姓名：张磊

学号：171491311

班级：硬件二班

实验题目：实验一

**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

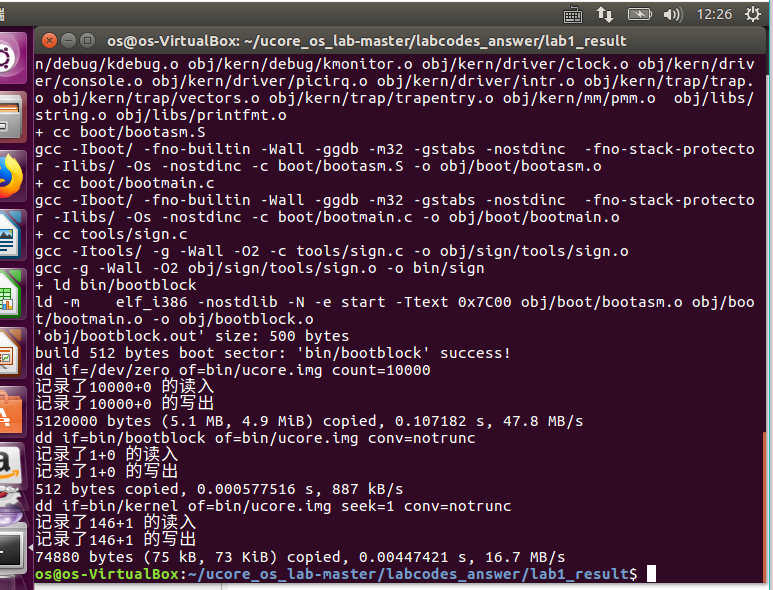
些工作。为此，我们需要完成一个能够切换到x86的保护模式并显示字符的bootloader，为启动操作系统ucore做准备。lab1

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。

**实验题目：**

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)

打开终端：输入make V=



1、首先把C的源代码进行编译成为.o文件，也就是目标文件  
2、ld命令将这些目标文件转变成可执行文件，比如此处的bootblock.out

3、dd命令把bootloder放到ucore.img count的虚拟硬盘之中

4、还生成了两个软件，一个 是Bootloader,另一个是kernel。

1. 使用qemu执行并调试lab1中的软件
2. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

由于BIOS是在实模式下运行的，因此需要在tools/gdbinit里进行相应设置，将其修改为:

**set arch i8086**

**target remote: 1234**

之后再执行make debug，就可以使用gdb单步追踪BIOS的指令执行了；具体调试结果如下图所示；有图可见在刚初始化的时候，cs，eip寄存器的数值分别被初始化为0xf000, 0xfff0, 即第一个执行的指令位于内存中的0xffff0处，该指令是一条跳转指令，跳转到BIOS的主题代码所在的入口；如图所示，使用GDB进行调试可以很方便地观察指令执行过程中的所有寄存器的数值变化；

2. 在初始化位置0x7c00设置实地址断点,测试断点正常。0x7c00是bootloader的入口位置，此时CPU仍然处于实模式下，因此只需要设置实地址断点在0x7c00处即可，此时需要的tools/gdbinit文件如下：

**set arch i8086**

**target remote: 1234**

**b \*0x7c00**

**continue**

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

1. 分析bootloader进入保护模式的过程。

第一步：屏蔽中断，设置串地址增长方向，把ds，es，ss寄存器置0

cli:中断允许标志IF为1时CPU能响应外部的可屏蔽中断请求,但是对非屏蔽不起作用 ，设置用STI 清除是CLI，这里清除，表示不响应可屏蔽中断

CLD：DF方向标志 为1 串按减 为0 串按加 STD置 CLD清DF

设置ds，es，ss寄存器为0.这里用了一个技巧，不管ax寄存器初始化为什么内容，通过xor ax,ax我都可以让ax置0

第二步：开启A20

这部分知识，在实验指导书上有详细说明，这里对这部分内容进行详细阐述：

首先，先不要管A20是什么，我们先了解一下8086结构的历史。之前的内存空间是比较小的，然后一开始的8086的地址线是20位，所以按照2的20次方是1M，所以按理来说，“按理”是指这20位二进制数的值就作为byte的地址的话，可以的寻址范围就是0-1M的范围，然而当时的寄存器是16位，所以不得不采用另外一种寻址方式：一个16位寄存器表示基地址 \* 16 + 另一个16位寄存器表示的偏移地址，这样计算，最大的寻址空间是 0xffff0（0xffff左移了四位） + 0xffff，最后结果是0x10ffef，大约是1088KB，那么这就比1024KB还要大，举个例子，当你的地址通过上面的计算方式得到0x100001这个地址，那么因为只有20根地址线，所以这个地址的最高位1根本无法表示出来，会发生“回卷”，也就是会获取地址为0x00001处的值。但是当时这个问题对于使用没有影响

