

实验一：系统软件启动过程

计31 张正 2013011418

一、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，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、实验内容

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？(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)

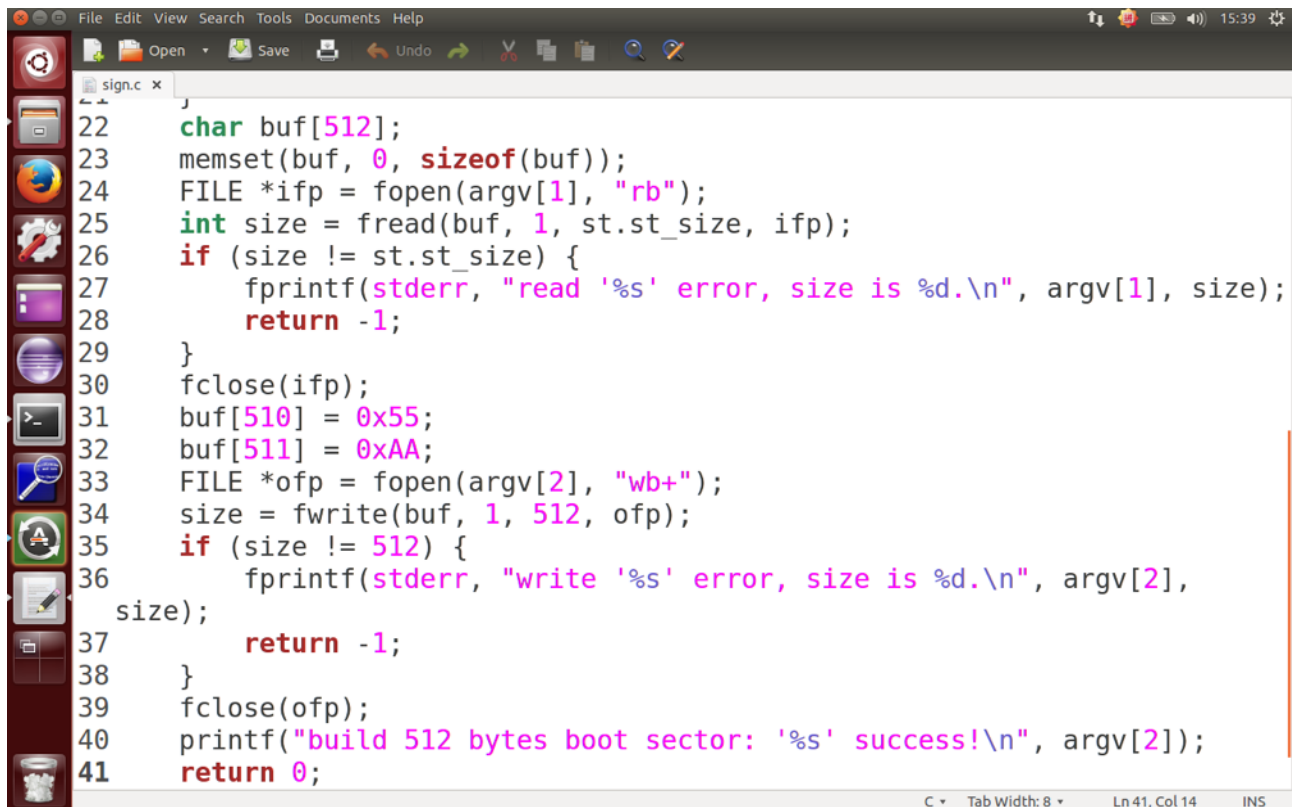
在命令行中输入“make V=”

```
File Edit View Search Terminal Help
+ cc kern/mm/pmm.c
gcc -Ikern/mm/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/mm/pmm.c -o obj/kern/mm/pmm.o
+ cc libs/printfmt.c
gcc -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -c libs/printfmt.c -o obj/libs/printfmt.o
+ cc libs/string.c
gcc -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -c libs/string.c -o obj/libs/string.o
+ ld bin/kernel
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.o
+ cc boot/bootasm.c
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.c -o obj/boot/bootasm.o
+ cc boot/bootmain.c
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o
+ cc tools/sign.c
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/sign
+ ld bin/bootblock
ld -m elf_i386 -nostdlib -N -e start -i text 0x1000 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o
'obj/bootblock.out' size: 472 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
512000 bytes (512 KB) copied, 0.0034209 s, 61.4 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.00011664 s, 4.6 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
138+1 records in
138+1 records out
70775 bytes (71 KB) copied, 0.00742163 s, 9.5 MB/s
[~/moococ/ucore_lab/labcodes/lab1]
moococ-->
```

