操作系统lab1

**班级：硬件一班**

**姓名：于华宇**

**学号：171491120**

系统软件启动过程

## 实验目的

我们将通过另外一个简单的软件-bootloader来完成加载并运行操作系统的工作。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，可以了解到：

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

ucore OS软件：编译运行ucore OS的过程，ucore OS的启动过程，调试ucore OS的方法

## 实验内容

1. 理解通过make生成执行文件的过程

需要通过静态分析代码来了解操作系统镜像文件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)

具体操作：

1. 打开Makefile，找到相关命令区域

Makefile作用：定义一系列规则来指定源文件的编译顺序，拥有特定语法规则，支持函数定义和函数调用，能够直接继承操作系统各种命令。

由ucore.img的命令可知，kernel和BootLoader文件是ucore.img所依赖的文件，所以需要对kernel和BootLoader进行详细学习分析。

分析方法主要有：直接分析代码内容和运行make命令，查看编译内容。两者对照得出Makefile中相关命令和参数的含义，并得出命令导致的结果。

查阅资料：-g 产生gdb调试

-c 编译生成机器码（不可执行）

-o 编译生成可执行文件

-t 把目标文件的时间更新，即把目标文件改成已编译状态

-s 生成汇编代码

$@ 代表当前规则中，触发命令后，会被执行的目标

ld：把一定量的目标文件跟档案文件连接起来。一般在编译一个程序时最后一步就是运行“ld”

