#### 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）

为了熟悉使用qemu和gdb进行的调试工作，我们进行如下的小练习：

（1）从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

（2）在初始化位置0x7c00设置实地址断点,测试断点正常。

（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。

（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。

提示：参考附录“启动后第一条执行的指令”

补充材料：

我们主要通过硬件模拟器qemu来进行各种实验。在实验的过程中我们可能会遇上各种各样的问题，调试是必要的。qemu支持使用gdb进行的强大而方便的调试。所以用好qemu和gdb是完成各种实验的基本要素。

默认的gdb需要进行一些额外的配置才进行qemu的调试任务。qemu和gdb之间使用网络端口1234进行通讯。在打开qemu进行模拟之后，执行gdb并输入

target remote localhost:1234

即可连接qemu，此时qemu会进入停止状态，听从gdb的命令。

另外，可能需要qemu在一开始便进入等待模式，则不再使用make qemu开始系统的运行，而使用make debug来完成这项工作。这样qemu便不会在gdb尚未连接的时候擅自运行了。

BIOS首先运行在16位实模式下，第一条指令是ljmp，执行这条指令后会跳到另外一个地方。gdb默认是32位线性地址模式，调试BIOS的16位代码（短地址）需要手动计算地址，计算公式如下：

Linear Addr=(cs<<4)+ip

如果CS=0xf000，EIP=0xe05b，则Linear Address=0xfe05b。

另外，为了正确反汇编16位指令，在gdb中执行

(gdb)set architecture i8086

(gdb)x/16i 0xfe05b

0xfe05b: cmpl $0x0,%cs:-0x2f2c

0xfe062: jne 0xfc792

⋮

1. gdb的地址断点。

在gdb命令行中，使用b \* [地址]便可以在指定内存地址设置断点，当qemu中的cpu执行到指定地址时，便会将控制权交给gdb。

1. 关于代码的反汇编

有可能gdb无法正确获取当前qemu执行的汇编指令，通过如下配置可以在每次gdb命令行前强制反汇编当前的指令，在gdb命令行或配置文件中添加：

define hook-stop

x/i $pc

end

即可。

1. gdb的单步命令。

在gdb中，有next, nexti, step, stepi等指令来单步调试程序，他们功能各不相同，区别在于单步的“跨度”上。

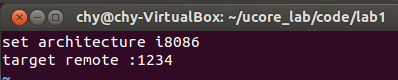
next：单步到程序源代码的下一行，不进入函数。

nexti：单步一条机器指令，不进入函数。

step：单步到下一个不同的源代码行（包括进入函数）。

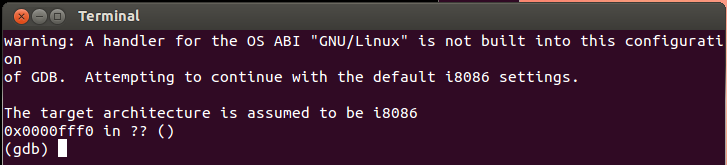
stepi：单步一条机器指令。

答：（1）首先修改tools/gdbinit文件为如下内容：

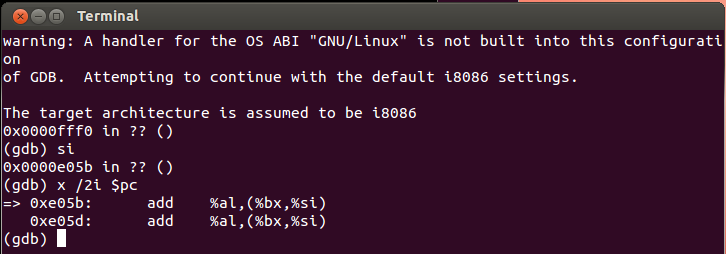


这个文件中保存的是gdb初始化时执行的指令，其中第一条指令表示BIOS进入8086的16位实模式方式，第二条指令的作用是与qemu通过TRP进行连接，端口号缺省为1234。

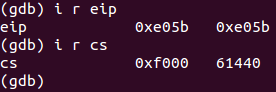
在lab1文件夹内执行make debug，会弹出gdb调试界面如下图：



然后可以在gdb调试窗口中使用si命令进行单步执行，同时使用x /2i $pc命令来打印断点处的两条指令，在这个命令中x是显示的意思，i是指令，/2i表示两条指令，$pc代表EIP寄存器，执行结果如图：

