader** 可以切换到 **X86** 保护模式，能够读磁盘并加载 **ELF** 执行文件格式，并显示字符。而这 **lab1** 中的 **OS** 只是一个可以处理时钟中断和显示字符的幼儿园级别 **OS**。

#### 练习 1：理解通过 **make** 生成执行文件的过程

##### 1. 操作系统镜像文件 **ucore.img** 是如何一步一步生成的

[1] 首先执行指令 **make "V="** 可以得到如下结果：



到若干.o 文件或其他文件，具体的每条命令及跟随的命令参数在 report 当中已经很完备，就不再赘述，下面再看一下在生成 bootblock 时的 sign 工具；

```
# create 'sign' tools
$(call add_files_host,tools/sign.c,sign,sign)
$(call create_target_host,sign,sign)

# -----

char buf[512];
memset(buf, 0, sizeof(buf));
FILE *ifp = fopen(argv[1], "rb");
int size = fread(buf, 1, st.st_size, ifp);
if (size != st.st_size) {
    fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
    return -1;
}
fclose(ifp);
buf[510] = 0x55;
buf[511] = 0xAA;
FILE *ofp = fopen(argv[2], "wb+");
size = fwrite(buf, 1, 512, ofp);
if (size != 512) {
    fprintf(stderr, "write '%s' error, size is %d.\n", argv[2], size);
    return -1;
}
fclose(ofp);
printf("build 512 bytes boot sector: '%s' success!\n", argv[2]);
return 0;
```

> 通过 sign.c 也可以间接了解到硬盘主引导扇区的规范格式，在 2 中也会提到。

## 2. 一个被系统认为是符合规范的硬盘主引导扇区的特征是什么

主引导扇区位于整个硬盘的 0 磁头 0 柱面 1 扇区，包括硬盘主引导记录 MBR (Master Boot Record) 和分区表 DPT (Disk Partition Table)。规范的主引导扇区特征如下：

- [1] 总大小为 512 字节，由主引导程序、分区表、结束标志三部分构成；
- [2] 引导程序，从 0x0 位置起共 446 字节（隐含 windows 磁盘签名）；
- [3] 分区表，占用 64 字节，是 MBR 中的重要结构；
- [4] 结束标志，扇区的最后两个字节“55AA”是 MBR 的结束标志。

```
zdx@Antique:~/文档/ucore_os_lab-master/labcodes
00000000 fcfa c031 d88e c08e d08e 64e4 02a8 fa75
00000010 d1b0 64e6 64e4 02a8 fa75 dfb0 60e6 010f
00000020 6c16 0f7c c020 8366 01c8 220f eac0 7c32
00000030 0008 b866 0010 d88e c08e e08e e88e d08e
00000040 00bd 0000 bc00 7c00 0000 c0e8 0000 eb00
00000050 8dfe 0076 0000 0000 0000 0000 ffff 0000
00000060 9a00 00cf ffff 0000 9200 00cf 0017 7c54
00000070 0000 8955 57e5 8956 53c6 d001 3153 89d2
00000080 f045 c889 35f7 7df0 0000 588d 2901 3bd6
00000090 f075 7573 f7ba 0001 ec00 e083 3cc0 7540
000000a0 baf3 01f2 0000 01b0 baee 01f3 0000 d888
000000b0 89ee bad8 01f4 0000 e8c1 ee08 d889 f5ba
000000c0 0001 c100 10e8 89ee bad8 01f6 0000 e8c1
000000d0 8318 0fe0 c883 eee0 20b0 f7ba 0001 ee00
000000e0 f7ba 0001 ec00 e083 3cc0 7540 8bf3 f00d
000000f0 007d 8900 baf7 01f0 0000 e9c1 fc02 6df2
00000100 3503 7df0 0000 eb43 5886 5e5b 5d5f a1c3
00000110 7df0 0000 3155 89c9 56e5 8d53 c514 0000
00000120 0000 eca1 007d e800 ff46 ffff eca1 007d
00000130 8100 7f38 4c45 7546 0f39 70b7 8b2c 1c58
00000140 c301 e6c1 0105 39de 73f3 8b18 0843 4b8b
00000150 8304 20c3 538b 25f4 ffff 00ff 11e8 ffff
00000160 ebf6 a1e4 7dec 0000 408b 2518 ffff 00ff
00000170 d0ff 00ba ff8a 89ff 66d0 b8ef 8e00 ffff
00000180 ef66 feeb 0014 0000 0000 0000 7a01 0052
00000190 7c01 0108 0c1b 0404 0188 0000 002c 0000
000001a0 001c 0000 fece ffff 009d 0000 4100 080e
000001b0 0285 0d42 4205 0387 0486 8345 0205 c38f
000001c0 c641 c741 c541 040c 0004 0000 001c 0000
000001d0 004c 0000 ff3b ffff 0075 0000 4600 080e
000001e0 0285 0d44 4205 0386 0483 0000 0000 0001
000001f0 0200 0000 0000 0000 0000 0000 0000 aa55
00002000
```

