Ucore lab1实验报告

姓名： 李胜宇

学号：404011

【练习1】

理解通过 make 生成执行文件的过程。（要求在报告中写出对下述问题的回答）

在此练习中,大家需要通过阅读代码来了解:

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】

1. 生成ucore.img需要kernel和bootblock

查看makefile文件出现下列代码，生成ucore.img需要先生成kernel和

bottblock。

# create ucore.img

UCOREIMG := $(call totarget,ucore.img)

$(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)

# >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

1. 生成kernel和bootblock

# create kernel target

kernel = $(call totarget,kernel)

$(kernel): tools/kernel.ld

$(kernel): $(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)

$(call create\_target,kernel)

# -------------------------------------------------------------------

# create bootblock

bootfiles = $(call listf\_cc,boot)

$(foreach f,$(bootfiles),$(call cc\_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))

bootblock = $(call totarget,bootblock)

$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)

@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)

$(call create\_target,bootblock)

# -------------------------------------------------------------------

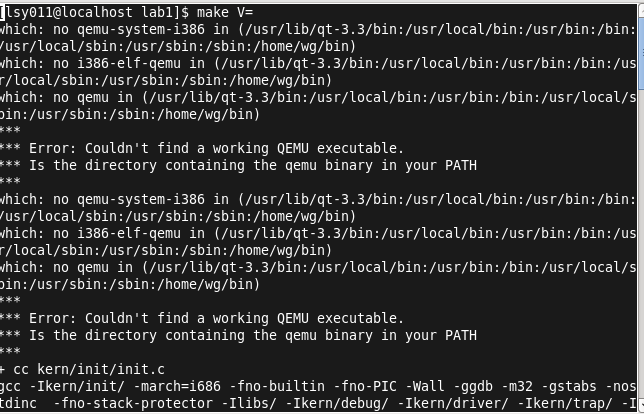
# create 'sign' tools

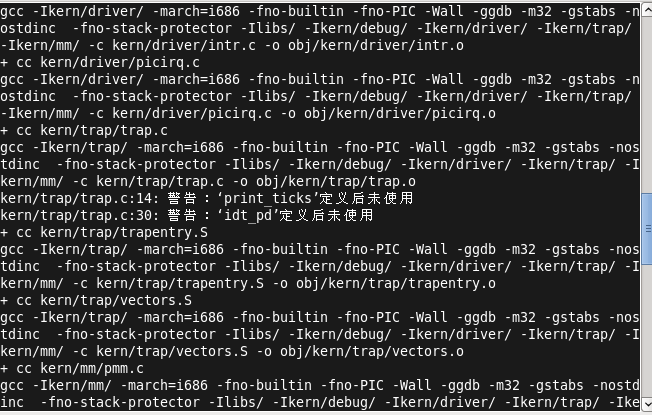
$(call add\_files\_host,tools/sign.c,sign,sign)

$(call create\_target\_host,sign,sign)

# -------------------------------------------------------------------

1. 运行make V=







1. 总结生成过程

1 编译所有生成bin/kernel所需的文件

2 链接生成bin/kernel

3 编译bootasm.S bootmain.c sign.c

4 根据sign规范生成obj/bootblock.o

1. 生成ucore.img

5、错误分析：

* 1. 输入make V=，显示内存不足。

解决办法：重装虚拟机。重装后，虚拟机可正常使用。

【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

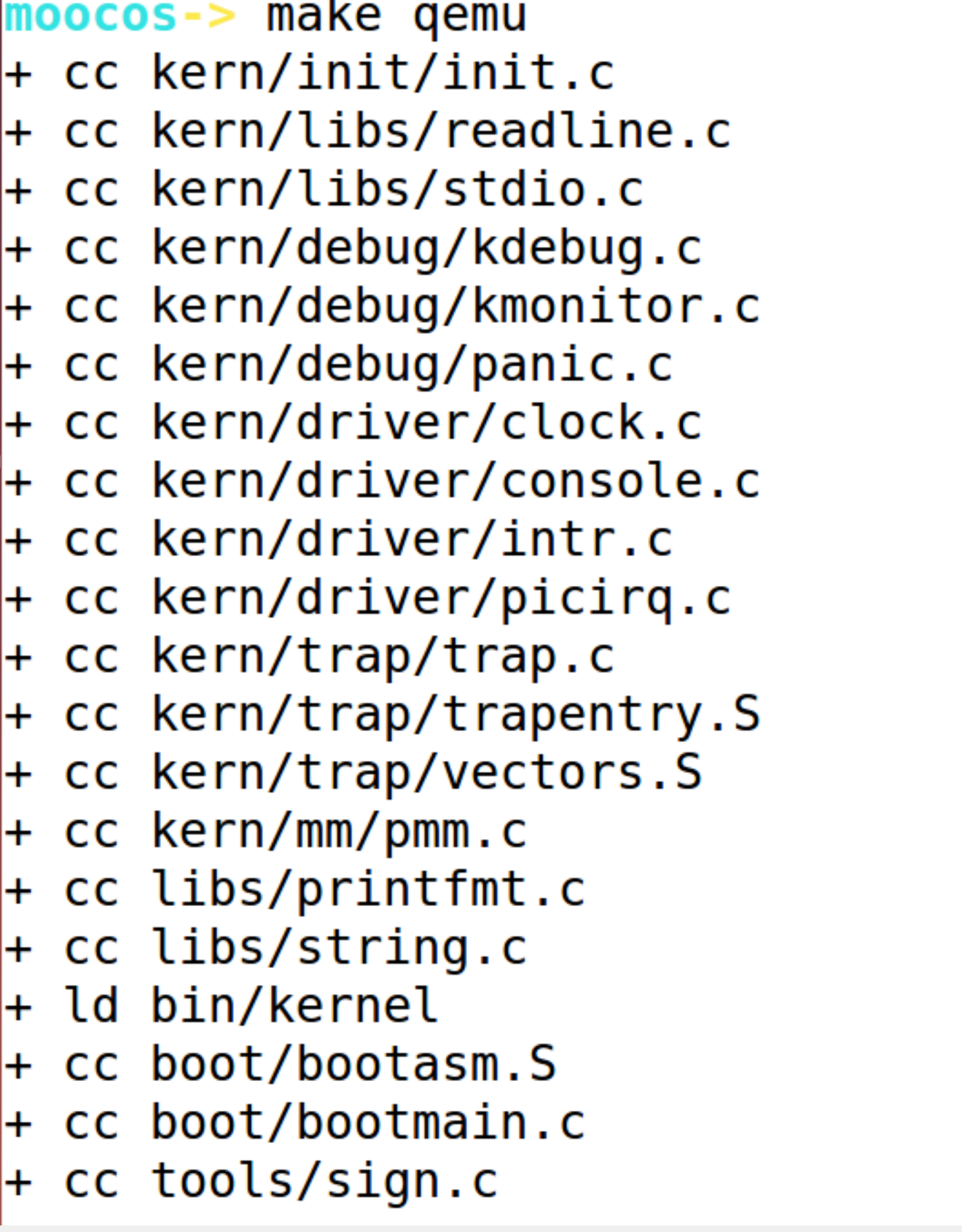
- 由不超过466字节的启动代码和不超过64字节的硬盘分区表加上两个字节的结束符组成