- 首先把C的源代码进行编译成为.o文件，也就是目标文件（红色方框内）
- ld命令将这些目标文件转变成可执行文件，比如此处的bootblock.out（蓝色方框内）
- dd命令把bootloader放到ucore.img count的虚拟硬盘之中
- 还生成了两个软件，一个是Bootloader，另一个是kernel。（紫色方框内）

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？

在/lab1/tools/sign.c中我们可以了解到



```
22 char buf[512];
23 memset(buf, 0, sizeof(buf));
24 FILE *ifp = fopen(argv[1], "rb");
25 int size = fread(buf, 1, st.st_size, ifp);
26 if (size != st.st_size) {
27     fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
28     return -1;
29 }
30 fclose(ifp);
31 buf[510] = 0x55;
32 buf[511] = 0xAA;
33 FILE *ofp = fopen(argv[2], "wb+");
34 size = fwrite(buf, 1, 512, ofp);
35 if (size != 512) {
36     fprintf(stderr, "write '%s' error, size is %d.\n", argv[2],
size);
37     return -1;
38 }
39 fclose(ofp);
40 printf("build 512 bytes boot sector: '%s' success!\n", argv[2]);
41 return 0;
```

规范的硬盘引导扇区的大小为512字节，硬盘结束标志位55AA

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。

改写Makefile文件

lab1-mon: \$(UCOREIMG)

\$(V)\$(TERMINAL) -e "\$(QEMU) -S -s -d in_asm -D \$(BINDIR)/q.log -monitor

stdio -hda \$< -serial null"

\$(V)sleep 2

\$(V)\$(TERMINAL) -e "gdb -q -x tools/lab1init"