> 上图为主引导扇区的 512 个字节存放的内容，结合之前的 sign.c 可以看到，末尾的两个字节存放的为 0xAA55，实际上即为 0x55AA 的小尾存放方式，高字节存放在高地址。

## 练习 2：使用 qemu 执行并调试 lab1 中的软件

### 1. 从 CPU 加电后执行的第一条指令开始，单步跟踪 BIOS 的执行

```
set architecture i8086
target remote :1234
```

[1] 首先修改 gdbinit 文件如上所示，由于 BIOS 启动过程是从实模式 (16 位模式) 开始的，故需要将此时的架构修改为 i8086 方能正常调试；

```
The target architecture is assumed to be i8086
0x0000ffff in ?? ()
(gdb) x/1i $pc
=> 0xffff:    add    %al,(%bx,%si)
(gdb) x/1i 0xffffffff
0xffffffff:  ljmp    $0xf000,$0xe05b
(gdb) si
0x0000e05b in ?? ()
(gdb) x/10i $pc
=> 0xe05b:    add    %al,(%bx,%si)
    0xe05d:    add    %al,(%bx,%si)
    0xe05f:    add    %al,(%bx,%si)
    0xe061:    add    %al,(%bx,%si)
    0xe063:    add    %al,(%bx,%si)
    0xe065:    add    %al,(%bx,%si)
    0xe067:    add    %al,(%bx,%si)
    0xe069:    add    %al,(%bx,%si)
    0xe06b:    add    %al,(%bx,%si)
    0xe06d:    add    %al,(%bx,%si)
(gdb) si
0x0000e062 in ?? ()
(gdb) x/1i $pc
=> 0xe062:    add    %al,(%bx,%si)

-----
IN:
0xffffffff:  ljmp    $0xf000,$0xe05b

-----
IN:
0x000fe05b:  cmpl    $0x0,%cs:0x6c48
0x000fe062:  jne     0xfd2e1

-----
IN:
0x000fe066:  xor     %dx,%dx
0x000fe068:  mov     %dx,%ss

-----
IN:
0x000fe06a:  mov     $0x7000,%esp

-----
IN:
0x000fe070:  mov     $0xf3691,%edx
0x000fe076:  jmp     0xfd165
```

[2] 修改完成后执行 make debug 进入 gdb 后即可开始调试，由于当前是实模式我

们直接执行 `x/1i $pc` 看到的指令并不是实际执行的指令，所以可以通过执行 `x/1i 0xffffffff` 来看第一条指令，即长跳转指令，后面跳转完成后同样可以查看多行指令，但由长跳转指令可得此处的指令同样不是实际执行的指令。

## 2. 在初始化位置 0x7C00 设置实地址断点，测试断点正常

```
The target architecture is assumed to be i8086
0x0000ffff in ?? ()
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) set architecture i386
The target architecture is assumed to be i386
(gdb) x/10i $pc
=> 0x7c00:      cli
    0x7c01:      cld
    0x7c02:      xor     %eax,%eax
    0x7c04:      mov     %eax,%ds
    0x7c06:      mov     %eax,%es
    0x7c08:      mov     %eax,%ss
    0x7c0a:      in      $0x64,%al
    0x7c0c:      test    $0x2,%al
    0x7c0e:      jne     0x7c0a
    0x7c10:      mov     $0xd1,%al
(gdb) █
```

[1] Report 中给出的方法略显繁琐，可以不通过修改 `gdbinit` 而是直接在 `gdb` 中执行指令的方式来设置断点并设置当前调试的 CPU，同时执行 `x/10i $pc` 也可以看到后面执行的 10 条汇编指令，可在 3 中进行比对。

## 3. 从 0x7C00 开始跟踪代码运行，将单步跟踪反汇编得到的代码与 `bootasm.S` 和 `bootblock.asm` 进行比较

```
# start address should be 0:7c00, in real mode, the beginning address of the running bootloader
.globl start
start:
.code16
cli                                     # Assemble for 16-bit mode
cld                                     # Disable interrupts
                                        # String operations increment

# Set up the important data segment registers (DS, ES, SS).
xorw %ax, %ax                          # Segment number zero
movw %ax, %ds                          # -> Data Segment
movw %ax, %es                          # -> Extra Segment
movw %ax, %ss                          # -> Stack Segment

# Enable A20:
# For backwards compatibility with the earliest PCs, physical
# address line 20 is tied low, so that addresses higher than
# 1MB wrap around to zero by default. This code undoes this.
seta20.1:
inb $0x64, %al                         # Wait for not busy(8042 input buffer empty).
testb $0x2, %al
jnz seta20.1

movb $0xd1, %al                         # 0xd1 -> port 0x64
outb %al, $0x64                        # 0xd1 means: write data to 8042's P2 port
```

```

# start address should be 0:7c00, in real mode, the beginning address of the running bootloader
.globl start
start:
.code16
    cli                    # Assemble for 16-bit mode
    cld                    # Disable interrupts
    cld                    # String operations increment

    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax          # Segment number zero
    movw %ax, %ds          # -> Data Segment
    movw %ax, %es          # -> Extra Segment
    movw %ax, %ss          # -> Stack Segment

    # Enable A20:
    # For backwards compatibility with the earliest PCs, physical
    # address line 20 is tied low, so that addresses higher than
    # 1MB wrap around to zero by default. This code undoes this.
seta20.1:
    inb $0x64, %al         # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.1
    outb %al, $0x64         # 0xd1 means: write data to 8042's P2 port

    movb $0xd1, %al        # 0xd1 -> port 0x64
    outb %al, $0x64