【练习2】

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。

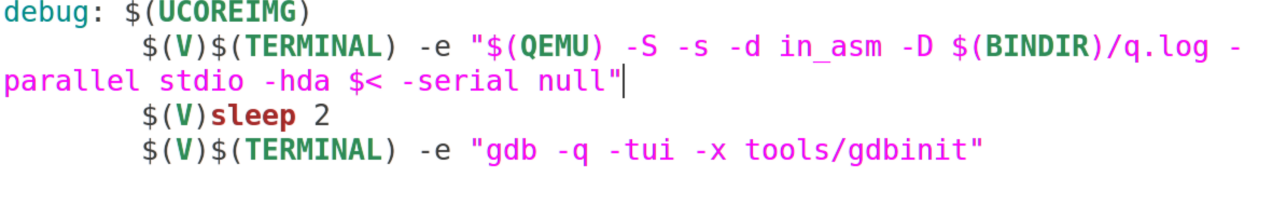
【练习2.1】从 CPU加电后执行的第一条指令开始，单步跟踪 BIOS的执行。

·执行make qemu指令，生成如下：



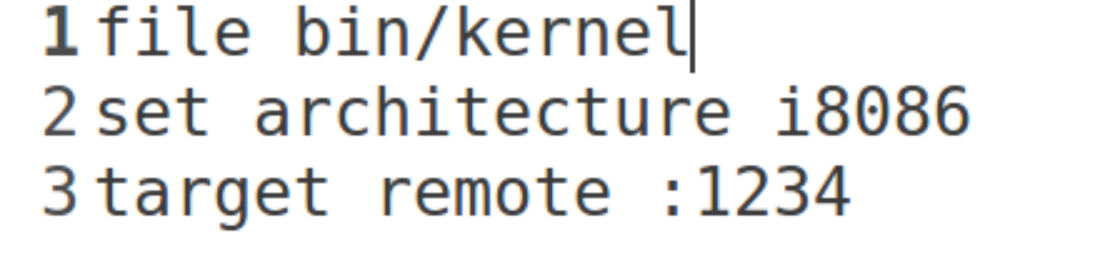
makefile中定义了debug的操作是启动QEMU、启动Terminal并在其中运行gdb。

·修改debug代码如下：



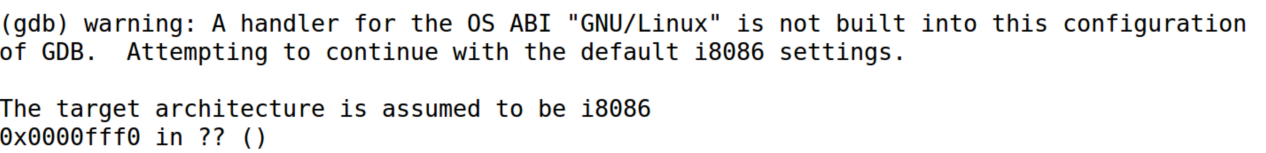
在调用qemu时增加-d in\_asm -D q.log参数，便可以将运行的汇编指令保存在q.log中。

·修改tools/gdbinit代码如下：



为防止qemu在gdb连接后立即开始执行，删除了tools/gdbinit中的“continue”行。

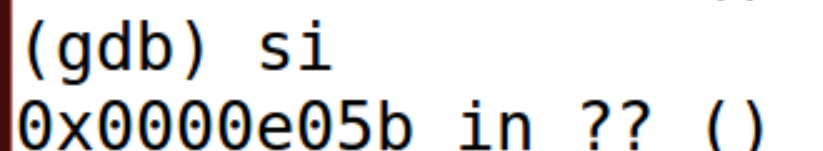
·执行make debug，这时会弹出一个QEMU窗口和一个Terminal窗口，显示如下图：



