# Lab1

**一、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）**

1.分析



定义变量，用UCOREIMG这个变量表示$(call totarget,ucore.img)



主体规则： (UCOREIMG):(UCOREIMG): (UCOREIMG):(kernel) (bootblock)分别是目标：前置条件，所以需要用到两个依赖文件，分别就是(kernel)和$(bootblock)



就是将/dev/zero下面的块文件拷贝到$@这个地方

也就是$(UCOREIMG)所在的空间，然后指定了拷贝10000个block。

这里要特别说明下/dev/zero代表的意义，/dev/zero，是一个输入设备，你可用它来初始化文件。该设备无穷尽地提供0，可以使用任何你需要的数目——设备提供的要多的多。他可以用于向设备或文件写入字符串0。所以这个指令相当于初始化ucore.img所要用的空间，整个通式相当于

dd if=源文件位置 of=目标文件位置；



这条指令相当于将$(bootblack)的内容拷贝到$(UCOREIMG)这里，且选项conv=notrunc我们查询可得知是表示不截短输出文件地输出

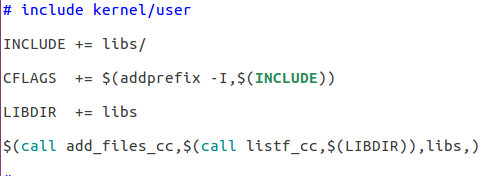


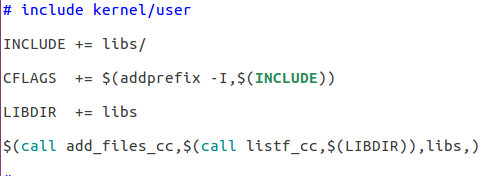
同样，相当于将$(kernel)的内容拷贝到$(UCOREIMG)这里，且选项内容是不截短输出文件地输出以及从输出文件开头跳过1个块后再开始复制。

这里之所以要用到seek＝1的原因就是因为前面第二条指令bootblock已经用到了一个块的空间了，为了不覆盖掉其内容，所以要选择跳过这个块区域的内容进行复制。

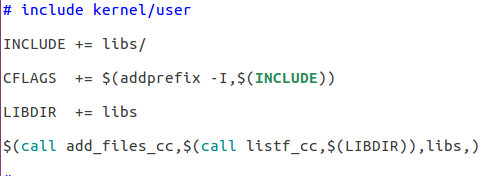


相当于调用函数

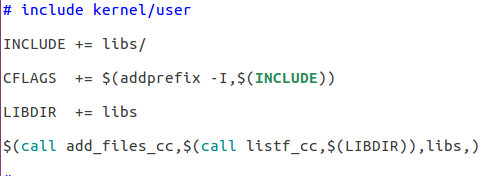
2.分析 



将libs/这个目录下的内容追加到INCLUDE这个变量的内容后面，相当于加上了libs/的内容到INCLUDE中，这个INCLUDE代表了一个变量，在makefile的87行include tools/function.mk 代表了包含的文件



将libs/追加到变量CFLAGS后，也就是直接指定了头文件的路径位置



追加到变量后

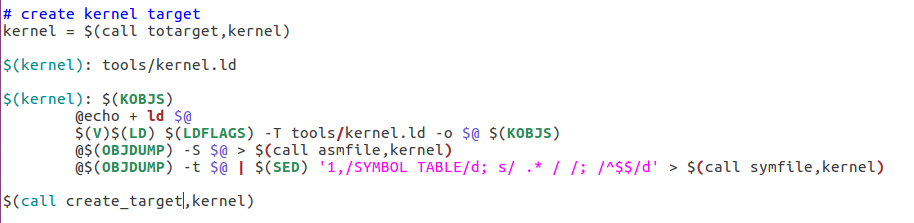
3.