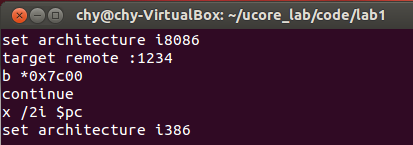


在gdb调试界面使用i r 寄存器可以显示寄存器的值：



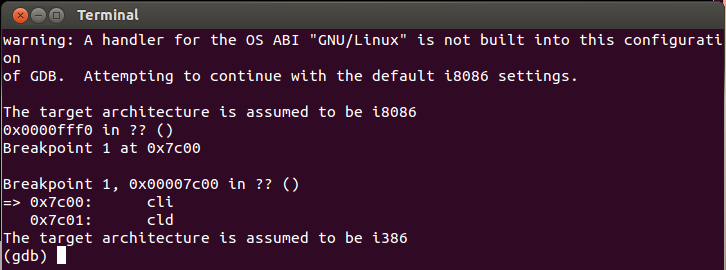
从图中可以看出，打印断点处的指令显示的是32位线性地址，不是实模式下的真实地址。

（2）再次修改tools/gdbinit文件如下：



其中第三条指令代表在0x7c00处设置断点，0x7c00是bootloader被加载入内存的起始地址，第四条指令表示开始执行直到遇到断点，最后一条指令表示调试CPU切换到80386模式，为启动ucore做准备。

执行make clean清除上次make产生的文件，再次执行make debug，这一次gdb直接停在了0x7c00处：



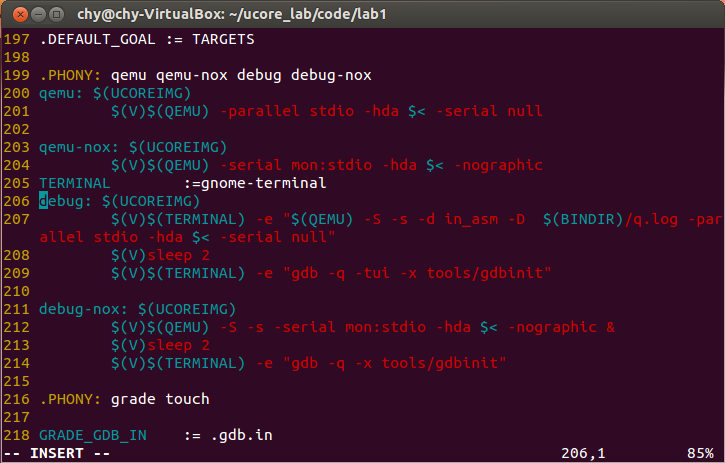
程序正常执行到了断点处。

（3）为了方便跟踪代码的运行，我们对Makefile文件进行了修改，将debug处的内容修改如下：

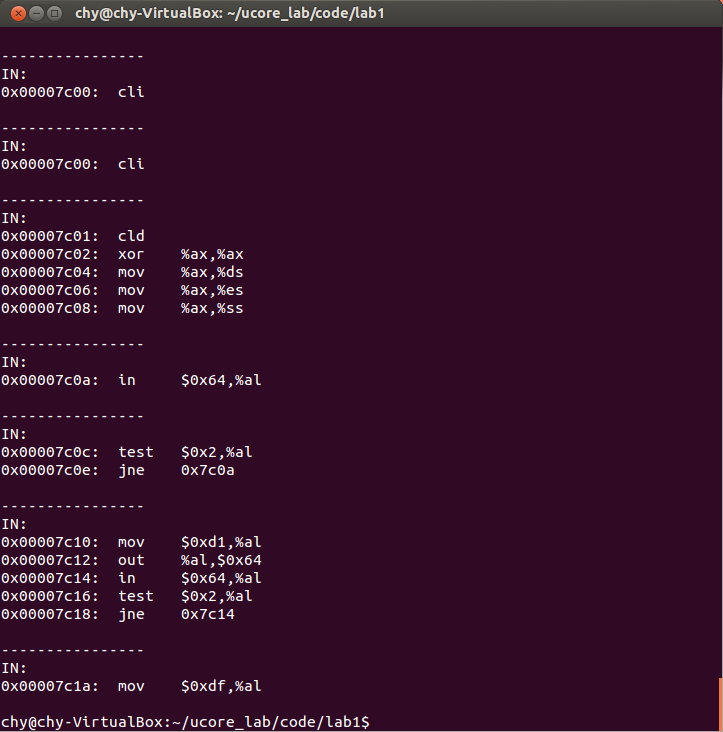
debug: $(UCOREIMG)

$(V)$(TERMINAL) -e "$(QEMU) -S -s -d in\_asm -D $(BINDIR)/q.log -par allel stdio -hda $< -serial null"