终端此时停在0x0000fff0的位置。

输入si命令单步调试。

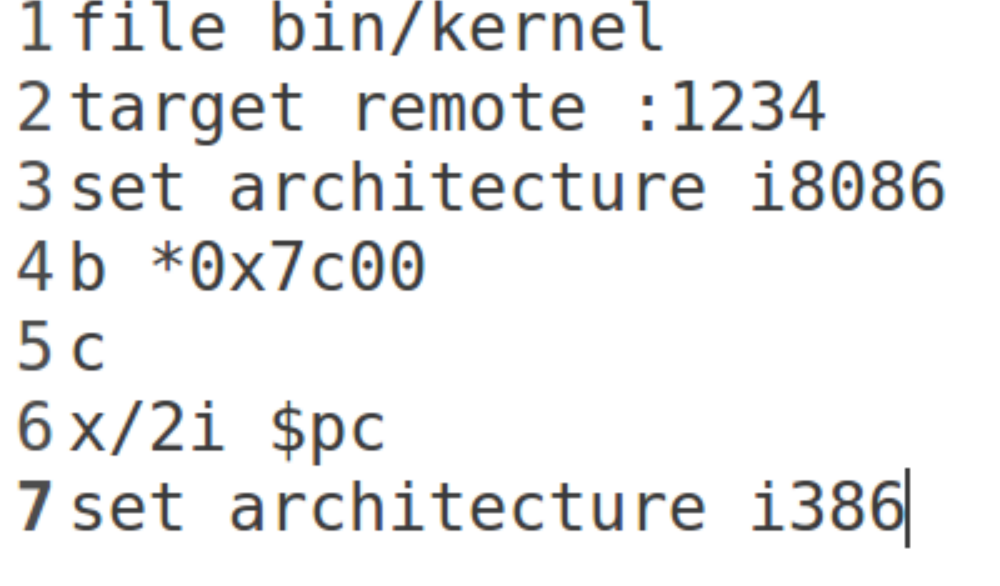
·终端打印的下一条命令的地址和内容：



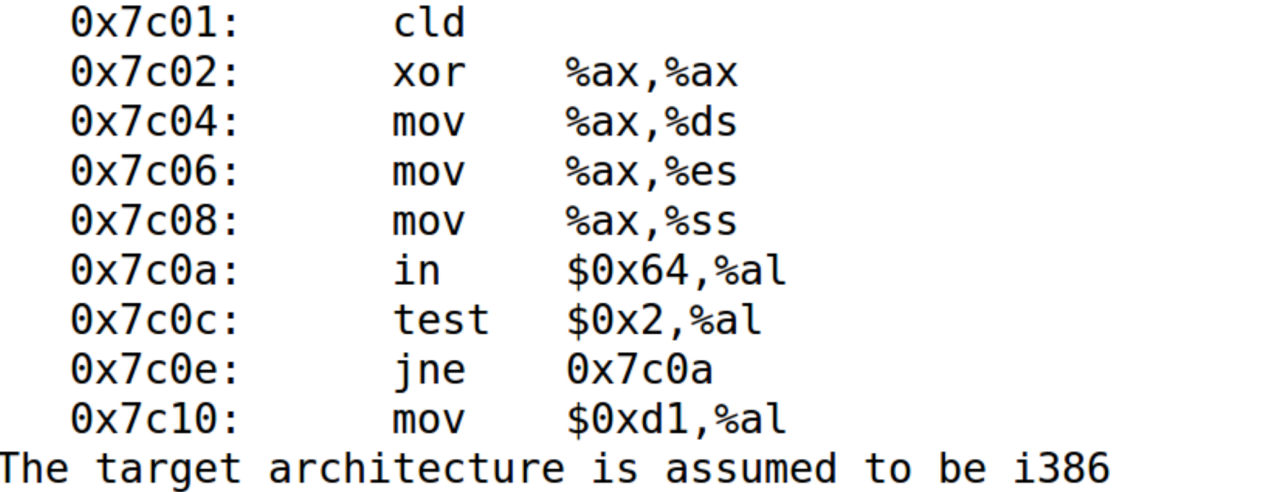
终端此时停在0x0000e05b位置。

【练习2.2】在初始化位置 0x7c00 设置实地址断点，测试断点正常。

·修改tools/gdbinit代码如下：