```

[1] 由 2 中的反汇编代码与上面两图中的汇编代码比对，可得二者一致。

#### 4. 自己找一个 bootblock 或内核中的代码位置，设置断点并进行测试

```

(gdb) b *0x00100860
Breakpoint 2 at 0x100860
(gdb) c
Continuing.

Breakpoint 2, 0x00100860 in ?? ()
(gdb) x/10i $pc
=> 0x100860: mov    %edx,%eax
    0x100862: add    %eax,%eax
    0x100864: add    %edx,%eax
    0x100866: shl    $0x2,%eax
    0x100869: mov    %eax,%edx
    0x10086b: mov    -0xc(%ebp),%eax
    0x10086e: add    %edx,%eax
    0x100870: mov    (%eax),%edx
    0x100872: mov    -0x14(%ebp),%eax
    0x100875: add    %eax,%edx
(gdb) 

```

```

100860: 89 d0      mov    %edx,%eax
100862: 01 c0      add    %eax,%eax
100864: 01 d0      add    %edx,%eax
100866: c1 e0 02   shl    $0x2,%eax
100869: 89 c2      mov    %eax,%edx
10086b: 8b 45 f4   mov    -0xc(%ebp),%eax
10086e: 01 d0      add    %edx,%eax
100870: 8b 10      mov    (%eax),%edx
100872: 8b 45 ec   mov    -0x14(%ebp),%eax
100875: 01 c2      add    %eax,%edx

```

[1] 将断点设置在 kernel 中的 0x00100860 的位置并 continue 执行，到达断点后查看代码可得反汇编结果与原汇编指令一致。

### 练习 3：分析 bootloader 进入保护模式的过程

#### 1. 为何开启 A20，以及如何开启 A20

[1] 首先根据实验指导书中 Lab1 的附录 A 可得，A20 的存在是为了保持向下兼容性，一开始时 A20 地址线控制是被屏蔽的，直到系统软件通过一定的 IO 操作打开它，开启之后才能访问高端内存，即保护模式下开关必须开启，且虽然使用的控制

芯片为 8042 键盘控制器，但实际与键盘并无关联。

[2] 由附录 A 进一步可得打开 A20 Gate 的具体步骤大致如下：

- 等待 8042 Input buffer 为空
- 发送 Write 8042 Output Port(P2)命令到 8042 Input buffer
- 等待 8042 Input buffer 为空
- 将 8042 Output Port(P2)得到字节的第 2 位置 1，然后写入 8042 Input buffer

```
# Enable A20:
# For backwards compatibility with the earliest PCs, physical
# address line 20 is tied low, so that addresses higher than
# 1MB wrap around to zero by default. This code undoes this.
seta20.1:
    inb $0x64, %al                # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al               # 0xd1 -> port 0x64
    outb %al, $0x64              # 0xd1 means: write data to 8042's P2 port

seta20.2:
    inb $0x64, %al                # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al              # 0xdf -> port 0x60
    outb %al, $0x60              # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to 1
```

[3] 由 bootasm.S 中的 Enable A20 部分代码可得，开启过程与附录 A 中描述相同。

## 2. 如何初始化 GDT 表

```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to physical addresses, so that the
# effective memory map does not change during the switch.
lgdt gdt_desc
7c1e:      0f 01 16                lgdtl  (%esi)
7c21:      6c                    insb  (%dx),%es:(%edi)
7c22:      7c 0f                jl    7c33 <protcseg+0x1>

# Bootstrap GDT
.p2align 2
gdt:
    SEG_NULLASM                # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg for bootloader and kernel
    SEG_ASM(STA_W, 0x0, 0xffffffff)      # data seg for bootloader and kernel

gdt_desc:
    .word 0x17                 # sizeof(gdt) - 1
    .long gdt                  # address gdt
```

[1] 初始化 GDT 直接通过加载全局描述符的 LGDT 指令来直接完成，解释出的汇编指令如上面第一张图所示，而 gdt\_desc 对应着一个内存地址，由于 GDTR 寄存器为 6 个字节，存放 GDT 表的内存起始地址和表长度，故执行此指令时会把 gdt\_desc 指向的内存地址开始的连续 6 个字节的数据存入 GDTR 寄存器。

