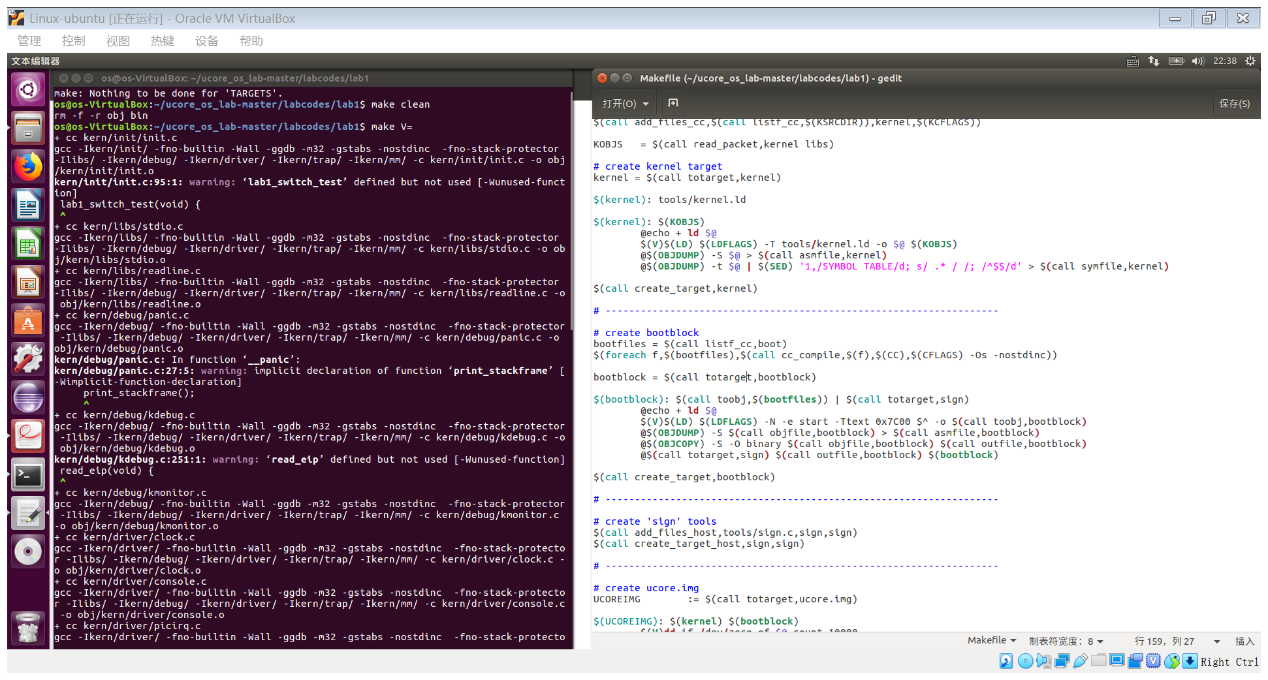
# 实验一

## 练习1：

输入make V=得到如下结果



我们发现ucore.img的生成和bootload和kernel的生成为基础

其中+cc （gcc指令）后接kernel或kern的指令是编译这些文件，生成.o文件，之后+ld（gcc指令）是链接这些文件

Bootload需要bootlock和sign作为工具，所以先编译和链接这两类文件方法同上。

之后编译和链接生成bootload，成功之后展示是如下信息

之后通过这些文件生成ucore.img。在makefile中就是带dd开头的语句。基本都是对磁盘写入操作。指令如下：

图片包含 户外

描述已自动生成

每一条dd开头的语句，也就是makefile中create ucore.img中的语句，分别代表含义

创建大小为10000个快的ucore.img,每块为512字节全部初始为0；

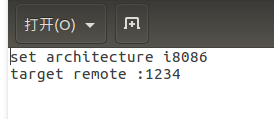
把bootlock的内容写入第一块；

从的二块开始写入kernel的内容

## 练习2：

### 1

a，修改tools/gdbinit文件如下：

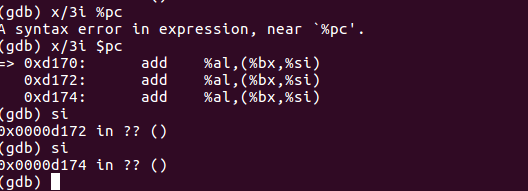


b，打开终端，进入lab1目录，输入指令 make debug

图片包含 屏幕截图, 监视器, 室内, 屏幕

描述已自动生成

c，单步执行，输入si可以查看bios的执行过程中cs和pc和内存地址变化，x/ni $pc (n为数字)可以反汇编，得出n步的汇编语言