CC:表示c编译器版本

LDFLAGS：表示库参数库选项

@只显示命令结果，忽略命令本身

-O2 编译时使用二级优化

-Os相当于-O2.5优化

-n 不执行参数，只是打印命令

-e 指明环境变量的值覆盖Makefile中定义的变量值

objdump –S 是输出源代码和反汇编（把机器语言转为汇编语言）出来的指令对照的格式。显示文件的header信息，还显示对应的十六进制文件代码。

objdump –T 显示文件的符号表入口

dd：创建指定大小的文件

if：读取file替换标准输入

of：写入file替换标准输出

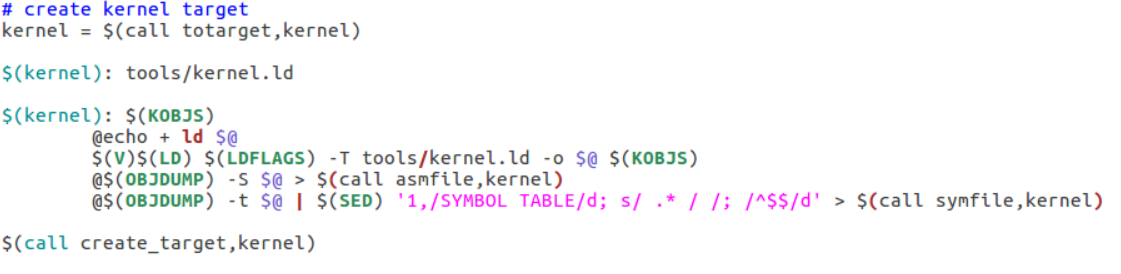
count：赋值多少输入块

conv:转换参数，用指定的参数转换文件

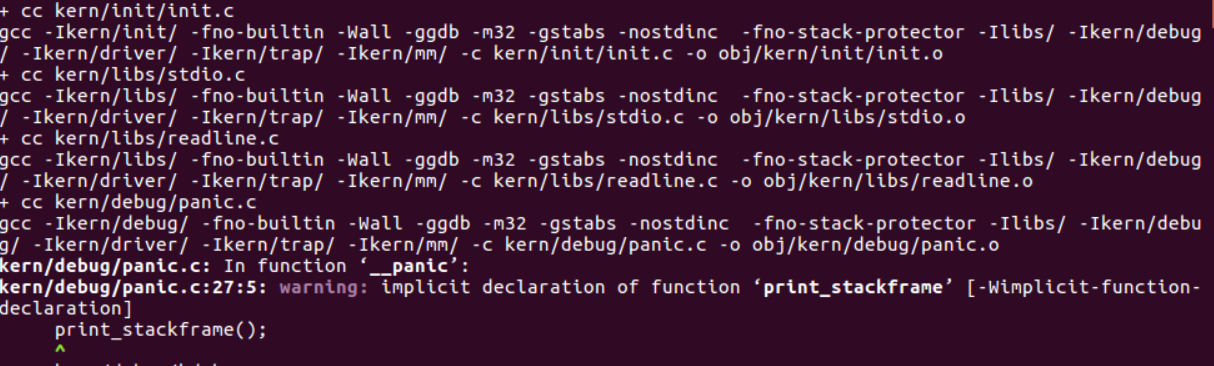
notrunc：不截短输出文件

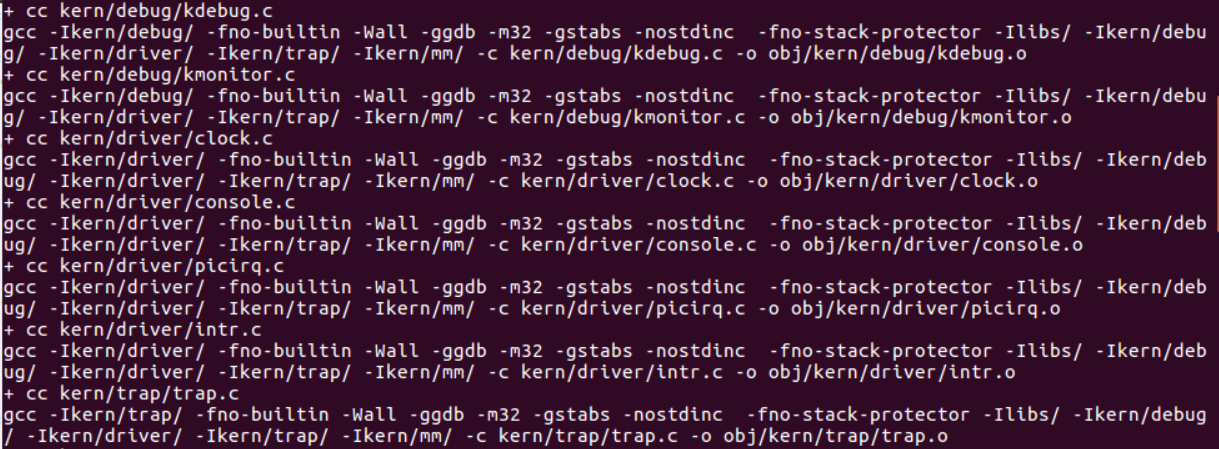
kernel：

Makefile中内容：

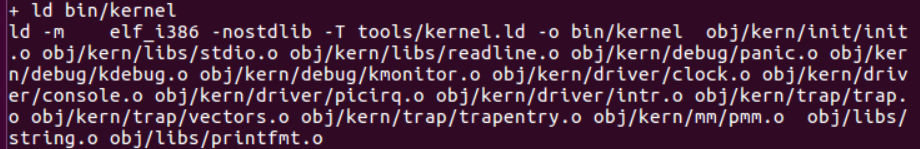


编译内容：









变量对照后我们发现$@（当前会被执行的目标）对应为bin/kernel，根据Makefile中变量的赋值，V对应@，LD对应$(GCCPREFIX)ld，LDFLAGS对应-m，KOBJS对应$(call read\_packet,kernel libs)，OBJDUMP对应$(GCCPREFIX)objdump，odjdump是Linux下的反汇编目标文件或者可执行文件的命令，SED对应sed。

第一条命令：@echo + ld $@

@echo 相当于输出，会打印语句中除了@echo以外的东西。

第二条命令：$（V）$(LD)$(LDFLAGS) –T tools/kernerl.ld –o $@ $(KOBJS)

用于编译链接文件。利用-T命令提供链接脚本，tools/kernerl.ld为链接脚本，执行ld命令，编译连接文件。

第三条命令：@$(OBJDUMP) –S $@ > $(call asmfile,kernel)

对文件进行反汇编，即把已经编译成机器语言的文件转为汇编语言。将结果存到call函数(对kernel执行asmfile函数)执行后的文件中。

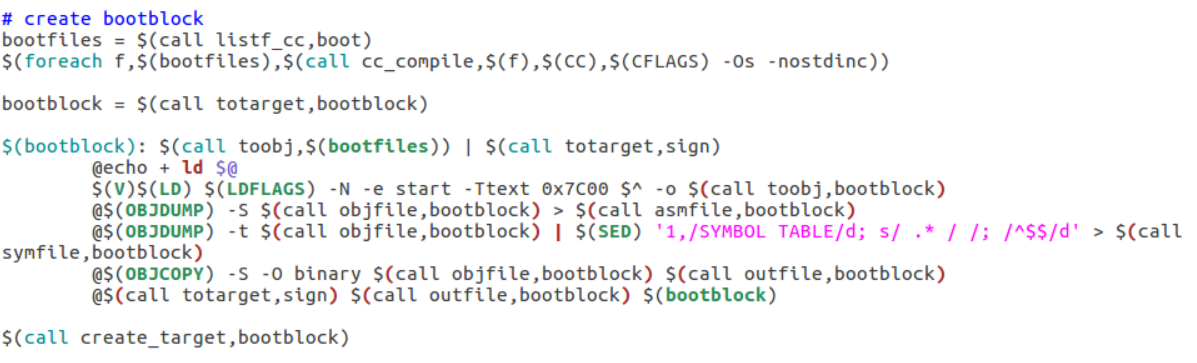
第四条命令:@$ (OBJDUMP) –t $@ | $(SED) ‘1,/SYMBOL TABLE/d; S/ .\*/ /; /^$$/d’> $(call symfile,kernel)