## 3. 如何使能和进入保护模式

```
movl %cr0, %eax
7c24:      20 c0                    and    %al,%al
orl $CR0_PE_ON, %eax
7c26:      66 83 c8 01            or     $0x1,%ax
movl %eax, %cr0
7c2a:      0f 22 c0                mov     %eax,%cr0
```

[1] 做好上述准备后，使能并从实模式切换到保护模式并不难，只需要将控制寄存器 CR0 中的 PE 位置 1 即可，置 1 过程如上图指令所示。



```

# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp $PROT_MODE_CSEG, $protcseg

.code32                                # Assemble for 32-bit mode
protcseg:
# Set up the protected-mode data segment registers
movw $PROT_MODE_DSEG, %ax             # Our data segment selector
movw %ax, %ds                         # -> DS: Data Segment
movw %ax, %es                         # -> ES: Extra Segment
movw %ax, %fs                         # -> FS
movw %ax, %gs                         # -> GS
movw %ax, %ss                         # -> SS: Stack Segment

# Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
movl $0x0, %ebp
movl $start, %esp
call bootmain

# If bootmain returns (it shouldn't), loop.
spin:
jmp spin

```

[2] 但执行完成之后，处理器虽然转入了保护模式，但 CS 中的内容仍然为实模式下代码段的段值而非保护模式下代码段的选择子，故取指令前应把代码段的选择子装入 CS，故紧接着会执行如上图所示长跳转指令及其他指令。同时后面还分配了堆栈空间，完成操作后调用 bootmain 将 kernel 加载进内存之中。

## 练习 4：分析 bootloader 加载 ELF 格式的 OS 的过程

### 1. bootloader 如何读取硬盘扇区的

[1] 首先要明确本实验中的 ELF 类型为用于执行的可执行文件(executable file)，用于提供程序的进程映像，加载到内存执行，读取硬盘扇区关联的函数即为 bootmain.c 中的 readsect 函数，而整个加载过程即对应 bootmain 函数，在 2 中会进一步分析。

第6位：为1=LBA模式；0=CHS模式 第7位和第5位必须为1

IO地址	功能
0x1f0	读数据，当0x1f7不为忙状态时，可以读。
0x1f2	要读写的扇区数，每次读写前，你需要表明你要读写几个扇区。最小是1个扇区
0x1f3	如果是LBA模式，就是LBA参数的0-7位
0x1f4	如果是LBA模式，就是LBA参数的8-15位
0x1f5	如果是LBA模式，就是LBA参数的16-23位
0x1f6	第0-3位：如果是LBA模式就是24-27位 第4位：为0主盘；为1从盘
0x1f7	状态和命令寄存器。操作时先给命令，再读取，如果不是忙状态就从0x1f0端口读数据

[2] 同时实验指导书中给出了磁盘 IO 地址和对应功能，如上图所示，也给出了一个扇区的流程如下所示：

- 等待磁盘准备好
- 发出读取扇区的命令
- 等待磁盘准备好
- 把磁盘扇区数据读到制定内存

```

/* readsect - read a single sector at @secno into @dst */
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready
    waitdisk();

    outb(0x1f2, 1);                    // count = 1
    outb(0x1f3, secno & 0xFF);
    outb(0x1f4, (secno >> 8) & 0xFF);
    outb(0x1f5, (secno >> 16) & 0xFF);
    outb(0x1f6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1f7, 0x20);                 // cmd 0x20 - read sectors

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

    // read a sector
    insl(0x1f0, dst, SECTSIZE / 4);
}

```



[3] 可以看到 bootmain.c 中的 readsect 函数如上所示，开启过程与上面的描述完全相符。对 0x1F2 的操作指定了每次读取 1 个扇区，对 0x1F3-0x1F6 的操作共同指定了读取的扇区号，对 0x1F7 的操作中 0x20 的指令用来读取扇区。

## 2. bootloader 是如何加载 ELF 格式的 OS

[1] 分析完读取硬盘扇区后可得，加载整个 kernel 的过程本质上就是循环读取扇区的过程，涉及到的函数即为 bootmain.c 中的 readseg 函数和 bootmain 函数。

```
/* *
 * readseg - read @count bytes at @offset from kernel into virtual address @va,
 * might copy more than asked.
 * */
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;

    // round down to sector boundary
    va -= offset % SECTSIZE;

    // translate from bytes to sectors; kernel starts at sector 1
    uint32_t secno = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of sectors at a time.
    // We'd write more to memory than asked, but it doesn't matter --
    // we load in increasing order.
    for (; va < end_va; va += SECTSIZE, secno++) {
        readsect((void *)va, secno);
    }
}
```