·运行make debug结果如下：

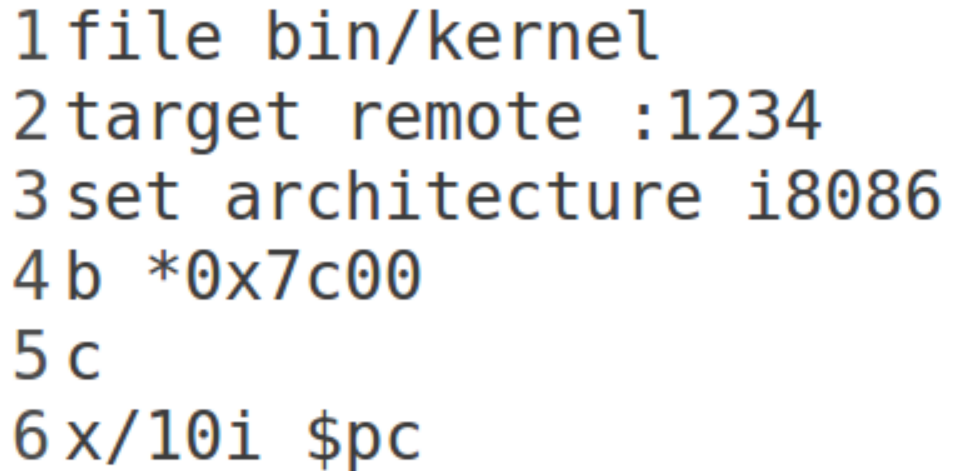


【练习2.3】从 0x7c00 开始跟踪代码运行，将单步跟踪反汇编得到的代码与 bootasm.S和 bootblock.asm进行比较。

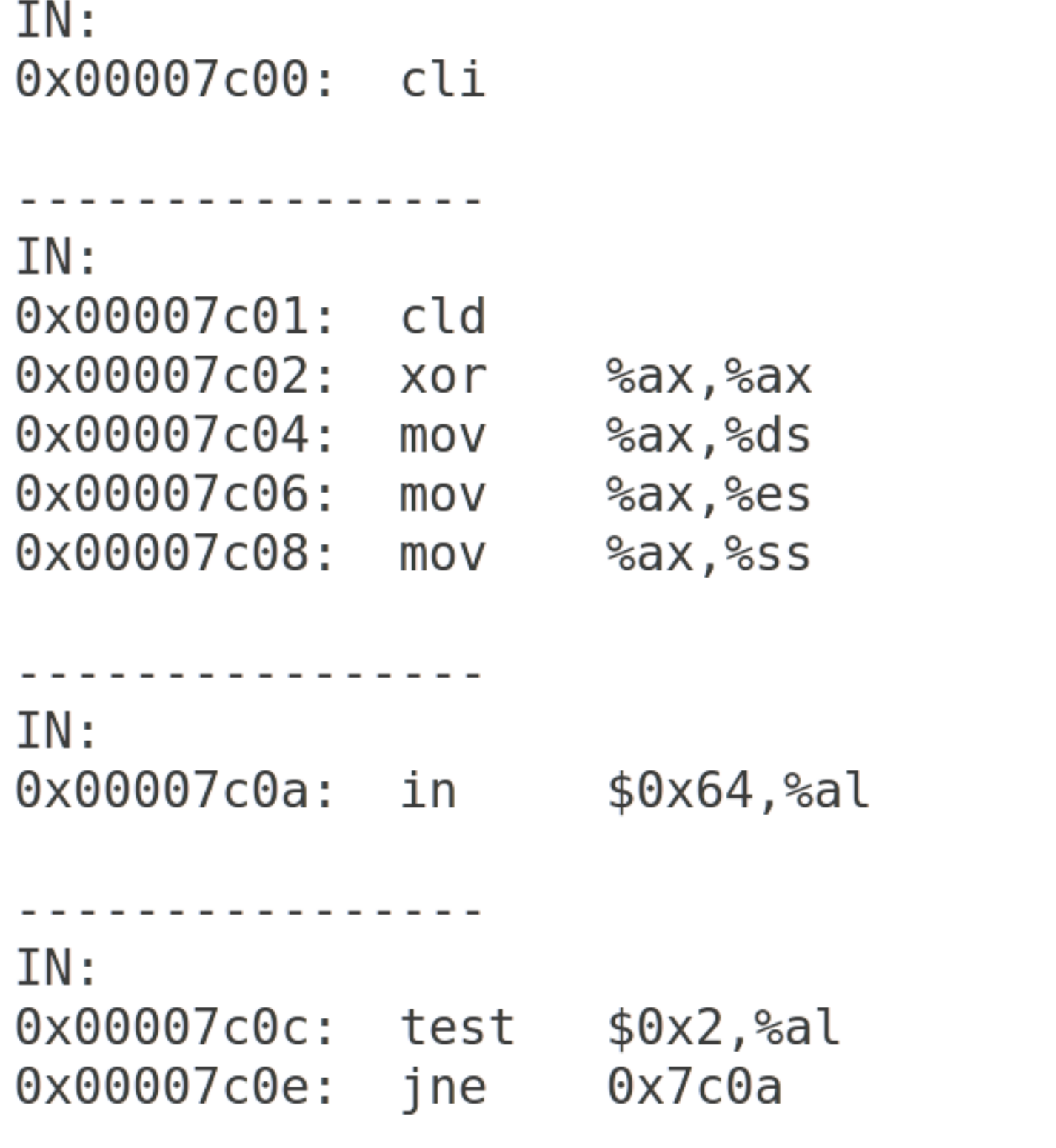
在调用qemu时增加-d in\_asm -D q.log参数，便可以将运行的汇编指令保存在q.log中。