或者对$@执行objdump-t,显示文件符号表入口的指令，或者执行sed命令替换文件，执行call函数（对kernel执行 symfile 函数）

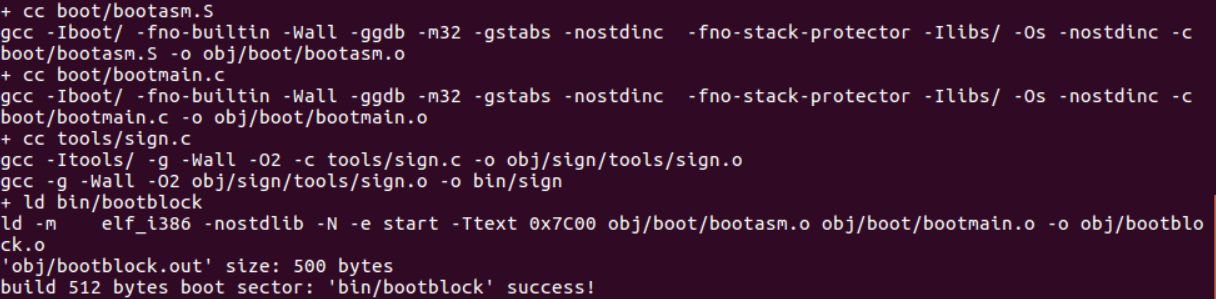
第五条命令:$(call create\_target,kernel)

执行call函数（对kernel执行create\_target）

BootLoader：



编译内容：



在执行命令前，编译文件bootasm.s bootmain.c sign.c 成为可执行文件

1、@echo + ld $@：

打印 +ld $@(bin/bootblock）

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

V对应@，LD对应$(GCCPREFIX)ld，LDFLAGS对应-m。打印命令-N -e start -Ttext 0x7C00 -e 指明环境变量的值覆盖Makefile中定义的变量值。执行ld命令链接文件。

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

对文件进行反汇编，即把已经编译成机器语言的文件转为汇编语言。执行call函数(对bootblock l执行objfile函数)得到机器语言，使其变为汇编语言。

4、@$(OBJDUMP) -t $(call objfile,bootblock) | $(SED) '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > $(call symfile,bootblock)：

显示bootblock可执行文件的符号表入口或者执行替换命令替换文件

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

运行objcopy命令，通过使用二进制为输出目标，将bootblock的目标文件和outfile文件输出成原始二进制文件。实质上是进行了一回输入目标文件内容的内存转储。

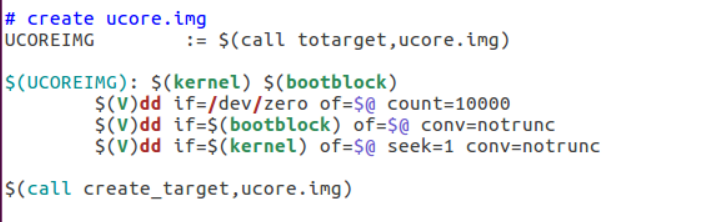
6、@$(call totarget,sign) $(call outfile,bootblock) $(bootblock)：

使用bootblock变量执行outfile，使用sign执行totarget操作。

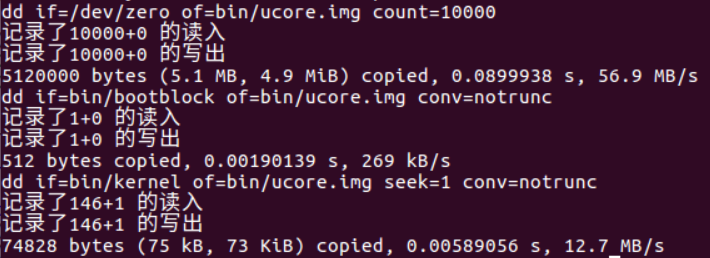
7、$(call create\_target,bootblock)：

用bootblock变量create\_target.