[2] readseg 函数如上所示，需要注意的是 secno 的设定是从 1 开始，原因是 0 扇区对应的是主引导扇区，从 1 扇区开始才是 kernel 部分。

```
/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // read the 1st page off disk
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // load each program segment (ignores ph flags)
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph++) {
        readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
    }

    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();

bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

    // do nothing */
    while (1);
}
```

```

struct elfhdr {
    uint magic; // must equal ELF_MAGIC
    uchar elf[12];
    ushort type;
    ushort machine;
    uint version;
    uint entry; // 程序入口的虚拟地址
    uint phoff; // program header 表的位置偏移
    uint shoff;
    uint flags;
    ushort ehsize;
    ushort phentsize;
    ushort phnum; //program header表中的入口数目
    ushort shentsize;
    ushort shnum;
    ushort shstrndx;
};

```

[3] bootmain 主函数如上所示，首先是读取 ELF 的头部，然后由 ELF 文件头格式要求可得要先比对 magic 是否等于 ELF\_MAGIC 来确认 ELF 合法性，确认后按照头格式中加载位置、入口信息等一系列值来将 ELF 加载进内存之中并找到内核的入口，全部完成后加载过程即结束，转入内核执行。

### 练习 5：实现函数调用堆栈跟踪函数

```

void
print_stackframe(void) {
    /* LAB1 YOUR CODE : STEP 1 */
    /* (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
     * (2) call read_eip() to get the value of eip. the type is (uint32_t);
     * (3) from 0 .. STACKFRAME_DEPTH
     * (3.1) printf value of ebp, eip
     * (3.2) (uint32_t)calling arguments [0..4] = the contents in address (uint32_t)ebp + 2 [0..4]
     * (3.3) cprintf("\n");
     * (3.4) call print_debuginfo(eip-1) to print the C calling function name and line number, etc.
     * (3.5) popup a calling stackframe
     * NOTICE: the calling function's return addr eip = ss:[ebp+4]
     *           the calling function's ebp = ss:[ebp]
     */
    uint32_t ebp = read_ebp(), eip = read_eip();

    int i, j;
    for (i = 0; ebp != 0 && i < STACKFRAME_DEPTH; i++) {
        cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
        uint32_t *args = (uint32_t *)ebp + 2;
        for (j = 0; j < 4; j++) {
            cprintf("0x%08x ", args[j]);
        }
        cprintf("\n");
        print_debuginfo(eip - 1);
        eip = ((uint32_t *)ebp)[1];
        ebp = ((uint32_t *)ebp)[0];
    }
}

```

[1] 首先看 print\_stackframe 函数，即从 0-当前深度将调用的所有的函数全部打印出来，包括 ebp、eip 和参数列表 args。

```

Special kernel symbols:
entry   0x00100000 (phys)
etext   0x0010353b (phys)
edata   0x0010ea16 (phys)
end      0x0010fd80 (phys)
Kernel executable memory footprint: 64KB
ebp:0x00007b38 eip:0x00100a37 args:0x00010094 0x00007b68 0x00100084
    kern/debug/kdebug.c:305: print_stackframe+21
ebp:0x00007b48 eip:0x00100d37 args:0x00000000 0x00000000 0x00007bb8
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b68 eip:0x00100084 args:0x00000000 0x00007b90 0xffff0000 0x00007b94
    kern/init/init.c:48: grade_backtrace2+19
ebp:0x00007b88 eip:0x001000a6 args:0x00000000 0xffff0000 0x00007bb4 0x00000029
    kern/init/init.c:53: grade_backtrace1+27
ebp:0x00007ba8 eip:0x001000c3 args:0x00000000 0x00100000 0xffff0000 0x00100043
    kern/init/init.c:58: grade_backtrace0+19
ebp:0x00007bc8 eip:0x001000e4 args:0x00000000 0x00000000 0x00000000 0x00103540
    kern/init/init.c:63: grade_backtrace+26
ebp:0x00007be8 eip:0x00100050 args:0x00000000 0x00000000 0x00000000 0x00007c4f
    kern/init/init.c:28: kern_init+79
ebp:0x00007bf8 eip:0x00007d72 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknown>: -- 0x00007d71 --

```

[2] 然后执行 make qemu 之后，可以看到关于堆栈调用部分的代码如上所示，每一层调用都会输出 ebp、eip 以及 args 的值，且指明调用函数的位置和关系。

```
// call the entry point from the ELF header
// note: does not return
((void (*)(void))(ELFHDR->e_entry & 0xFFFFFFFF))();
7d63:    a1 ec 7d 00 00    mov     0x7dec,%eax
7d68:    8b 40 18          mov     0x18(%eax),%eax
7d6b:    25 ff ff ff 00    and     $0xffffffff,%eax
7d70:    ff d0            call    *%eax
asm volatile ("outb %0, %1" :: "a" (data), "d" (port));
```

