

uCore实验一报告

黄郭斌 计科1502班 201507010206

一. 内容

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，读者可以了解到：

- 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

二. 目的

- 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。

三. 实验分析及结果

1. 练习一(理解通过make生成执行文件的过程)

列出本实验各练习中对应的OS原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。在此练习中，大家需要通过静态分析代码来了解：

1. 操作系统镜像文件ucore.img是如何一步一步生成的？(需要比较详细地解释Makefile中每一条相关命令和命令参数的含

义，以及说明命令导致的结果)

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？

[练习1.1]

- 生成 `ucore.img` 需要 `kernel` 和 `bootblock`

生成 `ucore.img` 的代码如下：

```
$(UCOREIMG): $(kernel) $(bootblock)
    $(V)dd if=/dev/zero of=$@ count=10000
    $(V)dd if=$(bootblock) of=$@ conv=notrunc
    $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

$(call create_target,ucore.img)123456
```

首先先创建一个大小为10000字节的块儿，然后再将 `bootblock` 拷贝过去。生成 `ucore.img` 需要先生成 `kernel` 和 `bootblock`

- 生成 `kernel`

而生成 `kernel` 的代码如下：

```
$(kernel): tools/kernel.ld
$(kernel): $(KOBJS)
    @echo "bbbbbbbbbbbbbbbbbbbb$(KOBJS)"
    @echo + ld $@
    $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
    @$ (OBJDUMP) -S $@ > $(call asmfile,kernel)
    @$ (OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $(call symfile,kernel)1234567
```

通过 `make V=` 指令得到执行的具体命令如下：

```
ld -m elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/init.o
obj/kern/libs/readline.o obj/kern/libs/stdio.o obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o
obj/kern/debug/panic.o obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/intr.o
obj/kern/driver/picirq.o obj/kern/trap/trap.o obj/kern/trap/trapentry.o obj/kern/trap/vectors.o
obj/kern/mm/pmm.o obj/libs/printfmt.o obj/libs/string.o1
```

然后根据其中可以看到，要生成 `kernel`，需要用GCC编译器将 `kern` 目录下所有的 `.c` 文件全部编译生成的 `.o` 文件的支持。具体如下：

```
obj/kern/init/init.o
obj/kern/libs/readline.o
obj/kern/libs/stdio.o
obj/kern/debug/kdebug.o
obj/kern/debug/kmonitor.o
obj/kern/debug/panic.o
obj/kern/driver/clock.o
obj/kern/driver/console.o
obj/kern/driver/intr.o
```

```
obj/kern/driver/picirq.o
obj/kern/trap/trap.o
obj/kern/trap/trapentry.o
obj/kern/trap/vectors.o
obj/kern/mm/pmm.o
obj/libs/printfmt.o
obj/libs/string.o12345678910111213141516
```

- 生成 bootblock

而生成 bootblock 的代码如下：

```
$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
    @echo "===== $(call toobj,$(bootfiles))"
    @echo + ld @@
    $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)
    @$ (OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
    @$ (OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)
    @$ (call totarget,sign) $(call outfile,bootblock) $(bootblock)1234567
```

同样根据 make V= 指令打印的结果，得到要生成 bootblock，首先需要生成 bootasm.o、bootmain.o、sign，下列代码为生成 bootasm.o、bootmain.o 的代码，由宏定义批量实现了。

```
bootfiles = $(call listf_cc,boot)
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))12
```

而实际的命令在 make V= 指令结果里可以看到。下述是由 bootasm.S 生成 bootasm.o 的具体命令：

```
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o1
```

下述是由 bootmain.c 生成 bootmain.o 的具体命令

```
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o1
```

至于上述命令的具体参数，查阅资料罗列如下：-ggdb 生成可供gdb使用的调试信息 -m32 生成适用于32位环境的代码 -gstabs 生成stabs格式的调试信息 -nostdinc 不使用标准库 -fno-stack-protector 不生成用于检测缓冲区溢出的代码 -Os 为减小代码大小而进行优化 -l

添加搜索头文件的路径 -fno-builtin 不进行builtin函数的优化

下列代码为生成 sign 的代码

```
$(call add_files_host,tools/sign.c,sign,sign)
$(call create_target_host,sign,sign)12
```

下面是生成 sign 具体的命令：

```
gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o
gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign12
```