在调用qemu时增加-d in_asm -D q.log参数，便可以将运行的汇编指令保存在q.log中。

在tools中增加lab1init，内容如下

file bin/kernel

target remote :1234

set architecture i8086

b *0x7c00

continue

x /2i \$pc

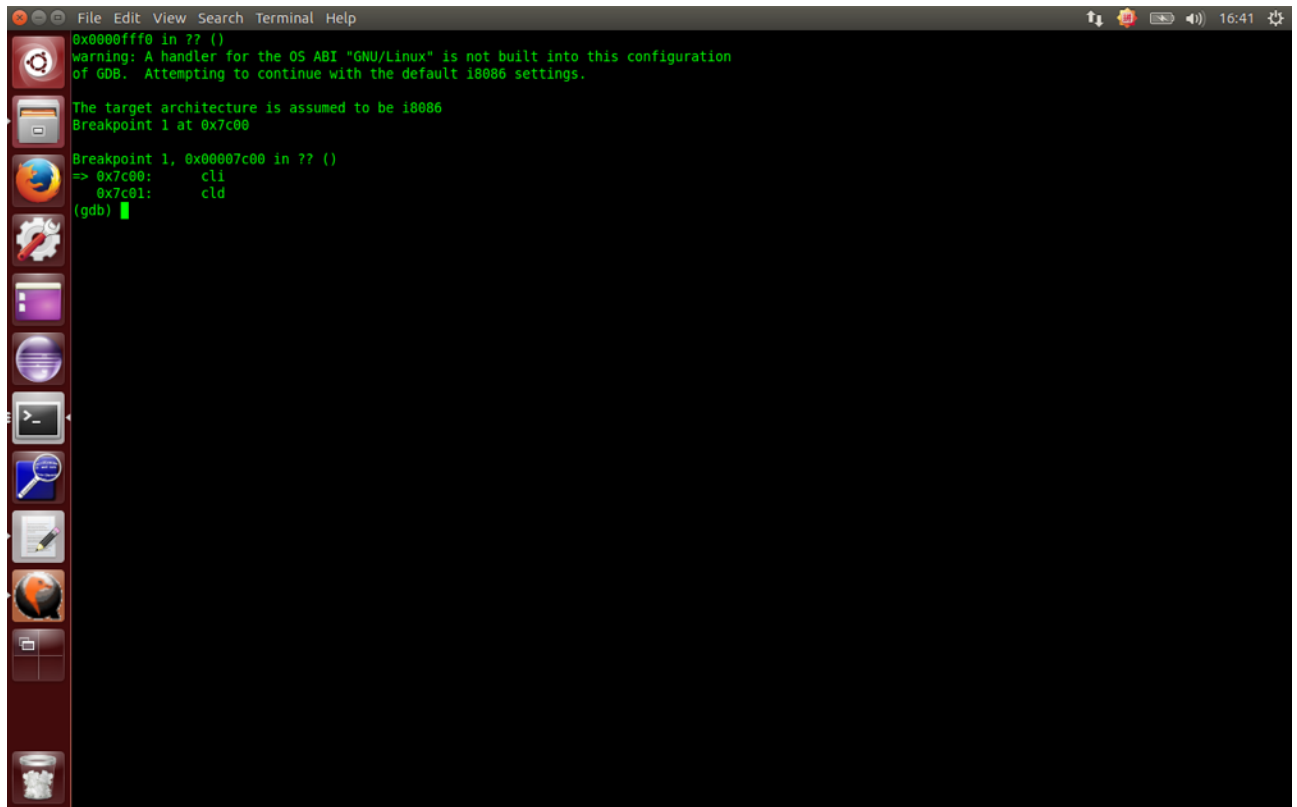
2 在初始化位置0x7c00设置实地址断点,测试断点正常。

语句：在lab1init中

b *0x7c00

continue

x /2i \$pc



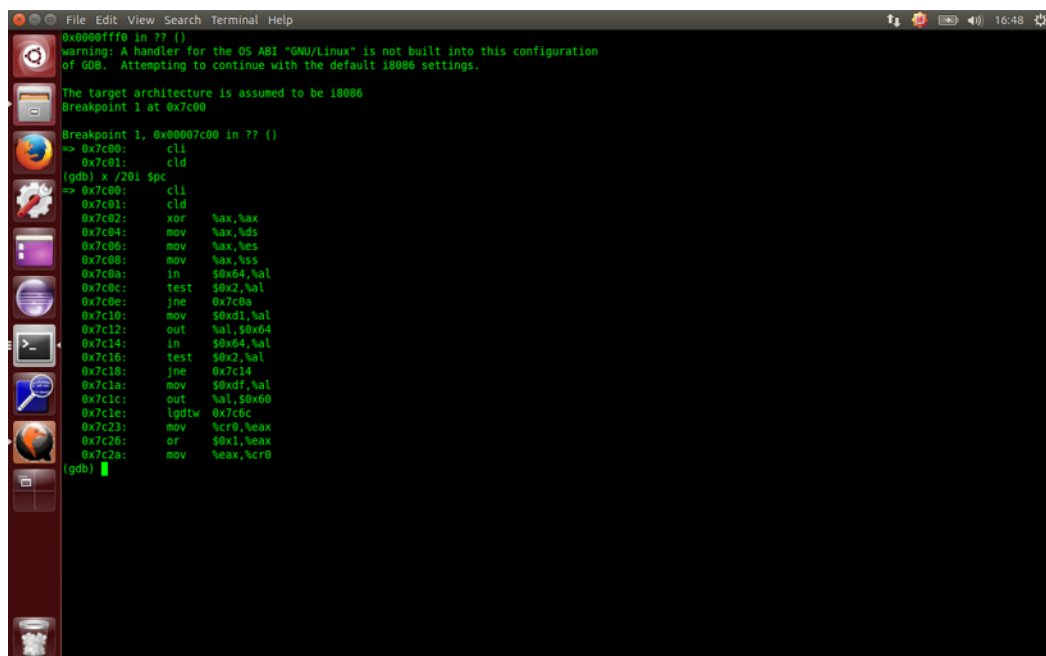
```
File Edit View Search Terminal Help
0x00000000 in ?? ()
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
Breakpoint 1 at 0x7c00

Breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00:  cli
    0x7c01:  cld
(gdb)
```