ucore.img：



编译内容：



ucore.img依赖的kernel和bootblock在上面已经处理完毕。

1、$(V)dd if=/dev/zero of=$@ count=10000

执行dd命令创建文件，输入/dev/zero文件中的内容到bin/ucore.img，赋给10000个输入块

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

执行dd命令创建文件，输入bin/bootblock文件中的内容到bin/ucore.img，不截短输出文件

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

执行dd命令创建文件，输入bin/kernel文件中的内容到bin/ucore.img

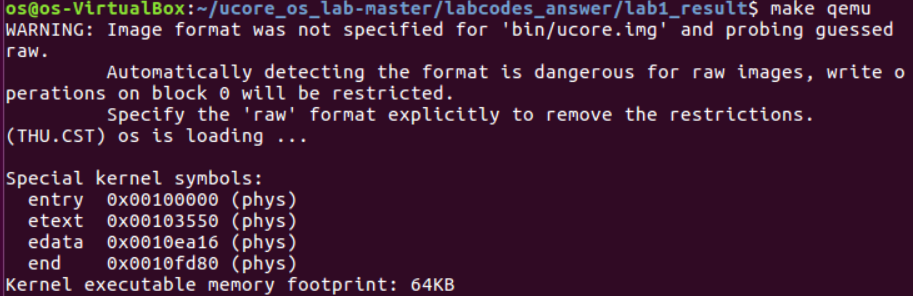
4、$(call create\_target,ucore.img)

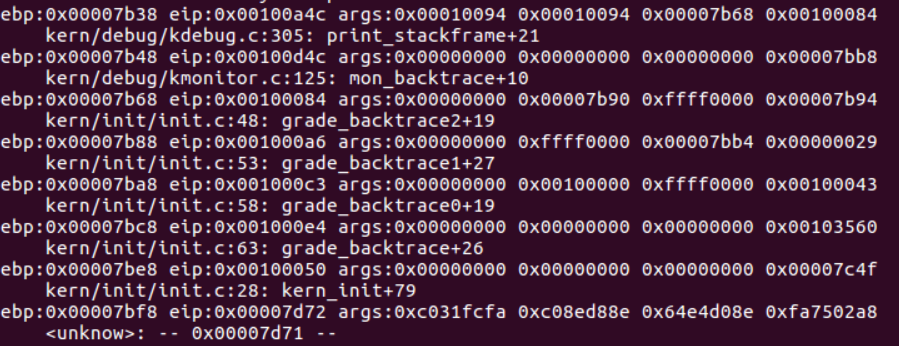
创建ucore.img文件

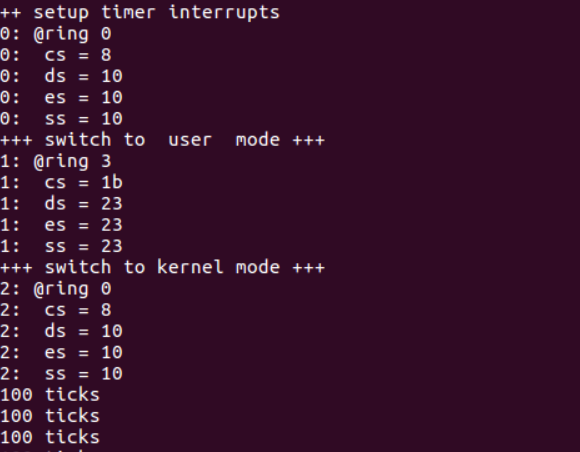
1. 使用qemu执行并调试lab1中的软件。

执行：

进入/home/os/ucore\_os\_lab-master/labcodes\_answer/lab1\_result目录下，输入make指令，编译。再输入make qemu指令运行。