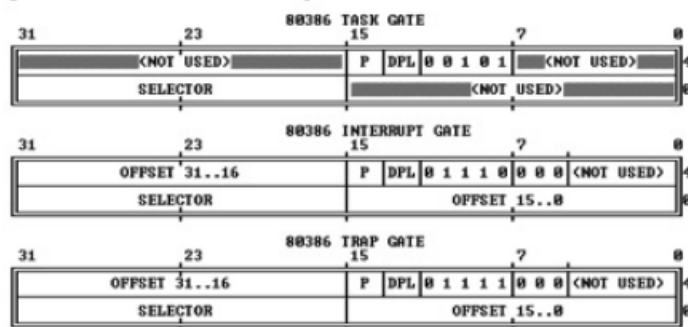
[3] 对于最后一行实际上为最开始的调用即 call bootmain 语句，由于初始化后堆栈为空，栈顶在 0x7C00 位置，故调用后压栈，栈顶指针变为 0x7BF8，即为 ebp 所示值，而 call 语句所在的位置为 0x7D70，如图所示，故 eip 指向下一条会执行的语句即 0x7D72。

## 练习 6：完善中断初始化和处理

### 1. 中断描述符表（保护模式下的中断向量表）中一个表项占多少字节，其中哪几位代表中断处理代码的入口

[1] 由实验指导书中断与异常部分的介绍可得，中断描述符表 IDT 是一个 8 字节的描述符数组，故可得其中每一个表项占 8 字节，CPU 把中断异常号乘 8 作为 IDT 的索引。

Figure 9-3. 80386 IDT Gate Descriptors



[2] 由实验指导书中给出的三种 Descriptor 可得，除去 Task-gate 此处未使用外，剩下两种均可统一成如下形式：

- 2-3 字节为段选择子
- 6-7 字节和 0-1 字节拼接成 4 字节的偏移量
- 4-5 字节用来决定 Descriptor 类型
- 可得除去 4-5 字节外剩余的 6 个字节联合给出中断处理代码的入口

### 2. 请编程完善 kern/trap/trap.c 中对中断向量表进行初始化的函数 idt\_init。在 idt\_init 函数中，依次对所有中断入口进行初始化。使用 mmu.h 中的 SETGATE 宏，填充 idt 数组内容。每个中断的入口由 tools/vectors.c 生成，使用 trap.c 中声明的 vectors 数组即可。

```

/* idt_init - initialize IDT to each of the entry points in kern/trap/vectors.S */
void
idt_init(void) {
    /* LAB1 YOUR CODE : STEP 2 */
    /* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?
     * All ISR's entry addrs are stored in __vectors, where is uintptr_t __vectors[] ?
     * __vectors[] is in kern/trap/vector.S which is produced by tools/vector.c
     * (Try "make" command in lab1, then you will find vector.S in kern/trap DIR)
     * You can use "extern uintptr_t __vectors[];" to define this extern variable which will be used later.
     * (2) Now you should setup the entries of ISR in Interrupt Description Table (IDT).
     * Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE macro to setup each item of IDT
     * (3) After setup the contents of IDT, you will let CPU know where is the IDT by using 'lidt' instruction.
     * You don't know the meaning of this instruction? just google it! and check the libs/x86.h to know more.
     * Notice: the argument of lidt is idt_pd. try to find it!
     */
    extern uintptr_t __vectors[];
    int i;
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    // set for switch from user to kernel
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
    // load the IDT
    lidt(&idt_pd);
}

```

[1] 按照题目要求给出的代码如上图所示，循环对 idt 内所有的中断入口进行初始化，完成后即可通过 LIDT 指令来加载 IDT。

3. 请编程完善 trap.c 中的中断处理函数 trap，在对时钟中断进行处理的部分填写 trap 函数中处理时钟中断的部分，使操作系统每遇到 100 次时钟中断后，调用 print\_ticks 子程序，向屏幕上打印一行文字“100 ticks”。

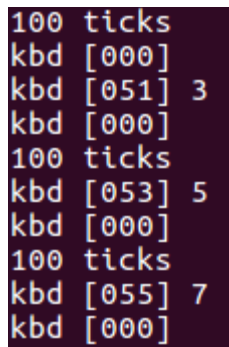
```

static void
trap_dispatch(struct trapframe *tf) {
    char c;

    switch (tf->tf_trapno) {
        case IRQ_OFFSET + IRQ_TIMER:
            /* LAB1 YOUR CODE : STEP 3 */
            /* handle the timer interrupt */
            kern/driver/clock.c
            * (2) Every TICK_NUM cycle, you can print some info using a function, such as print_ticks().
            * (3) Too Simple? Yes, I think so!
            ticks++;
            if (ticks % TICK_NUM == 0) {
                print_ticks();
            }
            break;
        case IRQ_OFFSET + IRQ_COM1:
            c = Cons_getc();
            cprintf("serial [%03d] %c\n", c, c);
            break;
        case IRQ_OFFSET + IRQ_KBD:
            c = Cons_getc();
            cprintf("kbd [%03d] %c\n", c, c);
            break;
    }
}

```

[1] 按照题目要求给出的代码如上图所示，当面临中断时要判断是属于哪一类中断，在实验中容易尝试出的为时钟中断和键盘中断，串口中断因为没有相关设备连接所以没有触发。



```

100 ticks
kbd [000]
kbd [051] 3
kbd [000]
100 ticks
kbd [053] 5
kbd [000]
100 ticks
kbd [055] 7
kbd [000]