由上可得，bios存储在内存中，cpu执行从0x0000fff0。

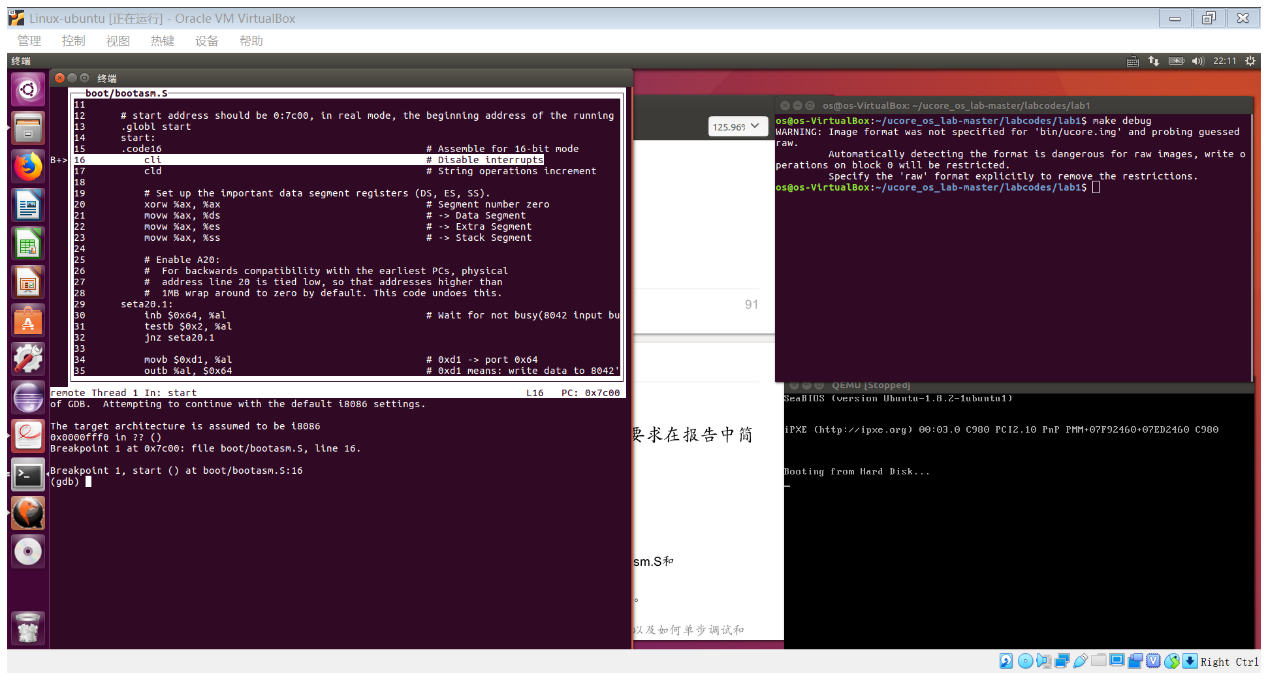
### 2、3

a，修改tools/gdbinit文件如下：

图片包含 屏幕截图

描述已自动生成

b，打开终端，进入lab1目录，输入指令 make debug



c，单步执行，输入si可以查看bios的执行过程中cs和pc和内存地址变化，x/ni $pc (n为数字)可以反汇编，得出n步的汇编语言

图片包含 屏幕截图

描述已自动生成

由上可知，0x7C00为主引导程序的入口地址，代码与bootasm.S一致。反汇编代码也与bootasm.s一致。

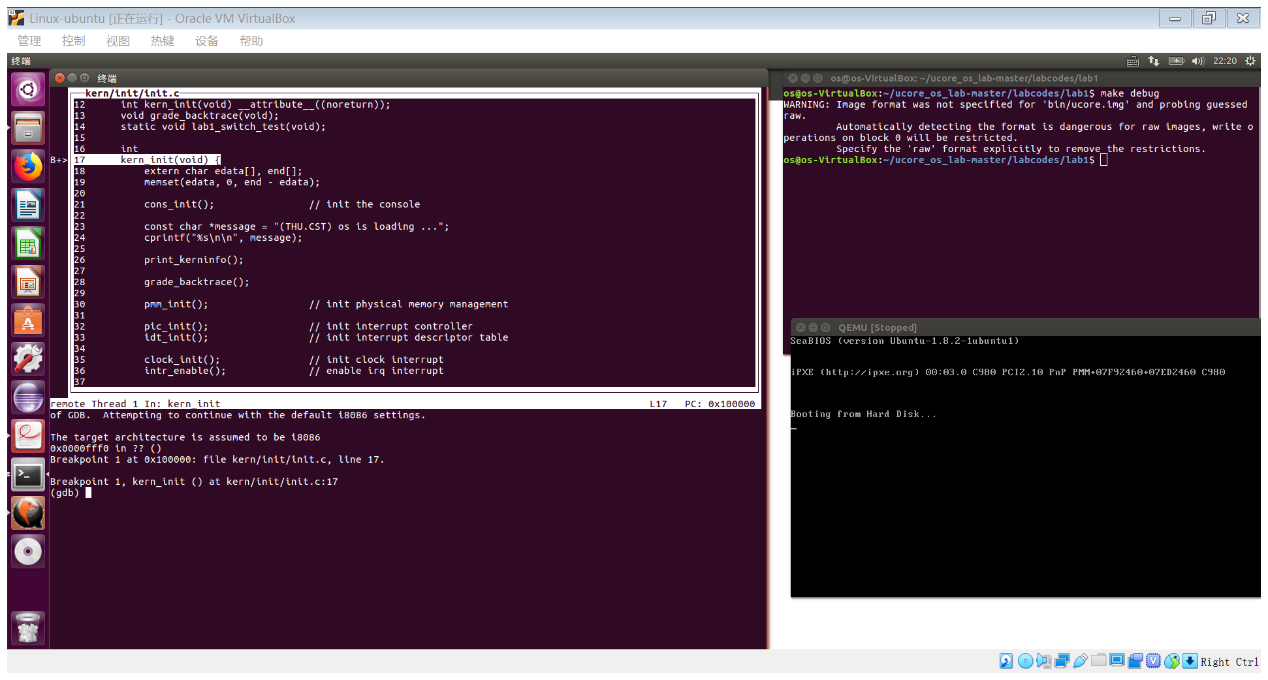
### 4

a，修改tools/gdbinit文件如下：

图片包含 屏幕截图

描述已自动生成

b，打开终端，进入lab1目录，输入指令 make debug



由上得知在内核入口处增加了断点，代码执行停在kern\_init函数。

## 练习3

a为何开启A20,以及如何开启A20

在i8086时代，CPU的数据总线是16bit，地址总线是20bit，寄存器是16bit，因此CPU只能访问1MB以内的空间。因为数据总线和寄存器只有16bit，如果需要获取20bit的数据, 我们需要做一些额外的操作，8086采用偏移4位的方法，使之能访问20bit的数据。理论上，20bit的地址可以访问1MB的内存空间(0x00000 - (2^20 - 1 = 0xFFFFF))。