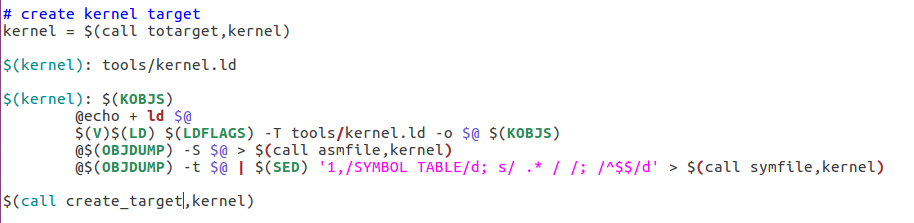


KINCLUDE和KSRCDIR两个变量的作用就是将kern目录也就是内核文件添加进makefile工程中

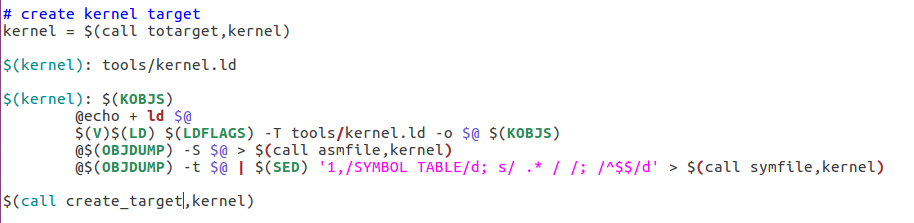
KCFLAGS的作用是将-I选项加到KINCLUDE的所有目录下

KOBJS的作用是将libs的内容追加到read\_packet这个变量中

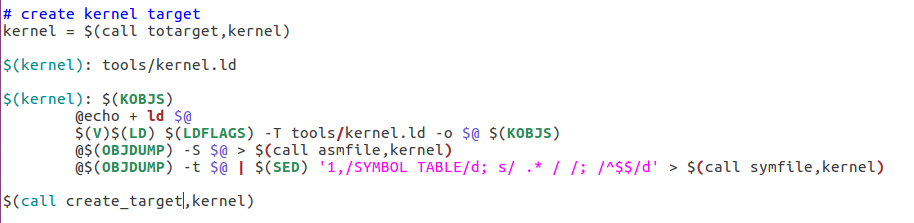
4. 



定义了kernel变量是call了一个kernel到totarget中



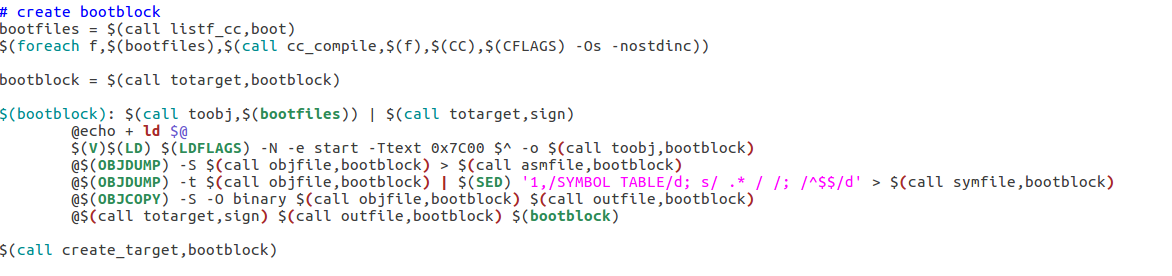
等价于@echo + ld $(kernel)



等价于@ld -m $(shell $(LD) -V | grep elf\_i386 2>/dev/null) -nostdlib -T tools/kernel.ld -o tools/kernel.ld $(KOBJS)

第一条定义了kernel变量是call了一个kernel到totarget中

5.



前面三条指令主要是变量的相关定义，bootfiles主要调用了两个函数关系，实现了将boot目录下的文件加上了.c或者.S的后缀

第二条makefile的功能是将boot下的文件与GCC的编译模版规则进行比对，查看到底是否有语法错误

而第三条是将bin 这个前缀加到boot目录文件前参与编译工作。

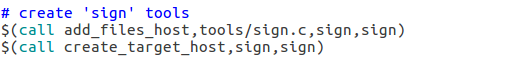
接着是主要的部分，bootblock的依赖文件主要这里分成了两个，两个之一都可以满足条件

前一个的作用是调用了toobj函数，实现了在.o文件前加obj 前缀的功能

而第二个是在依赖文件sign前加bin。所以此处还应该涉及到另一个bootblock的依赖文件sign，下面会一并分析。

生成规则的第一条是输出链接文件，第二条是准备将处理后的bootblock加载到内存地址0x7c00处开始运行

6.