有了上述的 `bootasm.o`、`bootmain.o`、`sign`。接下来就可以生成 `bootblock` 了，实际命令如下：

```
ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o1
```

参数解释如下：（不重复解释）
-m 模拟为i386上的连接器
-N 设置代码段和数据段均可读写
-e 指定入口
-Ttext 制定代码段开始位置

[练习1.2]

一个被系统认为是符合规范的硬盘主引导扇区的特征有以下几点：
- 磁盘主引导扇区只有512字节
- 磁盘最后两个字节为 `0x55AA`
- 由不超过466字节的启动代码和不超过64字节的硬盘分区表加上两个字节的结束符组成

2. 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。
2. 在初始化位置 `0x7c00` 设置实地址断点,测试断点正常。
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。
4. 自己找一个 bootloader或内核中的代码位置，设置断点并进行测试

首先通过 `make qemu` 指令运行出等待调试的qemu虚拟机，然后再打开一个终端，通过下述命令连接到 `qemu` 虚拟机：

```
gdb
target remote 127.0.0.1:123412
```

进入到调试界面

输入 `si` 命令单步调试，这是另一个终端会打印下一条命令的地址和内容

然后输入 `b*0x7c00` 在初始化位置地址 `0x7c00` 设置上断点

然后输入 `continue` 使之继续运行 这时成功在 `0x7c00` 处停止运行，然后我们查看此处的反汇编代码 对比此时 `bootasm.S` 中的起始代码，发现确实是一样的

这里多次的单步调试就不在截图赘述了。

3. 练习3(分析bootloader进入保护模式的过程)

BIOS将通过读取硬盘主引导扇区到内存，并转跳到对应内存中的位置执行bootloader。请分析bootloader是如何完成从实模式进入保护模式的。

提示：需要阅读小节“保护模式和分段机制”和lab1/boot/bootasm.S源码，了解如何从实模式切换到保护模式，需要了解：

- 为何开启A20，以及如何开启A20
- 如何初始化GDT表

- 如何使能和进入保护模式

首先我们先分析一下 `bootloader` :

关闭中断，将各个段寄存器重置

它先将各个寄存器置0

```
cli                # Disable interrupts
cld                # String operations increment
xorw %ax, %ax      # Segment number zero
movw %ax, %ds      # -> Data Segment
movw %ax, %es      # -> Extra Segment
movw %ax, %ss      # -> Stack Segment123456
```

开启A20

然后就是将A20置1，这里简单解释一下A20，当 A20 地址线控制禁止时，则程序就像在 8086 中运行，1MB 以上的地址是不可访问的。而在保护模式下 A20 地址线控制是要打开的，所以需要通过将键盘控制器上的A20线置于高电平，使得全部32条地址线可用。

```
seta20.1:
    inb $0x64, %al    # 读取状态寄存器,等待8042键盘控制器闲置
    testb $0x2, %al   # 判断输入缓存是否为空
    jnz seta20.1

    movb $0xd1, %al    # 0xd1表示写输出端口命令,参数随后通过0x60端口写入
    outb %al, $0x64

seta20.2:
    inb $0x64, %al
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al    # 通过0x60写入数据11011111 即将A20置1
    outb %al, $0x60
```

加载 GDT 表

```
lgdt gtdesc1
```

将 CR0 的第0位置1

```
movl %cr0, %eax
orl $CR0_PE_ON, %eax
movl %eax, %cr0
1234
```

长跳转到32位代码段，重装CS和EIP

```
ljmp $PROT_MODE_CSEG, $protcseg1
```

重装DS、ES等段寄存器等

```
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 Segment123456
```

转到保护模式完成，进入boot主方法

```
movl $0x0, %ebp
movl $start, %esp
call bootmain123
```

4. 练习4(分析bootloader加载ELF格式的OS的过程)