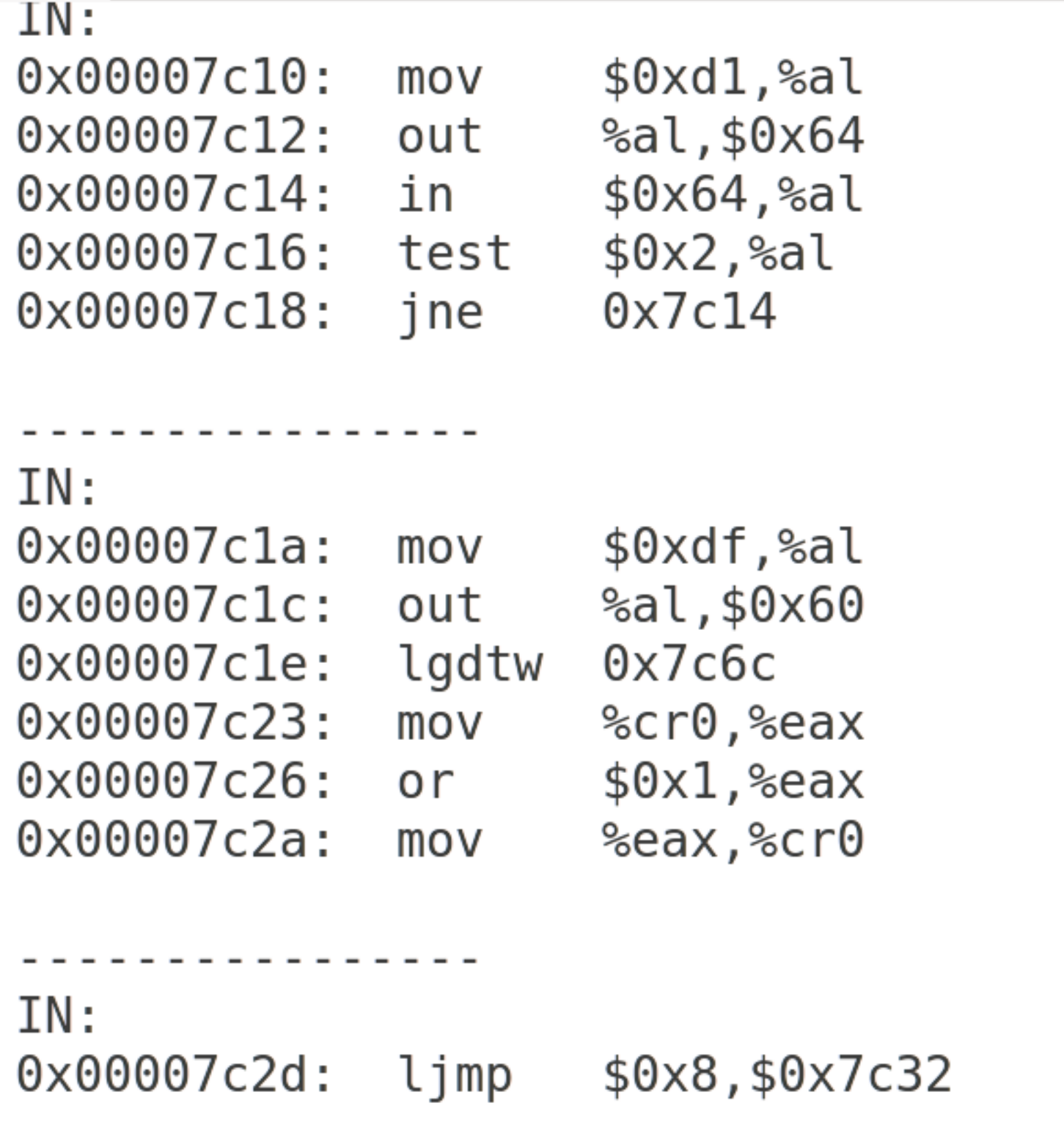
将执行的汇编代码与bootasm.S和bootblock.asm进行比较，看看二者是否一致。