第一条主要是将tools/sign下的文件加载到makefile工程中，主要用到了一系列在function.mk中定义的函数，具体可以查看

而第二条指令是调用了create\_target\_host函数

**二、使用qemu执行并调试lab1中的软件**

1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

①修改lab1/tools/gdbinit，内容为：

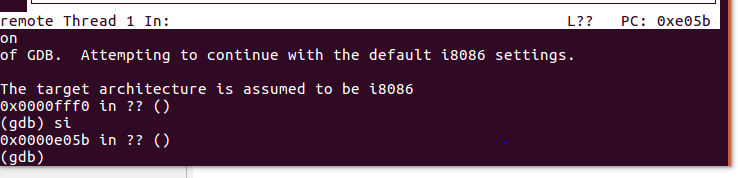
Set architecture i8086

Target remote :1234

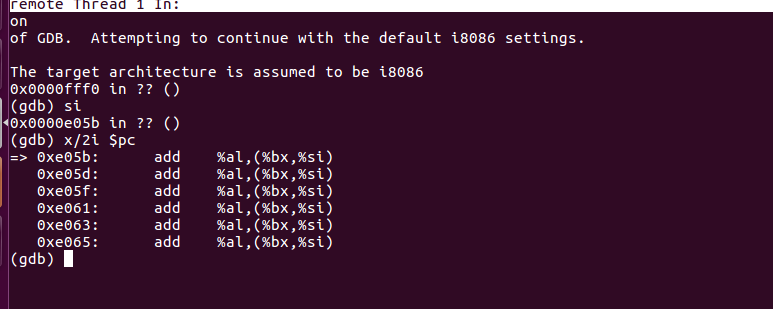
②在lab1目录下执行make debug

③在看到gdb的调试界面后，执行命令si

即可单步跟踪BIOS了

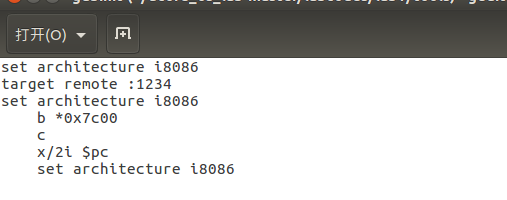


④在gdb界面下，执行x/2i $pc来查看BIOS的代码（此处我按了三次回车）

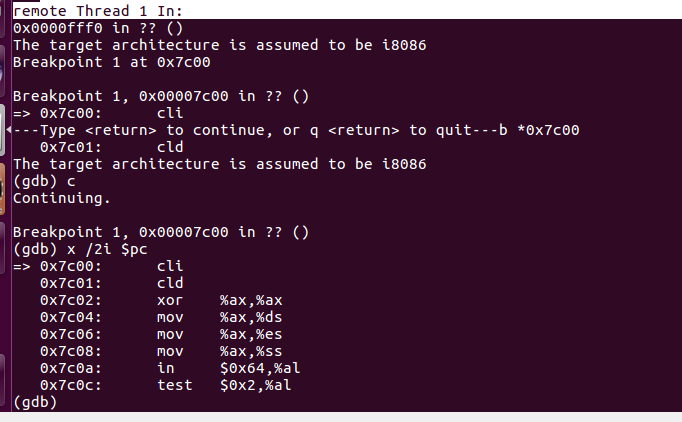


2.在初始化位置0x7c00设置实地址断点,测试断点正常。

①将gdbinit文件内容改成如下形式：

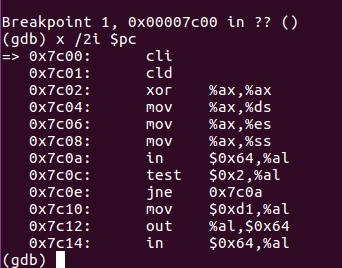


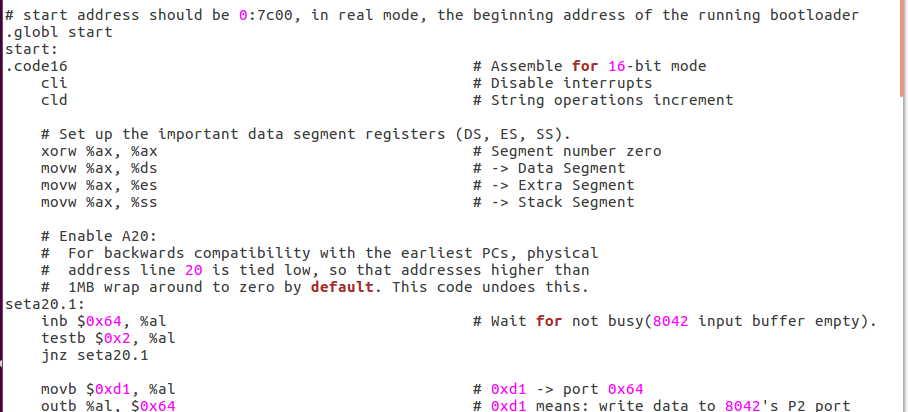
