**171491322 谷贺 lab1**

**实验一：系统软件启动过程**

**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**

1. CPU的编址与寻址: 基于分段机制的内存管理

2.CPU的中断机制

3.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

***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。

**对实验报告的要求：**

**\***基于markdown格式来完成，以文本方式为主。

\*填写各个基本练习中要求完成的报告内容

**\***完成实验后，请分析ucore\_lab中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别

**\***列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对 二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知 识点）

**\***列出你认为OS原理中很重要，但在实验中没有对应上的知识点

**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) 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？

输入make V=命令，出现以下命令：

1.+ cc kern/init/init.c

gcc -Ikern/init/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/init/init.c -o obj/kern/init/init.o

-m32:表示在64位系统下编译32位的二进制文件，需要额外的库支持

-fno-stack-protector：禁用栈保护措施

-Wall：编译后显示所有警告

-ggdb:产生gdb所需的调试信息

-nostdinc：告诉gcc不要包含libc等库的标准头文件

-gstabs：以stabs格式声称调试信息，但是不包括gdb调试信息

-fno-builtin：不接受没有\_builtin\_前缀的函数作为内建函数

-l:手动添加链接库

这个命令的作用：编译init.c文件，生成init.o文件

2.+ cc kern/libs/stdio.c

gcc -Ikern/libs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/libs/stdio.c -o obj/kern/libs/stdio.o

编译stdio.c文件，生成stdio.o文件

3.+ cc kern/libs/readline.c

gcc -Ikern/libs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/libs/readline.c -o obj/kern/libs/readline.o

编译readline.c文件，生成readline.o文件

4.+ cc kern/debug/panic.c

gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/panic.c -o obj/kern/debug/panic.o

kern/debug/panic.c: In function ‘\_\_panic’:

kern/debug/panic.c:27:5: warning: implicit declaration of function ‘print\_stackframe’ [-Wimplicit-function-declaration]

print\_stackframe();

^

编译panic.c文件，生成panic.o文件

5.+ cc kern/debug/kdebug.c

gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/kdebug.c -o obj/kern/debug/kdebug.o

编译kdebug.c文件，生成kdebug.o文件

6.+ cc kern/debug/kmonitor.c

gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/kmonitor.c -o obj/kern/debug/kmonitor.o

编译kmonitor.c文件，生成kmonitor.o文件

7.+ cc kern/driver/clock.c

gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/clock.c -o obj/kern/driver/clock.o

编译clock.c文件，生成clock.o文件

8.+ cc kern/driver/console.c

gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/console.c -o obj/kern/driver/console.o

编译console.c文件，生成console.o文件

9.+ cc kern/driver/picirq.c

gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/picirq.c -o obj/kern/driver/picirq.o

编译picirq.c文件，生成picirq.o文件

10.+ cc kern/driver/intr.c

gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/intr.c -o obj/kern/driver/intr.o

编译intr.c文件，生成intr.o文件

11.+ cc kern/trap/trap.c

gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/trap.c -o obj/kern/trap/trap.o

编译trap.c文件，生成trap.o文件

12.+ cc kern/trap/vectors.S

gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/vectors.S -o obj/kern/trap/vectors.o

编译vector.S文件，生成vectors.o文件

13.+ cc kern/trap/trapentry.S

gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/trapentry.S -o obj/kern/trap/trapentry.o

编译trapentry.S文件，生成trapentry.o文件

14.+ 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

编译pmm.c文件，生成pmm.o文件

15.+ 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

编译string.c文件，生成string.o文件

16.+ 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

编译printfmt.c文件，生成printfmt.o文件

17.+ ld bin/kernel

ld -m elf\_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/init.o obj/kern/libs/stdio.o obj/kern/libs/readline.o obj/kern/debug/panic.o obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/picirq.o obj/kern/driver/intr.o obj/kern/trap/trap.o obj/kern/trap/vectors.o obj/kern/trap/trapentry.o obj/kern/mm/pmm.o obj/libs/string.o obj/libs/printfmt.o

连接成kernel

18.+ cc boot/bootasm.S

gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o

编译bootasm.S文件，生成bootasm.o文件

19.+ 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

编译bootmain.c文件，生成bootmain.c文件

20.+ 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

把sign.c文件编译并连接成可执行文件sign

21.+ ld bin/bootblock

ld -m elf\_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o

'obj/bootblock.out' size: 500 bytes

build 512 bytes boot sector: 'bin/bootblock' success!