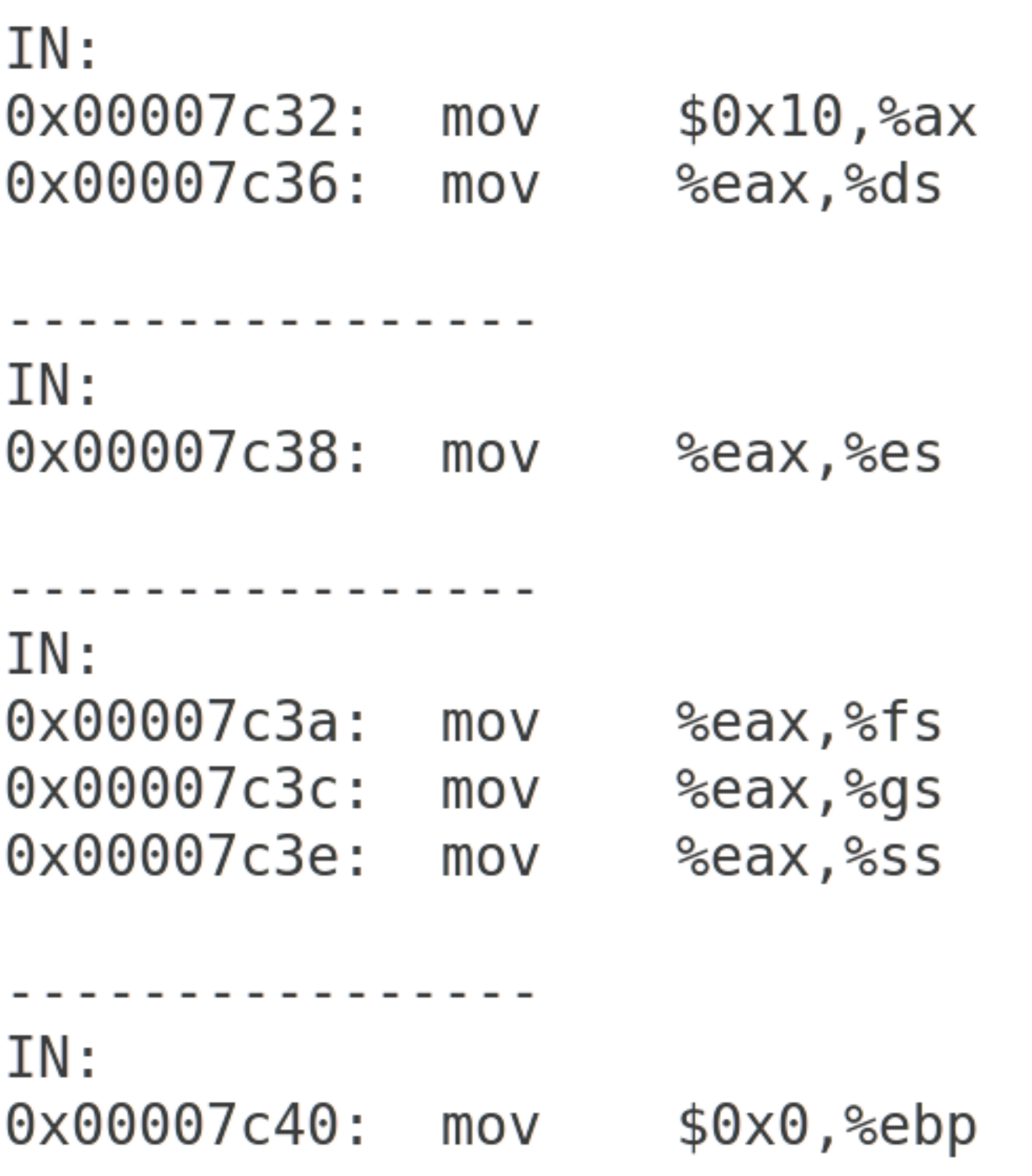
·修改tools/gdbinit代码如下：

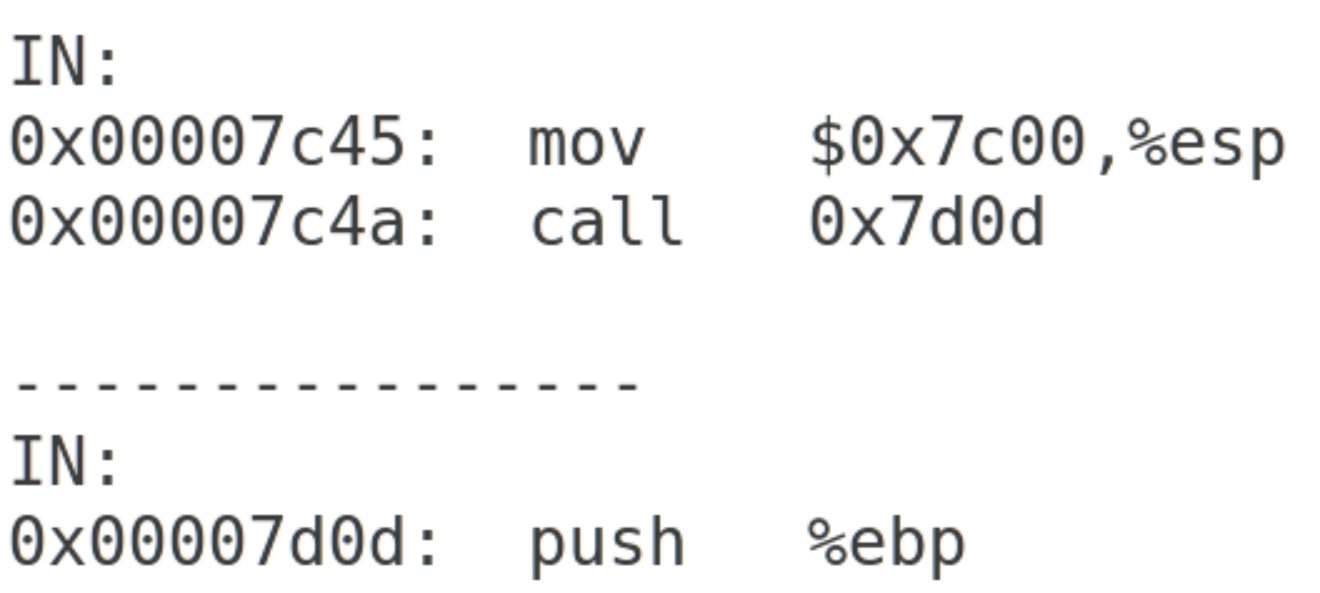


·在q.log中读到call bootmain前执行的命令，如下：









【练习3】分析从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。

1. 关闭中断，将各个段寄存器重置

·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 Segment

————————————————

1. 开启A20

将A20置1。

A20解释：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线置于高电位，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

1. 加载GDT表

lgdt gdtdesc

1. 将CR0的第0位置1

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

1. 长跳转到32位代码段，重装CS和EIP

ljmp $PROT\_MODE\_CSEG, $protcseg

1. 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 Segment

————————————————

1. 转到保护模式完成，进入boot主方法

movl $0x0, %ebp

movl $start, %esp

call bootmain

【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？
2. bootloader是如何加载 ELF格式的 OS？

这里主要分析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 & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);

}

((void (\*)(void))(ELFHDR->e\_entry & 0xFFFFFF))();

bad:

outw(0x8A00, 0x8A00);

outw(0x8A00, 0x8E00);

while (1);

}

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。

·下述为主要代码：

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进入保护模式并载入c程序bootmain

\* bootmain中readsect函数完成读取磁盘扇区的工作，函数传入一个指针和一个uint\_32类型secno，函数将secno对应的扇区内容拷贝至指针处

\* 调用waitdisk函数等待地址0x1F7中低8、7位变为0,1，准备好磁盘

\* 向0x1F2输出1，表示读1个扇区，0x1F3输出secno低8位，0x1F4输出secno的8~15位，0x1F5输出secno的16~23位，0x1F6输出0xe+secno的24~27位，第四位0表示主盘，第六位1表示LBA模式，0x1F7输出0x20

\* 调用waitdisk函数等待磁盘准备好

\* 调用insl函数把磁盘扇区数据读到指定内存

2. 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。

·下述为加载文件代码注释：

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 & 0xFFFFFF, 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 & 0xFFFFFF))();

bad:

..........

