**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）**

**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)**

(1)通过GCC编译器将Kernel目录下的.c文件编译成OBJ目录下的.o文件。

(2)ld命令根据链接脚本文件kernel.ld将生成的\*.o文件，链接成BIN目录下的kernel文件

(3)通过GCC编译器将boot目录下的.c,.S文件以及tools目录下的sign.c文件编译成OBJ目录下的\*.o文件。

(4)ld命令将生成的\*.o文件，链接成BIN目录下的bootblock文件。

(5)dd命令将dev/zero, bin/bootblock,bin/kernel 写入到bin/ucore.img

**练习2：使用qemu执行并调试lab1中的软件。（要求在报告中简要写出练习过程）**

**为了熟悉使用qemu和gdb进行的调试工作，我们进行如下的小练习：**

**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。**

问题1：从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行

1.tools/gdbinit的内容如下。可见，这里是对内核代码进行调试，并且将断点设置在内核代码的入口地址，即kern\_init函数

file bin/kernel

target remote :1234

break kern\_init

continue

1.为了从CPU加电后执行的第一条指令开始调试，需要修改tools/gdbinit的内容为：

set architecture i8086

file bin/bootblock

target remote :1234

break startcontinue

执行make debug，这时会弹出一个QEMU窗口和一个Terminal窗口，这是正常的，因为我们在makefile中定义了debug的操作正是启动QEMU、启动Terminal并在其中运行gdb。

debug: $(UCOREIMG)

$(V)$(QEMU) -S -s -parallel stdio -hda $< -serial null &

$(V)sleep 2

$(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"

1.Terminal窗口此时停在0x0000fff0的位置，这是eip寄存器的值，而cs寄存器的值为0xf000. （遇到一个问题：此时无法正确反汇编出代码，使用x来查询内存0xfff0处的值时显示全0，不知道什么原因）

The target architecture is assumed to be i8086

0x0000fff0 in ?? ()

Breakpoint 1 at 0x7c00: file boot/bootasm.S, line 16.

1.输入si，执行1步，程序会跳转到0xe05b的地方。查看寄存器也可以发现eip的值变为0xe05b，而cs的值不变，仍然是0xf000.

2.反复输入si，以单步执行。

问题2：在初始化位置0x7c00设置实地址断点,测试断点正常

1.直接在tools/gdbinit中设置了断点break start，由于boot loader的入口为start，其地址为0x7c00，因此这和break \*0x7c00效果是相同的。

2.设置断点后，输入continue或c，可以看到程序在0x7c00处停了下来，说明断点设置成功。

问题3：从0x7c00开始, 将反汇编代码与bootasm.S和bootblock.asm进行比较

1.反汇编的代码与bootblock.asm基本相同，而与bootasm.S的差别在于：

（1）反汇编的代码中的指令不带指示长度的后缀，而bootasm.S的指令则有。比如，反汇编 的代码是xor %eax, %eax，而bootasm.S的代码为xorw %ax, %ax

（2）反汇编的代码中的通用寄存器是32位（带有e前缀），而bootasm.S的代码中的通用寄存器是16位（不带e前缀）。

问题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

file bin/kernel

target remote :1234

set architecture i8086

b \*0x7d10

c

x/i $pc

set architecture i386

define hook-stop

x/i $pc

end

**练习3：分析bootloader进入保护模式的过程。（要求在报告中写出分析）**

**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：**

**1.为何开启A20，以及如何开启A20**

**2.如何初始化GDT表**

**3.如何使能和进入保护模式**

start:

# Step1：清理环境，射中重要段寄存器的初值。

.code16

cli

cld

xorw %ax, %ax

movw %ax, %ds

movw %ax, %es

movw %ax, %ss

# Step2：开启A20。通过将键盘控制器上的A20线置于高电位，全部32条地址线可用， 可以访问4G的内存空间。

seta20.1:

inb $0x64, %al # 等待8042键盘控制器不忙

testb $0x2, %al

jnz seta20.1

movb $0xd1, %al # 向8042端口发送0xd1

outb %al, $0x64

seta20.2:

inb $0x64, %al # 等待8042键盘控制器不忙input buffer empty). testb $0x2, %al

jnz seta20.2

movb $0xdf, %al # 开启A20

outb %al, $0x60

# Step3：初始化GDT，从实模式切换至保护模式

lgdt gdtdesc

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

# Step4：将处理器转至32位模式，跳转

ljmp $PROT\_MODE\_CSEG, $protcseg

.code32 # Assemble for 32-bit mode

protcseg:

# Step5：设置段寄存器，并建立堆栈

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

movl $0x0, %ebp

movl $start, %esp

# Step6：进入bootmain

call bootmain