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。

下图是反汇编得到的代码

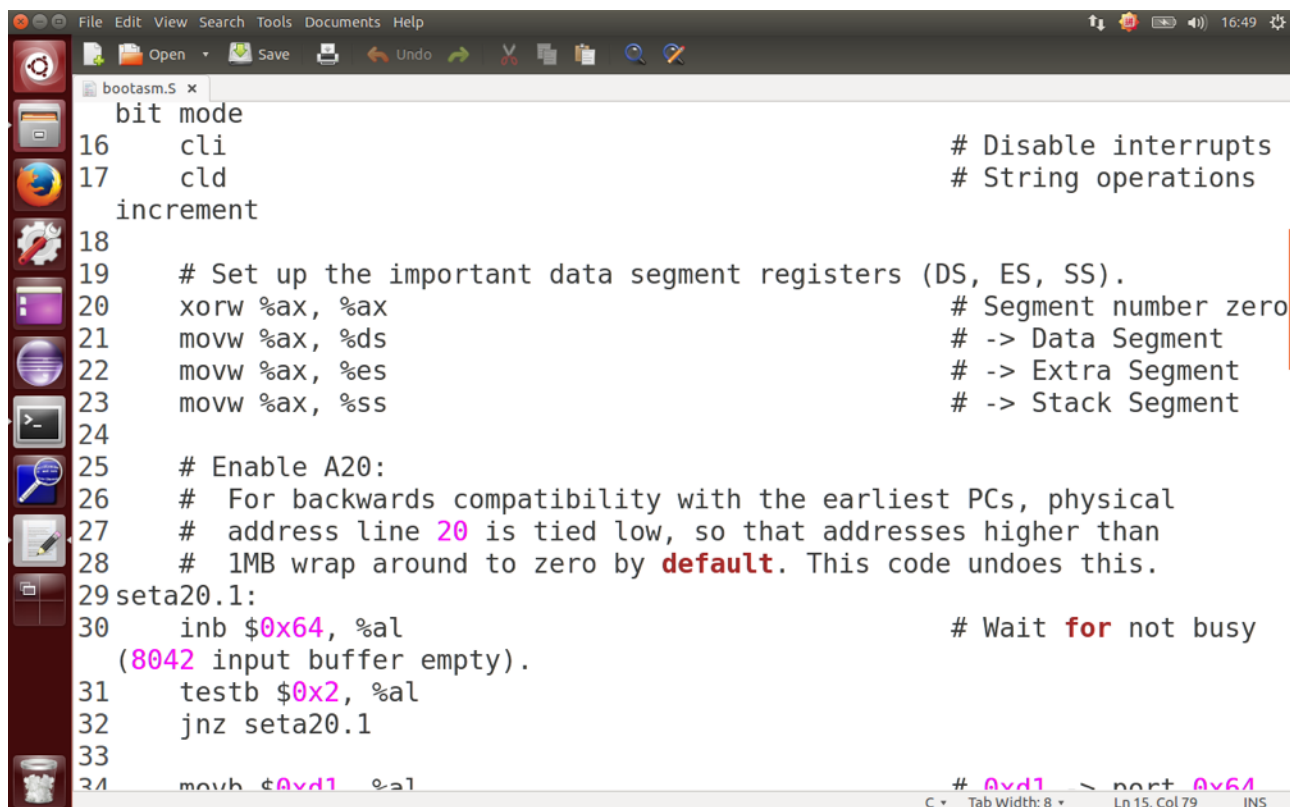


```
File Edit View Search Terminal Help
0x00000000 in ?? ()
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
Breakpoint 1 at 0x7c00

Breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00:  cli
    0x7c01:  cld
(gdb) x /20i $pc
=> 0x7c00:  cli
    0x7c01:  cld
    0x7c02:  xor    %ax,%ax
    0x7c04:  mov    %ax,%ds
    0x7c06:  mov    %ax,%es
    0x7c08:  mov    %ax,%ss
    0x7c0a:  in     $0x64,%al
    0x7c0c:  test   $0x2,%al
    0x7c0e:  jne    0x7c0a
    0x7c10:  mov    $0xd1,%al
    0x7c12:  out    %al,$0x64
    0x7c14:  in     $0x64,%al
    0x7c16:  test   $0x2,%al
    0x7c18:  jne    0x7c14
    0x7c1a:  mov    $0xdf,%al
    0x7c1c:  out    %al,$0x60
    0x7c1e:  lgdtw  0x7c6c
    0x7c20:  mov    %cr0,%eax
    0x7c22:  or     $0x1,%eax
    0x7c24:  mov    %eax,%cr0
(gdb)
```