但在实模式下, 这20bit的地址理论上能访问从0x00000 - (0xFFFF0 + 0xFFFF = 0x10FFEF)的内存空间。也就是说，理论上我们可以访问超过1MB的内存空间，但越过0xFFFFF后，地址又会回到0x00000。上面这个特征在i8086中是没有任何问题的(因为它最多只能访问1MB的内存空间)，但到了i80286/i80386后，CPU有了更宽的地址总线，数据总线和寄存器后，这就会出现一个问题： 在实模式下, 我们可以访问超过1MB的空间，但我们只希望访问1MB以内的内存空间。为了解决这个问题， CPU中添加了一个可控制A20地址线的模块，通过这个模块，我们在实模式下将第20bit的地址线限制为0，这样CPU就不能访问超过1MB的空间了。进入保护模式后，我们再通过这个模块解除对A20地址线的限制，这样我们就能访问超过1MB的内存空间了。默认情况下，A20地址线是关闭的(20bit以上的地址线限制为0)，因此在进入保护模式(需要访问超过1MB的内存空间)前，我们需要开启A20地址线(20bit以上的地址线可为0或者1)。

如何初始化GDT表

从bootasm.s可以看出这里所有GDT表项初始化为全段,除了空段

如何使能和进入保护模式

1. 等待8086 Input buffer为空;

2. 发送Write 8086 Output Port (P2)命令到8086 Input buffer;

3. 等待8086 Input buffer为空;

4. 将8086 Output Port(P2)得到字节的第2位置1,然后写入8086Input buffer;