**Ucore 实验的运行和调试**

**1. shell 下的编译运行调试**

**先进入一个 lab（例如 ucore\_os\_lab/labcodes\_answer/lab1\_result)，然和键入 make，就可以编**

**译了，make clean 清除编译的结果，make qemu可以运行代码。**

**调试呢稍微复杂一点，我们需要开两个 Terminal 窗口，都进入同一个 lab目录下，其中一个使用 qemu-**

**system-i386 -S -s -hda ./bin/ucore.img -monitor stdio 命名将虚拟机跑你来并等待调试操**

**作。项目中有两个可执行文件（obj/bootblock.o bin/kernel），一个对应于 bootLoader，一个对应**

**于操作系统内核，可以选择任意一个进行调试。在另一个窗口键入 i386-elf-gdb + 需要调试的文件，进**

**入 gdb 后，键入 target remote 127.0.0.1:1234 ,使得 gdb 与 qemu连接。然后就可以开始调试了。**

**b + 函数名 或者 内存地址 ： 表示添加断点**

**next ： 执行下一条语句**

**nexti： 执行下一条汇编语句**

**file +　文件名　：指定要调试的文件**

**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**)

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)

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。

2、生成kernel

而生成kernel的代码如下：

$(kernel): tools/kernel.ld

$(kernel): $(KOBJS)

@echo "bbbbbbbbbbbbbbbbbbbbbb$(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)

3、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)

1. **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？**

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：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**

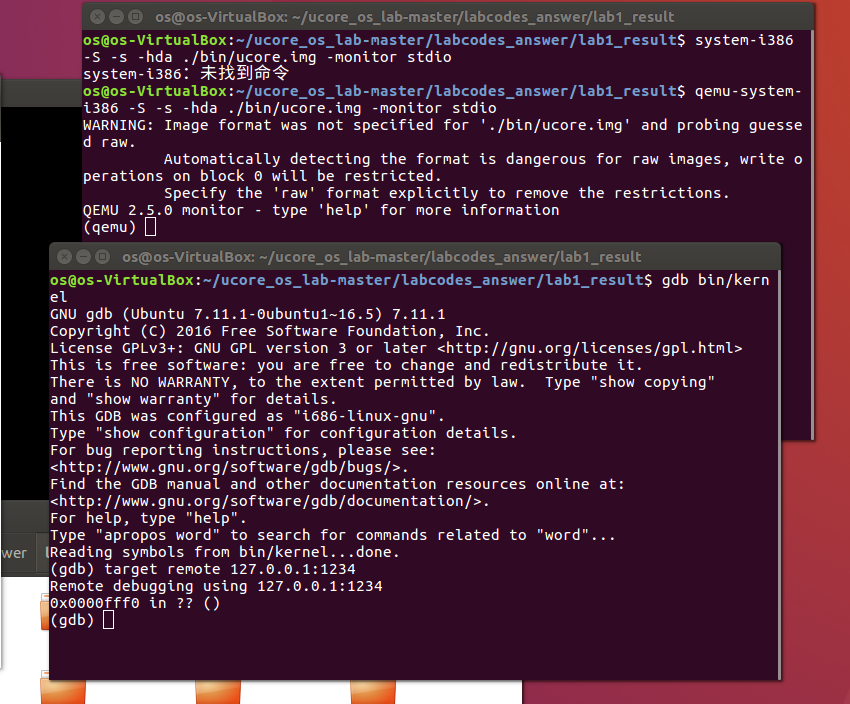
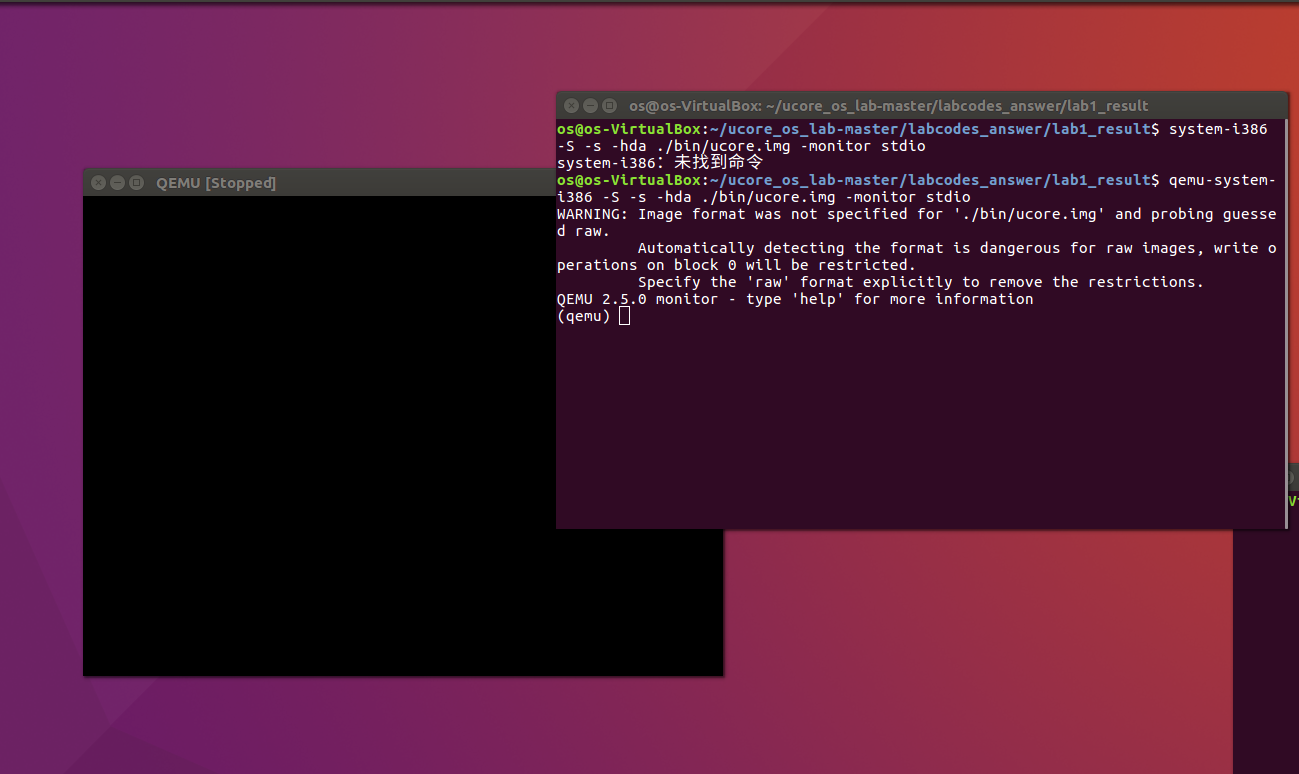
**bootblock.asm进行比较。**