根据sign规范生成bootblock

22.dd if=/dev/zero of=bin/ucore.img count=10000

记录了10000+0 的读入

记录了10000+0 的写出

5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.231004 s, 22.2 MB/s

创建大小为10000个块的ucore.img,初始化为0，每个块为512字节

23.dd if=bin/bootblock of=bin/ucore.img conv=notrunc

记录了1+0 的读入

记录了1+0 的写出

512 bytes copied, 0.00992629 s, 51.6 kB/s

把bootblock中的内容写到第一个块

24.dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc

记录了146+1 的读入

记录了146+1 的写出

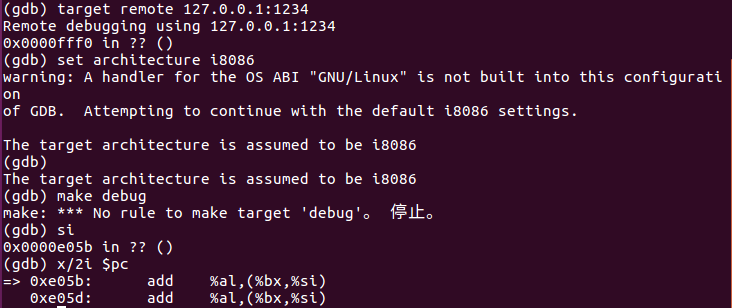
74880 bytes (75 kB, 73 KiB) copied, 0.017913 s, 4.2 MB/s

从第二个块开始写kernel中的内容

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）

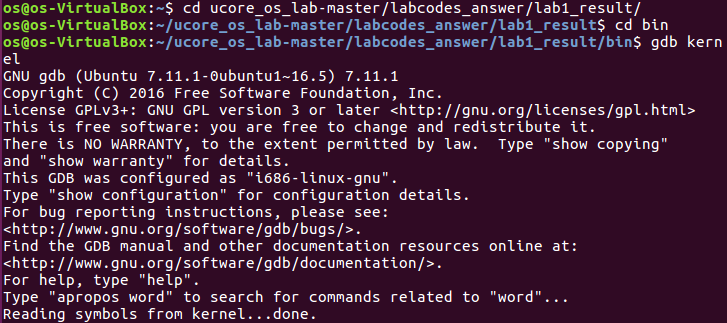
1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

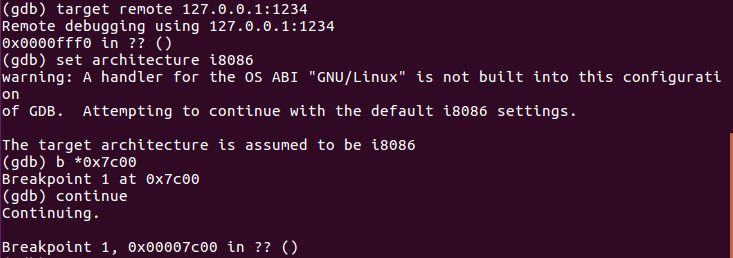
命令及运行结果如下图:

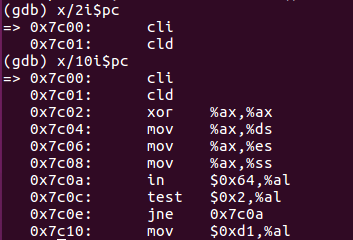


1. 在初始化位置0x7c00设置实地址断点，测试断点正常。

命令及运行结果如下图：

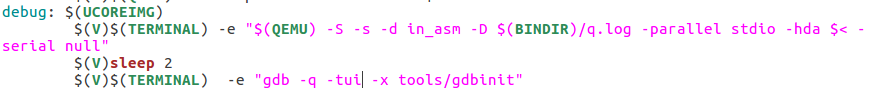






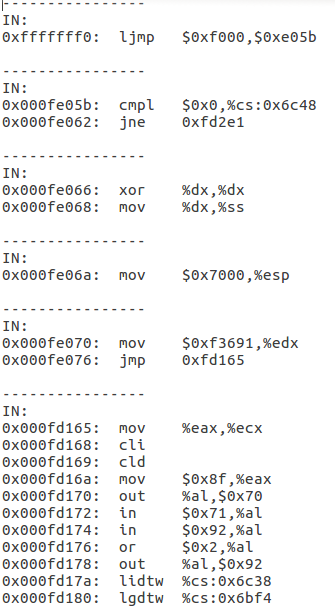
1. 从0x7c00开始跟踪代码运行，将单步跟踪反汇编得到的代码与bootasm.S和bootblock.asm进行比较。