下图是bootasm.S的代码



```
bootasm.S
bit mode
16 cli # Disable interrupts
17 cld # String operations
increment
18
19 # Set up the important data segment registers (DS, ES, SS).
20 xorw %ax, %ax # Segment number zero
21 movw %ax, %ds # -> Data Segment
22 movw %ax, %es # -> Extra Segment
23 movw %ax, %ss # -> Stack Segment
24
25 # Enable A20:
26 # For backwards compatibility with the earliest PCs, physical
27 # address line 20 is tied low, so that addresses higher than
28 # 1MB wrap around to zero by default. This code undoes this.
29 seta20.1:
30 inb $0x64, %al # Wait for not busy
    (8042 input buffer empty).
31 testb $0x2, %al
32 jnz seta20.1
33
34 movb $0xd1, %al # 0xd1 -> port 0x64
```

可以看到虚拟机中运行的汇编代码与bootasm.S和bootblock.asm中的代码相同。

- 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。
同上只需要改变断点的位置即可

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。

通过阅读/lab1/boot/bootasm.S

```
.globl start
```

```
start:
```

```
.code16
```

```
# 关中断,并清除方向标志,即将 DF 置“0”,这样(E)SI 及(E)DI 的修改为增量
```

```
cli
```

```
cld
```

```
# 清零各数据段寄存器:DS、ES、FS
```

```
xorw %ax, %ax
```

```
movw %ax, %ds
movw %ax, %es
movw %ax, %ss
```

使能 A20 地址线,这样 80386 就可以突破 1MB 访存现在,而可访问 4GB 的 32 位地址空间

seta20.1:

```
inb $0x64, %al          # 等待8042键盘控制器不忙
testb $0x2, %al
jnz seta20.1
```

```
movb $0xd1, %al
outb %al, $0x64
```

seta20.2:

```
inb $0x64, %al          # 等待8042键盘控制器不忙
testb $0x2, %al
jnz seta20.2
```

```
movb $0xdf, %al         # 打开A20
outb %al, $0x60
```

初始化gdt

lgdt gtdesc

进入保护模式

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

长跳转

```
ljmp $PROT_MODE_CSEG, $protcseg
```

.code32

protcseg:

设置段寄存器, 并建立堆栈

```
movw $PROT_MODE_DSEG, %ax
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

# 设置堆栈
movl $0x0, %ebp
movl $start, %esp      # 栈顶为0x7c00
# 进入bootmain, 不再返回
call bootmain
spin:
jmp spin

```

练习四 分析bootloader加载ELF格式的OS的过程

通过阅读bootmain.c, 了解bootloader如何加载ELF文件。通过分析源代码和通过qemu来运行并调试bootloader&OS,

- bootloader如何读取硬盘扇区的?
使用outb机器指令, 其实现是使用内联汇编, 采取IO空间的寻址方式
- bootloader是如何加载ELF格式的OS?