```

[2] 触发结果如上图所示，可以看到当没有键盘输入时每隔 100ticks 会自动触发中断，有键盘输入时会立刻触发键盘中断。

### 扩展练习 Challenge1 :

扩展 proj4, 增加 syscall 功能，即增加一用户态函数（可执行一特定系统调用：获得时钟计数值），当内核初始完毕后，可从内核态返回到用户态的函数，而用户态的函数又通过系统调用得到内核态的服务。

```

static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TODO
    asm volatile (
        "sub $0x8, %%esp \n"
        "int %0 \n"
        "movl %%ebp, %%esp"
        :
        : "i"(T_SWITCH_TOU)
    );
}

static void
lab1_switch_to_kernel(void) {
    //LAB1 CHALLENGE 1 : TODO
    asm volatile (
        "int %0 \n"
        "movl %%ebp, %%esp \n"
        :
        : "i"(T_SWITCH_TOK)
    );
}

static void
lab1_switch_test(void) {
    lab1_print_cur_status();
    cprintf("+++ switch to user mode +++\n");
    lab1_switch_to_user();
    lab1_print_cur_status();
    cprintf("+++ switch to kernel mode +++\n");
    lab1_switch_to_kernel();
    lab1_print_cur_status();
}

```

```

0: @ring 0
0:  cs = 8
0:  ds = 10
0:  es = 10
0:  ss = 10
+++ switch to user mode +++
1: @ring 3
1:  cs = 1b
1:  ds = 23
1:  es = 23
1:  ss = 23
+++ switch to kernel mode +++
2: @ring 0
2:  cs = 8
2:  ds = 10
2:  es = 10
2:  ss = 10

```

[1] 内核初始完毕后会执行 lab1\_switch\_test 函数，在其中会发生两次切换，首先输出当前（内核）状态，随后切换到用户态，输出状态后再切换回内核态并输出当前状态。

[2] 此处需要注意的是在 lab1\_switch\_to\_kernel 和 lab1\_switch\_to\_user 这两个函数中采用了内嵌汇编的写法，同时加入了 volatile 限定符确保汇编指令不被优化修改，经查阅资料了解，内嵌汇编中的：是用来分隔不同部分用的，在此处的两个例子中分隔的分别是汇编语句模板和输入部分，同时在模板中用 %0 作为占位符，在输入部分用 "i" 代表立即数将中断号填入 INT 中使其发生中断。

```
//LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
case T_SWITCH_TOU:
    if (tf->tf_cs != USER_CS) {
        switchk2u = *tf;
        switchk2u.tf_cs = USER_CS;
        switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
        switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;

        // set eflags, make sure ucore can use io under user mode.
        // if CPL > IOPL, then cpu will generate a general protection.
        switchk2u.tf_eflags |= FL_IOPL_MASK;

        // set temporary stack
        // then iret will jump to the right stack
        *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
    }
    break;
case T_SWITCH_TOK:
    if (tf->tf_cs != KERNEL_CS) {
        tf->tf_cs = KERNEL_CS;
        tf->tf_ds = tf->tf_es = KERNEL_DS;
        tf->tf_eflags &= ~FL_IOPL_MASK;
        switchu2k = (struct trapframe *) (tf->tf_esp - (sizeof(struct trapframe) - 8));
        memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
        *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
    }
    break;
```

[3] 上图所示即为中断处理切换状态时的代码，两段代码总体上比较对称，都需要在一开始判断是否已经处于目标状态，如果不是目标状态，那么需要修改相应的寄存器，根据 trapframe 的定义，需要修改的寄存器有 CS,DS,ES,ESP 等，同时根据 trapframe 中为补齐 4 字节加入的 padding 也可以验证之前的 55AA 的小尾存储的问题。

[4] 其中比较关键的还有对 EFLAGS 寄存器的修改，EFLAGS 寄存器的第 12/13 位为 IOPL(I/O privilege level field)，指示当前运行任务的 I/O 特权级，只有在运行任务的当前特权级(CPL)小于等于 I/O 特权级才允许访问 I/O 地址空间，而代码中 |=0x3000 和 &=~0x3000 的作用则分别是置 11 和置 00。

## 扩展练习 Challenge2 :

用键盘实现用户模式内核模式切换。具体目标是：“键盘输入 3 时切换到用户模式，键盘输入 0 时切换到内核模式”。基本思路是借鉴软中断(syscall 功能)的代码，并且把 trap.c 中软中断处理的设置语句拿过来。

```
//LAB1 CHALLENGE 2
case IRQ_OFFSET + IRQ_KBD:
    c = cons_getc();
    if (c == 48) {
        printf("kbd [%03d] %c\n", c, c);
        if (tf->tf_cs == KERNEL_CS) {
            tf->tf_cs = KERNEL_CS;
            tf->tf_ds = tf->tf_es = KERNEL_DS;
            tf->tf_eflags &= ~FL_IOPL_MASK;
            switchu2k = (struct trapframe *) (tf->tf_esp - (sizeof(struct trapframe) - 8));
            memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
            *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
            printf("+++ switch to kernel mode +++\n");
            print_trapframe(tf);
        }
    }
    else if (c == 51) {
        printf("kbd [%03d] %c\n", c, c);
        if (tf->tf_cs == USER_CS) {
            switchk2u = *tf;
            switchk2u.tf_cs = USER_CS;
            switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
            switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;
            switchk2u.tf_eflags |= FL_IOPL_MASK;
            *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
            printf("+++ switch to user mode +++\n");
            print_trapframe(tf-1);
        }
    }
    else printf("kbd [%03d] %c\n", c, c);
    break;
```