修改makefile文件如下图：

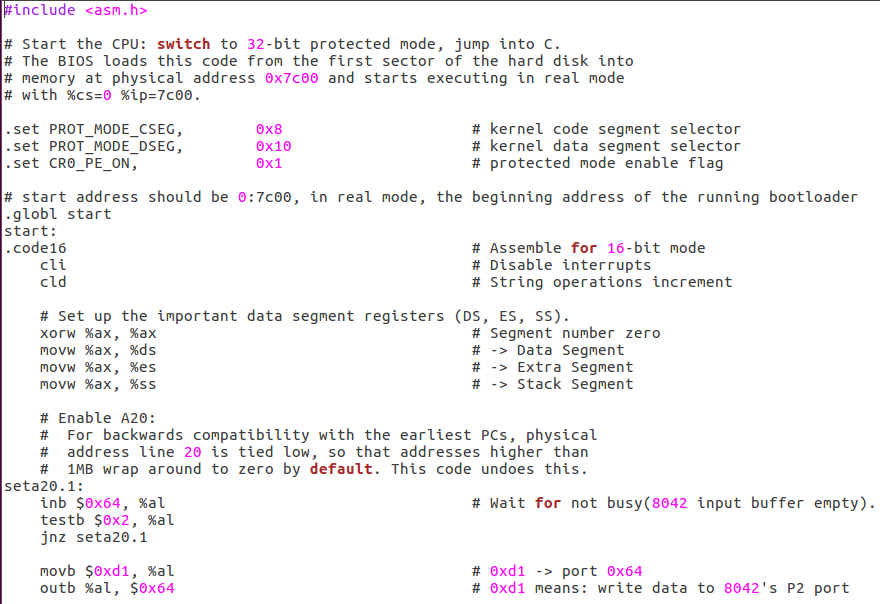


并执行make debug命令，得到q.log文件

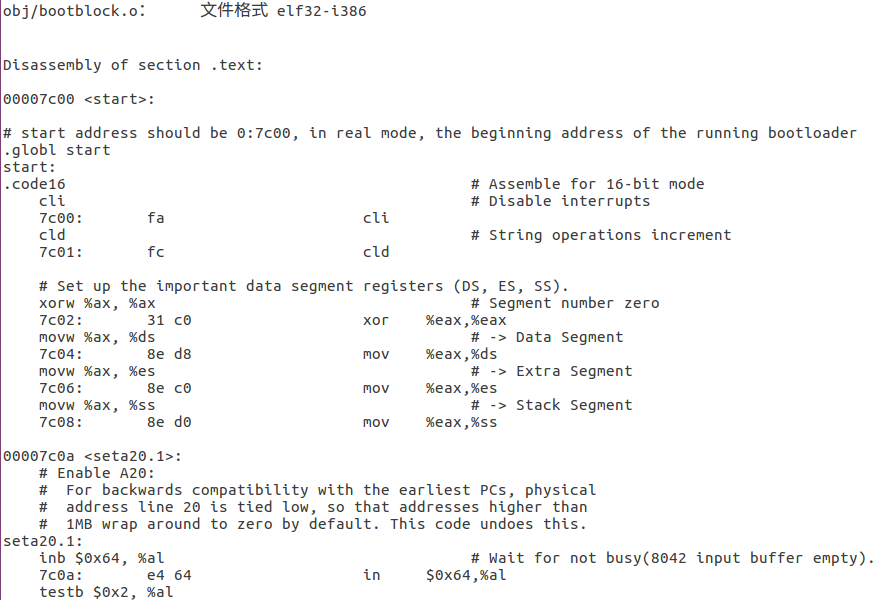
q.log文件：



Bootasm.S文件：



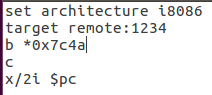
Bootblock.asm文件：



q.log中的代码与bootasm.S和bootblock.asm代码相同。

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。

将gdbinit文件修改如下：



执行如下命令：



运行结果为：



**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

1. 为何开启A20，以及如何开启A20

开启A20原因：初始时A20为0，只能访问1MB地址，而将A20地址线置1后，才可能访问4G内存。

如何开启：

从`%cs=0 $pc=0x7c00`，进入后首先清理环境：包括将flag置0和将段寄存器置0

.code16

cli

cld

xorw %ax, %ax

movw %ax, %ds

movw %ax, %es

movw %ax, %ss

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: # 等待8042键盘控制器不忙

inb $0x64, %al #

testb $0x2, %al #

jnz seta20.1 #

movb $0xd1, %al # 发送写8042输出端口的指令

outb %al, $0x64 #

seta20.1: # 等待8042键盘控制器不忙

inb $0x64, %al #

testb $0x2, %al #

jnz seta20.1 #

movb $0xdf, %al # 打开A20

outb %al, $0x60 #

2.如何初始化GDT表

初始化GDT表：一个简单的GDT表和其描述符已经静态储存在引导区中，载入即可

lgdt gdtdesc

3.如何使能和进入保护模式

进入保护模式：通过将cr0寄存器PE位置1便开启了保护模式

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

4.将CRO的第0位置1  
movl %eax, %cro  
5.长跳转到32位代码段，重装cs和EIP  
ljmp $PROT\_MODE CSEG, Sprotcseg.  
6.重装DS,ES等段寄存器等  
movw  $PROT MODE DSEG, %ax   
movw  %ax, %ds  
movw %ax, %es  
movw %ax. %fs  
movw %ax, %gs  
movw %ax, %ss

7.转到保护模式完成，进入boot主方法。

movl  $oxo, %ebp

movl $start, %esp

call bootmain