分析bootloader加载ELF格式的OS的过程 \1. bootloader如何读取硬盘扇区的？ \2. bootloader是如何加载ELF格式的 OS？ 这里主要分析是 `bootmain` 函数，

```
bootmain(void) {
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }
    struct proghdr *ph, *eph;
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph++) {
        readseg(ph->p_va & 0xFFFFF, ph->p_memsz, ph->p_offset);
    }
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFF))();
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    while (1);
}
```

bootloader读取硬盘扇区

根据上述 `bootmain` 函数分析，首先是由 `readseg` 函数读取硬盘扇区，而 `readseg` 函数则循环调用了真正读取硬盘扇区的函数 `readsect` 来每次读出一个扇区，如下，详细的解释看代码中的注释：

```

readsect(void *dst, uint32_t secno) {
    waitdisk(); // 等待硬盘就绪
    // 写地址0x1f2~0x1f5,0x1f7,发出读取磁盘的命令
    outb(0x1F2, 1);
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20);
    waitdisk();
    insl(0x1F0, dst, SECTSIZE / 4); // 读取一个扇区
}

```

bootloader加载 ELF 格式的 OS

读取完磁盘之后，开始加载 ELF 格式的文件。详细的解释看代码中的注释。

```

bootmain(void) {
    .....
    // 首先判断是不是ELF
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }
    struct proghdr *ph, *eph;

    // ELF头部有描述ELF文件应加载到内存什么位置的描述表，这里读取出来将之存入ph
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;

    // 按照程序头表的描述，将ELF文件中的数据载入内存
    for (; ph < eph; ph++) {
        readseg(ph->p_va & 0xFFFFFFFF, ph->p_memsz, ph->p_offset);
    }
    // 根据ELF头表中的入口信息，找到内核的入口并开始运行
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFFFF))();
bad:
    .....
}

```

5. 练习5(实现函数调用堆栈跟踪函数)

完成 `kdebug.c` 中函数 `print_stackframe` 的实现，可以通过函数 `> print_stackframe` 来跟踪函数调用堆栈中记录的返回地址。

函数堆栈的原理

理解函数堆栈最重要的两点是：栈的结构，以及 `EBP` 寄存器的作用。

一个函数调用动作可分解为零到多个 `PUSH` 指令（用于参数入栈）和一个 `CALL` 指令。`CALL` 指令内部其实还暗含了一个将返回地址压栈的动作，这是由硬件完成的。几乎所有本地编译器都会在每个函数体之前插入类似如下的汇编指令：

```
pushl %ebp
movl %esp,%ebp12
```

这两条汇编指令的含义是：首先将 `ebp` 寄存器入栈，然后将栈顶指针 `esp` 赋值给 `ebp`。 `movl %esp %ebp` 这条指令表面上看是用 `esp` 覆盖 `ebp` 原来的值，其实不然。因为给 `ebp` 赋值之前，原 `ebp` 值已经被压栈（位于栈顶），而新的 `ebp` 又恰恰指向栈顶。此时 `ebp` 寄存器就已经处于一个非常重要的地位，该寄存器中存储着栈中的一个地址（原 `ebp` 入栈后的栈顶），从该地址为基准，向上（栈底方向）能获取返回地址、参数值，向下（栈顶方向）能获取函数局部变量值，而该地址处又存储着上一层函数调用时的 `ebp` 值。

现在做一下更完整的解释：

函数调用大概包括以下几个步骤：- 1、参数入栈：将参数从右向左（或从右向左）依次压入系统栈中。- 2、返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。- 3、代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。- 4、栈帧调整 - 4.1保存当前栈帧状态值，已备后面恢复本栈帧时使用（`EBP` 入栈）。- 4.2将当前栈帧切换到新栈帧（将 `ESP` 值装入 `EBP`，更新栈帧底部）。- 4.3给新栈帧分配空间（把 `ESP` 减去所需空间的大小，抬高栈顶）。

而函数返回大概包括以下几个步骤：- 1、保存返回值，通常将函数的返回值保存在寄存器 `EAX` 中。- 2、弹出当前帧，恢复上一个栈帧。- 2.1在堆栈平衡的基础上，给 `ESP` 加上栈帧的大小，降低栈顶，回收当前栈帧的空间 - 2.2将当前栈帧底部保存的前栈帧 `EBP` 值弹入 `EBP` 寄存器，恢复出上一个栈帧。- 2.3将函数返回地址弹给 `EIP` 寄存器。- 3、跳转：按照函数返回地址跳回母函数中继续执行。