读一个扇区的流程可参看bootmain.c 中的 readsect 函数实现。大致如下:

1. 读 I/O 地址 0x1f7,等待磁盘准备好;
2. 写 I/O 地址 0x1f2~0x1f5,0x1f7,发出读取第 offseet 个扇区处的磁盘数据的命令;

3. 读 I/O 地址 0x1f7,等待磁盘准备好;
4. 连续读 I/O 地址 0x1f0,把磁盘扇区数据读到指定内存。

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);
}

```

最后在bootmain函数中完成加载ELF格式os的操作：

- 1: 读取ELF的头部
- 2: 判断ELF文件是否是合法
- 3: 将描述表的头地址存在ph
- 4: 按照描述表将ELF文件中数据载入内存
- 5: 根据ELF头部储存的入口信息，找到内核的入口(不再返回)

练习五 实现函数调用堆栈跟踪函数

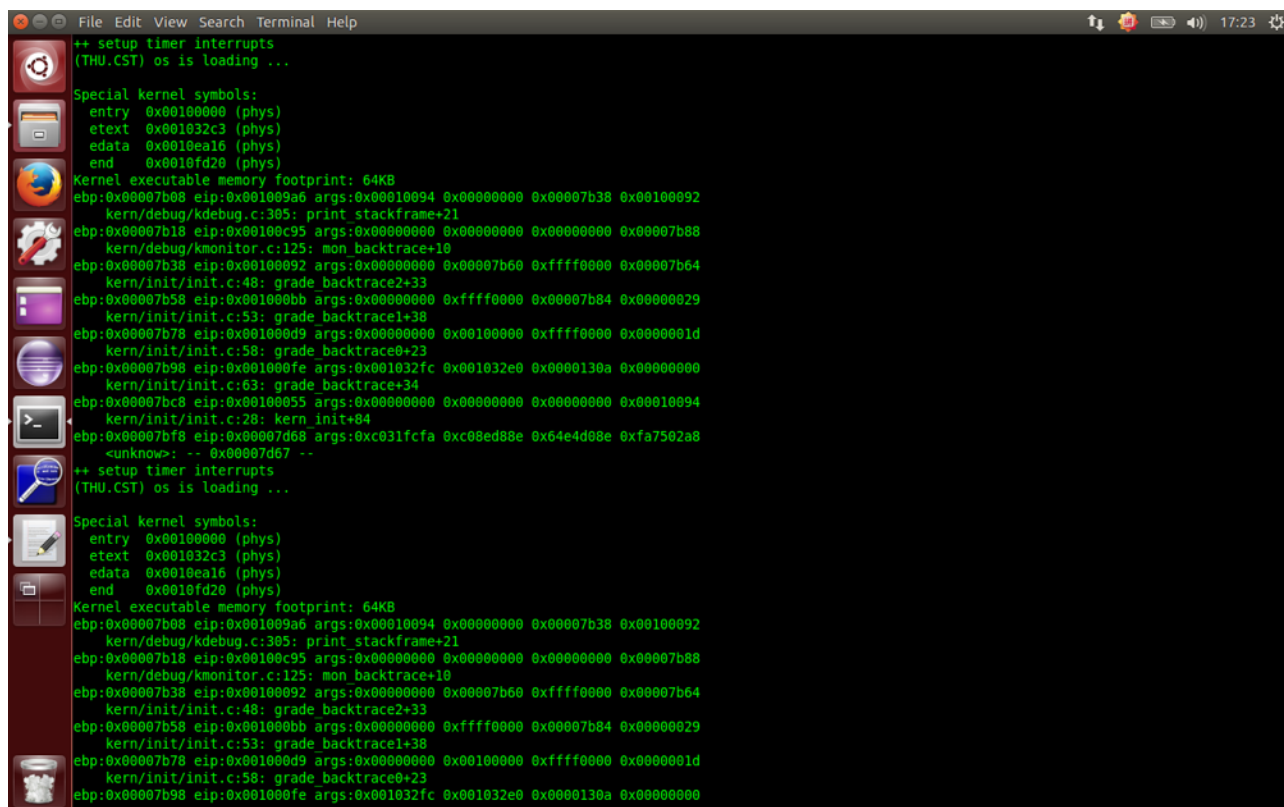
我们需要在lab1中完成kdebug.c中函数print_stackframe的实现，可以通过函数print_stackframe来跟踪函数调用堆栈中记录的返回地址。