ebp：扩展基址指针寄存器(extended base pointer)　其内存放一个指针，该指针指向系统栈最上面一个栈帧的底部

eip：IP是指令寄存器，存放当前指令的下一条指令的地址。CPU该执行哪条指令就是通过IP来指示的。

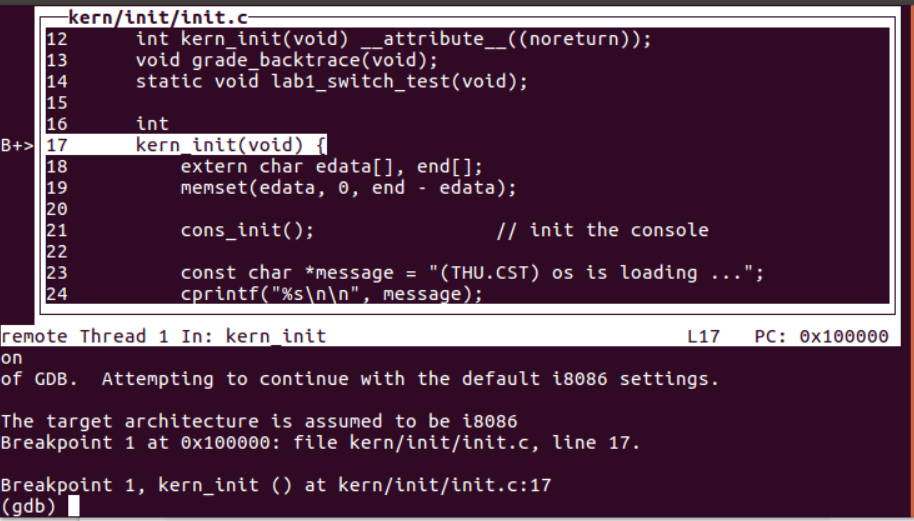
注：当pc机加电开始启动时，处理器在实模式下自检，开始执行物理地址为0Xffff0即ROM-BIOS的起始地址处的代码，一条跳转指令,通过跳转指令跳到BIOS例行程序起始点。BIOS做完计算机硬件自检和初始化。然后把引导设备的第一个扇区加载到地址0x7c00，然后CPU控制权会转移到那个地址继续执行。至此BIOS的初始化工作做完了,进一步的工作交给了ucore的bootloader。

调试：

1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

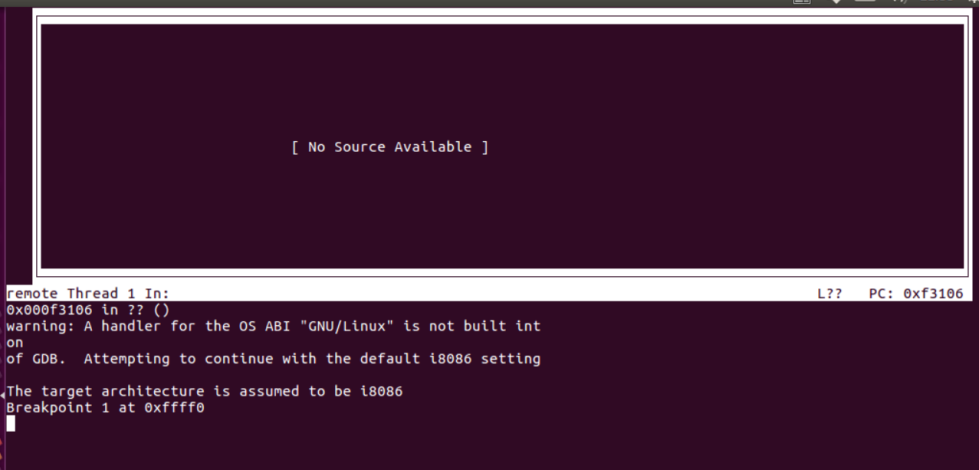
a、首先修改lab1/tools/gdbinit，添加set architecture i8086，target remote：1234