而由此我们可以直接根据 `ebp` 就能读取到各个栈帧的地址和值，一般而言，`ss:[ebp+4]` 处为返回地址，`ss:[ebp+8]` 处为第一个参数值（最后一个入栈的参数值，此处假设其占用 4 字节内存，对应 32 位系统），`ss:[ebp-4]` 处为第一个局部变量，`ss:[ebp]` 处为上一层 `ebp` 值。

print_stackframe 函数的实现

首先我们直接看到 `print_stackframe` 函数的注释：

```
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 funciton's return addr eip = ss:[ebp+4]
     *           the calling funciton's ebp = ss:[ebp]
     */
}
```

这样我们直接根据注释以及之前的相关知识就能比较简单的编写成程序，如下所示：

```
void print_stackframe(void) {

    uint32_t ebp=read_ebp();//(1) call read_ebp() to get the value of ebp. the type is
```



```

(uint32_t)
uint32_t eip=read_eip();//(2) call read_eip() to get the value of eip. the type is
(uint32_t)
int i;
for(i=0;i<STACKFRAME_DEPTH&&ebp!=0;i++){//(3) from 0 .. STACKFRAME_DEPTH
    printf("ebp:0x%08x   eip:0x%08x ",ebp,eip);//(3.1)printf value of ebp, eip
    uint32_t *tmp=(uint32_t *)ebp+2;
    printf("arg :0x%08x 0x%08x 0x%08x 0x%08x",*(tmp+0),*(tmp+1),*(tmp+2),*
    (tmp+3));//(3.2)(uint32_t)calling arguments [0..4] = the contents in address (uint32_t)ebp +2
    [0..4]

    printf("\n");//(3.3) printf("\n");
    print_debuginfo(eip-1);//(3.4) call print_debuginfo(eip-1) to print the C calling
    function name and line number, etc.
    eip=((uint32_t *)ebp)[1];
    ebp=((uint32_t *)ebp)[0];//(3.5) popup a calling stackframe
}
}

```

6. 练习6(完善中断初始化和处理)

1.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？ 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。注意除了系统调用中断 (`T_SYSCALL`) 以外，其它中断均使用中断门描述符，权限为内核态权限；而系统调用中断使用异常门描述符。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。 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`。

[练习6.1]

中断描述符表一个表项占8字节。其中0~15位和48~63位分别为 `offset` 的低16位和高16位。16~31位为段选择子。通过段选择子获得段基址，加上段内偏移量即可得到中断处理代码的入口。

[练习6.2]

这里这里主要就是实现对中断向量表的初始化。注释如下：

```

void idt_init(void) {
    /* LAB1 YOUR CODE : STEP 2 */
    /* (1) Where are the entry adrrs of each Interrupt Service Routine (ISR)?
     *   All ISR's entry adrrs 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!
    */
}

```

重点就是两步 第一步，声明__vectors[],其中存放着中断服务程序的入口地址。这个数组生成于vector.S中。 第二步，填充中断描述符表IDT。 第三部，加载中断描述符表。 对应到代码中如下所示：

```

void idt_init(void) {
    extern uintptr_t __vectors[]; //声明__vectors[]
    int i;
    for(i=0; i<256; i++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
    lidt(&idt_pd); //使用lidt指令加载中断描述符表
}

```

这里的 SETGATE 在 mmu.h 中有定义，

```

#define SETGATE(gate, istrap, sel, off, dpl) 1

```

简单解释一下参数

gate：为相应的 idt[] 数组内容，处理函数的入口地址 istrap：系统段设置为1，中断门设置为0
sel：段选择子 off：为 __vectors[] 数组内容 dpl：设置特权级。这里中断都设置为内核级，即第0级

[练习6.3]