但是后来，随着内存空间的不断增大，地址线也逐渐增加到32位，为了保持向下兼容（至今未理解这个向下兼容是具体表示什么意思），他们采取的做法是在第20根地址线（A20）上做了一个开关，当A20被使能时，它是一个正常的地址线，当他被disable时，它永远为0，所以这就引入了在一开始A20使能的问题，在保护模式下，要访问高端内存，一定要打开这个开关，否则第21个bit总是为0，那只能访问奇数M的空间了

在了解A20如何打开之前，先对这个体系的地址空间做一个了解：在一开始只有1M内存时，这个部分内存是被分为低端的640KB的常规内存，和高端的384KB的内存，这部分内存一开始是被设计用来作为ROM和系统设备的地址区域。（好像是IBM当时认为内存不会到现在这么大，才把高地址的384KB作这样用）这个设计为之后内存容量的增大带来了麻烦。因为这384KB是ROM和系统设备的地址空间，那么内存会被这部分分开，0-640KB ，1M-最大内存，不连续了，为了解决这个问题，采用了这样的办法：系统加电后，先让ROM有效（即这部分地址空间是给ROM的），此时取出ROM的内容，然后再让RAM有效，把这部分内容保存到RAM的这部分地址空间中，这就是所谓的ROM shadowing

接下来讲A20的相关操作。之前说A20是第21根地址线的值，实际上，是由一个8042键盘控制器来控制的A20 Gate（据说是找不到其他可以控制的地方了），而8042芯片内部有三个端口，其中一个是Output Port，而A20Gate就是Output Port端口的bit 1，所以要控制A20使能，其实就是通过读写端口数据，使得这个bit的值为1

还是需要介绍这个芯片的读写方式：

首先，这个芯片有两个外部端口，0x60h和0x64h，就相当于读写操作的地址了。

读Output Port，需要向64h发送0d0h命令，然后从60h读取Output port的内容

写Output Port，需要先向64h发送0d1h命令，然后向60h写入Output Port的内容

同时我们还需要检查当前缓冲区是否有数据，如果有正在处理的数据，那么肯定需要等待数据处理完才可以，所以还需要知道我们可以通过读取0x64h的数据，获取这个芯片的状态，如果这个状态为0x2（这是规定），说明还有数据没有处理完

实际上还需要知道更多的命令。包括关闭键盘输入等，但是在ucore的实现中，并没有这么麻烦

所以我们可以看这部分代码，很容易就理解，这部分代码就是按照下面的顺序打开A20的：

inb $0x64, %al 就是读取当前状态到 al寄存器，然后testb $0x2, %al，就是检查它当前状态是否标志位0x2被设置了，配合下面这个跳转，当这个标志位为0，或者说是当前输入缓冲区没有数据了，就不跳转继续执行了

movb $0xd1, %al outb %al, $0x64，按照之前说的，先向64h发送0xd1命令， 表示要写

其实不太清楚，这里为什么还要等待8042的input buffer没有数据了，中断已经关闭了呀，应该没有什么会影响到input buffer了吧

movb $0xdf, %al outb %al, $0x60，把0xdf写入0x60，这样A20就打开了

初始化GDT表，使用lgdt gdtdesc 即可，gdtdesc的定义了解一下，定义了空描述符，数据段和代码段

进入保护模式，就是让cr0寄存器的PE为1

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

通过长跳转，更新CS寄存器的基地址

ljmp $PROT\_MODE\_CSEG, $protcseg

其中protcseg是一个label

这里还要注意PROT\_MODE\_CSEG和PROT\_MODE\_DSEG，这两者分别定义为0x8和0x10，表示代码段和数据段的选择子，注意段选择子的结构，前13位是index，正好这里分别对应1和2（和之前全局描述符表的顺序一致），然后后三位是0，表示全局的，而且dpl为0

设置段寄存器，并建立堆栈

注意这里建立堆栈，ebp寄存器按理来说是栈帧的，但是这里并不需要把它设置为0x7c00，因为这里0x7c00是栈的最高地址，它上面没有有效内容，而之后因为调用，ebp会被设置为被调用的那个函数的栈的起始地址，这里就不用管它了。

1、为何开启A20,以及如何开启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。在保护模式下 A20地址线控制是要打开的。为了使能所有

地址位的寻址能力,必须向键盘控制器8042 发送一个命令。键盘控制器

8042将会将它的的某个输出引脚的输出置高电平,作为A20地址线控制的

输入。一-旦设置成功之后,内存将不会再被绕回(memorywrapping)这样我们

就可以寻址intel 80286 CPU支持的16M 内存空间，或者是寻址intel 80386

以上级别CPU支持的所有4G内存空间了。

3、如何使能进入保护模式?   
通过长跳转指令  
ljmp $PROT\_ MODE\_ CSEG, $protcseg进入了保护模式。  
进入保护模式之后还有一个步骤:把所有的数据段寄存器指向上面的GDT  
描述符表中的数据段( 0x10)。