```

void
print_stackframe(void) {
    /* LAB1 2013011418 : STEP 1*/
    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;
        //(uint32_t)calling arguments [0..4] = the contents in address
        (uint32_t)ebp + 2 [0..4]
        for (j = 0; j < 4; j++) {
            cprintf("0x%08x ", args[j]);
        }
        cprintf("\n");
        print_debuginfo(eip - 1);
        /*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];
    }
}

```


运行结果



```
File Edit View Search Terminal Help
++ setup timer interrupts
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x001032c3 (phys)
  edata  0x0010ea16 (phys)
  end    0x0010fd20 (phys)
Kernel executable memory footprint: 64KB
ebp:0x00007b08 eip:0x001009a6 args:0x00010094 0x00000000 0x00007b38 0x00100092
  kern/debug/kdebug.c:305: print_stackframe+21
ebp:0x00007b18 eip:0x00100c95 args:0x00000000 0x00000000 0x00000000 0x00007b88
  kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b38 eip:0x00100092 args:0x00000000 0x00007b60 0xffff0000 0x00007b64
  kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b58 eip:0x001000bb args:0x00000000 0xffff0000 0x00007b84 0x00000029
  kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 args:0x00000000 0x00100000 0xffff0000 0x0000001d
  kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe args:0x001032fc 0x001032e0 0x0000130a 0x00000000
  kern/init/init.c:63: grade_backtrace+34
ebp:0x00007bc8 eip:0x00100055 args:0x00000000 0x00000000 0x00000000 0x00010094
  kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
  <unknown>: -- 0x00007d67 --
++ setup timer interrupts
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x001032c3 (phys)
  edata  0x0010ea16 (phys)
  end    0x0010fd20 (phys)
Kernel executable memory footprint: 64KB
ebp:0x00007b08 eip:0x001009a6 args:0x00010094 0x00000000 0x00007b38 0x00100092
  kern/debug/kdebug.c:305: print_stackframe+21
ebp:0x00007b18 eip:0x00100c95 args:0x00000000 0x00000000 0x00000000 0x00007b88
  kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b38 eip:0x00100092 args:0x00000000 0x00007b60 0xffff0000 0x00007b64
  kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b58 eip:0x001000bb args:0x00000000 0xffff0000 0x00007b84 0x00000029
  kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 args:0x00000000 0x00100000 0xffff0000 0x0000001d
  kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe args:0x001032fc 0x001032e0 0x0000130a 0x00000000
```

可以看出得到的输出与正确输出一致

输出中，堆栈最深一层为

ebp:0x00007bf8 eip:0x00007d68 args:0x00000000 0x00000000 0x00000000
0x00007c4f

<unknown>: -- 0x00007d67 --

其对应的是第一个使用堆栈的函数，bootmain.c中的bootmain。

bootloader设置的堆栈从0x7c00开始，使用"call bootmain"转入bootmain函数。

call指令压栈，所以bootmain中ebp为0x7bf8。

练习六 完善中断初始化和处理

1 中断描述符表（也可简称为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？

中断向量表一个表项占用8字节，其中2-3字节是段选择子，0-1字节和6-7字节拼成位移，入口地址=段选择子+段内偏移量。

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。

可以在/lab1/kern/mm/mmu.h中可以找到SETGATE函数，查找其具体操作。

```

void
idt_init(void) {
    /* LAB1 YOUR CODE : STEP 2 */
    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); //设置IDT
    }
    lidt(&idt_pd); //载入IDT表
}

```

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”。

在时钟中断部分添加如下代码

```

ticks++; //一次中断累加1
if (ticks % TICK_NUM == 0) {
    print_ticks();
}

```

可完成实验

拓展实验

内核态切换到用

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)
    );
}

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;
}

```

用户态切换到内核态:

```

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

```

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;
    }

```