②运行make debug可得：



断点正常

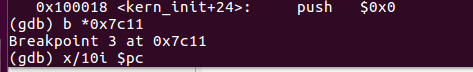
3.从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较。

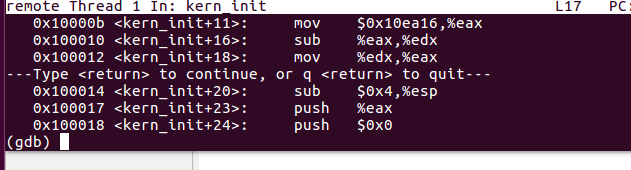




比较结果：相同

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。

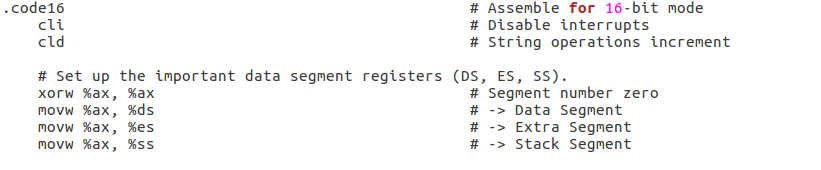




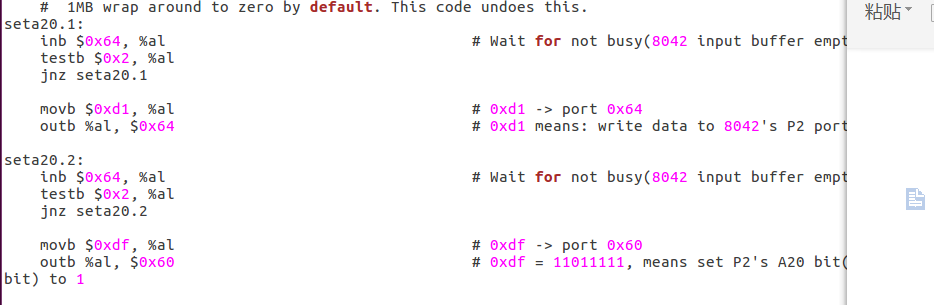
**三、分析bootloader进入保护模式的过程**



1. 清理环境：包括将flag置零和将段寄存器置零



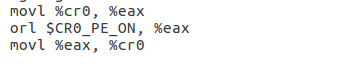
1. 开启A20：通过将键盘控制器上的A20线置于高电位，全部32条地址线可用，可用访问4G的内存空间。



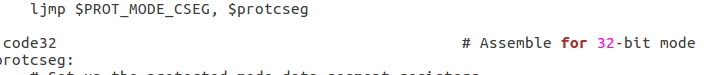
1. 初始化GDT表



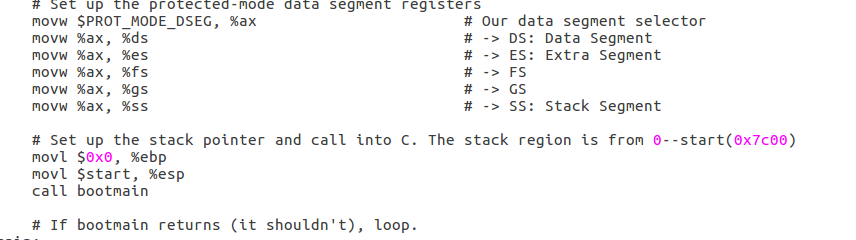
1. 进入保护模式：通过将cr0寄存器PE位置1便开启了保护模式



1. 通过长跳转更新cs的基地址



1. 设置段寄存器，并建立堆栈



1. 转到保护模式完成，进入boot主方法