}

bootloader通过bootmain函数完成ELF格式OS的加载。

\* 调用readseg函数从kernel头读取8个扇区得到elfher

\* 判断elfher的成员变量magic是否等于ELF\_MAGIC，不等则进入bad死循环

\* 相等表明是符合格式的ELF文件，循环调用readseg函数加载每一个程序段

\* 调用elfher的入口指针进入OS

【练习5】

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。

1. 函数堆栈的原理

理解函数堆栈最重要的两点：

1. 栈的结构
2. 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。

·汇编指令如下：

pushl %ebp

movl %esp,%ebp

这两条汇编指令的含义：

1. 将ebp 寄存器入栈
2. 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。

函数调用包括：

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：
   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）。
   2. 将当前栈帧切换到新栈帧（将ESP值装入EBP，更新栈帧底部）。
   3. 给新栈帧分配空间（把ESP减去所需空间的大小，抬高栈顶）。

函数返回包括：

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。
   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. 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。
   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。

直接根据ebp能够读取到各个栈帧的地址和值，查询得知，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。

2、print\_stackframe函数的实现

·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 (unit32\_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]

\*/

}

————————————————

·根据注释编写成程序，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

cprintf("ebp:0x%08x eip:0x%08x ",ebp,eip);//(3.1)printf value of ebp, eip

uint32\_t \*tmp=(uint32\_t \*)ebp+2;

cprintf("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 (unit32\_t)ebp +2 [0..4]

cprintf("\n");//(3.3) cprintf("\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

}

}

————————————————

【练习6】

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，权限为陷阱门描述符。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】

这里主要就是实现对中断向量表的初始化。

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!

\*/

}

————————————————

第一步，声明\_\_vertors[]，其中存放着中断服务程序的入口地址。这个数组生成于vertor.S中。

第二步，填充中断描述符表IDT。

第三步，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\_\_vertors[]

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);

