**操作系统实验1**

171491205 汪昊程 硬件1班

前言积累篇：

Linux入门实验笔记

前言：为什么要写这个笔记？

两年的大学生涯已经过去，大三上学期的课程可谓深有难度，此中定然包括了我们这门课程--操作系统。

在前一个月的学习中，我发现，如果只是泛泛的了解操作系统学习的命令，学之用之但不理解之，仅靠课上所学，根本没办法理解这晦涩难懂的知识。如果运气好当天理解了，两天后不回顾必忘。所以我现在发现学习os这门课程，一是掌握理论知识，二是掌握实际操作，我写这个笔记也是为了目的2.虽然会较以前相比有相当大的time cost，但是学习不在多而在精。每天多花时间总结，将来必有收获。

----------------------------------2019.10.13----------------------------------------

[基础命令篇]

ls命令：将你当前目录下的所有文件名展示在终端上。按照字母大小排列。文件夹显示蓝色，文件显示绿色。

pwd命令：显示当前的目录

echo命令：在屏幕上输出字符。该命令的特点在于，你可以在屏幕上输入你的参数“”<--像左面这样用引号扩起来。

cat:打开一个目标文件。注意：我使用的时候如果打上后缀，会报错。具体需不需要打后缀再议

。

cp:复制一个目标文件。格式：cp 你要复制的文件名 [空格] 副本的文件名。之后可以用cat命令来检测你是否复制成功。

mv:把目标文件的内容挪到另外一个文件中。（a to b)类似于windows系统的“剪切”。经过测试，无论那个“另外一个文件”是否存在，原来的那个文件都消失了。

cp与mv的关系：cp是单纯的复制，而mv类似于剪切（文件与文件之间的剪切)

touch:新建一个文件。{问题：为什么我无论后面打什么格式，新建的文件永远都是txt?}

mkdir:新建一个目录，好像就是创建了一个文件夹。

rm:删除文件/目录。慎用！

ps:查询当前的进程。

-----------------------------基础命令篇未完待续---------------------------------

正文部分：

**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)

答：

create ucore.img #创建ucore.img

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

$(UCOREIMG):$(kernel)$(bootblock) #UCOREIMG依赖kernel和bootblock

$(V)dd if=/dev/zeroof=$@ count=10000

$(V)dd if=$(bootblock) of=$@ conv=notrunc

$(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

$(call create\_target,ucore.img)

#dd表示链接，将文件连接在一起，if=inputfile，if=/dev/zero 表示空。of=$@中，￥@是个变量，表示ucore.img,count=10000表示开辟了一个10000的空间，存到我们将来要编辑的Ucore.img。if又把bootblock和kernel放到了ucore.image

# 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)

#前面是目标，后面是依赖的文件

@echo + ld $@

$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)

@$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)#如何编译汇编文件

@$(OBJDUMP) -t $(call objfile,bootblock) | $(SED) '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > $(call symfile,bootblock)#如何编译.c文件

@$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)

@$(call totarget,sign) $(call outfile,bootblock) $(bootblock)

$(call create\_target,bootblock)

# create kernel target

kernel = $(call totarget,kernel) #最总的目标文件obj/kernel

$(kernel): tools/kernel.ld #最总的目标文件规则

$(kernel): $(KOBJS)

@echo + ld $@

$(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)#链接目标的一些文件，生成ELFi386的内核文件，并且使用kernel.ld链接器脚本

@$(OBJDUMP) -S $@ > $(call asmfile,kernel)

@$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > $(call symfile,kernel)#最终的目标文件去掉符号表等信息，并输出符号表信息，汇编文件信息，和输出信息。

$(call create\_target,kernel)

# create bootblock#启动扇区的翻译，过程与内核差不多，唯一的区别是对破译后的扇区进行签名。最后字节0X55aa

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) #前面是目标，后面是依赖的文件

@echo + ld $@

$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)

@$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)

@$(OBJDUMP) -t $(call objfile,bootblock) | $(SED) '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > $(call symfile,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 #在内核工具目录中，sign.c用来给扇区签名的小工具，使用host是因为该工具是在特定操作系统下的下的工具，所以编译过程和内核编译完全不同。