b、在lab1目录下执行 make debug

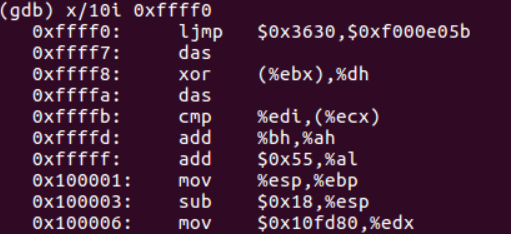


发现第一处断点不在0xffff0处，于是找到gdbinit文件，修改第一个断点为

b \*0ffff0得到下图。发现执行已经跳过0xffff0

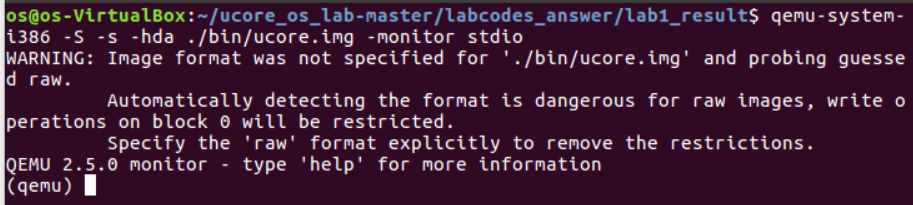


c、经过查询代码得BIOS中部分代码

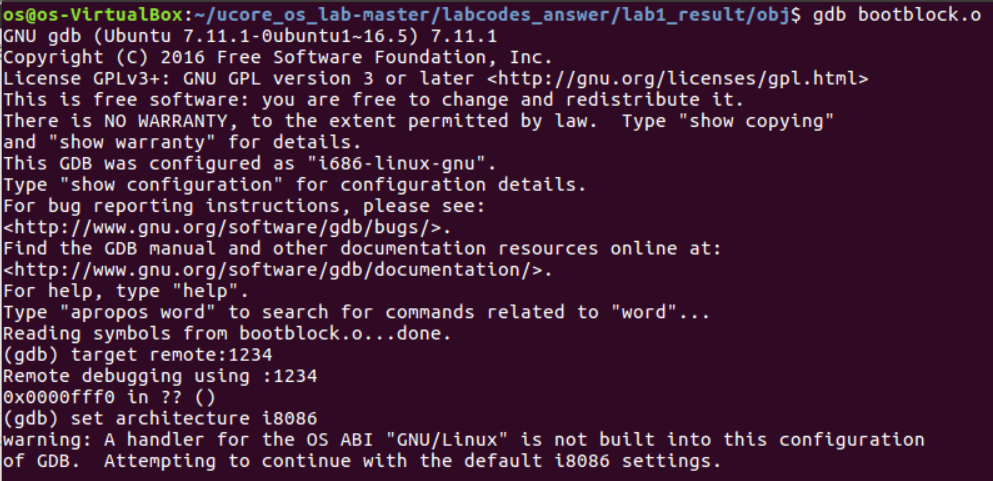


1. 在初始化位置0x7c00设置实地址断点,测试断点正常

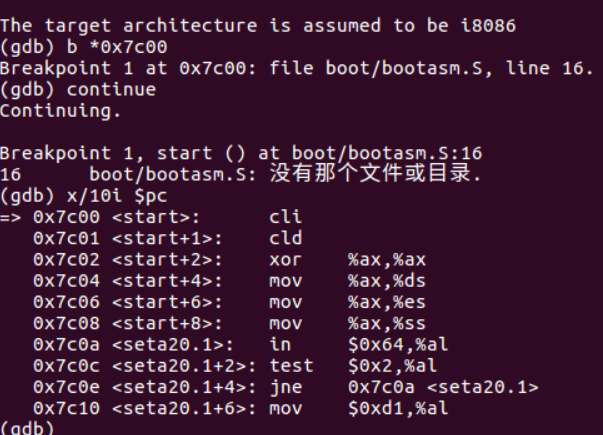
进入qemu模拟器



打开另一个终端，用gdb调试，并连接qemu



设置断点



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。

反汇编代码：

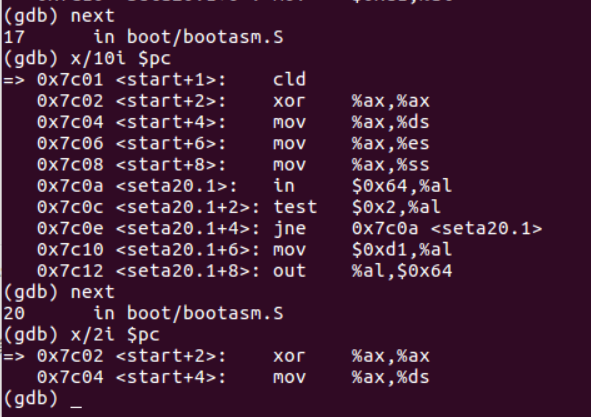
在gdb命令行或配置文件中添加:

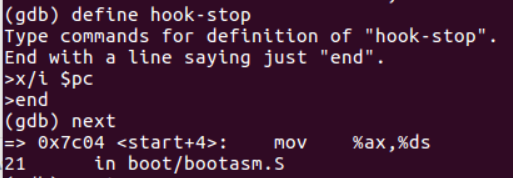
define hook-stop

x/i $pc

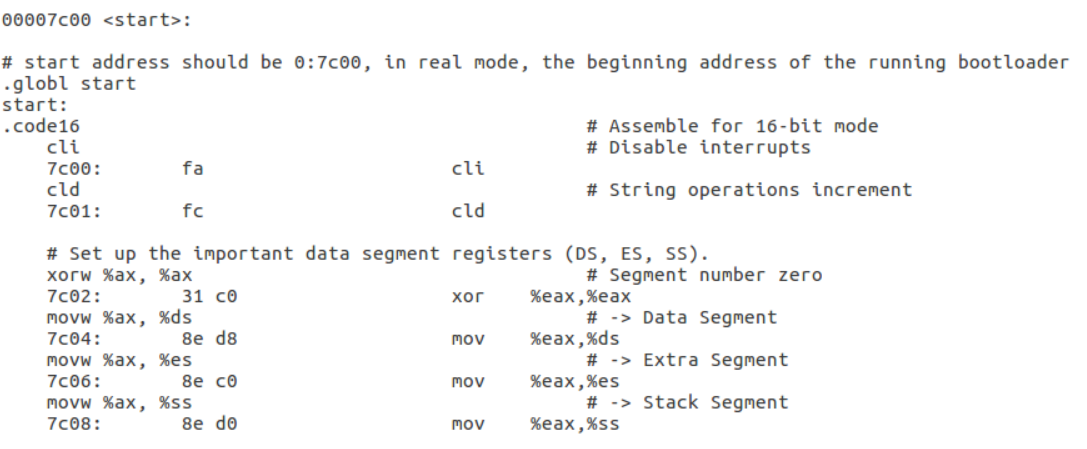
end

跟踪：





bootblock.asm



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。

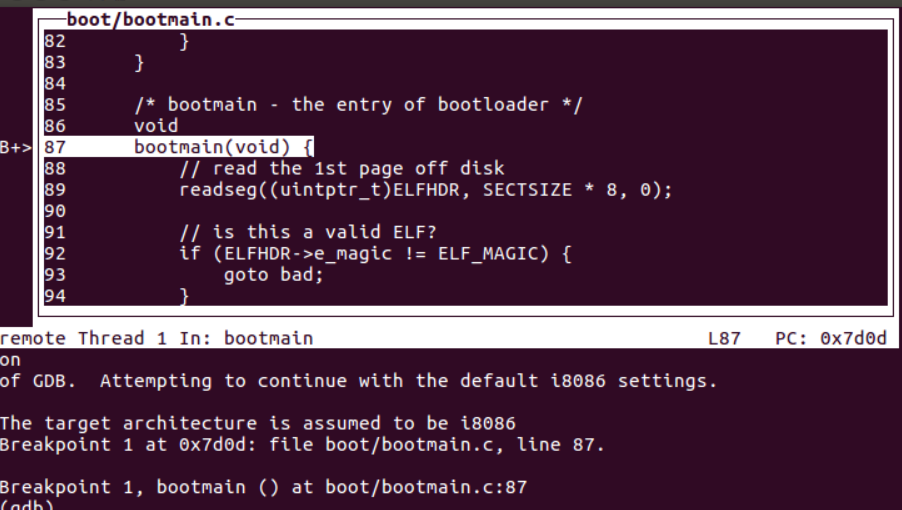
file obj/bootblock.o

target remote :1234

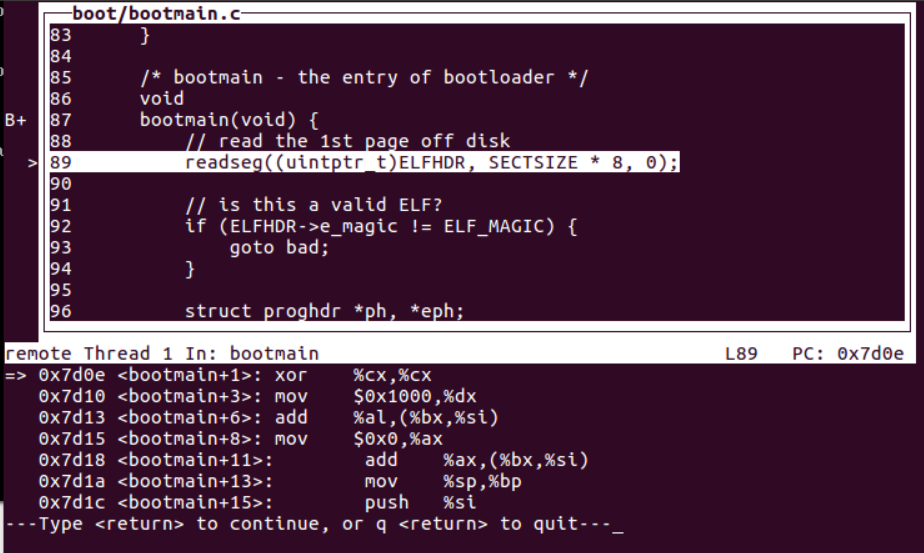
set architecture i8086

break bootmain

continue



输入si，再查看代码



1. 分析bootloader进入保护模式的过程。

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。

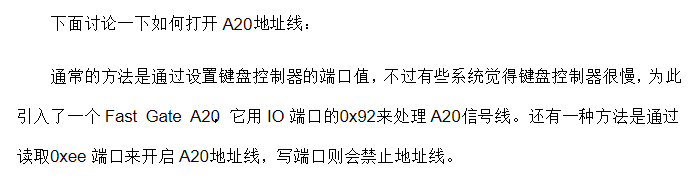
提示：需要阅读小节“保护模式和分段机制”和lab1/boot/bootasm.S源码，了解如何从实模式切换到保护模式，需要了解： 为何开启A20，以及如何开启A20 如何初始化GDT表 如何使能和进入保护模式