[1] 代码实现如上所示，实现时采用的方法为考虑键盘中断的分支，但不影响原中断的效果，先输出当前键入字符，如果输入为 0/3 且需要发生模式切换就会按照 T\_SWITCH\_TOU/TOK 的方式来进行切换，实验结果如下图所示。

```

kbd [048] 0
kbd [000]
100 ticks
kbd [051] 3
+++ switch to user mode +++
trapframe at 0x7b40
edi 0x00010094
esi 0x33010094
ebp 0x00007b78
oesp 0x0000000c
ebx 0x00100200
edx 0x00010094
ecx 0x00010094
eax 0x00000000
ds 0x---001e
es 0x---7b8c
fs 0x---2038
gs 0x---7b80
trap 0x00010094 (unknown trap)
err 0x00000000
eip 0x00007b94
cs 0x---000c
flag 0x00007be8 ZF,SF,TF,IF,OF,NT,IOPL=3
esp 0x00102ac4
ss 0x---f920
kbd [000]=51){
100 ticks f("kbd [%03d] %c\n", c, c);
kbd [051] 3
100 ticks f_cs=USER_CS){
100 ticks switchk2u = *tf;
kbd [000] switchk2u.tf_cs = USER_CS;
100 ticks switchk2u.tf_ds = switchk2u.tf_es = sw
kbd [048] 0
+++ switch to kernel mode +++
trapframe at 0x10fd34
edi 0x00000000
esi 0x00010094
ebp 0x00007be8
oesp 0x0010fd54
ebx 0x00010094
edx 0x00103747
ecx 0x00000000
eax 0x00000003
ds 0x---0010
es 0x---0010
fs 0x---0023
gs 0x---0023
trap 0x00000021 Hardware Interrupt
err 0x00000000
eip 0x0010006e
cs 0x---0008
flag 0x00000206 PF,IF,IOPL=0
kbd [000]
100 ticks
100 ticks

```

[2] 可以看出，在不需要状态切换时只是单纯的键盘中断，需要状态切换时会调用 print\_trapframe 打印出各类寄存器的状态，可以通过 EFLAGS 寄存器的值看出状态切换的结果，并且多次切换时可保证仍然正常工作。



## 实验感想：

1. 本实验和以往的实验都不太一样，因为总体实验的难度比较大，所以大部分时间都是以阅读给出的代码和实验结果为主，更多的是去分析和理解，很容易让自己陷入我好像看起来都懂了的样子，但是事实上要深入挖掘实验内容的话还是有很多要认真学习的。因为实验给了很多参考资料，所以在实验过程中也有着非常多的细节，其实很多都已经在实验参考书或者一些地方给出了，比如：

[1] 关于进入保护模式时的 A20 - 操作系统实验指导书 Page 105

[2] 80386 的任务/中断/陷阱门描述符 - 操作系统实验指导书 Page 99

2. 实验过程要有充分的耐心，因为大部分内容都是之前未接触过的，再加上直接接触一个完整的项目文件，不可避免的会产生阅读上的困难，这种时候就要不厌其烦的百度和 google，不管是 OS 还是 xv6 还是 ucore 都有大量前人做过的工作，可以从中学取经验和教训作为自己的收获，比如搜索如下指令：

[1] x/2i \$pc 为 gdb 调试过程中的一条指令，查阅资料可得其中的含义：

- x 为 examine 的缩写
- 2 表示要查看的内存单元个数
- i 指令地址格式
- pc 即 program counter，程序计数器

[2] asm volatile(...:....:....); 为 gcc 在 c 语言中内嵌汇编的方式，具体含义：

- asm 表示后面的代码为内嵌汇编
- volatile 表示编译器不优化代码，后面的指令保留原样
- ()内由三个:分隔四部分分别为汇编语句模板/输出/输入/破坏描述部分
- 汇编语句模板内用%0-%9 可以最多指定 9 个占位符

对于每一条指令都可以深入挖掘学到相关的很多知识，不仅仅在 OS 这门课或者这些实验当中可以用到，更多的是积累解决问题的能力。

3. 实验串联课程较多，可以当做对于过往课程（数据结构、微机原理）的整理复习和对于将来需要补充课程（编译原理、嵌入式）的积累，需要认真学习体会