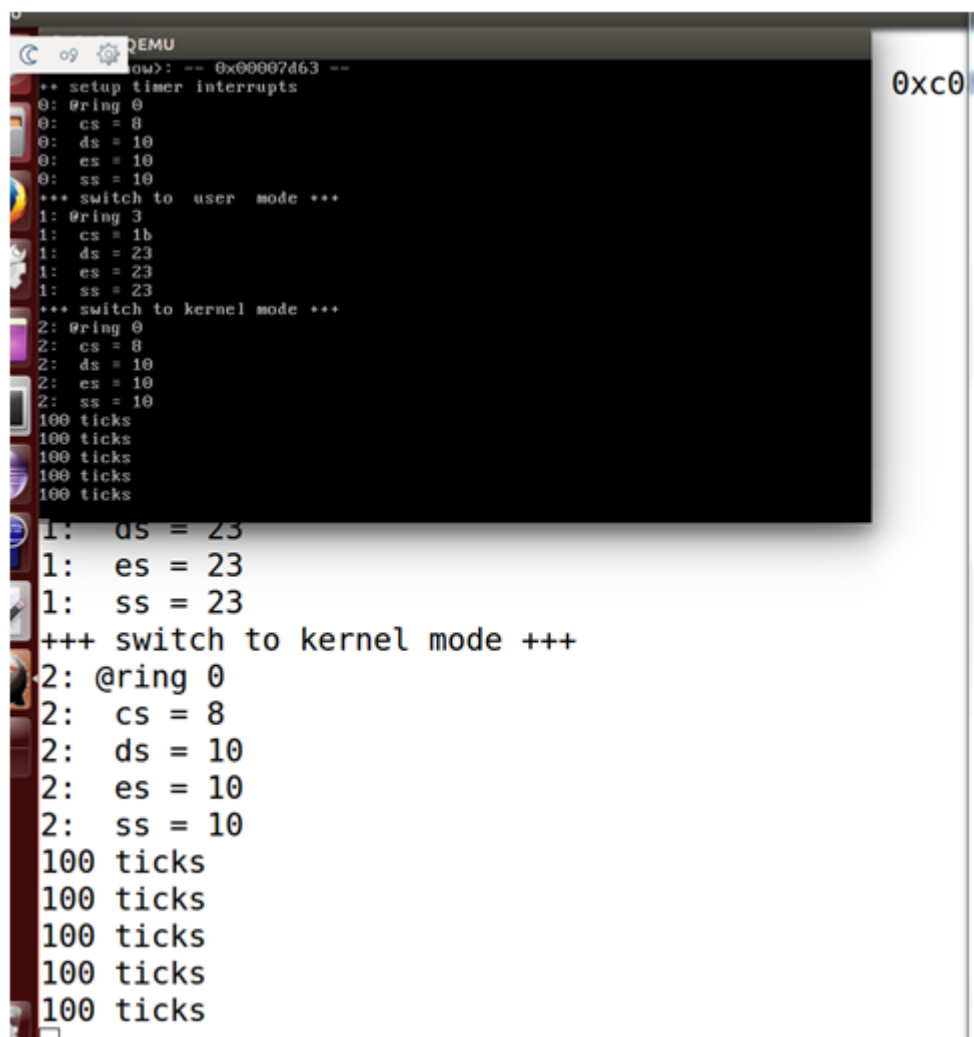
这里根据指导书查看函数 trap_dispatch，发现 print_ticks() 子程序已经被实现了，所以我们直接进行判断输出即可，如下（见注释）：

```

.....
.....
case IRQ_OFFSET + IRQ_TIMER:
    ticks++; //每一次时钟信号会使变量ticks加1
    if (ticks==TICK_NUM) { //TICK_NUM已经被预定义成了100，每到100便调用print_ticks()函数打印
        ticks-=TICK_NUM;
        print_ticks();
    }
    break;
.....
.....

```

7. 实验结果

A screenshot of a debugger window titled 'DEMU'. The main pane shows assembly code with comments and register values. The code includes instructions for setting up timer interrupts, switching between user and kernel modes, and executing a series of 'ticks'. The registers (CS, DS, ES, SS) are shown for different instructions. A right-hand pane displays the value '0xc0'.

```
ow>: -- 0x00007d63 --
** setup timer interrupts
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
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
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
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

四. 实验体会

通过本次实验，让我了解了bootloader是什么，系统是如何启动的，OS呼叫IO设备是通过中断实现的，已经中断的跳转和中断处理程序，IDT是什么以及如何初始化它们，如何使用它们等内容。因为是第一次读操作系统的源码，Linux环境已经有些忘了，环境也装了很久，加上对这个实验要干什么就理解了很久，导致这个实验做了很久，也耽搁了后面的实验。

不过总的太说，做完这个实验的收获还是很大的，通过这次实验，我不仅了解了关于实验方面的内容，也熟悉了Linux环境和一些相关工具（如源码阅读工具sourceInsight，Linux shell，包括我现在写报告使用的Markdown（也是实验要求中需要用的）等）。