1. 实模式与保护模式
2. 实模式 在bootloader接手BIOS的工作后，当前的PC系统处于实模式（16位模式）运行状态，在这种状态下软件可访问的物理内存空间不能超过1MB，且无法发挥Intel 80386以上级别的32位CPU的4GB内存管理能力。 实模式将整个物理内存看成分段的区域，程序代码和数据位于不同区域，操作系统和用户程序并没有区别对待，而且每一个指针都是指向实际的物理地址。这样，用户程序的一个指针如果指向了操作系统区域或其他用户程序区域，并修改了内容， 那么其后果就很可能是灾难性的。通过修改A20地址线可以完成从实模式到保护模式的转换。
3. 保护模式 只有在保护模式下，80386的全部32根地址线有效，可寻址高达4G字节的线性地址空间和物理地址空间，可访问64TB（有 2^14个段，每个段最大空间为2^32字节）的逻辑地址空间，可采用分段存储管理机制和分页存储管理机制。这不仅为存储共享和保护提供了硬件支持，而且为实现虚拟存储提供了硬件支持。通过提供4个特权级和完善的特权检查机制，既能实现资源 共享又能保证代码数据的安全及任务的隔离。

保护模式与实模式差别：一、保护模式提供段间的保护机制，防止程序间胡乱访问地址带来的问题。二、访问的内存空间变大

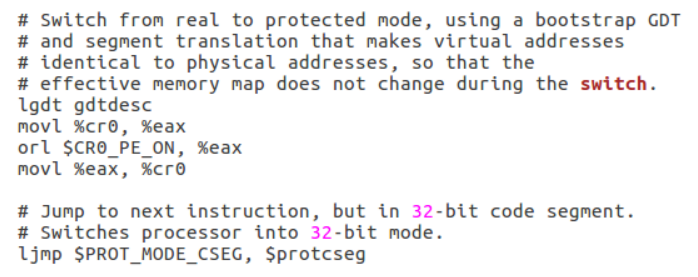
注：为何开启A20，以及如何开启A20，如何初始化GDT表，如何使能和进入保护模式。

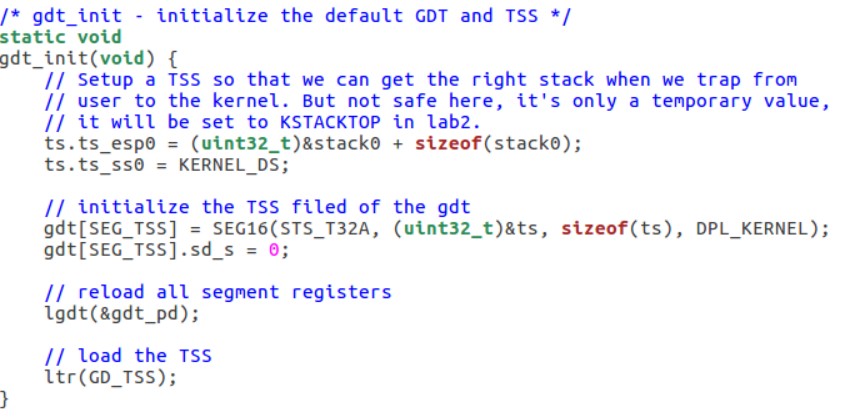
出现80286以后，为了保持和8086的兼容，PC机在设计上在第21条地址线（也就是A20）上做了一个开关，当这个开关打开时，这条地址线和其它地址线一样可以使用，当这个开关关闭时，第21条地址线（A20）恒为0，这个开关就叫做A20 Gate，很显然，在实模式下要访问高端内存区，这个开关必须打开，在保护模式下，由于使用32位地址线，如果A20恒等于0，那么系统只能访问奇数兆的内存，即只能访问0--1M、2-3M、4-5M......，这显然是不行的，所以在保护模式下，这个开关也必须打开。



全局描述符表GDT:

全局描述符表的是一个保存多个段描述符的“数组”,其起始地址保存在全局描述符表寄存器GDTR中。GDTR长48位,其中高32位为基地址,低16位为段界限。由于GDT不能有GDT本身之内的描述符进行描述定义,所以处理器采用GDTR为GDT这一特殊的系统段。注意,全局描述符表中第一个段描述符设定为空段描述符。GDTR中的段界限以字节为单位。对于含有N个描述符的描述符表的段界限通常可设为8\*N-1。在ucore中的boot/bootasm.S中的gdt地址处和kern/mm/pmm.c中的全局变量数组gdt[]分别有基于汇编语言和C语言的全局描述符表的具体实现。





bootloader进入保护模式的过程

先打开A20，再执行lgdt把gdt表的基地址加载进gdtr寄存器。设置寄存机cr0中PE位（保护模式的开关），后执行指令进入保护模式。