1. **自己找一个bootloader或内核中的代码位置，设置断点并进行测试。**

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 127.0.0.1:1234

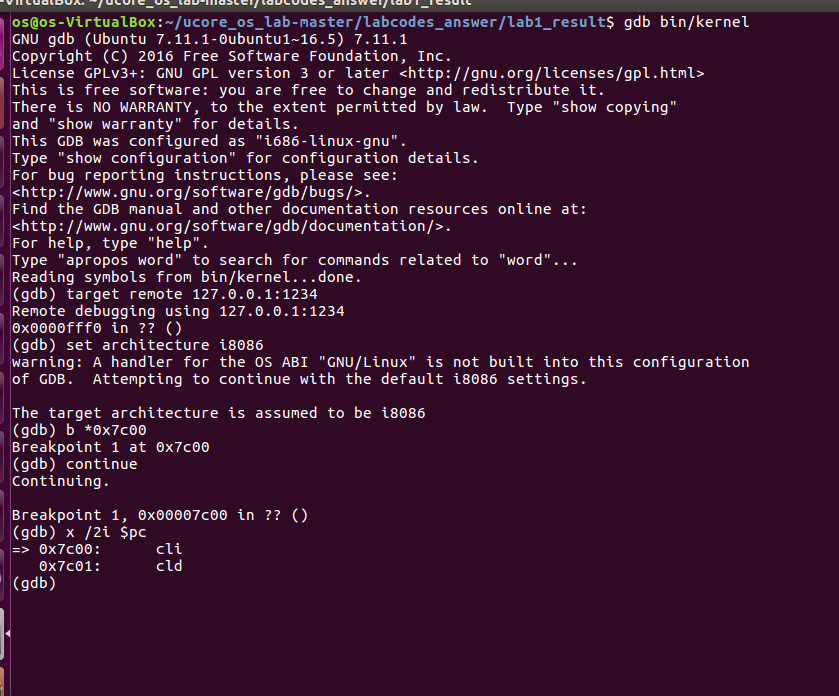
进入到调试界面：

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，对比此时bootasm.S中的起始代码，发现确实是一样的



**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：**

**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**

bootloader：

1、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 Segment

2、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

3、加载GDT表

lgdt gdtdesc

4、将CR0的第0位置1

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

5、长跳转到32位代码段，重装CS和EIP

ljmp $PROT\_MODE\_CSEG, $protcseg

6、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

7、转到保护模式完成，进入boot主方法

movl $0x0, %ebp

movl $start, %esp

call bootmain