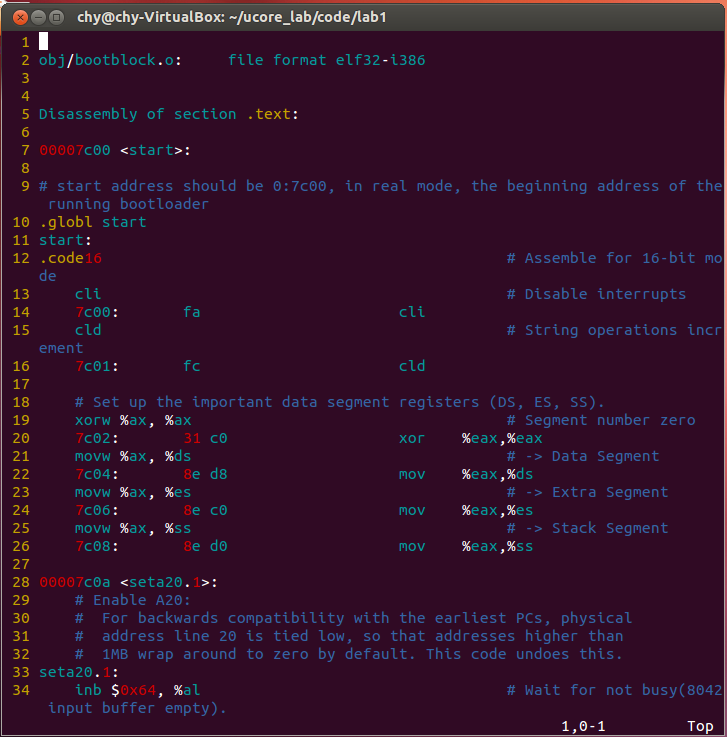
$(V)sleep 2

$(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"

主要变化是第207行增加了-d in\_asm -D q.log参数，可以保存运行的汇编指令到q.log文件中，方便后续比较。此次make debug后得到了bin/q.log文件，其部分内容如下：



图中展示了0x7c00后执行的部分指令，我们打开obj/bootblock.asm查看代码如下：

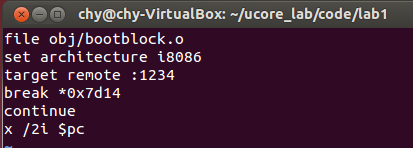


然后再打开boot/bootasm.S查看代码如下：

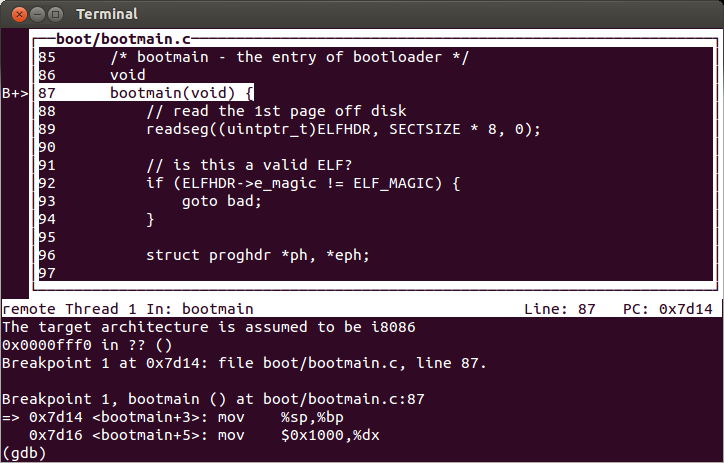


容易发现，q.log显示断点之后的代码与bootasm.S、bootblock.asm中的代码是一样的。也就是说，bootblock被安装在了0x7c00处的内存空间中。以Intel 80386为例，计算机加电后，CPU从物理地址0xFFFFFFF0（由初始化的CS:EIP确定，此时CS和IP的值分别是0xF000和0xFFF0）开始执行。在0xFFFFFFF0这里只是存放了一条跳转指令，通过跳转指令跳到BIOS例行程序起始点。BIOS做完计算机硬件自检和初始化后，会选择一个启动设备（例如软盘、硬盘、光盘等），并且读取该设备的第一扇区（即主引导扇区或启动扇区）到内存一个特定的地址0x7c00处，然后CPU控制权会转移到那个地址继续执行。

（4）要在0x7d14处设置断点，修改tools/gdbinit文件如下：



执行make debug后可以发现，此处是bootmain函数的入口地址：



参考资料：

[1] 唐源棕．ucore lab1[OL]．<https://blog.csdn.net/tangyuanzong/article/details/78595854>，2017-11-21．