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

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

总结：

-t

--touch:这个参数是吧目标文件的时间更新，但不更改目标文件。也就是说，make假装编译目标，但不是真正编译目标，只是把目标变成已经编译的状态。

-n

不执行参数，这些参数只是打印命令，不管目标是否更新，把规则和连带规则打印出来，但是不执行

-q：该参数的行为行为是找目标

-b

-m作用是忽略与其他版本make的兼容性

-d:输出make的调试信息

练习2：Qumu执行并调试lab1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。

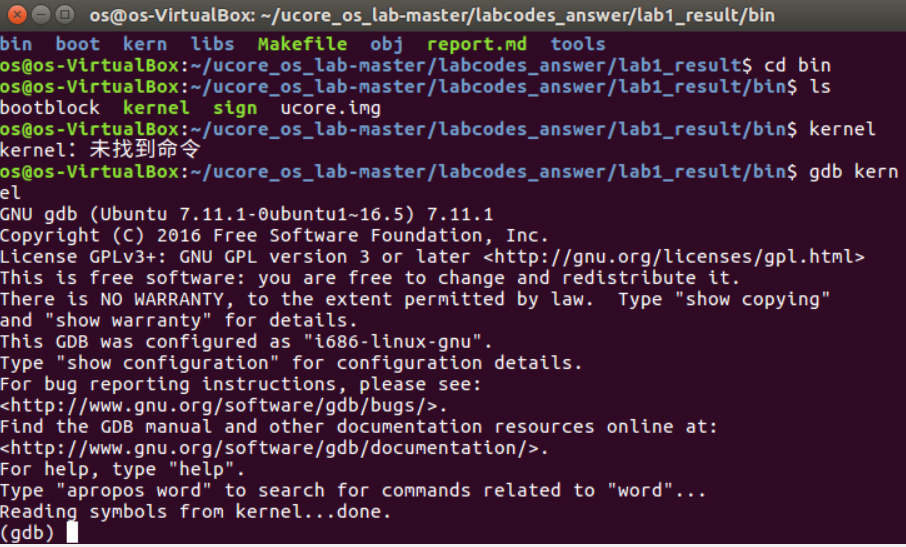
2. 在初始化位置0x7c00设置实地址断点,测试断点正常。

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。

打开init1，文件显示如下：





老师，这个练习2我感到有点懵圈。。就是照做了一遍也不知道自己在干啥

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

解答：

# start address should be 0:7c00, in real mode, the beginning address of the running bootloader从0x7c00开始运行bootloader

.globl start

start:

.code16

cli # Disable interrupts关中断

cld # String operations incremen

# Set up the important data segment registers (DS, ES, SS).三个段寄存器置零

xorw %ax, %ax # Segment number zero

movw %ax, %ds # -> Data Segment

movw %ax, %es # -> Extra Segment

movw %ax, %ss # -> Stack Segment

lgdt gdtdesc

#将CR0寄存器第一位置1来开启保护模式

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

# Jump to next instruction, but in 32-bit code segment.

# Switches processor into 32-bit mode.

#长跳转到32位模式下的下一条指令，将$PROT\_MODE\_CSEG加载到CS中，将$protcseg加载到IP中

ljmp $PROT\_MODE\_CSEG, $protcseg

.code32 # Assemble for 32-bit mode

protcseg:

# Set up the protected-mode data segment registers

#将数据段选择子加载到ax中

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

#调用bootmain

movl $0x0, %ebp

movl $start, %esp

call bootmain

spin:

jmp spin

# Bootstrap GDT

.p2align 2

#全局描述符表

gdt:

SEG\_NULLASM # null seg

#代码段描述符

SEG\_ASM(STA\_X|STA\_R, 0x0, 0xffffffff)

#数据段描述符

SEG\_ASM(STA\_W, 0x0, 0xffffffff)

#全局描述符表对应的描述符

gdtdesc:

.word 0x17 # sizeof(gdt) - 1

.long gdt # address gdt

部分代码学习了csdn上大佬的成果，站在巨人的肩膀上~！