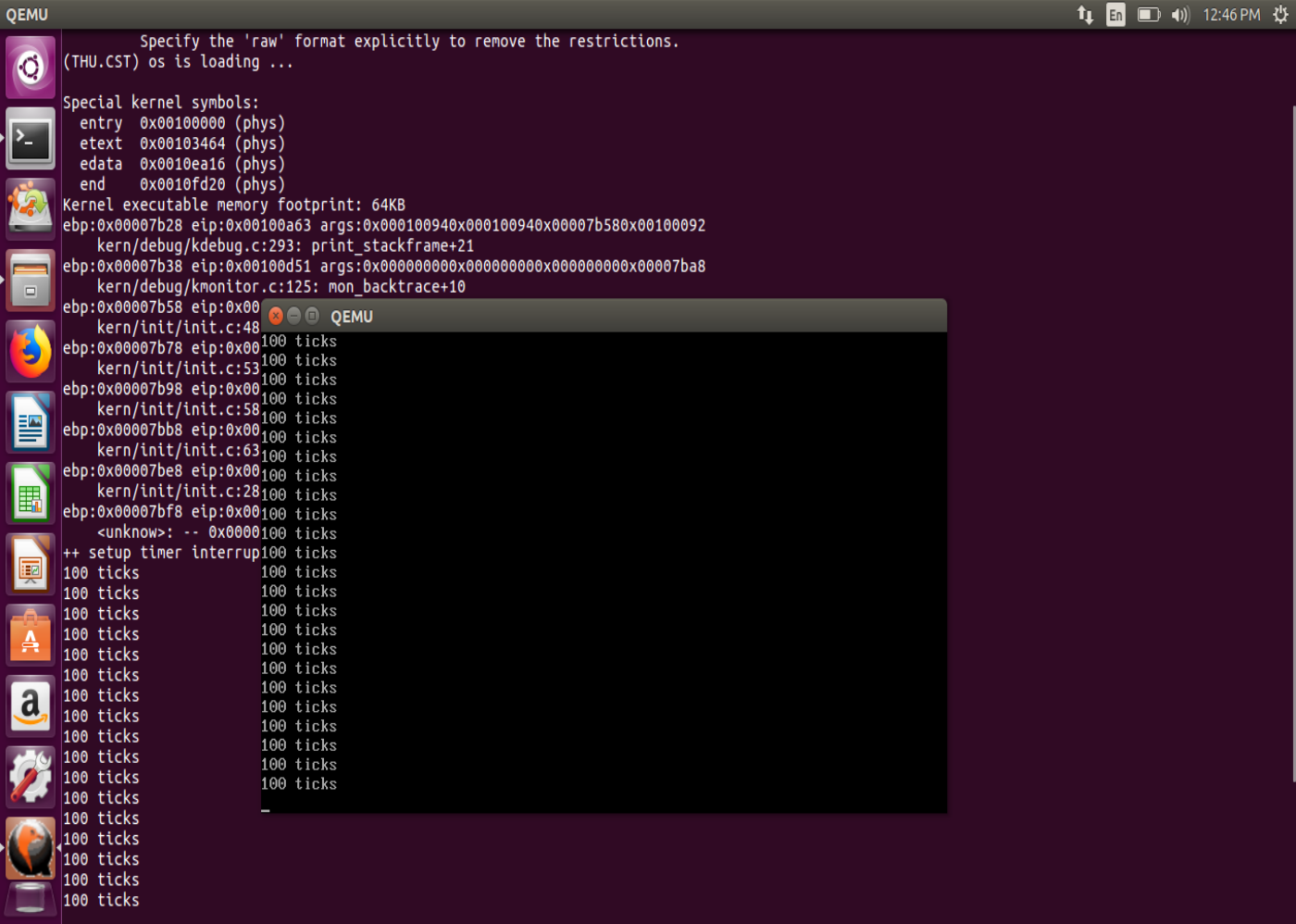
lidt(&idt\_pd);//使用lidt指令加载中断描述符表

}

·这里的SETGATE在mmu.h中有定义：

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

·运行结果如下：



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

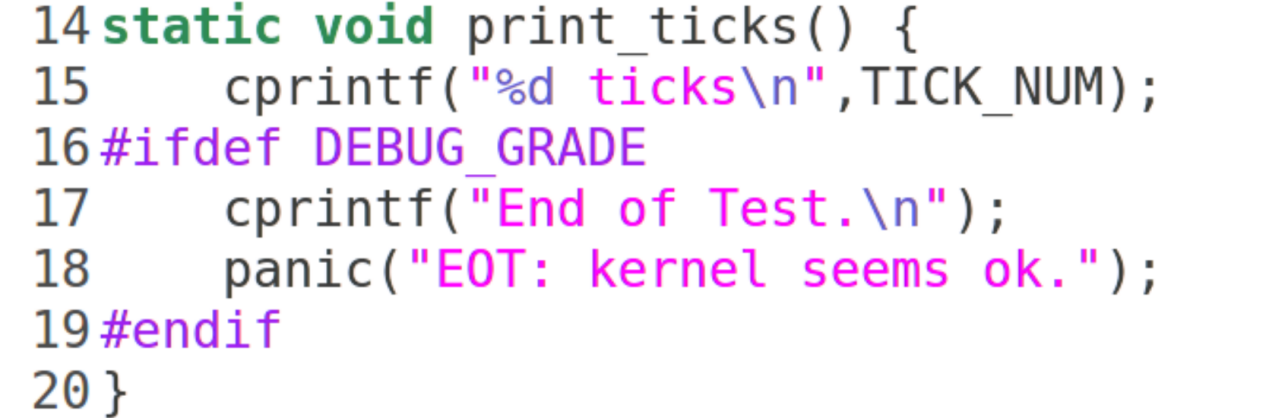
【练习6.3】

trap函数只是直接调用了trap\_dispatch函数，而trap\_dispatch函数实现对各种中断的处理，题目要求我们完成对时钟中断的处理。

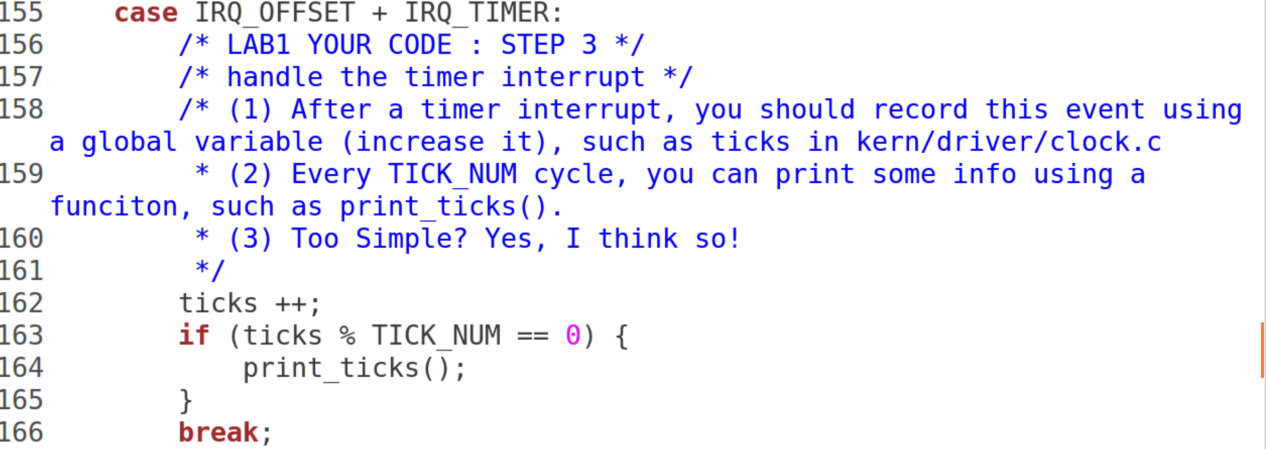
实现操作：定义一个全局变量ticks，每次时钟中断将ticks加1，加到100后打印"100 ticks"，然后将ticks清零重新计数。

print\_ticks()子程序被实现。

·代码实现如下：



·定义一个全局变量ticks，每次时钟中断将ticks加1，加到100后打印“100 ticks”的代码实现如下：



·实现如下：

